# MODIFIED MCLAREN-MARSAGLIA PSEUDO-RANDOM NUMBER GENERATOR AND STOCHASTIC KEY AGREEMENT

By

RICHARD LLOYD CHURCHILL

Bachelor of Science in Chemistry and Philosophy

Oklahoma State University

Stillwater, Oklahoma

1980

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2011

MODIFIED MCLAREN-MARSAGLIA PSEUDO-

RANDOM NUMBER GENERATOR AND

STOCHASTIC KEY AGREEMENT

Thesis Approved:

Dr. H. K. Dai

Thesis Adviser

Dr. John P. Chandler

Dr. Douglas Heisterkamp

Dr. Mark E. Payton

Dean of the Graduate College

## TABLE OF CONTENTS

Chapter Page

# LIST OF TABLES

# LIST OF FIGURES

Figure Page

# CHAPTER I

# INTRODUCTION

The intent of this thesis is to provide a description and implementation of a scheme whereby two parties may agree upon a set of random binary digits, which may be used for cryptographic purposes, and which provides sufficient security in the process of agreement to regard the result as reasonably cryptographically strong.  The difficulty in providing a new approach to solving this problem warrants some justification, as well as a discussion of why it is difficult.

## Some Basics Regarding Cryptographic Systems and Their Goals

Cryptographic systems are a critical part of the overall modern security environment.  While encryption algorithms, or ciphers, have a long history in communications, they play an increasingly critical role in modern communications.  This increase may be ascribed to the importance of telecommunications networks, such as the Internet, as described by Metcalfe's Law [1].  This "rule of thumb" states that the value of a telecommunications network increases proportionally to the square of the number of compatible items attached to it.  Various formulations of this "law" substitute terms such as "user" and "devices" for "compatible items." But, as a rule of thumb and a description of the ability of networks to increase in value as they grow, along with increasing the value of attached items, these alternate formulations are interchangeable according to circumstance.  This increase in value often entails the need to protect the items information attached cryptographically.

The fundamental goals of cryptographic systems are confidentiality, integrity and availability [2]. Confidentiality assures that information is known only to those trusted with it. Integrity assures that it is not inappropriately altered. Availability assures that it can be accessed when needed. By extension, more refined cases of each may be identified, such as authentication and non-repudiation. Authentication includes such concerns as assurance that entities are who or what they claim to be. Non-repudiation deals with assurance that those who have handled information, or more importantly originated it or agreed to it, cannot later deny having done so. In seeking to achieve the latter two goals, confidentiality plays a critical role. “Secrets” play a role in assuring availability by preventing unauthorized destruction or alteration of information, as well as assurance of origin.

Cryptographic mechanisms and protocols rely heavily upon the difficulty of guessing or predicting numbers, passwords or pass phrases. In defining protocols and mechanisms, these numbers are often specified as being secret and in many cases random. These random numbers play a large number of pivotal roles in cryptographic protocols, including as keys for encryption and decryption, nonces, initialization vectors and more. Many passwords and pass phrases being used to generate numbers that hopefully display good apparent randomness. In practice such numbers rarely are truly random, as they are generated by pseudo-random number generators (PRNGs), or using chaotic processes related to user actions. Thus, the process of generating numbers that are either truly random or sufficiently apparently so is one of the central problems of cryptographic systems. In many cases, these random numbers must be shared in some manner for the cryptographic protocols to work correctly. Such is the case with session keys, which may be used to encrypt the bulk of communications between two parties.

Cryptographic keys play a central role in security, as was first described clearly by Auguste Kerckhoffs [3] [4] and further described by Claude Shannon [5]. In brief, the fundamental principles he laid out are that in communication using ciphers the cipher system must be assumed to be known, and therefore the security of the system rests in the keys used.

## Session Keys and Public Key Encryption

A session key is a key used to allow the parties participating in a communication to efficiently encrypt and decrypt the traffic of that communication.  Such communications are typically conducted using a symmetric cipher.  A symmetric cipher is one that uses the same key to both encrypt plaintext and decrypt the resultant ciphertext.  Since both the participants must have copies of this key to communicate, yet the key must be kept secret for the contents of the exchanges to be secure from unauthorized parties, the participants must have some means by which to either agree upon or communicate the session key among them.

Historically, there have been many approaches to deal with the problems posed by session keys, but the present state of cryptographic art uses a specific class of ciphers to facilitate such exchanges.  These are public key cryptosystems.  Such systems rely on algorithms that are assumed to be difficult to reverse.  They use separate but related keys to encrypt and decrypt messages.  Due to the nature of their underlying mathematical problems, the algorithms are generally slow, when compared to symmetric key ciphers, and are thus not suitable replacements for symmetric key systems where performance is an issue.  This is typically the case when extended communications will occur or are voluminous, and when data streams are involved.

The relationship between the keys of key pairs in a public key system, one key called public, the other private, is that each member of a pair may be used to reverse encryption by its mate: The public key will decrypt messages encrypted with the private key, and the private key will decrypt messages encrypted using the public key.  Further, it is assumed that though the keys are derived from the same material, the private key cannot be easily deduced from the public key.

The assumption of difficulty is often referred to as the Diffie-Hellman assumption or conjecture, which is related to the Diffie-Hellman problem. This was developed in relation to the development of the Diffie-Hellman key exchange [6][7][8] (which discussed in greater detail later in this chapter). The Diffie-Hellman problem is simple. Given a large integer $g$ (called the generator) and large prime $n$, such that $g$ is primitive mod $n$ (in simple terms, all integers in the interval $[1, n)$ can be expressed as integral powers of $g$), a pair of values $g^x$ mod $n$ and $g^y$ mod $n$, determine $g^{xy}$ mod $n$, when $x$ and $y$ are not known. The assumption or conjecture is that this is a difficult problem to solve for the selected generator used by the public key cipher system in question. While the problem itself is applicable only to mathematically-based public key systems that utilize exponentiation, the conjecture, when generalized to mathematics-based public key systems, is common to all such systems that are presumed secure. If the mathematical problem upon which a system is based is subsequently found to be less difficult than previously believed, the system is correspondingly weakened.

An example of presumed difficulty less than at first believed is the Diffie-Hellman key exchange protocol, which is based on the discrete logarithm problem. While Diffie-Hellman is still believed to be secure if the keys are properly selected, the selection process is more critical than at first believed, and significant efforts and progress at solving discrete logarithm problems has been made [9][10].

Another public key system is the ubiquitous RSA [11], developed by Rivest, Shamir and Adelman. It is based on the difficulty of determining the prime factors of very large numbers. While the factorization problem is still considered hard, the ever increasing power of computers, combined with the development of massively parallel attack schemes, has continually eroded the effective strength of keys of shorter length. Where 512-bit RSA keys might once have sufficed, the European Union recommended that after 1998 only keys of 1024 bits and longer should be used. Further, R.D. Silverman of RSA Laboratories stated in 2000 [12], “We do not believe that any public key size specified today should be used to protect something whose lifetime is more than 20 years." Of particular concern should be the development of an algorithm for quantum computing platforms by Peter Shor. It provides a means of computing the prime factors of numbers in polynomial time [13]. While quantum computing remains formative, and the error term in this algorithm grows with increases in the size of the number to be factored, we must anticipate that these problems will eventually be resolved, rendering RSA in effective for cryptographic purposes. Even if these problems are not completely resolved, and the error term in Shor's algorithm cannot be tightly limited, any decrease in the search space that may result from its application will still constitute a potentially crippling weakening of the RSA protocol.

This conclusion should be extended to all mathematically based systems. It is safer to assume that there will always be a better mathematician, and computing power will always increase. A compounding problem is that discovery of a faster way to solve any of these problems will not necessarily be made public, as it may be to the advantage of the discoverers to keep the development secret. This is clearly demonstrated by the sale of German Enigma machines to other nations by the victorious Allies following World War II, without the disclaimer that the Allies had broken the Enigma cipher system [14].

## Bennett and Brassard Key Agreement and Continuing Research

In 1984, Bennett and Brassard published a paper [15] proposing means by which two parties could perform a series of communications whereby they would agree on a set of random bits securely. The claim that these could then be used as a one-time pad (OTP, discussed in chapter 2) for encryption was significant, as a correctly implemented OTP system is provably perfect [5].

The Bennett-Brassard system utilizes a dedicated optical fiber channel between the two correspondents, plus a public side-channel. Both ends of this channel must be equipped with a polarizing filter that can be switched between orientations rapidly, and a random number generator.  One end (Alice's) is equipped with a photo emitter capable of emitting single photons. The other is equipped with a detector capable of detecting individual photons.  Alice and Bob first agree which of the possible polarizations of photons represent binary 1's and which 0's. Alice starts the process by sending a stream of individual photons, each polarized to one of the agreed orientations randomly, with Alice recording the orientations.  Bob uses his random number generator to "guess" the orientation of each incoming photon.  As this guess is expressed as an orientation of the polarizing filter he possesses, Bob will either detect a photon, having guessed right, or not, having guessed wrong.  Bob records both his guesses and whether or not he received a photon corresponding to that guess.

After Alice has sent some agreed number of photons, Bob sends Alice his guesses as to the orientations of the photons.  As Alice knows the orientations of the photons sent, she confirms which guesses were correct.  Additional steps allow Alice and Bob to confirm that bits (referred to as qubits in a quantum mechanical application) were in fact received correctly, and that nobody was attempting to eavesdrop on the process.

Due to the properties of polarized photons (as sent over the dedicated channel), the scheme is provably secure against passive eavesdropping on that channel, since any measurement of those photons by an eavesdropper will perturb the results, while passive eavesdropping cannot determine the orientation of the photons.

While the scheme is not without problems, such as the need for the dedicated optical fiber channel, it is interesting both because it was the first effective quantum cryptographic protocol proposed, and it does not rely upon a mathematical problem that is assumed to be hard to solve. While it might be broken via a radically improved understanding of quantum behaviors, the present understanding of these behaviors preclude a successful break.

The Bennett-Brassard scheme, as well as subsequent quantum cryptographic proposals, highlights some currents in modern cryptography. One current is the continuing effort to produce ever stronger, more secure means of communicating sensitive information. Another is the importance, the value, and the difficulty, of two parties being able to agree upon a set of random numbers that may be used in establishing cryptographically secure communications.

Regarding the first current, the previously mentioned efforts to find faster ways to solve the mathematical problems underlying public key systems, the on-going cryptanalysis targeting current symmetric ciphers, and the efforts to develop newer, stronger ones, illustrate at least a perception of risk and a desire to mitigate such risk before it becomes substantial. An examination of the Advanced Encryption Standard (AES) competition and continuing analysis of its winner helps clarify the issue.

In 1997, the U.S. National Institute of Standards and Technology (NIST) announced a competition for a replacement for the Data Encryption Standard (DES), followed later in the year by a request for submissions. By mid-1998, fifteen proposals were submitted to NIST, and then to the public for analysis. From these, five finalists were selected, with limited numbers and types of corrections to the proposals to deal with flaws identified in the first round. From the finalists, the Rijndael algorithm was selected in 2000 as the new AES algorithm. Since then, the AES cipher has been subject to continuing cryptanalysis, with improvements in analytical techniques gaining steadily, if slowly [16][17].

The second current includes the effort to develop communications schemes that utilize quantum mechanical phenomena in order to achieve security. The Bennett-Brassard proposal was the first entrant in the quantum field. Subsequent proposals exploit phenomena such as quantum entanglement. While some progress has been made in this realm, there are also significant problems, not least the potential for statutory restrictions.

A further illustration of the second current is the number and variety of PRNGs that have been and continue to be developed. These developments are in no small part due to the problems with existing generators, including performance issues and exploitable flaws.

## Issues and Goals in Cryptology and Randomness

If the pseudo-random number generator (PRNG) used in a cryptographic application is well designed, correctly implemented and properly used, there is generally no significant loss in security due to its substitution for a random number generator (RNG). The problems for the cryptographer are to determine when the use of a PRNG is appropriate and safe, which PRNG to use and how to use it safely.

Where “true” random numbers are required in the absence of a suitable hardware random number generator (RNG), chaotic processes, based upon user input device events and timing, are typically used. But, these are incapable of generating large volumes of random material. They are at best expedients for limited cases, such as providing the seed for a PRNG, which is then used to generate longer sequences of pseudo-random values.

Techniques utilizing physical phenomena continue to be developed, and improve the generation rates that can be achieved. Yet, the problem for most applications remains generation of large quantities of numbers that are sufficiently apparently random, and thus “secure,” by software means alone, and thus by PRNGs.

By use of the adjective “secure” for a pseudo-random number, a cryptologist means a number that, when taken as a member of a sequence generated by a PRNG, displays certain characteristics relative to the sequence and its place in that sequence, depending on the application to which the number will be applied. Ideally, it should be impossible to determine the value of any element of such a sequence, even if given the algorithm used to generate it, plus all elements of the sequence preceding and following it, but not the seed or state of the generator that produced the sequence.  It should even be impossible to determine any element of the sequence given the state of the generator immediately after it was generated.

Such goals are extremely difficult to achieve, and when achieved may be too expensive, in one or more senses, to be practical for a given application.  Fortunately, not all applications for random values in cryptology require ideal characteristics.  Thus, cryptologists are left with the problem of developing or selecting PRNGs that are suitable for varied applications and environments.

From the degree to which a PRNG achieves the ideal characteristics sought, we draw the distinction between "strong" generators and "weak" generators, and must deal with the problem of determining the relative strength of different generators.  "Cryptographically strong" generators are those that most nearly attain the ideal goals.

## Pseudo-Random Number Generators (PRNGs)

Due to the close relationship between PRNGs and synchronous stream ciphers, such ciphers are dealt with in the brief survey found in Chapter II: Many synchronous stream ciphers utilize PRNGs, generally in complex combinations, as the sources of the running keys.  Any "good" PRNG (meaning "cryptographically strong") can be used as a basis for a synchronous stream cipher.  For purposes of the implementation contained in this thesis, specific PRNGs are used as feeds for mixing algorithms.  But, as the mechanism described is a selection and mixing process that uses multiple PRNGs as sources, as well as an RNG as a source of noise, the basic process is in reality agnostic regarding the constituent PRNGs used in any implementation of the scheme, so long as those PRNGs satisfy the requirements for the level of security to be achieved, and all parties involved have identical implementations.  It is therefore appropriate to be cognizant of the types and varieties of PRNGs (and implicitly stream ciphers) available, as well as their relative advantages and drawbacks.

The study of PRNGs, both for purposes of developing better ones and identifying the weaknesses of existing ones, is the stuff of the intelligence and security agencies around the world that seek to read others' communications or keep their own nations' secure, as well as academia, corporations and interested individuals, including criminals.  While research has provided a number of PRNGs that are to varying degrees considered cryptographically strong, all such PRNGs fall into either of two categories, both of which entail very real problems.  The vast majority are periodic, which means that the output stream eventually repeats itself in a fixed pattern of fixed length.  Such repetitive sequences are sometimes called linear recursions.  In contrast, aperiodic generators do not repeat in a fixed, recurring pattern, even though subsequences will recur at irregular intervals.

## Periodic Generators

The most common forms of PRNGs are periodic.  Such generators fairly rapidly reach a state that serves as the entry-point into a cycle that is repeated ad infinitum.  Many start in such a state.  In general, the length of the cycle a generator may achieve depends on the size of the internal state of the generator.  For a given internal state size, and barring external inputs, the size of that state sets an upper bound for the length of the cycle that may be achieved.  But, the relationships between state size and cycle length, and cycle length and cryptographic strength are not monotonically increasing functions.  Some generators can fall into degenerate cycles that are far shorter than might be expected based on the size of the internal state.  (See the discussion of non-linear feedback shift registers in Chapter II for additional discussion of this problem.)

Some PRNGs with very large internal states and long periods are far weaker, in cryptographic terms, than other PRNGs with shorter periods.  For example, consider the cases of the Mersenne Twister (MT) and Blum-Blum-Shub (BBS) PRNGs.  The MT generator is exceptionally fast in operation, and has an exceptionally long period.  Yet, it is cryptographically weak.  Its internal state can be determined easily, given a sample of several hundred consecutive outputs, since each successive output reveals a distinct portion of the internal state of the generator, and that state changes substantially only once every 624 32-bit outputs.  In large part due to performance constraints, almost all implementations of the BBS generator have smaller internal states and shorter periods than the MT generator, yet they are cryptographically stronger.  This is because only a small fraction of the internal state is ever seen in the output stream, and these portions are in a sense "disjoint," the internal state of the generator being substantially altered with each output.  (Both the MT and BBS generators will be discussed in greater detail in Chapter II.)

The presence of a repeating cycle of outputs is a flaw in PRNGs for many applications.  Like the MT generator, many PRNGs are subject to effective cryptanalysis given a relatively small sample of consecutive outputs, as so much information about the internal state of the generator may be derived from those outputs.  Even for cryptographically strong generators, the sheer volume of outputs required for some applications (encrypted video, for example) may exhaust a PRNG's cycle.  Once the cycle has been exhausted, it may not be necessary to determine the internal state, or even the generator used.  Also, since any repeating cycle can be treated as a linear recursion, and a linear feedback register generated from that recursion will reproduce the recursion, even discovery of subsequences by an attacker may render a generator compromised.  (See Chapter II for a discussion of the Berlekamp-Massey algorithm.)

## Aperiodic Generators

The less studied class of PRNGs is comprised of aperiodic generators.  The most easily recognized members of this class are based upon irrational numbers.  As these numbers cannot be expressed as simple fractions, they have infinite, non-repeating sequences of digits in the fractional portion of their representation, regardless of the integer base used to express them. While techniques exist for the calculation of arbitrary segments of the fractional portion of some irrational numbers, the space and time required becomes large as the order of the digits decreases.

Thus, though it may be possible to calculate vast numbers of digits in such manner, the volume and time demands of many cryptographic applications can make this impractical.

A newer group of members of this class are based upon the structure of quasi-crystals. Quasi-crystals are similar to crystals in that they have a distinct structure displaying many of the characteristics of crystals, but they lack the precise, regular lattice of true crystals. The small variations in structure can be used as a source for non-repeating series of values (think of drawing a line across a quasi-crystal's face, then measuring the distance of successive "closest" atoms to that line, from the line), or as a computational model (determining the same distances). Therefore, quasi-crystals can, at least theoretically, provide infinite length sequences of non-repeating values. Practical considerations, however, intervene. Just as an actual quasi-crystal has limited dimensions (however vast these may be in terms of the number of atoms involved), the state and precision of a computational model of a quasi-crystal will limit the sequence to a repeating cycle, once the computational state duplicates any prior state, and increasing the state size or numerical precision of the model eventually limits the practicality of the approach in any case.

Thus, both of these groups still encounter the same fundamental problems. All real computing environments are finite. And as with conventional PRNGs, computer environments have a finite number of possible states. The number of theoretically possible states is simply far larger than for most well-studied conventional PRNGs. Therefore, absent external input, once a system enters a state previously entered, it will proceed to the same state it entered following the prior occurrence of that state, and computation of the desired sequences will eventually fail.

It is should be evident that in order to achieve effective aperiodicity, without running afoul the limitations of cost, space, etc., a PRNG must utilize some form of external random or chaotic stimulus.  Means by which such external stimuli may be injected into a PRNG will be discussed in Chapters III and IV.

## Evaluating Generators

Regardless of the specific PRNG in question, some means of evaluating the strength of that PRNG is needed.  How can it be determined whether it is effective at generating sequences that are sufficiently apparently random to justify treating them as random?  There must be some means of determining success, and more importantly to identify failure.  The growing importance of the Internet and the commerce it supports has forced governments and governmental agencies, as well as standards bodies, to address this question publicly.  While many nations have produced or adopted standards or specifications, it is reasonable to limit concern to a few from major industrial nations or duly recognized standards bodies.  Such standards must provide descriptions of their requirements, include testing procedures that are reasonably consistent and thorough, and help in defining goals further.

Three specifications for evaluating PRNG are considered here.  These are the U.S. Federal Information Processing Standard (FIPS) publication 140-2 [18], Institute of Electrical and Electronics Engineers (IEEE) specification 1363, and Bundesamt für Sicherheit in der Informationstechnik (BSI) AIS 20, version 1 [19].  The National Institute for Science and Technology (NIST) is another source of information regarding test suites.

## BSI AIS 20

The Bundesamt für Sicherheit in der Informationstechnik (BSI – Germany’s Federal Office for Information Security) has published a set of requirements for four classes of deterministic random number generators [19].  While other standards and specifications exist (i.e. FIPS 140-2), the problem is often finding an understandable description of the classification system, and of the means of testing proposed algorithms’ compliance with the requirements.  The BSI “Application

Notes and Interpretation of the Scheme," AIS 20, version one, "Functionality classes and evaluation methodology for deterministic random number generators," provides such descriptions.

In this and other standards, a PRNG is defined by a 5-tuple $(S, R, \varphi, \psi, p_A)$ comprised of a finite set of generator states $(S)$, a set of possible outputs $(R)$, a state function $(\varphi : S \to S)$, an output function $(\psi : S \to R)$ and a probability metric of the random distribution of the seed $(p_A)$.

This application note defines four classes of deterministic random number generators: K1, K2, K3 and K4. (For our purposes, deterministic random number generators are identically pseudo-random number generators, and we will use the latter term, though quotations may include either.) Membership in any class implies membership in the next lower class, with the K4 class being the "strongest" in cryptographic terms. Thus, any K4 class generator is a member of K3, and recursively K2 and K1.

Without delving into the detailed requirements for each class, the base, additive requirements for these classes are as follows.

> K1 – There is a high probability that disjoint subsequences of the output stream are distinct.
>
> K2 – The output must pass the tests specified in the AIS 20 document.
>
> K3 – The entropy of the seed probability metric must be adequately high, and it must be "practically impossible" for an attacker to determine the predecessor and successor outputs of any subsequence of outputs shorter than the cycle of the generator, given the defining tuple, but not the state of the generator at any point in the subsequence.

> K4 – “It must be practically impossible for an adversary to work out the predecessor random number $r_{i-1}$ from knowledge of the internal state $s_i$. The adversary's assumed attack potential depends here on the strength of mechanism. Even using the most advanced know-how currently available, the probability of guessing (realized by a reasonable partial exhaustion) may at most be negligibly greater than if $s_i$ were not known. It is assumed that the adversary knows the defining 5-tuple.” (The salience of the requirement regarding predecessor internal states will be discussed later, in conjunction with Vernam ciphers, One-Time-Pads and Claude Shannon’s work.) The objective of this is, “Protection against reconstruction of old random numbers from a known internal state.”

BSI AIS 20, version 1, specifies the tests for randomness that must be passed to satisfy the requirement for membership in the K2 class. These are labeled T1 through T5. Basic descriptions of these tests are as follows.

> T1 – monobit test – the number of one-valued bits, in 20,000 bits of output, is in the range (9654, 10346), and the number of zero-valued bits is in the same range.
>
> T2 – poker test – the values of four-bit groups (nybbles) are near equally distributed.
>
> T3 – run test – the numbers of runs of ones, or of zeroes, of various lengths, fall within statistically acceptable ranges.
>
> T4 – long run test – no runs of 34 bits in length or longer occur.
>
> T5 – autocorrelation test – there can be no consistent correlation between bit values at regular intervals.

These tests must not be regarded as “complete” in any sense, but serve as a basic set that provides reasonable assurance of the ability of a PRNG to produce output sequences of moderate length that display reasonably apparently random characteristics, and satisfy the requirement for the K2 class of generators.  Source code in C for these tests is contained in Appendix B.

## FIPS Publication 140

The U.S. government has a long-established standards agency, the National Institute of Standards and Technology (NIST), which, among many other standards, has published a growing series of Federal Information Processing Standards (FIPS).  NIST's FIPS publication 140 series, comprised of versions 140-1, 140-2 and 140-3 (Draft), is the U.S. Federal standard addressing the security of cryptographic modules, including PRNGs.  As does the BSI AIS 20 specification, the series defines security levels and requirements for meeting them.  But, rather than addressing PRNGs only, the FIPS 140 series addresses a much broader range of topics, including PRNGs.  Specifically, Annex C: Approved Random Number Generators (Draft), July 21, 2009, lists a total of six deterministic random number generators that have been approved.  No non-deterministic ones are listed.  No specific tests for validation of PRNGs are given, but the discussion of their use is important in understanding why the problem of designing or selecting PRNGs is critical to cryptographic systems.

Relative to the requirements of the FIPS publication 140 series, the NIST has the Cryptographic Module Validation Program (CMVP), which performs testing on cryptographic modules as a means of assuring users of the characteristics and strength of software and hardware modules.  The CMVP is a function of the NIST's Computer Security Division, Computer Security Resource Center (CSRC).  The CMVP utilizes accredited test laboratories to test the performance of characteristics of modules, rather than providing test suites.

What guidance the NIST, in the form of the CSRC, provides is reference to applicable international standards: ISO/IEC19790 and ISO/IEC 24759, which were derived from FIPS 140-1 and 140-2.  The whole is thus circular, and not as supportive of self-testing as might be desired.

## IEEE Standards 1363

The Institute of Electrical and Electronics Engineers (IEEE) is an accredited international standards body, and as such has forayed into the field on many fronts, both sponsoring periodicals and conferences with associated proceedings. The primary standards developed by IEEE regarding security and encryption are the IEEE 1363 specification family, composed of IEEE 1363-2000 IEEE Standard Specifications for Public Key Cryptography and IEEE 1363a-2004 Standard Specifications for Public Key Cryptography – Amendment 1, Additional Techniques. Continuing efforts include proposed standards IEEE P1363.1 Draft Standard for Public-Key Cryptographic Techniques Based on Hard Problems over Lattices, IEEE P1363.2 Draft Standard for Specifications for Password-based Public Key Cryptographic Techniques and IEEE P1363.3 Draft Standard for Indentity-based Public Key Cryptography Using Pairing.

While these standards deal with numerous public-key crypto-systems, specifying protocols and algorithms, virtually all of which require use of either random values of random number generators, they are largely silent on the issues pertaining to such generators, relying on other sources for such issues. It is perhaps an indication of the difficult of the overall problem of random number and pseudo-random number generators that the issue is deferred to other sources. Still, these standards are interesting from many perspectives, and particularly their discussions of and reliance upon “hard problems.” Despite the interesting character of these standards, and their dependence upon quality random numbers, they are not particularly useful for the present problem.

## Other Test Software

Test suites for the quality of a sequence of pseudo-random outputs are problematic, since the fact that all pseudo-random generators are deterministic, but the “good” ones are of sufficient complexity that any correlations that may exist in their output streams may be difficult to identify, and thus to test for. The problem for those developing tests for any problem beyond the simple and clearly understood set (such as those identified and tested for in the AIS 20 test suite) is thus to identify potential correlations, probabilistic skewing, etc., and develop tests for them, without necessarily knowing whether any generator may display the hypothesized problem. Yet any such problem, once identified, may present an exploitable opening in some number of generators.

Considerable and on-going academic and practical study of the problem has produced a substantial body of literature. Working groups sponsored by governmental agencies contribute to this body on a regular basis.

The National Institute for Science and Technology (NIST) maintains a list of “commercial” PRNG test batteries, but with the caveat that these are not endorsed by NIST. These are the following.

The pLab Project – a Web site maintained by Peter Hellekalek of the University of Salzberg, with a test suite maintained by Richard Simard of the University of Montreal.

The Information Security Research Centre – a project maintained by Information Security Institute of the Queensland University of Technology.

The Information Security Research Centre (Crypt-X) – a subproject of the Information Security Institute of the Queensland University of Technology that deals with “black box” testing of random number generators (viewed as stream ciphers), block ciphers and key generators.

The DIEHARD Test Suite – a test suite and project maintained by Florida State University, and a direct ancestor of the DIEHARDER test suite (which is not listed by NIST).

FIPS 140-2 – dealt with above.

ENT – a project of John Walker, Fourmilab, Switzerland.

Any of these may be appropriate as a source of more complete and formal testing of a random or pseudo-random number generator's output, but are regarded as beyond the present scope, which focuses on the mixing and selection methods described in Chapter IV.

## The Problem Addressed

The inspiration for this thesis is the Bennett-Brassard protocol, and on the belief that problems presently regarded as hard may prove less so in the future. Relying as Bennett-Brassard does upon quantum physical properties and specialized hardware, its wide-spread implementation is presently problematic. In particular, the problems of generating individual photons and detecting them at great distances with sufficient consistency to allow agreement on long strings of random bits are consequential. Single mode optical fibers of sufficient quality to reliably carry individual photons over distances of thousands of miles do not at present exist, and certainly have not been laid in sufficient numbers to afford dedicated two-party links for any meaningful population. Similarly, the equipment necessary to reliably produce single photons travelling in the right direction to pass through a polarizing filter and then down an optical fiber is common and affordable. Neither is the equipment to reliably detect individual photons. Thus, the question addressed here is whether and how at least some of the qualities found in Bennett-Brassard may be achieved without resorting to quantum phenomena and putatively hard mathematical problems.

The scheme described and implemented here utilizes a set of PRNGs, coupled with a selection and mixing process, applied to a string of random bits shared by two or more parties, to agree upon a shorter string of bit values, with the property that the mechanism injects aperiodicity into the selection and mixing process. As the overall scheme is indifferent to the PRNGs used, Chapter II is a survey of stream ciphers. This is appropriate since especially for synchronous stream ciphers, we may regard stream cipher algorithms as being PRNGs. Chapter 3 examines bit selection and mixing functions based on the McLaren-Marsaglia PRNG. These functions provide the basis for the overall schemes selection and mixing functions. Chapter IV describes the implementation of the scheme, with explanations of the components and their interactions. Chapter V deals with the questions of performance, and with the security of the scheme.

## Some Existing Alternatives Based Upon Mathematical Problems

As noted at several points in this thesis, key agreement protocols (which inherently includes key exchange or transmission protocols) already exist. They are critical elements in the existing cryptographic landscape. But, what are they? And, why look for an alternative?

As previously discussed in lesser detail, the Diffie-Hellman [6][7][8] key agreement protocol was the first publicly developed and discussed key-exchange protocol, and is either the basis of or useable in several other key-exchange protocols. Its security is based on the presumed difficulty of calculating discrete logarithms. As it is so central to the development of several protocols, and illustrates some of the problems the protocol presented in this thesis attempts to address, it is worth a longer, more detailed discussion.

The Diffie-Hellman protocol is as follows.

By prior arrangement, Alice and Bob select two numbers, $n$ and $g$, such that $n$ is a large prime, and $g$ (called the generator) is primitive mod $n$. These numbers need not be kept secret. When Alice and Bob need an agreed secret key $k$, they do the following.

1. Alice chooses random integer $x$, computes $X = g^x \bmod n$, and sends $X$ to Bob, while Bob chooses random integer $y$, computes $Y = g^y \bmod n$, and sends $Y$ to Alice. Both $x$ and $y$ should be ‘large.’

2. Alice computes $k = Y^x \bmod n$, and Bob computes $k' = X^y \bmod n$.

Since $n$ is prime and $g$ is primitive mod $n$, we have

$$k = Y^x \bmod n = g^{y^x} \bmod n = g^{xy} \bmod n = g^{x^y} \bmod n = X^y \bmod n = k'.$$

The presumed security derives from the problem of calculating discrete logarithms, base $g$, which is related to the problem of factoring large integers, hence the requirement that $n$ be large. Since an eavesdropper will at most know $n$, $g$, $X$, and $Y$, but neither $x$ nor $y$, the protocol is believed secure against eavesdropping, absent efficient means to determine discrete logarithms base $g$. But, a successful guess of either $x$ or $y$ will produce k, compromising the communications.

Analysis [20] has shown that not only must $n$ be both large and prime, $(n - 1) / 2$ must also be prime. While this assures that the Pohlig-Hellman algorithm [20] will not reveal $x$ or $y$, it also limits the number of primes that may be used, as the selection of $n$ is limited by the Sophie Germain primes. Whether this is an exploitable fact remains an open question, but given the fact that the distribution of primes becomes progressively sparser among integers as magnitude increases, further thinning of eligible primes cannot be regarded as wholly harmless. Even with selection of a $g$ that is primitive mod $n$, problems arise, as this reveals low-order bits of $x$ and $y$ [21]. As a result, $g$ may be selected based upon its generation of a large subset of $n$, rather than being primitive.

An additional problem [21] is that if an attacker can cause any level of bias in the selection processes for the integers $x$ and $y$ used by Alice and Bob, the discrete logarithm problem can be simplified, thus weakening the system.

One of the reasons the Diffie-Hellman protocol is not used widely for key-exchange is that it provides no authentication, and is thus susceptible to a man-in-the-middle attack. If an attacker can intercept $X$ and $Y$ and substitute his own values for these, he is in a position to read all traffic between Alice and Bob. Even if he is quickly detected, the information already intercepted may prove damaging. Authentication can be added to the protocol, but involves use of certificates, which add their own problems to the overall scheme.

Knapsack problems as a basis for public key systems were first explored (at least publically) by Merkle and Hellman [22][23], and extended to include authentication by Shamir [24]. The problem is based upon a super-increasing series. That is, the series *S* is super-increasing if for any element $s_i \in S, s_i > \sum_{j=1}^{i} s_j$. Given a sum of members of *S*, it is easy to solve the problem of which elements of *S* are elements of the sum. (This is the easy knapsack problem.) At the same time, it is possible to construct a non-super-increasing series that is still increasing, from the super-increasing series, such that the two series have the same solution. The non-super-increasing series problem is much harder. The pair of series comprise a key pair, with the super-increasing series being the private key, and the non-super-increasing series the public key. Unfortunately, though the mathematics is interesting, the solution isn't as the knapsack problem has been successfully attacked, and by several cryptographers, as have variations on the theme. Schneier [25] discusses many of these attacks and variants in some detail, therefore the details are left to that resource.

The most common public-key system in current usage is the RSA key exchange protocol, which is a member of the broad family of RSA protocols. Developed by Rivest, Shamir and Adleman [11], it relies upon the difficulty of factoring very large numbers into their constituent primes. The core concepts of the mechanism are as follows.

To generate keys, select two large primes *p* and *q*, and calculate $n = pq$ and $\phi = (p - 1)(q - 1)$. Select an integer value *e*, the encryption exponent, such that $\gcd(e, \phi) = 1$, then calculate *d*, the unique decryption exponent, using the extended Euclidean Algorithm. (The integers *d* and *e* are multiplicative inverses of each other mod $\phi$, since $de$ mod $\phi = 1$.) The public key is $n$ and $e$, and the secret key is $n$ and $d$.

The ciphertext $c$ of a plaintext message $m$ (treated as a number in the range [0, $n$ -1]) is obtained by computing $m^e$ mod $n = c$. The plaintext message is obtained from the ciphertext by computing $c^d$ mod $n = m$.

As noted elsewhere, the strength of this system has continually eroded, though it hasn't been broken: the problem is one of increasing computational power. There is a problem, though. As stated by Schneier [25], "It is *conjectured* that the security of RSA depends on the problem of factoring large numbers." There is, at present no proof that factorization is required, only the present understanding of the problem which indicates that this is so. As previously noted, Shor's algorithm [26], or some other advance in factoring may yet prove a serious problem for RSA's security. In any case, the RSA algorithm tends to be very slow, particularly when using long keys, which are also presumably the more secure ones.

Menezes, van Oorschot and Vanstone [21] address one way in which the factoring problem might be avoided. Any method that can derive $d$ directly from $e$ and $n$ would significantly simplify factoring $n$. Whether such a method may be discovered is an open question.

Another problem discussed in [21] is "unconcealed" messages, which are defined as messages that encrypt to themselves – $m = m^e$ mod $n$. Two easily identified cases of this are when $m$ equals 0 or 1. Other unconcealed messages are more difficult to identify beforehand, and are dependent upon $p$, $q$, $d$ and $e$. The number of such messages can be computed exactly as $(\gcd(e-1, p-1) + 1)(\gcd(e-1, q-1) + 1)$. Fortunately, this number tends to be small, with random or small $e$, but does require that ciphertexts be compared with the plaintext in order to prevent sending any unconcealed messages. A convenient way to deal with unconcealed messages is to add a random nonce and re-encrypt.

Other potential problems discussed in [21] include those related to small values for $d$ or $e$ (when sending identical or similar messages to more than one recipient), small message spaces, and common modulus issues.

In particular, a common modulus creates an opportunity for more efficient factorization of a modulus. If multiple key-pairs use the same modulus, obtaining the $d$ and $e$ as a pair from any one of the individuals using that modulus simplifies factoring that modulus. Then, using the public ($e$) encryption exponent and modulus yields the private key component $d$.

Additional issues addressed include the recommendations that the difference between $p$ and $q$ not be small (since then $p \approx q$ and $p \approx \sqrt{n}$), and that both $p$ and $q$ be "strong" primes ($p$ is strong if $p$ – 1 has a large prime factor $r$, $p$ + 1 has a large prime factor, and $r$ – 1 has a large prime factor).

Pohlig and Hellman [27][20] developed an asymmetric cipher system that is not actually a public key system. Both members of a key pair must remain secret. As with RSA, it relies on the property that $C = P^e \bmod n$ and $P = C^d \bmod n$, where $e$ and $d$ are inverse relative to a selected modulus, $P$ is the plaintext and $C$ is the ciphertext. Its security will be broken if an efficient means for calculating $e = \log_P C \bmod n$ can be found, though some security remains so long as $n$ remains secret, as an attacker must determine $n$ before launching such an attack.

While Pohlig-Hellman is an alternative to the scheme developed and discussed in this thesis, it is slow by comparison, suffering the same order of performance as RSA encryption. (While $e$ may be selected to minimize the number of operations to be performed, $d$, its corresponding decryption exponent, is unlikely to share that characteristic, and thus require more operations in the decryption than in the encryption.) It may also be viewed as an adjunct, allowing either a form of super-encryption of portions of the data-stream used to agree upon a random bit sequence, or as a means performing periodic re-keying.

Rabin public key encryption [28] is a “provably” secure system, since there exists a proof that, for a passive eavesdropper, the recovery of plaintext from a given ciphertext is equivalent to factoring. It isn’t, however, based upon the factoring problem. Rather, it is based on the problem of finding square roots modulo $n = pq$, where $p$ and $q$ are large primes congruent to 3 mod 4. The primes $p$ and $q$ are the private key, while $n$ is the public key. Note that this dispenses with the encryption and decryption exponents used in RSA, but at the cost of a more complicated decryption mechanism. A secondary problem is that it cannot be used to encrypt random bit strings, since decryption produces a set of four candidate solutions. In the case of a message that is a random bit string, there is no way to determine which is correct, unless a known marker is added to the original message. Therefore, this system is not discussed further in its original form, here, but in a modified form developed by Williams [29] that addresses this deficiency.

In the Rabin-Williams public key system $p$ and $q$ are again large primes, but now we have that $p \equiv 3 \bmod 8$ and $q \equiv 7 \bmod 8$. Also, $N = pq$. The revised scheme uses the Jacobi symbol/function, and is also provably equivalent to factoring. Several other variants have been developed, but all suffer from one serious problem. They all fall to a chosen plaintext attack. Thus, they cannot be used in any situation where an attacker can select the message to be encrypted, such as when they are used in digital signatures.

ElGamal [30][31] is yet another eponymous public key system. It relies on the problem of discrete logarithms in a finite field. Select a large prime $p$, and two random numbers $g$ and $x$, both less than $p$, and calculate $y = g^x \bmod p$. The public key is $y$, $g$ and $p$, while the private key is the public key plus $x$. To encrypt a message $m$ (which is in the range $[0, p - 1]$), select a random $k$ relatively prime to $p - 1$. Compute $a = g^k \bmod p$ and $b = y^k m \bmod p$. The ciphertext is $a$ and $b$, which are each of the same length as $p$ and $m$. The plaintext is recovered as $m = b/a^x \bmod p$. As Schneier [25] notes, this is Diffie-Hellman, but with $y$ added to the key via the multiplication by $y^k$.

While the schemes discussed thus far are generally problems involving exponentiation in one form or another, other approaches have been proposed. Robert McEliece [32] developed a scheme based on Goppa codes, which are a class of error-correction codes. While the scheme has several advantages, and is one of the earliest publically proposed public key systems, it is rarely used due to the size of the public keys required, and more recently due to the attacks that have been developed against it. The public key is an $n$ by $n$ matrix $G = SG'P$, where $S$ is a $k$ by $k$ non-singular matrix, G' a $k$ by $n$ generator matrix for a Goppa code correctly up to $t$ errors, and P is an $n$ by $n$ permutation matrix. The private key is the three matrices $S$, $G'$ and $P$. Encryption with the public key of message $m$ of $n$ bits is accomplished by calculating $c = mG + z$, where $c$ is the resulting ciphertext, and $z$ is a random $n$ bit vector with a Hamming distance relative to $m$ of $t$ or fewer bits. Decryption is accomplished by calculating $m'$ such that $m'G$ has a Hamming distance of $t$ or less with respect to $cP^{-1}$, using the Goppa decoding algorithm, then calculating $m = m'S^{-1}$.

While this is much faster than RSA and several other public key systems, the public key for a minimal secure key pair is $2^{19}$ bits, and ciphertexts are twice as long as their corresponding plaintexts. At first, there were unsubstantiated claims of successful attacks against McEliece [33], though the veracity of the claims are suspect, as they include no support for those claims. However, subsequent efforts have proven more effective, with the best attack being a parallel attack requiring no communications between nodes developed by Bernstein, Lange and Peters [34], involving a work factor of only about $2^{60}$. While this attack is against the original parameters published by McEliece, it must be expected that subsequent attacks against the later versions will follow at some point.

The problems with encryption based on error correcting codes lies in the fact that the encrypted material must contain sufficient information to both convey the message and allow detection and correction of the "errors" in the ciphertext. Such problems are discussed by Chabaud [35].

Yet another direction that has been explored involves the use of discrete polynomials in finite fields. Koblitz [36] and Miller [37] independently proposed system based upon elliptical curves that fall into this category, and a great deal of work has ensued.

Elliptical curves have the advantage of being relatively fast algorithms, but also have problems. As this topic is very rich and complex, a full discussion is beyond the present scope. Interested readers are urged to reference Rosing [38] for an accessible (if older) discussion of the implementation, performance and problems elliptic curve cryptography, or the somewhat more recent Hankerson, Menezes and Vanstone [39] for further information. A detailed search of current literature is also recommended, as developments are near continuous.

Other polynomial-based systems have been developed outside the area of elliptic curves. Kravitz-Reed [40], Müller-Nöbauer [41][42] (using Dickson polynomials), Lidl- Müller [43][44] and LUC (by Smith andLennon) [45][46] are examples.

Note that in all the above schemes, the cryptanalytic problem faced by any attacker is a single problem that is presumed to be "hard," and thus to impart cryptographic strength. In the case of the Rabin public key system, the proof of strength is really just a proof that the problem is as difficult as a problem for which there is no proof of difficulty. While any or all of these problems may ultimately be proven to be as hard as presumed, this cannot be counted upon. Absent a proof that a problem is in fact hard, its use in a cryptosystem is an example of the Diffie-Hellman assumption, and thus suspect. The history of cryptology is, if anything, a history of presumed hard problems being found less difficult than thought.

Of a different form are public key systems based upon finite automatons, as have been developed by Tao Renji [47][48][49][50]. The strength of these systems is based upon the problem of factoring two composed finite automatons. As such automatons may be implemented as matrix operations, composition are relatively straightforward and result in manageable operations. As discussed by Renji, if one of the automatons is non-linear, but of relatively weak form (possessing a weak inverse), composition with another automaton, even a linear one, results in an automaton without an inverse.

The keys for such systems are large, when compared to RSA keys offering comparable security, and this makes the approach unsuitable for some applications, but the operations using those keys are generally much faster than RSA encryption and decryption.  But, as the difficulty of factoring composed automatons is presumed, rather than proven, concerns about the validity of the implicit Diffie-Hellman assumption remain, and it remains worth considering mechanisms that do not incorporate such an assumption in the arguments for their cryptographic strength.

### Zero-Knowledge Proofs and Key Agreement Protocols

Of interest in many applications is the problem of how one participant in a conversation can prove knowledge of some fact without revealing the substance of that fact.  An example where this is the case is as follows.

One party (Alice) has information a second party (Bob) wants, but Alice wants to be paid for that information.  Rather than risking not being paid by revealing the information to Bob before she is paid, Alice insists on being paid first.  Bob is concerned that payment might be made, then nothing revealed by Alice as she does not possess that information.  Therefore, he insists on proof that Alice actually possesses the information first.  For example's sake, let us assume Alice has an account with a business news service that allows her to see stock prices in real-time, while Bob can only see them with a 30 minute delay.  Alice is willing to share her account access with Bob if he pays her for that access.  Bob can get Alice to prove she has such an account by asking her questions that she can answer correctly, in a timely manner, only if she has the access she claims.

Bob asks Alice the exact bid and ask prices for a set of stocks at specific times over the course of a trading day.  If Alice has the access she claims, she will be able to send Bob the required information effectively immediately upon receipt of the queries.  Bob can then compare the times the responses were sent to the time the queries were sent, and the reported prices to the actual prices he sees 30 minutes after the query/response pair.  If Bob selects stocks with volatile prices, and Alice is always right, sending the information upon receipt of the query, she probably has the access she claims.  The more times she is correct, the more certain that she has the access claimed.  However, if she is significantly wrong at all, or is off in terms of time by a meaningful amount, it indicates that she does not have the claimed access.

Note that in the above process, knowledge is passed by Alice to Bob, but not information regarding her account password.  The information she passes to Bob is obtained using her password to her account, but contains nothing derived from her password.  Thus, the process is zero-knowledge with regard to the password Alice is using.

Zero-knowledge proofs serve very useful purposes other than as described in the example above. One of these is authentication of identity without revealing personal information.  Another related use is proof of possessing a password without actually transmitting the password.  This latter category is of interest here, as these techniques leads to protocols for agreement among two or more parties upon a secure key, though these are not, in a strict sense, zero-knowledge proofs.

As pointed out by Schneier [25], an unfortunate fact of zero-knowledge proofs of identity is that they are susceptible to man-in-the-middle attacks.  The process is simple.  If Mallory wishes to convince Bob that she is Alice, she places herself where she can intercept all traffic between Alice and Bob.  When Bob asks Alice a question using the protocol, Mallory intercepts the question, but passes it on to Alice, posing as Bob.  Alice answers the question, and Mallory again intercepts the message, and passes it to Bob, posing as Alice.  As the protocol continues, Mallory repeats this process as necessary, until Bob is satisfied that Alice has correctly proven her identity.  Mallory has no idea what Alice knows, but Bob is now convinced that Mallory is Alice, and Mallory is free to exploit the deception.  The same applies to related protocols, provided Mallory can successfully interpose herself between the correspondents.  The primary exception is in key agreement protocols, where, unless Mallory is able to break the encrypted traffic, or as in the case of Diffie-Hellman based systems substitute her own exponents in for Alice's and Bob's, she will have convinced one or both that she is the other, but remain unable to read the traffic using the resulting keys.

Zero-knowledge key agreement protocols are not, in the strictest sense, zero-knowledge proofs [21], in that they exchange information derived from the passwords or other materials used in arriving at an agreed key.  However, the information is generally sufficiently abstracted, through hashing, modular arithmetic, etc., that this material is at least presumably difficult to determine from the exchanged information.  Still these protocols provide an alternative to the material presented in Chapter IV.

Bellovin and Merritt [51][52] presented several methods for agreeing upon a secret key based upon a shared password.  Some of these proved weak, but those that were not were developed further, expanding to include client-server environments.  The basic idea is for one party to encrypt a public key that is "ephemeral" (meaning it will be used only for the present key negotiation, or "once") using the shared password.  This is encrypted ephemeral public key is sent to the second party, who decrypts the public key using the shared password.  The two parties then negotiate a session key using the ephemeral key pair.

This approach strengthens more usual public key schemes by making the ephemeral public key public in no sense.  Thus, any cryptanalysis must start with breaking the encryption of the ephemeral.  Once that is accomplished, an attack upon the ephemeral public key can commence. This has the advantage of increasing the effective cost of breaking the key exchange process by ensuring that the initial cost of cracking the password-based encryption still leaves the problem of breaking a new public key each time the protocol is invoked.

The liabilities of these protocols include the costs of repeatedly generating public/private key pairs, and of the actual operation of the public key protocol in negotiating the session key. Further, the problem of the Diffie-Hellman conjecture remains. If a truly trivial solution to the public key system used is found, breaking the password-based encryption essentially solves the whole protocol. Conversely, so long as an effective attack against the public key protocol is non-trivial, the cost of a continuing attack upon communications between the two (or more) parties may remain effectively prohibitive, relative to the value to be derived from the attack.

Jablon [53][54] developed a scheme called "Simple Password Exponential Key Exchange," or SPEKE. This is essentially Diffie-Hellman key exchange, but with the generator $g$ derived from a previously agreed password and a large, randomly selected safe prime (meaning a prime $p$ such that $2p + 1$ is also a prime) via a hash function. Since the parties to the exchange will derive the same key if and only if they use the same password, a key verification protocol will allow the two to determine that they indeed share the password used.

As this scheme requires an attacker to know the generator $g$, or the password and prime used to generate $g$, which are not public information, it offers a layer of protection not present in the purely public key Diffie-Hellman protocol, but once $g$ is discovered by whatever means, any effective attack on Diffie-Hellman will also break this protocol. The users of this protocol also have the overhead of the Diffie-Hellman protocol as an inherent cost of using the scheme.

Unfortunately for general application of SPEKE, a U.S. patent (6226383) was issued to Jablon on May 1, 2001.

It is interesting to note that Blum, Feldman and Micali [55] proved that two parties who share a common random sequence of bits possess enough information for a non-interactive zero-knowledge proof of identity: communication during the proof is unidirectional, from Alice to Bob, after which Bob knows Alice is who she claims to be. Their protocol is based upon a three-color coloring of a graph using information derived from the shared material. Unfortunately, full exploration of their protocol is well beyond the scope of this thesis.

One way to understand the validity of such a scheme is to consider a vastly simplified zero-knowledge proof of identity based upon two parties sharing a One-time Pad. Assume Alice needs to prove she is who she is to Bob. As the two share a One-time Pad, she could simply send a message to Bob that is intelligible, once decrypted. So long as Bob believes only Alice has a copy of the pad, the fact that he is able to decrypt an intelligible message is sufficient proof of identity. This is not zero-knowledge in the sense that Alice has revealed a portion of the pad to Bob (who already knows it), and a third party who can obtain a copy of the plaintext can determine the key material used. (Thus Alice cannot send Bob a message that is fixed in content, or very predictable.) However, it is in the sense that she has not revealed the remainder of the pad she and Bob share. A stronger approach is for Alice to send Bob a random string encrypted with the One-time Pad. Bob must be able to identify the plaintext as "correct," which would require additional interaction, but for the fact that the One-time Pad provides such material. Thus, if Alice wishes to prove her identity to Bob to within a $(2^n - 1) / 2^n$ probability, she can XOR the first $n$ bits from the One-time Pad she shares with Bob with the next $n$ bits of that pad, and send the result to Bob. Upon receipt of this $n$-bit string, Bob XOR's it with the first $n$ bits from his pad. If the result equals the next $n$ bits from his pad, he is correspondingly certain that Alice in fact is Alice.

Again, this is not a substitute for Blum-Feldman-Micali, or any comparable protocol, but should demonstrate clearly that shared secrets composed of some number of random bits provide strong, non-interactive proof of identity.

# CHAPTER II

## A SURVEY OF STREAM CIPHERS

Ciphers may be classified in a number of ways. The most direct approach starts with how the keys are handled, then addresses the ways the keys and plaintexts interact to arrive at a ciphertext. This avoids the problems of trying to classify them based upon algorithms, which are complicated by the fact that different algorithms can arrive at the same result. For example, any cipher, using any number Pseudo-Random Number Generators (PRNGs), that yields a repeating or periodic output sequence of key material is logically equivalent to some Linear Feedback Shift Register (LFSR) of equal linear complexity. (This fact is discussed in conjunction with LFSRs later in this chapter.)

The most basic divide between classes of cipher systems is that between secret key systems and public key systems. Secret key systems are also often called symmetric key systems, since the same key is used both to encrypt a plaintext message and to decrypt the resulting ciphertext. Thus, the algorithms used for encryption and decryption are easily reversible, given the secret key. The security messages encrypted by symmetric key systems relies entirely upon the strength of the algorithm and the keys being kept secret, shared only by trusted parties. As all parties who must be able to decrypt a given message must have a copy of the key used to encrypt the message, the agreement upon a key, whether by negotiation, physical delivery, or any other means, in a manner that maintains the secrecy of the key, is one of the central problems of secret key cipher systems.

Public key systems are markedly different from symmetric key systems, and not entirely public, despite the name. They utilize algorithms that are reversible (else decryption would be impossible), but not easily so with the same key. Rather, they utilize key pairs, and algorithms that utilize both members of a key pair to complete an encryption/decryption cycle. So long as the "private" member of a pair cannot practically be derived from the "public" member of the pair, the public key may be made reasonably freely available to anyone. Due to the "complementary" nature of the key pairs, either may be used to encrypt a message which can then be relatively easily decrypted using the other member of the pair. Thus, the public key can then used to securely send messages to parties holding the corresponding private key, as only the parties able to use the private key will be able to easily decipher messages encrypted using the public key. The reverse is not the case. Messages encrypted using the private key must be presumed easily deciphered by anyone, since the public key is expected to be public knowledge. But, so long as the private key is held only by those who "should" have it, the result is a message that can be trusted to be from such a "proper" private key holder. Due to this property, in many digital signature protocols, a public key is contained as plaintext in the digital signature associated with the signed and encrypted message, so that the signature may be authenticated and thus the message. For publicly accessible documents, the encrypted "message" need not be the entire body of the document. It may be a suitably secure message digest generated from the document signed using the system.

If two parties each have distinct key pairs, and publish their respective public keys, secure bidirectional communication can be achieved by means of double encryption. An example of this is the following. Alice wishes to send Bob a message, and wants to be sure that Bob will know that the message could only have been sent by her, as well as that only Bob can decipher it. She has Bob’s public key, and has the necessary tools to use that public key, as well as having a key pair of her own for this public key system. As the two have shared their public keys (the mechanism of such sharing is unimportant for this discussion), she knows Bob can decrypt messages encrypted using her private key, as well as being able to decrypt messages encrypted using his public key. Therefore, she encrypts her message using her private key, then encrypts that ciphertext again using Bob’s public key. She then sends the doubly encrypted ciphertext to Bob, and tells him that it is from her, and the order in which she used their respective keys to encrypt the message. Bob then decrypts the ciphertext using his private key, then decrypts the result using Alice’s public key. Assuming both Alice and Bob have been suitably careful, and nobody has obtained either’s private keys by nefarious means, Bob will know that only Alice could have sent the message, and only he could have decrypted it.

As wonderful as this sounds, public key systems are not used for routine communications. They are too slow in operation for rapid, high volume communications, much less for high volume real-time communications. Instead, symmetric key systems are used for the majority of communications, as they are typically “cheaper” in terms of time and processing requirements. Public key systems are typically used to agree upon or exchange the secret “session” keys used for the bulk of communications, as well as for digital signatures and such, which are much lower volume activities.

Secret/symmetric key systems may be divided into two broad categories: block ciphers and stream ciphers. The simplest way to differentiate between the two types has to do with the size of the data elements each encrypts. Block ciphers typically deal with fixed-length blocks of data that are comparatively large, compared to stream ciphers. These blocks are rarely less than 64 bits in length, and generally of some power of two in size. Two commonly used block ciphers are the algorithms defined by the Data Encryption Standard (DES) and Advanced Encryption Standard (AES), and the International Data Encryption Algorithm (IDEA), which use 64-bit, 128-bit and 64-bit blocks respectively.   In contrast, stream ciphers typically deal with bits, or small, individual tokens, such as bytes, in the communications medium they are used in.

A second distinguishing characteristic is often described in terms of time.  Block cipher algorithms operating in Electronic Codebook (ECB) mode are not concerned with the temporal ordering or sequence of the material enciphered.  They perform the transform algorithm using the secret key to encrypt or decrypt blocks, without regard to any other blocks that have been or will be encrypted using the same key.  Neither are they primarily concerned with real-time requirements.  While efficiency and speed of encryption and decryption are important considerations, emphasis is on exploiting large block sizes and keys to achieve desired characteristics, such as uniform probability distribution and diffusion effects.  Block ordering effects are imposed by operating modes superimposed on top of the basic block cipher transformation.  (See Appendix A for a discussion of block cipher modes.)  Only when encryption modes involving feedback, counter behavior and the like are used is there a strong sense of ordering, with resultant effects in the contents of the ciphertext blocks produced.

Stream ciphers are designed to deal with data streams that are often inherently ordered, and, when intended for use in applications that require real-time behavior, are designed to support the necessary data rates with the available computational resources.  Even the names of the two classes of stream ciphers explicitly address their synchronization behaviors.

The distinction between block and stream ciphers isn’t always clear.  Block ciphers are sometimes used as the bases of stream ciphers.  In Counter Mode (CM) (see Appendix A) a block cipher transform is used as a pseudo-random number generator (PRNG) to generate a running key, resulting in a synchronous stream cipher. Cipher Feedback (CFB) and Output Feedback (OFB) modes use a block cipher to construct self-synchronized stream ciphers. Conversely, self-synchronizing stream ciphers in many ways resemble block ciphers running in CFB or OFB mode, though they usually operate on much smaller blocks. (See the following discussion for the distinction between synchronous and self-synchronizing stream ciphers.)

The greatest differences between block ciphers and stream ciphers are seen at the extremes of the two classes of ciphers. The most fundamental of these is that stream ciphers implicitly entail some form of memory, while block cipher algorithms do not. This can be seen in the facts that stream ciphers use a keystream generator, successive outputs from the generator are derived from the key plus the state of the generator at the start of a cycle, the state is updated during or at the end of the cycle, and the state of the generator is retained for use in the next cycle. As stated above, a block cipher algorithm only performs a transform on a block, based upon the key, retaining no state information (except perhaps a fixed key schedule derived from the key) between successive iterations. Any memory associated with a block cipher is an artifact of the operating mode in which the transform is used.

## Stream Ciphers

Stream ciphers may be divided into two distinct classes. The first type generates a keystream, which is then logically combined (usually by an exclusive-or (XOR)) with the plaintext bits or bytes in sequence. The second uses previously generated ciphertext in the process of generating the keystream used to encrypt subsequent plaintext. These are referred to as synchronous and self-synchronizing stream ciphers, respectively. The reason for this nomenclature involves the fact that, since the keystream of a synchronous stream cipher is generated independently of the plaintext encrypted with it, the sender and recipient must have their respective copies of the cipher algorithm synchronized in order for decryption to be successful. The next state of a synchronous stream cipher is in a sense “closed,” and entirely determined by the immediate prior state of the cipher algorithm, regardless of the prior plaintext. In contrast, for a self-synchronizing stream cipher, the next state is determined by the prior state including the effects from prior ciphertext generated, plus the data element being encrypted.

One way to see the difference between a synchronous and a self-synchronizing cipher is via the following sets of functional definitions [21]. A synchronous stream cipher may be described as follows.

$$\begin{aligned}
\sigma_{i+1} &= f(\sigma_i, k),\\
z_i &= g(\sigma_i, k),\\
c_i &= h(z_i, m_i).
\end{aligned}$$

Here, $\sigma_0$ is the initial state (as determined by key $k$), $f$ the next state function, $z_i$ the $i$-th element of the keystream produced by the generator function $g$, $m_i$ the $i$-th plaintext symbol, and $c_i$ the $i$-th ciphertext symbol. Self-synchronizing stream ciphers can be described as follows.

$$\begin{aligned}
\sigma_i &= (c_{i-t}, c_{i-t+1}, \ldots, c_{i-1},),\\
z_i &= g(\sigma_i, k),\\
c_i &= h(z_i, m_i).
\end{aligned}$$

The symbols here have the same meaning as before, except that $\sigma_0 = (c_{i-t}, c_{i-t+1}, \ldots, c_{i-1})$ is the initial state, which may or may not be secret. The material composing the initial state is often referred to as an Initialization Vector (IV). IVs arise frequently in cryptographic systems, including in various block cipher modes, hash functions and authentication protocols.

Since each element of the keystream is determined by the prior $t$ ciphertext symbols plus the key $k$, we can see that, even if symbols are inserted into or deleted from a message stream, the keystream generator $g$ will resynchronize after $t$ correct ciphertext symbols are received in proper order.

It is worth reinforcing the fact that we are dealing with two distinct keys: the formal key and the keystream or “running key” constructed by the keystream generator. For both synchronous and self-synchronizing ciphers, any keystream generated is effectively the product of a PRNG. In fact, any PRNG may be used as a keystream generator, or as a component of one. Conversely, stream ciphers may be used as PRNGs, or as components, in both cryptographic and non-cryptographic applications.

Another important observation regarding stream ciphers is that while the running key resulting from operation of a strong stream cipher and a specific initialization key may be used for a significant span of time, re-use of the initialization key is typically a very bad idea. Such re-use results in the re-use of the same sequence of keystream bits or bytes. Ciphertexts encrypted with the same running key sequences can be XOR’ed together to produce an XOR of the plaintexts, which represents a vast simplification of the problem for an attacker. In comparison, a key for a block cipher may typically be used repeatedly, for extended periods, so long as the mode used is not one that acts as a stream cipher.

## Introduction to a Brief Survey of Modern Stream Ciphers

As the bulk of this thesis revolves around PRNGs used as stream ciphers, any alternate implementation’s effectiveness must depend in part on the PRNGs utilized. We recognize the relationship between stream ciphers and PRNGs, and will here briefly examine several stream ciphers, including some that are poor ciphers, but good PRNGs in terms of bit-wise randomness. We will not examine any prior to the Vernam cipher, which may be regarded as the first “modern” stream cipher, as it was expressly developed for use in automated electronic telecommunication, and many stream ciphers are in fact Vernam ciphers using specific PRNGs as keystream generators, or components thereof.

While an understanding of the means and methods of cryptanalysis is important for analyzing cipher systems, regardless of the type, mention and description of specific cryptanalytic attacks and processes will be limited to points where they are of specific interest, as the field is both broad and deep.  For a survey of ciphers it is more important to know that a system has known weaknesses than to know the details of them in depth.  The details of such matters are left to the references cited, which deal with them in much greater depth than is appropriate here, as well as specifically to Bruce Schneier's *Applied Cryptography*, which served as the initial source for much of the following material.

## Vernam Ciphers and One-Time Pads

Gilbert S. Vernam, working for AT&T during World War I, developed a scheme for automatically encrypting the Baudot character codes for use in the then current teletype systems.  In its more generalized forms, this scheme and the descendant ciphers now bear his name.  In practical terms, we can consider all synchronous stream ciphers to be Vernam ciphers, and all self-synchronizing stream ciphers as Vernam ciphers with feedback.

In its original form, a specialized teletype machine with a paper tape reader was used to logically combine, character by character, messages typed on the machine, with a key contained on a paper tape mounted in the reader, in an operation that is effectively an XOR.  The result of this was a ciphertext, which was transmitted over the TTY system to a recipient, or recorded on a paper tape which could later be fed into a teletype machine for transmission.  Upon receipt, depending on the equipment at the receiving station, either of two scheme could be used.  In the first, the receiving teletype machine could, if equipped with a paper tape reader and a copy of the key tape, perform the inverse operation to produce the plaintext.  In the second, a paper tape copy of the ciphertext could be produced.  This tape could then be fed into a combiner, equipped with a tape reader and key tape, which could then perform the inverse operation using the key tape to produce the plaintext.  As the key tape could be made as long as desired, within the limits of practical usability, with arbitrary key contents, the result was a relatively robust, automated system that required little knowledge and effort on the part of the operators.  Indeed the teletype operators did not need to be involved in the encryption and decryption processes at all, thereby increasing the security of the system if they were excluded.

This system is extremely general, and adaptable to any bit- or token-oriented system of communication.  As such, it is regarded as one of the most important developments in the history of cryptography.  However, it suffers from serious flaws.  The most important of these is that if the key is not random, displaying some structure that can be predicted, or if the key is re-used, so that different ciphertexts generated with the same key segment may be compared and analyzed, the result is a weak cipher.  In cases where the logical combining operation is an XOR, it is easy to describe the process by which a Vernam cipher may be broken.

If two ciphertexts encrypted with the same key are aligned with respect to the key, then XORed, the result is an XOR of the two plaintexts.  Where characters in the two plaintexts match, in terms of relative position in the respective streams, the result is a null.  The $\chi^2$ (chi-square) test may be used to determine whether a trial alignment is correct.  Once the alignment has been determined, many of the cryptanalytic tools that are effective against older poly-alphabetic ciphers are effective, but “educated guesses” can radically simplify the process.  By simply selecting a small number of words that have a high probability of appearing in either ciphertext, then XOR’ing these in successive positions in the XOR of the two ciphertexts, a correct alignment will result in plaintext.  This process is easily automated.  The more ciphertexts that are found using the same key segment, the more rapid and effective the process becomes.

Captain (later Major General) Joseph Mauborgne, U.S. Army Signal Corps, soon observed that, if the Vernam cipher key tape was random, and of equal or greater length than the plaintext, the resulting ciphertext would be difficult or impossible to break if the key were never re-used. Though it was not called such at the time, this specific implementation of the Vernam cipher became known as the One-Time Pad system, or simply OTP.

Claude Shannon's 1949 paper on cryptography [5] deals with the problems of cryptanalysis of OTP, as well as the more general questions of the relationships between key, block and plaintext lengths and the overall cryptographic strength of any cipher system. In that paper he proved several things, including that a properly implemented OTP is unbreakable and thus perfect. Further, any perfect cipher system must be a homologue of OTP. The requirements for a properly implemented OTP are that the key is secret, random, as long as the plaintext, never re-used, and destroyed upon use.

In the absence of the OTP key used to encipher a plaintext, all possible plaintexts of equal or shorter length as the ciphertext are equally as probable solutions, meaning no solution is possible, regardless of the cryptanalytic methods used. It is also impossible to determine whether a specific plaintext corresponds to a specific ciphertext with any reliability beyond whether the plaintext is of equal or shorter length, absent additional explicit evidence linking the two. Given these facts, if an innocuous text of the same length as a ciphertext is retained, and both the actual plaintext and OTP key destroyed, there is no way to prove that the innocuous text was not the plaintext, absent other definitive evidence.

The effect of encrypting a byte with OTP is the same as randomly selecting one of 256 mono-alphabetic substitution ciphers and applying it to the plaintext byte, since byte values must range from 0x00 to 0xFF (8 bit hexadecimal 0 to FF, using standard C notation). This point is worth reinforcing. There are only $2^8 = 256$ possible cipher key bytes for byte encryption using the XOR encryption function, and one of these (0x00) leaves the plaintext unchanged. This is a very small number when compared with the $256! - 1$mono-alphabetic substitution ciphers that are possible with random mapping.  This is true of all Vernam ciphers that use the XOR operation as the combining function for the plaintext and keystream.  The same problem applies to other simple combining functions, such as byte-wise addition or subtraction modulo 256.  While this is not a problem in correctly implemented OTP systems, for non-OTP Vernam ciphers, the result is a much simplified process of guessing the keystream bytes when multiple messages encrypted using the same keystream segment are available.

The primary problems with OTP are not cryptanalytic in nature.  They are logistical and operational.  OTP requires an amount of key material equal to the amount of traffic to be encrypted.  The key material must be truly random, and never reused.  The key material must be distributed in an absolutely secure manner, and completely, irreversibly destroyed upon use.  A requirement for simultaneous, bidirectional communication compounds the logistical problem, and loss of synchronization can cause significant problems.

While storage media, including flash drives, Digital Video Discs and Blu-ray® Discs, now offer simple means of storing and transporting massive amounts of data, the complete destruction of such media can be problematic, as well as expensive, as even small fragments of surviving material may be sufficient to decrypt portions of ciphertexts that may be "embarrassing."  This can be seen via the VENONA program, which was a joint effort among several Western allies to extract information from masses of Soviet OTP traffic.  Despite a very low yield, the insights gained were reckoned as well worth the decades of effort spent.

The success of the VENONA program is useful in illustrating the difficult of correctly implementing and operating an OTP system over long periods of time.  The success of the VENONA program became possible when the Soviets were forced to take shortcuts in operating their OTP system due to logistical problems, including the difficulty of generating and distributing large volumes of random key material.

Apart from other logistical problems, the physical transport of such media represents a significant risk, when an adversary possesses high motivation and ample resources. And, despite the high capacities of modern media, encryption of high-resolution video streams will rapidly exhaust key material regardless of the media used for distribution.

## Electro-Mechanical Ciphers

The Twentieth Century also saw the development of electro-mechanical encryption technology. Two inventions in particular deserve note: the Enigma and Lorenz machines. Both were used by the Germans during World War II, were successfully broken by Allied intelligence services, and are exemplars of the problems associated with such systems.

While there are significant differences in the rotors used in these machines, as well as other rotor-based equipment, the basic structure is the same. Each rotor has a set of electrical contacts on one side that are connected in an arbitrary or random pattern to contacts on the other side of the rotor. Thus, the connections through the rotors change as the rotors are rotated relative to the surrounding hardware. When a series of such rotors are used, and they are independently rotated as plaintext is enciphered, the combinatorial effects can be quite large. Yet, the basic operation is the same. At any given moment prior to an input key being pressed, the circuits effectively embody one mono-alphabetic substitution cipher. Pressing an input key selects the corresponding ciphertext or plaintext character, and upon release reconfigures the rotor system to select another mono-alphabetic cipher for use in encrypting the next input character.

Both the Enigma and Lorenz systems exhibit a fundamental problem with rotor-based electro-mechanical cipher systems. While a rotor may embody any arbitrary set of connections between the opposite sides of the rotor, the rotors are themselves static devices. Once a rotor has been manufactured, the mapping is fixed. While it is possible to produce sets of rotors that implement all possible mappings, this isn’t practical. Rotors are bulky and expensive. Thus, the number of different rotors tends to be small, and the internal mappings of the rotors part of the “secret” required to maintain security.  Using the rotors in series (or with the added complication of bidirectional traversal, as in the Enigma machines), with differing step rates for the rotors and regular changes to the rotors in the working set and their order in that set, complicates the encryption achieved.

The lack of flexibility in mechanical devices is a serious liability.  The World War II Allies’ success in breaking both the Enigma and Lorenz machines used by the Germans, demonstrates the liabilities of such devices.  In particular, the breaking of both of these cipher machines demonstrates of the validity of Kerckhoffs’ principle that the security of a cipher system must depend on the secrecy of the keys used, rather on the secrecy of the mechanism.

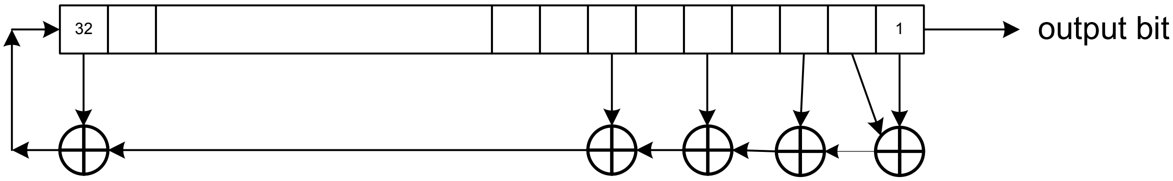
In the end, the flexibility that may be achieved in digital systems, and particularly in software-based digital systems, has turned cipher development away from relatively inflexible electro-mechanical systems.

## Digital Stream Ciphers

Digital stream cipher systems gain a considerable advantage over manual and mechanical systems in the fact that digital processing allows a much wider range of operations, with substantially lower risk of error.  While many digital stream ciphers retain the Vernam cipher’s XOR operation as the combining function (and thus are in fact instances of Vernam ciphers), with the attendant limitation on character mappings, the potential for generating very good pseudo-random keystreams is a substantial advantage over the earlier electro-mechanical systems, as well as the original Vernam cipher with looped paper tapes.  As a result, there are now several decades and untold thousands of man-hours of research on digital ciphers, including digital stream ciphers.

## Linear Feedback Shift Register (LFSR)

One of the most common and useful mechanisms in cryptography is the linear feedback shift register, or LSFR. The technique is perhaps best described in conjunction with a diagram.



A Linear Feedback Shift Register

The crossed circle represents the bit-wise logical exclusive or (XOR) operation.

The operation of an LFSR is simple. Specific bits in the register are designated "taps." When the LFSR is pumped for a bit value, the bits corresponding to the taps are sampled and XORed together to produce a new bit. The register is then shifted in whatever direction specified, and the new bit inserted at the end opposite the shift direction. The output is the bit at the end opposite the inserted bit.

An LFSR will have a maximal period only under specific conditions. The selection of taps is critical to achieving this goal.

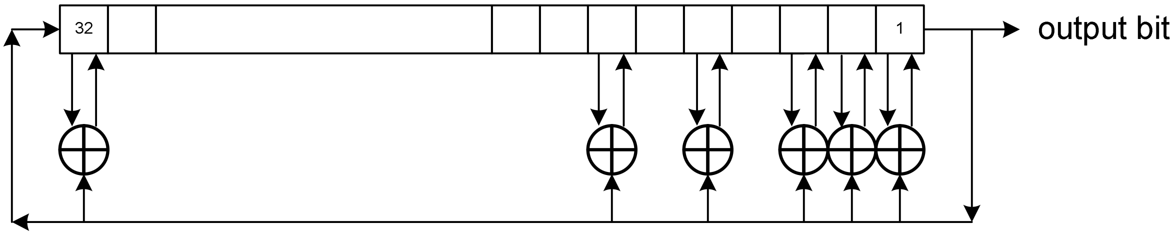
If the LFSR register is treated as an array of coefficients of a polynomial in the Galois field GF(2), plus an implicit 1, and the taps correspond to the non-zero coefficients of a primitive polynomial of degree *n*, where *n* is the length of the shift register, the resulting LFSR will have a maximal period equal to $2^n - 1$. For the present purposes, we may regard a primitive polynomial as a polynomial that is irreducible (meaning it has no real factors), divides $x^{2^{n-1}} + 1$, and does not divide $x^d + 1$ for any *d* that divides $2^n - 1$.

LFSRs are easily implemented in hardware. In software, they can be slow. Schneier [25] gives the following sample in C for an LFSR for the polynomial $p(\mathrm{x}) = \mathrm{x}^{32} + x^{7} + x^{5} + x^{3} + x^{2} + x + 1$. (This matches the above figure and the following discussion regarding Galois LFSRs.)

```
int LFSR()
{
     static unsigned long ShiftRegister = 1;
     /* Anything but 0. */
     ShiftRegister   =   ( ( ( (ShiftRegister >> 31)
                             ^ (ShiftRegister >>  6)
                             ^ (ShiftRegister >>  4)
                             ^ (ShiftRegister >>  2)
                             ^ (ShiftRegister >>  1)
                             ^ (ShiftRegister))
                           & 0x00000001) << 31)
                       | (ShiftRegister >> 1);
     return ShiftRegister & 0x00000001;
}
```

C code for a software-based LFSR

As can be readily seen, this is considerable of work for a single bit of output. There are two relatively simple ways this can be at least partially rectified. The first is to use what is called the "Galois configuration" for implementing the LFSR, as illustrated by the following diagram and code sample.



Galois configuration of an LFSR

Here is a code sample, based on Schneier's example.

```
int Galois_LFSR()
{
     static unsigned long ShiftRegister = 1;
     retVal = 0;
     if (ShiftRegister & 0x00000001)
     {
          ShiftRegister = ((ShiftRegister >> 1)
                         ^ 0x80000057)| 0x80000000;
          retVal = 1;
     }
     else
          ShiftRegister >>= 1;

     return retVal;
}
```

This is still considerable work to produce a single bit. But, as many PRNGs and stream ciphers utilize two or more LFSRs, implementing Galois configurations in parallel can speed the process, particularly if all the parallel LFSRs use the same primitive polynomial. They can also be implemented to support differing tap vectors for each, and even differing lengths, at some cost in performance. (This is done in the implementation presented in Appendix C.)

### Linear Complexity

The concept of linear complexity arises directly from the study of LFSR generators. Any finite generator (meaning a generator with a fixed, finite number of internal states) will produce a finite, repeating output stream. Regardless of the length of the period, the output stream represents a linear recursion. Thus, for any finite generator, it is possible to construct an LFSR that exactly duplicates the output behavior of the generator. This can be seen in the trivial case of a generator with period of *p* bits being mimicked by a shift register *p* bits long, with a single tap at bit 1.

The linear complexity of a generator is the length of the shortest LFSR that can produce the same output stream. The concept is useful in determining whether a generator may be cryptographically useful, but is not a determining factor. A high linear complexity is necessary for a cryptographically strong generator, but having a high linear complexity is far from being proof of strength. On the other hand, a low linear complexity is sure indication that a generator is cryptographically weak. A common example, noted by Rueppel [56] and others [25], is the following sequence.

$$\sum_{i=1}^{\infty} \frac{1}{10^{\sum_{j=1}^{i} j}} = 0.101001000100001...$$

This sequence has unbounded linear complexity, since it has no period, and thus cannot be generated by an LFSR, but is not cryptographically strong. It is too predictable, once the pattern is discerned, and the occurrence of 1's is far too sparse for most applications.

Part of the importance of the linear complexity of a generator is that it determines how easily the cipher can be broken by the Berlekamp-Massey algorithm. [57] If a generator has a linear complexity of *L*, this algorithm can generate the coefficients of an LFSR that will duplicate the behavior of the generator, given 2*L* consecutive bits, in order, of the output sequence generated by it.

It should be apparent that the period of a generator will limit its linear complexity. For a maximal LFSR of length *n*, *n* = *L*, but for any LFSR the length of that LFSR is the upper bound for its linear complexity. Thus, a good cryptographic generator must have both a long period and high linear complexity. These two characteristics must be coupled with a third requirement: that without knowledge of the current state, yet with knowledge of the prior output, an attacker must not be able to guess the value of the next bit output with greater probability than 0.5.

Other research into cryptographic complexity has produced interesting and useful results. The maximum order complexity of a sequence, as defined by Jansen and Boekee [58], is the length of the minimal feedback shift register (of any type, not just linear ones) that can produce the sequence. For random sequences of length *n*, the expected maximum order complexity will be approximately 2 *lg n*. Jansen and Boekee also presented a linear-time algorithm for computing this value.

The Ziv-Lempel complexity measure [59] quantifies the rate of appearance of new patterns within sequences, and may well prove to be a good test of the overall randomness of sequences. But, this approach had limited implications in the analysis of linear recursions with regard to producing corresponding generators into the 1990s.

Further development of these and other algorithms along the lines of the Berlekamp-Massey algorithm, must be expected to yield progressively stronger tools for cryptanalysis of linear recursions generated by any type of shift register, linear or non-linear, or by any other means. Still, Berlekamp-Massey remains a powerful and useful tool. Any linear recursions can be reduced to an LFSR by it, and it thus serves as a good initial tool in the cryptanalysis of PRNGs.

As a consequence of this discussion, it should be seen that stream ciphers that produce linear recursions will likely suffer increasing vulnerability to analytical attacks. Therefore, practical means of introducing meaningful aperiodicity into keystreams is of increasing interest.

## LFSR Based Stream Ciphers

As noted, any PRNG can be used to generate the keystream for a Vernam cipher, and as such will generally bear the name of the PRNG used. Combining PRNGs to produce non-linear results, or simply to complicate matters, is a frequently used approach. Given their ease of implementation, that they are easily altered or tapped at points other than the standard output, and that they are able to replicate the behavior of any generator producing a linear recursion, LFSRs are frequently used in such composite systems, despite the fact that a single LFSR, no matter its length, is almost axiomatically a cryptographically weak PRNG.

The manner of composition of constituent PRNGs is important. The first example given below is used to illustrate this fact, while others will be discussed more briefly.

## Geffe Generator and Correlation

This is a simple multiplexer scheme using three LFSRs. It is also a cautionary tale. The Geffe generator [60] consists of two LFSRs used as feeds to a multiplexer, while a third LFSR is used as the selector, yielding a non-linear output mechanism.

If the feed LFSRs, designated *A* and *B*, yield bits *a* and *b* respectively, and the selector generator *S* yields bit *s*, the ouput *o* is given by the equation $o = (s \wedge a) \oplus (\neg s \wedge b)$.

While this may appear effective in obfuscating the feed generators, notice that half the time the output will be *a*, but that when *b* is the output, *a* equals *o* half of the time. The same is true of the output relative to *b*. This can be seen in the following table.

| *s* | *a* | *b* | *o* | *a = o* | *b = o* |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | T | T |
| 0 | 0 | 1 | 1 | F | T |
| 0 | 1 | 0 | 0 | F | T |
| 0 | 1 | 1 | 1 | T | T |
| 1 | 0 | 0 | 0 | T | T |
| 1 | 0 | 1 | 0 | T | F |
| 1 | 1 | 0 | 1 | T | F |
| 1 | 1 | 1 | 1 | T | T |

Correlation in the Geffe generator

This means that there is a high correlation between the feeds and the output, as describe by E.L. Key [61], and by Zeng, et al. [62][63] An attacker can isolate either feed generator for attack. If the tap sequences are known, an unsuccessful guess at the state will produce a 0.5 rate of agreement with the output stream, while a successful guess will produce a rate of agreement of 0.75. The feed generators can be attacked in sequence, which then reveals the output of the selector. This can then be attacked any number of ways, depending on the generator used, as shown by Zeng, et al., as well as via the Berlekamp-Massey algorithm.

Generalizing the Geffe generator does not improve matters much. With *n* fill generators and *lg n* generators (or a single generator clocked at *lg n* times the rate of the others) used to select which fill generator will provide the final output (essentially implementing an *n*-to-1 multiplexer), the outputs of each fill generator will correlate with the final output at a rate of (1/*n*) + 0.5, since each fill generator will produce 1/*n* output bits, but will otherwise match the output bit half the time.

This behavior serves as a good introduction to the problem of correlation, and to correlation immunity. As can be seen with the Geffe generator, what may seem to be an effective scheme of mixing two or more generators can have unfortunate characteristics that are not readily apparent to an inexperienced person.

It is worth noting that, apart from the fact that it is easily reversible, the XOR operation is commonly used in cryptography due to its lack of strong correlation between the individual inputs and the result. This can be seen in the following table.

| a | b | c | $a \oplus b \oplus c = x$ | a = x | b = x | c = x |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | T | T | T |
| 0 | 0 | 1 | 1 | F | F | T |
| 0 | 1 | 0 | 1 | F | T | F |
| 0 | 1 | 1 | 0 | T | F | F |
| 1 | 0 | 0 | 1 | T | F | F |
| 1 | 0 | 1 | 0 | F | T | F |
| 1 | 1 | 0 | 0 | F | F | T |
| 1 | 1 | 1 | 1 | T | T | T |

Correlation and the XOR operation

Despite the positive characteristics of the XOR operation, it must still be used with care, as it does not in itself protect against correlation attacks, nor guarantee high linear complexity, as will be seen below.

The problem of selecting a combining function is far from simple. Correlation immunity is not a matter of “it is or it isn’t,” but of degree. A combining function is considered to be *m*-th order correlation immune if, for all subsets of the *m* random variables (*m* less than *n*), the output of the combining function of *n* variables (including the *m* random ones) is statistically independent of the *m* random variables. [56]

Unfortunately, as Rueppel [56] explains in great detail, high correlation immunity also tends to reduce the linear complexity of a generator. If we define the non-linear order of a generator as the maximum number of terms *n* appearing in the algebraic expression of the combining function, and recognize that the linear complexity of the aggregate generator tends to increase with higher non-linear order, we can see that high degrees of correlation immunity tend to reduce the linear complexity of the aggregate. In fact, if the generator output is balanced (as many ones as zeroes), the non-linear order of the generator must be less than or equal to $n - m - 1$, for $1 \leq m \leq (n - 2)$.

A partial solution to this trade-off is to incorporate memory into the combining function, which tends to obscure correlations between component generators and the aggregate output by spreading the effects of component variables over more outputs. This will eventually lead to the discussion of Feedback with Carry Shift Registers (FCSR), which incorporate memory in the form of additive carries.

### Pless Generator

This generator [64] uses eight LFSRs to drive four JK flip-flops, acting as non-linear combiners, and interleaves the resulting bits in an attempt to avoid exploitation of the behavior of flip-flops: the output identifies both the input and the next output. As this falls to both a divide-and-conquer attack (exploiting the interleaving) [65] and a correlation attack [66], it demonstrates that mere multiplication of LSFRs does not necessarily increase the strength of the resulting generator.

### Jennings Generator

A markedly different approach [67][68][69], this generator combines two LFSRs, used laterally instead of sequentially, plus a multiplexer and a filter/mapping function between one LFSR and the multiplexer inputs. The other LFSR provides the selector/address feeds for the multiplexer. One bit is selected by the multiplexer for each cycle. As each LSFR is clocked, the tap-bits of both are modified and the result shifted, so that the input and address bits vary significantly with time.  This shows that complexity does not afford security, as it falls to both a meet-in-the-middle consistency attack [70], and a linear consistency attack [62][71].

### Stop-and-Go Generators

There are several generators that use one or more of a set of LFSRs as means of controlling the clocking of other LFSRs in the set.  The simplest is the Beth-Piper Stop-and-go generator [72] which uses the first generator to clock the second, the output of which is XORed with the output from the third.  The first and third LFSRs are clocked by the same signal.  Despite the otherwise good characteristics of the XOR used to generate the final output, it is subject to a correlation attack by Zeng, et al. [63]

The Alternating Stop-and-Go generator [73] also uses three LFSRs, which must be of different lengths, with the clocking of the second and third controlled by the first.  When the output from the first is a one, the second LFSR is clocked; when it is zero, the third is clocked.  The final output is the XOR of the output of the second and third generators.  While there is a correlation attack against this generator, it is not very effective, as the correlation is weak, so that the generator remains relatively sound, and has been proposed for use in other schemes as recently as 2009 [74].

The Bilateral Stop-and-Go generator [62] uses only two LFSRs of the same length, but they interact with regard to clocking.  When either LFSR yields a one at time *t*, and a zero at *t* + 1, its mate is not clocked at *t* + 2.  The output is the XOR of the two LFSRs.  Analysis of this generator has concluded that the linear complexity is roughly its period.  In the above referenced paper, Zeng, et al. stated that "… no evident key redundancy has been observed in this system." Unfortunately, from a software perspective this is not a very useful generator.  While physically simple, it can be slow in software when compared to many alternatives, due to the need to handle clocking of the two LFSRs separately.

## Decimating and Shrinking Generators

Another approach involves discarding portions of the output of an LFSR in order to conceal the pattern produced by the underlying LFSR.  Such generators, using single LFSRs, may also be referred to as self-decimating.  Generators of this type have been proposed by Rueppel [75] and by Chambers and Gollman [76].  An inherent problem with this approach is that the resultant output streams are still linear recursions, and thus still subject to analysis using the Berlekamp-Massey algorithm, if sufficient portions of the keystream can be isolated, while the periods of the generators are effectively shortened by the discarding process.

The basic approach in self-decimating generators is to use some portion of the state of the LFSR to control how the LFSR is clocked.  When the function applied to the state yields a zero, the LFSR is clocked some number of times ($d$), else it is clocked a different number of times ($k$).  While the shortening of the period is important, the selection of $d$ and $k$ relative to the LFSR used is especially important.  If $d$, $k$, $p$ (the period of the LFSR) and the output sequence of the LFSR are pathologically related, and there exist integers $i$ and $j$ such that $id + jk = p$, and the function used to clock the LFSR yields $i$ occurrences of $d$ clocking and $j$ of $k$, the period of the generator will be reduced by $i(d – 1) + j(k – 1)$ bits, with result that the linear complexity of the decimating generator may be significantly less than that of the LFSR.  The problems with both self-decimating generators were amply revealed by Zeng, et al. [63].

Generalization of this leads to the shrinking approach.  These schemes use multiple LFSRs.  Here, the set of LFSRs is clocked, and the output from one generator (or subset of generators as determined by some function) is used to determine whether the output of another generator (or subset) will be used as the output of the aggregate, resulting in the discarding of roughly half the possible outputs.  Generators of this type have been proposed by Coppersmith and Grossman [77][78], and a self-shrinking, single LFSR variant by Meier [79].

Shrinking generators have a problem with regard to their output rates.  If the gating LFSR produces a sequence of zeroes, which indicate the output of the other is to be discarded, no output will be generated by the aggregate generator.  Thus, there may be relatively substantial gaps in the stream's timing.

The self-shrinking generator uses successive output bits from a single LFSR.  If the first of a pair of bits is a zero, the second is discarded, and two more bits generated, and the process repeated until the first bit of the pair is a one.  If the first bit is a one, the second is used as the output.  As the concept is not restricted to use of LFSRs, it should be noted that if the period of the PRNG is even, the period of the self-shrinking generator will be roughly half that of the PRNG used.

### Multispeed Inner-Product Generator

A generator with a similar motivation to the self-decimating generators, but moving in the opposite direct with regard to use of bits was propose by Massey and Rueppel [80].  Rather than clocking some number of time and skipping most of those bits, two LFSRs are clocked at a given frequency and d times that frequency, respectively, the results AND'ed, then XOR'ed.  Zeng, et al. [63], demonstrated that this was not very effective, as the internal state of the generator can be determined from $n_1 + n_2 + \log_2 d$ bits of the output, where $n_1$ and $n_2$ are the lengths of the two LFSRs, and $d$ the frequency multiple used.

### Gollmann Cascade Generator

This generator is related to the stop-and-go generators.  It uses a series of LFSRs to modify the clocking of successive LFSRs, and is related to the Stop-and-Go generators.  Proposed by Gollman [81][82], the linear complexity can be quite high, but as the number and interplay of the generators creates a complex implementation in software, they are not of interest here.

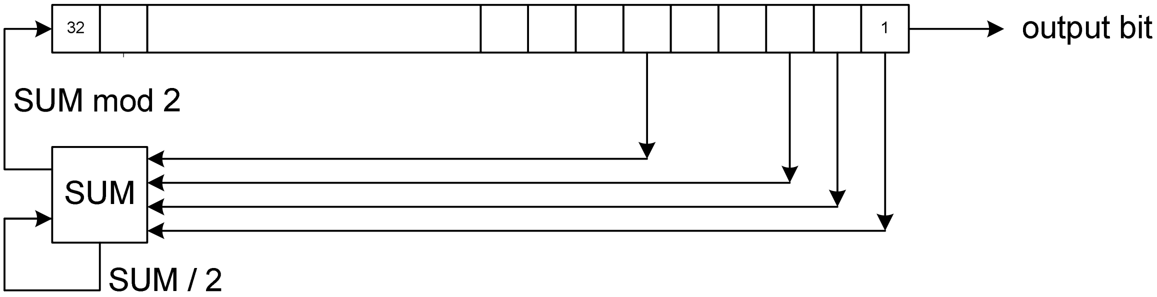
### Summation and Threshold Generators

Threshold generators operate on the assumption that a large number of independent generators can obfuscate the operation of the individual generators [83]. If an odd number of generators, each with relatively prime lengths and using primitive polynomials, are allowed to "vote" on the output result, the output will be whichever value (ones or zeroes) was produced by the majority of the component generators. The problem is that the fact of "voting" means that there will be correlation between the output and a majority of the LFSRs for each output bit. Thus, as Schneier observes [25], the generator effectively leaks information about the component LFSRs with each output bit.

The summation generator, proposed by Rueppel [56][84] couples two LFSRs. The output at any given time is the sum of the outputs of the two LFSRs, plus the "carry" from the prior sum. It has been shown that this is equivalent to a feedback with carry shift register, and can be analyzed as such [85], as well as being subject to correlation attacks [86][87][88]. More generators could easily be used and their outputs summed, but these remain feedback with carry shift registers built by other means.

### Feedback with Carry Shift Register (FCSR)

Feedback Shift Registers are not constrained to only use linear feedback that is an XOR of the feedback bits [21][25]. A very natural extension of the approach is the Feedback with Carry Shift Register. In this mechanism, the feedback bits are summed with the prior sum divided by 2 (the "carry" register), to produce a new sum, and use the lowest order bit (*sum* mod 2) as the "re-fill" bit for the shift register.



Example of a Feedback with Carry Shift Register (FCS) [25]

Since the carry register would otherwise overflow, there must be at least *lg t* bits in the carry register for *t* taps.

FCSR registers are distinguished from LFSRs in several regards beyond the use of the carry register in place of the XOR of tap bits.  Among these is the shortened period relative to the internal states – some states that are otherwise plausible may never occur in the cycle of the FCSR.  While some of these states may occur, they do not occur within the repeating cycle of the generator.  They occur only during an initial series of iterations when the FCSR has yet to settle into its cycle.

Perhaps the worst characteristic of FCSRs is that not all initial states will produce a maximum length cycle [21][25].  It is even possible, for some FCSRs, to create situations where the output degenerates to a short, fixed string, as well illustrated in two examples by Schneier.  Thus, there is a problem with "weak keys," such as sometimes found in block ciphers, potentially made far worse when "disastrous" initial states are selected, as again illustrated by Schneier's examples, which devolve rapidly to constant streams of ones.

The maximum period for a FCSR will be $q - 1$, with $q$ as determined by the following equation, if the value $q$ is prime, and has a primitive root of 2.

$$q = 2^{a_1} q_{a_1} + 2^{a_2} q_{a_2} + 2^{a_3} q_{a_3} + \ldots + 2^n q_n - 1$$

where the assorted $a_i$ are the tap numbers, up through *n*.

FCSRs are susceptible to many of the analytical tools available against LFSRs, with some adaptation [89][85][90][91], including a modified form of the Berlekamp-Massey algorithm, though the analysis may need to be delayed until the FCSRs involved have entered their repeating cycles. At worst, the Berlekamp-Massey algorithm may be used to produce a LFSR that is functionally equivalent to a given FCSR, once the initial extra-cyclic outputs have been excluded. (This LFSR may be very long!) Many varieties of LFSR-based generators exist in corresponding forms based either only upon FCSRs, or an intermingling of FCSRs and LFSRs.

### Nonlinear-Feedback Shift Registers (NFSR)

The feedback functions used with shift registers need not be limited to XOR and additive carry functions [21][25]. Any function on the binary field is suitable, so long as the resulting implementation produces a suitable output sequence. Thus, if we consider the full range of Boolean functions on *n* variables, where *n* is the length of a shift register, we have a much richer vocabulary of feedback shift registers available than described by the *LFSR* and FCSR classes, and enter the realm of the Non-linear Feedback Shift Register (NFSR). The number of Boolean functions of *n* variables is quite large for any large *n*, and is given by $booleans(n) = 2^{2^n}$. As large as this number is, for large *n*, the only functions of interest are those that produce a balanced output stream. This is a much smaller number than all possible Boolean functions on *n* variable, but is still quite large.

$$balanced\_booleans(n) = \binom{2^n}{2^{n-1}}.$$

The same problem arises with NFSRs as with FCSRs, in that they are more difficult to analyze, and are prone to producing undesirable behaviors, such as short periods or constant output streams, when poorly designed or are initialized with weak keys. Still, for any PRNG or stream cipher using one or more LFSRs, there is a corresponding design using NFSRs of some form, and the resulting implementations remain subject to analysis with the Berlekamp-Massey algorithm (though the result may be an extremely long LFSR) and others.

As with all deterministic algorithms, the upper bound on the period of any NFSR is $2^n$, where $n$ is the number of bits in the internal state. In the case of LFSRs, this is the length of the shift register. In FCSRs it includes the state of the carry mechanism. Any additional state information in an NFSR would also apply, but without altering the upper bound relative to the size of the internal state.

### A5

The A5 cipher family [92] was one of the most widely used ciphers families, due to its use in the GSM digital cellular telephone standard. The grouping is not due to inherent features across the A5 ciphers, but their application domain in GSM phone systems.

There are two stream ciphers in the family: A5/1 and A5/2. (There is also the KASUMI cipher that is often referred to as A5/3, though this is technically a block cipher and not within the present scope.) Both A5/1 and A5/2 are relatively weak, with A5/2 cipher particularly so [93].

A5/1 is constructed using three LFSRs of lengths 19, 22 and 23, and polynomials $x^{18} + x^{17} + x^{16} + 1$, $x^{21} + x^{20} + 1$ and $x^{22} + x^{21} + x^{20} + x^7 + 1$, respectively, though in reversed order from the previous discussion of LFSRs. The outputs of the three are XOR'ed to produce the output. As a complication, clocking of the three LSFRs is not synchronized. Rather, bits 8, 10 and 10 in the respective LFSRs are used to control clocking. Individual LFSRs are clocked whenever the clocking bit agrees with one or more of the other clocking bits. Thus, A5/1 is a Stop-and-Go generator, though it uses tapping points other than the outputs of the LFSRs to control the clocking.

The key length of the A5/1 is 54 bits, while the internal state is 64 bits.  This must be regarded as an unfortunately short key, as there is a very effective attack that can be performed in real-time [94].  Known plaintext attacks are also effective, and for older GSM phones protocol flaws can be exploited to great effect [95].

A5/2 is constructed from four LFSRs, but is so weak that it is breakable in real-time with inexpensive equipment [95].

### SEAL

The Self-optimized Encryption Algorithm, or SEAL, was developed by a well-regarded cryptanalyst named Don Coppersmith, along with Phillip Rogaway [96][97].  It is patented, with the patent rights held by IBM.

A cipher being developed by a respected cryptanalyst is often a good sign.  In this case, it seems to be a very good sign, but not quite good enough.  Coppersmith has since helped to develop Scream [98], which is described as “a more secure SEAL,” in the paper where it was described.  Both borrow much from block ciphers.

SEAL is highly optimized for software implementation, and has the marked advantage that it actually defines a function family, so that individual outputs may be calculated directly, rather than in sequence.  This makes SEAL at least somewhat useful as a cipher for storage media, since individual 32-bit portions of file of up to 64 Kbytes may be decrypted directly.  It is also useful in environments where occasional messages are lost in transmission, since the state of the cipher need not be stored, only the key and the value *n*.

SEAL also has the reputation of being the fastest reasonably-strong stream cipher available, being able to encipher and decipher at a rate of better than two bits per CPU clock cycle, on modern 32-bit processors.

SEAL incorporates three tables, called R, S, and T (of 256, 256 and 512 32-bit entries each), and uses the Secure Hash Algorithm (SHA) to massage the 160-bit key to fill them. Table T is in practical terms an S-box. It also requires several registers, plus an additional up to 64 Kbytes of memory, so despite being efficient in software, it requires enough space that there are many space-constrained applications for which it is not practical.

Cryptanalysis has not fully broken SEAL, but both $\chi^2$ and correlation attacks have made progress against it [99][100]. Much of this progress is explicitly due to the restricted size of table *T*, as three iterations of transforms using T produce a range of only $2^{27}$ possible values, rather than the more desirable $2^{32}$.

### Scream

As noted in the discussion of SEAL, Scream [98] borrows heavily from block ciphers, and particularly from the Advanced Encryption Standard (AES). Scream exists in two forms, both of which have round functions based on AES, but modified for 64-bit blocks, rather than 128-bit ones. They differ in the S-boxes used. Scream-F uses the AES S-boxes as defined. Scream uses S-boxes that are derived from the key.

Apart from the AES features, the structure of Scream is quite similar to SEAL. It uses a 128-bit key, with a 128-bit nonce, and is almost as efficient as SEAL in terms of its rate of encryption and decryption.

### WAKE

David Wheeler’s Word Auto Key Encryption cipher [101] is, for obvious reasons, called WAKE. It is a stream cipher with feedback, driven by a relatively simple PRNG producing 32-bit blocks. Thus, it achieves a very high encryption rate. However, it borrows greatly from block cipher concepts, particularly in its use of an S-box.

The S-box is constructed from the key, and is composed of 256 32-bit entries, with the highest-order bytes consisting of all possible 8-bit values. Construction of the S-box is ill-defined by Wheeler, but can be performed as the implementer chooses, in any case, using any reasonably good PRNG or source of random data, provided the highest-order bytes of the entries display the required characteristic.

The remainder of the structure is composed of a set of four 32-bit registers, with four instances of a mixing function defined as follows.

$$z = M(x, y) = ((x + y) >> 8) \oplus S_{(x+y)\wedge 255}.$$

Here, "$a >> b$" is a right shift of $a$ by $b$ places, and "$S_{(x+y)\wedge 255}$" indicates the contents of the S-box at the position indicated by the lowest order eight bits of $x + y$.

The encryption and update cycle is as follows.

$$\begin{aligned}
c_i &= p_i \oplus D_i, \\
A_{i+1} &= M(A_i, c_i), \\
B_{i+1} &= M(B_i, A_{i+1}), \\
C_{i+1} &= M(C_i, B_{i+1}), \\
D_{i+1} &= M(D_i, C_{i+1}).
\end{aligned}$$

The simplicity and speed of the cipher are its downfall, as it is vulnerable to both chosen plaintext and chosen ciphertext attacks [21][25]. Also, the feedback mechanism is not designed to provide the self-synchronization found in some stream ciphers.

## Complexity-Theoretic Stream Ciphers

Rather than limit the design of stream ciphers to simple mechanisms arranged in depth, these ciphers seek to use what are believed to be "hard" problems as the basis for generators. They can also be referred to as "number theoretic" generators, as they utilize what are believed to be hard to solve number theoretical problems as their basis. Several have been designed, with four being listed by Schneier [25]: Shamir's PRNG [102], the Blum-Micali generator [103], RSA [104][105] and Blum-Blum-Shub (BBS) [106]. All four of these are based on the use of large prime numbers, and three (Shamir's, RSA and BBS) are related closely enough to be regarded as variations on the RSA theme.

These last three use large primes $p$ and $q$, just as the RSA public key cipher system does, but with slight changes, or with no changes at all, as in Shamir's and RSA PRNG. BBS requires that $p$ and $q$ be congruent to 3 *mod* 4, and rather than using the equation $x_{i+1} = x_i^e$ mod $N$ (where $N = pq$) uses $x_{i+1} = x_i^2$ mod $N$, with the lowest-order bit being used as the output.

BBS has a number of interesting properties, not least of which is that it is possible to compute $x_i$ directly, rather than sequentially. Further, it can be shown that for a given N, lg(lg(N)) bits are indistinguishable from random. But, more importantly, these lg(lg(N)) bits are unpredictable in sequence both forward and backward, making BBS a particularly good cryptographic PRNG.

## Congruential Generators

A very common PRNG that must be addressed, despite being unsuitable as a cryptographic PRNG or as a stream cipher, was introduced by Lehmer [107]. Linear congruential generators (LCG) are simple, fast and ubiquitous for non-critical applications. All LCGs are described by the recursion

$$X_{i+1} = (aX_i + b) \bmod m,$$

where $X_i$ is the $i$-th value in the sequence, and $a$, $b$ and $m$ are constants.

Selection of the constants $a$, $b$ and $m$ is critical. If chosen correctly, the LCG will have a range from 0 to $m - 1$ and a period of $m$. Such LCGs are said to be maximal period generators. Knuth [108] spends considerable space discussing the selection of these constants, and the reader is recommended to read that discussion if interested in LCGs.  But, as Schneier [25] points out, LCGs are predictable.  In his words, "Unfortunately, linear congruential generators cannot be used for cryptography; they are predictable."

Schneier [25], as well as Menezes, van Oorschot and Vanstone [21] cite numerous papers on the cryptanalysis of LCGs, and also of closely related generators.  Of particular interesting is Joan Boyar's cryptanalysis [109][110] of quadratic and cubic generators, which have the following forms.

$$X_{i+1} = (aX_i^2 + bX_i + c) \bmod m, \text{ and } X_{i+1} = (aX_i^3 + bX_i^2 + cX_i + d) \bmod m.$$

Others extended Boyar's methods to polynomial congruential generators [111][112][113]. Discarding low-order bits does not materially improve matters. [114]

The only conclusion that may be safely drawn about congruential generators in cryptographic applications is that they should not be used in any critical application requiring any level of security.

# CHAPTER III

## MIXING AND MATCHING USING A MCLAREN-MARSAGLIA THEME

While it is easy to design an algorithm to produce what one believes to be a good PRNG, the fact is that it is very easy to be wrong.  In some cases, the problem with a specific generator is that an implementer uses it in a situation for which it was never intended, and thus is ill-suited.  A good example of this is a generator proposed by McLaren and Marsaglia in 1965 [115], and subsequently discussed by Donald Knuth [108] under the name Algorithm M.  The algorithm has a very simple structure, and allows the integration of two PRNGs into a single generator.  Given Knuth's discussion of the algorithm, some implementers believed (and continue to believe) it would disrupt the underlying patterns inherent in both component generators, yielding a composite generator that is cryptographically strong.  While the concept is interesting, and very useful in many applications, implementation of the algorithm as described has proven unsuitable for cryptographic applications, as will be discussed in detail later.

Attempts have been made to modify the concept, primarily by altering the way the central feature of the mechanism (a shuffling array) is managed.  But, certain of the problems with the approach remain after such changes.  It is contended here that the concept's failure in cryptographic applications is not fundamental to the concept of shuffling, which may be viewed as a form of non-linear combining operation.  The specific implementation issues result in four related, cryptanalytically exploitable flaws.  To varying degrees, these problems may be ameliorated by appropriate changes in implementation, and generalizations of certain aspects of the mechanism. As one of these altered forms is exploited in the key-agreement scheme presented here, it is necessary to first discuss the algorithm and why it is cryptographically weak.

### The McLaren-Marsaglia Algorithm

As described by Knuth [108], the algorithm uses three objects: two generators to produce sequences $<X_n>$ and $<Y_n>$, and a shuffling table $V$ of $k$ entries. The mechanism is initialized by filling $V$ with the first $k$ values from $<X_n>$. We quote Knuth's description of the algorithm directly as follows, noting that the value $m$ used references a linear congruential PRNG being used to generate the $<Y_n>$ sequence.

> "M1. [Generate $X$, $Y$] Set $X$ and $Y$ to the next members of the sequences $<X_n>$ and $<Y_n>$, respectively.
>
> M2. [Extract $j$] Set $j \leftarrow kY/m$, where $m$ is the modulus used in the sequence $<Y_n>$; that is, $j$ is a random value, $0 \leq j < k$, determined by $Y$.
>
> M3. [Exchange] Output $V[j]$ and then set $V[j] \leftarrow X$."

Subsequently, Knuth stated the following.

> "On intuitive grounds, it appears safe to predict that the sequence obtained by applying Algorithm M will satisfy virtually *anyone's* requirements for randomness in a computer-generated sequence, because the relationship between nearby terms of the output has been almost entirely obliterated. Furthermore, the time required to generate this sequence is only slightly more than twice as long as it takes to generate the sequence $<X_n>$ alone."

The problem with intuition is that it is sometimes wrong. In the present instance, relying upon Knuth's first statement has led some to rely upon this intuitive view of the McLaren-Marsaglia algorithm as useful in cryptography. This is not the case, as was well shown by Charles T. Retter [116][117]. The algorithm was never intended for such use, as shown in the original paper by McLaren and Marsaglia [115]. Still, the algorithm is relatively efficient, as per Knuth's second statement, and the fundamental concept has distinct merits. If we view the shuffling array $V$ as being akin to non-linear combining functions, and generalize on that basis, the idea of combining two PRNGs using a non-linear combiner is useful, and is discussed at length in the following sections.

(Note that from this point forward we will drop the subscripted *n* in <$X_n$> and <$Y_n$> as unnecessary and potentially confusing. Also, <*X*> and <*Y*> may be used synonymously for the generators used to produce *X* and *Y*, as well as the sequence the generator produces.)

## Cryptanalysis of McLaren-Marsaglia

There are at least four flaws in McLaren-Marsaglia generators when used for cryptographic purposes. These flaws are closely related. Three were effectively exploited by Charles Retter in his key-search attack [117]. The first flaw is that all outputs from the composite generator are unaltered outputs from <*X*>, as *V* contains only values generated by <*X*>. The re-ordering of the <*X*> generator outputs leaves those values unaltered, and they remain identifiable as outputs from <*X*>, and <*X*> only. In terms such as discussed in Chapter II, there is a complete correlation between the output of a McLaren-Marsaglia generator and the output of its <*X*> generator, if we ignore the re-ordering. As a result, <*X*> may be cryptanalyzed separately from <*Y*>.

The second flaw, as per Retter [116], is that the average number of iterations of the composite generator between the point when a specific element from <*X*> is placed in *V* and its appearance in the output stream will be roughly *k*. So long as the size of *V* (which is *k*) is significantly smaller than *R*, the range of values elements in <*X*> may assume, this is an exploitable feature of the composite generator, since for the overall distribution of individual values in <*X*> will otherwise be one occurrence per *R* values.

The third flaw is that the initial contents of *V*, once replaced by new values, have no bearing on subsequent outputs. The entry in *V* used to provide the next output of the composite generator is solely determined by <*Y*>, and the contents of that entry are replaced. Thus, even if we chose to initialize *V* with truly random values, the effects of these initial values are completely lost once all *k* entries in *V* have been updated over the course of repeated invocations. If we contrast this with what would happen were we to exclusive-or (XOR) the freshly generated *X* with the contents of the entry in *V*, we can see in this case that the initial contents of V persist in effect, rather than disappearing. The effect of this one change on Retter's strategy is significant, as it would be necessary to guess the contents of *V* as well as the state of the <*X*> generator. Notice, though, that if an attacker has a list of outputs from the composite generator, these were entries in *V*, and thus strong clues to the state of *V* at the start of the sequence, particularly for small values of *k*.

The fourth flaw is an implementation issue, rather than a conceptual one. Many implementations of McLaren-Marsaglia use a mixing array of 32-bit values. This has the dual effect of limiting the number of elements in the <*X*> sequence that may be stored in *V* and making individual elements in that sequence clearly distinguishable in their occurrences within the <*X*> sequence. On the former point, given a shuffling array of 32 bit entries, in most applications it is impractical to provide a shuffling array sufficiently large as to contain more than a small fraction of all possible 32-bit values. For example, if $k = 2^{10}$, there can be no more than $2^{10} / 2^{32} = 2^{-22}$ of all possible 32-bit values in *V*. Assuming a uniform distribution of output values from <*X*>, any given value will only occur, on average once every $2^{32}$ outputs from <*X*> and the composite McLaren-Marsaglia generator. Thus, as illustrated below in examining Retter's attack, it is relatively easy to generate trial output sequences for <*X*> and determine whether the guess it represents is a good candidate as a solution for <*X*>. Using smaller values in the shuffling array doesn't necessarily solve this problem, as Retter illustrated his attack against an implementation using 8-bit values, and a shuffling array not much smaller than 256 entries. Thus, it can be seen that the size of the shuffling array relative to the range of values it may contain is an important consideration, also as discussed below.

To understand how these problems interact, consider a sequence <*T*> of $2^{32}$ 32-bit values such that no 32-bit value occurs more than once in <*T*>. If we take this sequence, concatenate copies repeatedly any number of times, and then apply a special “permutation” or shuffle to obtain the sequence <*S*>, such that given a value at position *i* in the original, unshuffled sequence, that value is displaced to the “right” some random number of positions and the average displacement is *d* where *d* is small relative to $2^{32}$, with elements that would be shifted past the end of the sequence wrapped around to the start, then rotate the whole sequence <*S*> a random number of places, the result may seem reasonably random. However, we can still determine the relative alignment of segments in this sequence to the original sequence <*T*>.

The way this alignment can be determined is simple. We select a value at some location in <*S*> then find that value in the original sequence <*T*>. We align <*T*> to <*S*> using that pair of locations as indices, and shift <*T*> left one position. We then examine the distances from the values in <*T*> to their nearest occurrences in <*S*>. Since we know that *m* is small relative to $|<T>| = 2^{32}$, if no distances are negative, then <*T*> and <*S*> are aligned. If any nearest distances are negative, we shift the alignment of <*T*> in that direction by |*d*| + 1 positions, where |*d*| is the absolute value of largest negative distance. We can then verify the alignment by again checking the displacements from the occurrence of a value in <*T*> to the nearest occurrence in <*S*>, as aligned. If none of the distances is negative, <*T*> is properly aligned relative to <*S*>.

The process just described is in essence Retter’s attack [116][117], which first isolates the <*X*> sequence, and thus the generator producing it, for cryptanalysis. While Retter’s description starts with the assumption that a McLaren-Marsaglia generator is being used to generate a stream cipher, and includes a known-plaintext attack as the starting point, a more rigorous starting point is that the output sequence is known, as this conforms with a strong form of Kerckhoffs' axiom: everything is known, except the key (in this case the state/seed of the McLaren-Marsaglia generator). Retter’s cryptanalysis of the McLaren-Marsaglia generator also differs from the alignment problem described in that only a portion of <*S*>, *k* (the size of *V*), and the generators for <*X*> and <*Y*> are known, and the average delay between insertion of a value into *V* and its output from the composite generator is as well.

Retter's demonstration of the attack is based on a simplified model, with the <*X*> sequence being byte values, rather than 32-bits or larger. Despite a shuffling array size of 100 entries, and a long maximum delay, the fact that the table size is smaller than the range of values bytes may assume, resulting in the average delay from a value's insertion to its appearance in the output stream being equal to *k*, the correlation between a correct key/seed guess and average delay remains very strong.

In Retter's attack, a random seed for the <*X*> generator is selected, and a sequence of outputs generated based on that seed. These values are then checked against a known sample sequence from <*S*>. If there are no or few appearances of the generated values in the sample, we select another random seed, generate a sequence of outputs using it, and try again. Otherwise, the delays between the appearances of values in the trial sequence and their occurrence in the output stream are determined and averaged. If the average is near the table size, *k*, the guess is close, and can be refined. If the average is large, approaching *R*/2, a new seed and trial sequence is again generated, and the process repeated. This attack closes on a solution for the state of the <*X*> generator rapidly, when compared to a brute force attack on *V*, <*X*> and <*Y*>, as it is bounded by the state size of the <*X*> generator alone.

Note that the key for a stream cipher and the state are not necessarily identical in size. The key is used to initialize the mechanism, but there may be additional initialization material that is not part of the key, yet may be changed as a result of the ciphers operation over time. In Retter's published form of his attack, the key and state are synonymous, but this need not be the case.

The time required for Retter's attack varies with the resources available, the size of the sample output from the McLaren-Marsaglia generator, the period and complexity of the <*X*> generator, and *k*.  Cryptanalysis of the <*Y*> sequence can then be done separately.

## The Bays-Durham Generator

The McLaren-Marsaglia generator is not the only shuffling algorithm.  Immediately following his description and discussion of McLaren-Marsaglia, Knuth [108] presents an algorithm developed by Bays and Durham [118], titling it "Algorithm B."  We shall follow Knuth's description.

Like the McLaren-Marsaglia generator, the Bays-Durham generator incorporates a shuffling array *V*.  However, utilization of this shuffling vector is different from McLaren-Marsaglia, as it uses the same PRNG to both fill shuffling array *V* and to select entries in *V* for output.

In operation (assuming the <*X*> generator is a linear congruential generator) the Bays-Durham generator is initialized by filling the shuffling array *V* (of *k* entries) with the first *k* outputs from the <*X*> generator, and setting  generator, and setting *Y* to the next (*k* + 1) output.  Index *j* is then calculated as $j = \lfloor kY / m \rfloor$, where *m* is the modulus of the <*X*> generator.  Set *Y* equal to *V*[*j*]. Pump the <*X*> generator for a value and set *V*[*j*] to that value.  Output *Y*.

For non-cryptographic applications, this is a very good PRNG.  As only one PRNG is required, and the overhead of indexing the shuffling array is low (comparable to McLaren-Marsaglia), it is faster than McLaren-Marsaglia, assuming the same <*X*> generator is used for both.

For cryptographic purposes (a use for which it was never intended), this approach suffers the same liabilities with regard to the size and contents of the shuffling array that McLaren-Marsaglia does, but adds a new liability in that the output at any given point tells a cryptanalyst what shuffling array entry will provide the next output.

For the purposes of this thesis, the Bays-Durham approach does not offer the same opportunities for injection of aperiodicity, nor of non-linear combining functions, as does the McLaren-Marsaglia scheme. It is therefore noted as considered, and not discussed further, though it might be used as part of a constituent generator for the protocols discussed in Chapter IV.

## Modifying McLaren-Marsaglia

Various suggestions have been made as to how the McLaren-Marsaglia approach may be improved, in the cryptographic sense or otherwise. A proposal by Tyanev, Petkova and Tyaneva [119] is an example. As noted in the previous section, one of the basic flaws in the McLaren-Marsaglia approach is that the outputs of the composite generator are outputs of the <*X*> generator, another is the size of the shuffling array relative to the range of values the outputs may assume, and a third is the unaltered passage of inputs from <*X*> through the mechanism to the output stream. To varying degrees, such modifications address these problems. As interesting as many of these are, they complicate the cryptanalysis of a McLaren-Marsaglia generator only by increments. Indeed, the first of two modifications by Tyanev, et al. [119] merely changes the way the <*Y*> sequence is used to access the shuffling array *V*, which does nothing to complicate Retter's attack. Thus, in order to rehabilitate the McLaren-Marsaglia concept, other modifications may appropriately be examined.

Perhaps the most serious flaw of McLaren-Marsaglia in many implementations is the size of the shuffling array relative to the range of output values. Even with schemes that hash two or more entries in *V* together, the size of the shuffling array strongly affects the strength of the result. For example, if we assume the output function selects three entries in the shuffling array *V* (each entry being 32-bits long) and performs a hashing operation using the contents of those entries, assuming that there are no repetitions of values in *V*, if $k = 64$, we can still have at most $64^3 =$ 262,144 different values as possible outputs for any invocation, assuming no collisions.

Significant as this number is, it is still a small fraction of the range of values a 32-bit integer may assume.

With such considerations, it appears prudent to find other ways to increase the available range of outputs from *V*, at any point in the period of the composite generator, relative to the range of all possible outputs. Fortunately, there is a simple approach that accomplishes this in cryptographic applications, and two considerations lead in the same direction.

First, it can be seen that the effectiveness of Retter's attack [117] derives from the difference in expected delays in the appearance of values in the output stream, given wrong guesses as to the state of the <*X*> generator, and correct or near-correct guesses. (A "near correct" guess is one that corresponds to the state of the generator near the point in the sequence for which we are making a guess.) This difference derives from the values of *k* and *R*. As Retter showed, with $k = 100$ and $R = 256$, good guesses result in delays averaging *k* iterations after insertion, and bad ones in delays averaging *R*/2. But, if $k \geq R/2$, this attack cannot readily differentiate between good and bad guesses.

Second, whereas many implementations of the McLaren-Marsaglia algorithm treat the output from the <*X*> generator as a stream of 32-bit or larger values, Retter [116][117] and Schneier [25] implement *V* as an array of bytes, while Knuth [108] does not discuss the size of the entries in *V* at all. We are not bound by the implementation of the <*X*> generator to any specific size of elements in *V*, provided we are not dealing with floating point values, or with values that must conform to some non-uniform distribution. We can treat the <*X*> stream as we choose, as bits, nybbles, bytes, etc., and remain fully in the spirit of McLaren-Marsaglia. The interpretation of a bit or group of bits is, after all, completely arbitrary. Thus, we may treat <*X*> as a bit stream, and impose any organization on it we choose.

An example of how we can do this is treating the outputs from the <*X*> generator as groups of bytes which can be regrouped as we choose. Thus, where we might have chosen $k = 256$ due to space constraints when dealing with a shuffling array of 32-bit values, we could use the same space for an array of bytes where $k = 1024$. While one effect is unfortunate – it takes 40 bits to independently index four bytes – there are positive effects in terms of dealing with Retter's attack. A 256-entry array of 32-bit values can contain at most $2^8 / 2^{32} = 2^{-24}$ of all possible 32-bit values. On the other hand, a 1024-entry array of random bytes will contain an average of four of each of the possible values. Since the probability of a specific byte value being absent from a byte-array of this size is $((2^8 - 1)/2^8)^{1024} \cong 0.018173,$ there is a small but reasonable probability that any possible 32-bit value may be produced by randomly selecting four entries and concatenating them to produce a 32-bit value. This is the approach used by Schneier [25], who presents the algorithm using a table of $2^{13} = 8192$ entries, with the recommendation "… the larger the better."

Viewed as a byte-stream, the output will tend to contain occurrences of every possible byte value, on average, once per 256 bytes output. This means that the average delay for an arbitrary wrong guess of the seed of the <*X*> generator will display an average delay of $R/2 = 128$ bytes, as predicted by Retter, but the average delay before the appearance of any one byte value and an arbitrary point in the output stream will also be 128 bytes. We cannot readily differentiate between any two bytes of the same value and their point of origin in the <*X*> byte sequence, unless there is an uneven distribution of the value in that sequence that results in large gaps between clusters of occurrences. While a statistical analysis of the distribution of byte values in the <*X*> sequence and the output stream may well yield positive results in trying to isolate the <*X*> generator, increasing the number of entries in *V* will tend to reduce the effectiveness of this approach by decreasing the probability that *V* will become seriously depleted with respect to any byte value.

Here is an implementation of the byte-based construction of 32-bit outputs, with *k* = 1024.

```
unsigned long Xn();         // generator for <X>
unsigned long Yn4(int *in); // generator for <Y>, mod’ed to provide four
                            // indices in the range 0 to 1023, inclusive

unsigned char V[1024];      // the byte-structured shuffling array

unsigned long BitMatchStream()
{
    int                     indices[4]; // for the indices used
    int                     i;          // a counter
    unsigned long           fill;       // used to re-fill V slots
    unsigned long           out    = 0; // for the result

    Yn4(indices);
    fill = Xn();

    for (i = 0; i < 4; i++)
    {
        out = (out << 8) | (unsigned long) (V[indices[i]]);
        V[indices[i]] = (unsigned char) (fill & 0x000000ffL);
        fill = fill >> 8;
    }

    return out;
}
```

Byte-based McLaren-Marsaglia with 32-bit outputs

Some of the flaws of the McLaren-Marsaglia scheme remain even with this approach. In particular, the contents of *V* do not contribute meaningful added complexity in terms of the size of the internal state that must be determined for the composite generator. A correct guess of the states of the <*X*> and <*Y*> generators at any point will eventually, once the initial contents of *V* have been replaced, allow determination of the new contents of *V*, and lead to a complete solution. Fortunately, the opportunities for modifying the McLaren-Marsaglia mechanism have not been exhausted, as will be discussed later.

As a 1024-byte shuffling array is too large for some applications, treating the <*X*> data stream as a stream of even smaller units may be considered, though at the cost of additional problems in indexing elements in *V*. Treating the <*X*> sequence as a stream of 4-bit nybbles is useful, as *V* may require less space while decreasing the probability that an arbitrary value is not present in *V*. This effect may be seen in the following table. Note that even if byte locations are used to store the nybbles in *V*, a 256-entry table of nybbles would occupy the same space as a 256-entry byte table, with a dramatic reduction in the probability that any given value would be absent, but require twice the number of indexing bits.\

The following table offers a comparison between shuffling arrays with 4-bit and 8-bit entries. As with prior discussions, *V* is the shuffling array (viewed her as a set of values), *k* is the size of *V* in entries, and

$$P(x \notin V) = ((2^b - 1)/2^b)^k$$

is the probability that a particular *b*-bit value is not present in *V*. As can be seen, for a given table size (relative to the entry size) the smaller entry size offers a slightly lower probability that a value will be absent from *V* for even small array sizes. This advantage increases with increasing table size (again relative to the entry size).

| bits/entry | | | | | | |
|---|---|---|---|---|---|---|
| 8 | *k* | 256 | 512 | 1024 | 2048 | 4096 |
| | $P(x \notin V)$ | 3.67E-01 | 1.35E-01 | 1.82E-02 | 3.30E-04 | 1.09E-07 |
| 4 | *k* | 16 | 32 | 64 | 128 | 256 |
| | $P(x \notin V)$ | 3.561E-01 | 1.268E-01 | 1.608E-02 | 2.584E-04 | 6.68E-08 |

Probability a value *x* is absent from *V* based on size of *V* and bits per entry

## Bit-Selection

When the <*X*> input stream is treated purely as a bit-stream, the problem of indexing becomes acute. Something as simple as a 32-bit bit-wise shuffling array would appear to require a total of 40 bits of indexing material to produce a byte of output. The bits to be used must also be assembled into the required size blocks, and bit positions in *V* refilled from <*X*>. An alternative to indexing is an idea from Bennett and Brassard. In their proposed scheme for agreeing upon a One-Time Pad using quantum phenomena, they use a system of bit-matching. When a random bit (encoded as a single polarized photon) sent by Alice is correctly guessed (via the photon's polarization) by Bob, the value of that bit is used as a bit in the key-stream being generated. There are several ways this type of bit-matching may be adapted to the present problem.

The first method is to treat both the <*X*> and <*Y*> sequences as bit-streams, and do the same bit-match operation as used in the Bennett-Brassard paper: corresponding bits in the two bit-streams that match in value are used in the output stream. Another is to again treat both sequences as bit-streams, and select the bit values in <*X*> corresponding to 1-bits in <*Y*> for the output stream (though the same effect is obtained if we select bit values in <*X*> based on 0 bits in <*Y*>). Other selection and output functions are also applicable. All such methods have much the same effect, though selecting different bits, and thus generating different output streams. Each requires, on average, the use of two <*X*> sequence bits, and two <*Y*> sequence bits to generate one output bit. A code sample illustrating how this may be done is presented here.

```
unsigned long Xn();         // generator for <X>
unsigned long Yn();         // generator for <Y>

unsigned long BitMatchStream2()
{
    static unsigned long    xyMask = 0;
    static unsigned long    xBuff  = 0;
    static unsigned long    yBuff  = 0;
    unsigned long           oBuff  = 0;
    int                     oCnt   = 0;
    unsigned long           val    = 0;

    while (oCnt < 32)           // while needing more bits for output ...
    {
        if (xyMask == 0)        // indicates buffers are empty
        {
            xBuff  = Xn();      // refill buffs, set bit select at first
bit
            yBuff  = Yn();
            xyMask = 0x80000000;
        }

        val = xBuff & xyMask;   // grab a bit in <X> stream

        if ((yBuff & xyMask) == val)    // check for a bit match and do
                                        // the following if matched
        {
            oBuff  = (oBuff << 1) | ((val != 0) ? 1 : 0);
            oCnt++;
            xyMask = xyMask >> 1;
        }
    }

    oCnt = 0;

    return oBuff;
}
```

Bit selection from <*X*> based on matching <*Y*> bits

This technique does not match the McLaren-Marsaglia structure, since we do not have a shuffling array. Thus, for a single pass through the <*X*> sequence, the bits output are in the same order they were in within <*X*>. This must be regarded as a liability, since an attacker would have roughly half the bits in some segment of <*X*> in their correct order, given a sample of the resulting output stream. The number of complete bit sequences from <*X*> that could yield a given sample is large, especially given the fact that, though the length of the segment from which the sample's bits were extracted will be unknown, the length will tend towards twice the number of bits in the sample, particularly as the sample size increases. The number of possible source bit-string pairs that could generate a known *n*-bit bit-stream is given by the following, where *C* is the number of corresponding bit sequences.

$$C = \sum_{i=n}^{\infty} \binom{i}{n} 2^{i-n}.$$

Note that there is an assumed ordering of the two strings, which is appropriate for the cryptographic environment. Thus, if string *A* = 0110 and *B* = 0101, we do not regard this as being the same as *A*' = 0101 and *B*' = 0110, even though the result is the same for both string pairs. In the cryptanalysis of any such implementation in a PRNG, the sources of the respective strings are highly relevant.

While the summation is infinite, in practical terms an attacker would be concerned only with the region where *i* is near 2*n*. Still, this is not very helpful to an attacker. Consider the following case. A bit-matching process yields the 4-bit pattern 0101. If we know that this was generated from a pair of 8-bit sequences, the above equation tells us that there are

$$\binom{8}{4}2^{8-4} = \left(\frac{8!}{4!4!}\right)2^4 = 70*16 = 1120$$

8-bit string pairs that could have generated this four-bit pattern. As *n* increases, the ratio of potential generating pairs to the number of values $2^{2n}$ can assume increases rapidly, though certainly not as rapidly as $2^{4n}$, which is the number of possible 2*n*-bit pairs, ignoring matches.

One of the possible <*X*> generating streams is 00110011. Since any 0-bit is indistinguishable from any other, as is any 1-bit from any other 1-bit, we are left with 16 different patterns selecting the known bits that would produce that pattern, and, even if we know the source pattern was the one used, no means by which to conclude which bits were in fact used in the output. Such a determination requires knowledge of the <*Y*> bit-stream that was used.

The following table illustrates the combinatorial growth rate for an *n*-bit output stream, assuming only 2*n* bit input streams (as in the *i* = 2*n* case in the above summation formula) were used to generate the output. Note that the righ-hand shows the relative growth rate of the factorial piece of the problem set – the distribution of the selected bits within the input strings.

| $n$ | $2^{2n}$ | $(2^n)((2n)!/(n!)^2)$ | $(2n)!/(n!)^2 2^n$ |
|---:|---:|---:|---:|
| 4 | 256 | 1120 | 4.375 |
| 8 | 65536 | 3294720 | 50.273 |
| 16 | 4.29E+09 | 3.939E+13 | 9171.759 |
| 32 | 1.84E+19 | 7.871E+27 | 4.27E+08 |
| 64 | 3.4E+38 | 4.418E+56 | 1.3E+18 |

Growth of number of candidate input string pairs,
assuming *n* bits of output and input strings of 2*n* bits.

Stirling's approximation for large factorials is also useful for understanding the relative growth rates, and is given here.

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}$$

From this it can be clearly seen that the number of corresponding string pairs increases rapidly, with increasing *n*, since we can substitute this approximation into the formula for corresponding string pairs. For the case of *n* bits generated from 2*n*-bit strings, we have the following approximation for the factorial portion of the formula.

$$\binom{2n}{n} = \frac{(2n)!}{n!n!} \approx \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right)^{2}} = \frac{2\sqrt{\pi n}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} = \frac{2^{2n}}{\sqrt{\pi n}}.$$

This gives us the following approximation for the number of corresponding input strings of 2*n* bits, yielding an output string of *n* bits, where *n* is large.

$$\binom{2n}{n}2^{n} \approx \frac{2^{2n}}{\sqrt{\pi n}}2^{n} = \frac{2^{3n}}{\sqrt{\pi n}}.$$

A useful observation here is that if the sequences <*X*> and <*Y*> are random bit-streams, the bits output by the above algorithm will also be random. If these bits-streams are produced by PRNGs, but they are bit-wise apparently random (meaning that there is no practical means by which to accurately predict whether the next bit to be generated will be a 1 or a 0 except to know the state of the PRNG at that point in the sequence, even with knowledge of all prior outputs), the output from the bit-matching algorithm will likewise be bit-wise apparently random. We can then reasonably conclude that if the generators for <*X*> and <*Y*> are both “good” (meaning bit-wise apparently random, satisfying various tests of randomness, etc.) and independent (they do not produce the same output stream nor cycle period, and there is no internal relationship between their separate outputs), successful cryptanalysis of the output stream from the bit-matching process, in order to determine upcoming bits in their correct positions with reasonable likelihood, should require that both generators be solved.

With a standard McLaren-Marsaglia implementation using byte or larger entries in *V* and *k* significantly less than *R* / 2 (*R* being the range of possible values for entries in *V*, and *k* the table size), the fact that a correct solution for <*X*> yields a rolling window of possible outputs, solving <*Y*> may be unnecessary, as the probability of a correct prediction of the next output will be related to 1 / *k*, rather than 1 / *R*, as would be the case for a truly random output.

There is a potential downside to this scheme. If we consider cases where the periods of <*X*> and <*Y*> differ, the composite period will be equal to the least common multiple (LCM) of the two periods. Given a very large LCM (such as when the periods of the two generators are relatively prime), we should anticipate all bits in the <*X*> sequence to be eventually used in the composite output stream, or at least the vast majority being present. Since, segment-wise, the bits will be ordered, there may well be a way to exploit the ordering of the bits to reconstruct the original sequence produced by <X>. This must be regarded as a potentially serious flaw, barring some mechanism to further obscure the ordering of the bits, though at present we do not anticipate this being exploitable.

There is an alternative to straight bit-matching that uses the same concept as a means of virtual indexing, and partially shuffles the bits obtained from <*X*>, and more closely conforms to the McLaren-Marsaglia model. In this approach, <*X*> and <*Y*> are still treated as bit-streams, but the <*X*> bits are used to refill locations in a bit-table that serves as *V*, while bits from <*Y*> are used to select bits in *V* for output. A code sample follows, illustrating this technique.

```
unsigned long Xn();         // generator for <X>
unsigned long Yn();         // generator for <Y>

unsigned long BitMatchShuffle()
{
    static unsigned long    iFlag  = false;
    static unsigned long    xMask  = 0;
    static unsigned long    xBuff  = 0;
    static unsigned long    yMask  = 0;
    static unsigned long    yBuff  = 0;
    static unsigned long    vArray = 0;
    unsigned long           oBuff  = 0;
    int                     oCnt   = 0;
    unsigned long           val    = 0;

    if (iFlag == false)     // init shuffling array if not already done
    {
        vArray = Xn();
        iFlag  = true;
    }

    while (oCnt < 32)       // until enough bits are gathered ...
    {
        if (xMask == 0)     // refill <X> buffer if spent & set selector
        {
            xBuff  = Xn();
            xMask  = 0x80000000;
        }

        if (yMask == 0)     // refill <Y> buffer if spent & set selector
        {
            yBuff  = Yn();
            yMask  = 0x80000000;
        }

        val = vArray & yMask;       // grab a shuffling array bit value

        if ((yBuff & yMask) == val) // if <Y> bit matches, copy bit to
output
                                    // and update accordingly
        {
            oBuff = (oBuff << 1) | ((val != 0) ? 1 : 0);
            oCnt++;
            vArray = (vArray & !yMask) | ((xBuff & xMask) ? yMask : 0);
            xMask = xMask >> 1;
        }

        yMask = yMask >> 1;    // shift to next mask bit
    }

    oCnt = 0;

    return oBuff;
}
```

Bit-matching via a shuffling array

Here, the output stream is a bit-wise “permutation” of <*X*> in much the same way that the original form of McLaren-Marsaglia shuffles the <*X*> to produce a “permutation” of <*X*>, with two important exceptions. First, a bit inserted into the table is only tested for possible output once per cycle through the table, which may prove to be a flaw. Second, if *V* is initialized with random bit values, the effects of these bits will tend to persist well past their replacement in *V*:

these bits, and the subsequent states of *V* are important elements in the overall state and output cycle of the composite generator.

The persistence of effect can be seen by considering a single bit position, and the effect of changing the value of that bit. Given an initialization of *V*, if we change one bit from a 1 to a 0, we will change the point of insertion of the next output bit value from <*X*> by either changing from insertion of that bit at that position to not inserting it, or from non-insertion to insertion. In the former case (non-insertion), the bit value will be inserted at a later point in the process, likely in a different position in *V*, and its effect in the output stream is thus displaced, as well as the bits it and its successor values in that position will be tested against from <*Y*>. In the latter case (insertion), the same argument applies, with the bit value from <*X*> being inserted in a different position, along with the resultant change in bits from <*Y*> it and the position's (in *V*) successor values will be tested against.

The distribution of bits from the source stream (<*X*>) into the permuted output stream follows a reasonably consistent pattern. Since the point of comparison in a shuffling array of *k* bits returns to the same location only once every *k* comparisons, normal operation of the process will force a minimum shift that averages *k*/2 positions in the output stream, and the probability that the bit will be output as a result of any given test of its position (and value) in the shuffling array is 0.5, assuming the <*Y*> bit-stream is random.

Since all the bits from <*X*> appear in the output stream of this shuffling algorithm, it would seem that an attacker is given a substantial edge in determining <*X*>. This is not the case. An attacker faces the problem that any two bits of the same value are indistinguishable from each other, and averages are just that – averages. An attacker cannot easily tell whether a particular bit value in the output stream originated from any given point in the <*X*> input sequence. Depending on how many bits are output between the point a bit is placed in the shuffling array and its first examination as a potential bit-match, the first possible appearance of that bit may be anywhere from being the very next to *k* bits later, and there is no guarantee it will be output in that interval.

The probability distribution for any number of bits from zero to *k* – 1 being output before a bit position can be re-tested against the next <*Y*> sequence bit for use as output is given by the binomial distribution. For *k* = 32, the average number of bits output before the same position is retested is 15.5, and the probability that some number of bits outside the 14-to-18 range will be output before a given bit is first examined is better than 0.47, but there is only a 0.5 probability that the bit will appear at that point in that cycle through testing the bits in *V*. If a larger bit-vector is used (say *k* = 64), both the separation between placement of a bit in *V* and its first test, and the number of bits that may be output between tests of the bit in *V* increase.

The probability that a given bit is the first bit output after its insertion into the shuffling table is simple, when assuming random <*X*> and <*Y*>, and is the probability that no other bit is used as output before it is. Since the probability that the contents of any given bit position will be used as output is 0.5 (assuming random <*Y*>) for a table of *k* bits that bit position in questions probability of use as next output source in the first cycle through the table (following the prior use/insertion) is $0.5^{k-1} * 0.5 = 0.5^k$. For each successive pass, the probability decreases by a factor of $0.5^k$, so the probability that the position will provide the next output bit value by the following.

$$\sum_{i=1}^{\infty} \frac{1}{2^{ik}}$$

Calculation of the probability of an individual bit in *V* having a specific delay in the output sequence greater than zero is non-trivial. The calculation involves infinite series and binary distributions over large sequences. For example, if we want to know the probability that a bit will appear in the output stream the second time it is tested, and is the *i*-th bit (*i* < *k*) output after its initial insertion into *V*, with *k* bits in the shuffling array, we end up determining the probability for all distributions of *i* – 1 bits output in 2(*k* – 1) trials, times the probability that the position is not used in the first pass, times the probability that it is used in the second. The overall probability is, then, a summation of an infinite series of such calculations.

How effective this approach may be in countering Retter's attack is related to the same question with regard to other output unit sizes. In Schneier's words [25], "… the bigger the better," when it comes to the dimensions of *V*. We expect that for large shuffling tables the effectiveness will tend to decline with increasing values of *k*, when *k* > *R*. In the case of bit-shuffling, it might well prove that an analysis of local "bit density," meaning the ratio of 1's and 0's in segments of <*X*> and the output, will provide an effective means of attack, since the ratio of 1's to 0's will in *V* at any given time will tend to be reflected in the output sequence.

It must be recognized that true bit-wise sampling of the contents of *V* is a slow process. For every *n* bits of output from the composite generator, *n* bit replacements in *V*, and an average of 2*n* bit-tests, must be performed. A tabular approach can speed this some, as illustrated in the following code sample, and introduces a step in the development of more complex mixing methods.

```
unsigned long Xn();
unsigned long Yn();

int bits[16][16] = {
{0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
{0,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,  1,  0,  1},
{0,  0,  2,  1,  0,  0,  2,  1,  0,  0,  2,  1,  0,  0,  2,  1},
{0,  1,  1,  3,  0,  1,  1,  3,  0,  1,  1,  3,  0,  1,  1,  3},
{0,  0,  0,  0,  4,  2,  2,  1,  0,  0,  0,  0,  4,  2,  2,  1},
{0,  1,  0,  1,  2,  5,  1,  3,  0,  1,  0,  1,  2,  5,  1,  3},
{0,  0,  2,  1,  2,  1,  6,  3,  0,  0,  2,  1,  2,  1,  6,  3},
{0,  1,  1,  3,  1,  3,  3,  7,  0,  1,  1,  3,  1,  3,  3,  7},
{0,  0,  0,  0,  0,  0,  0,  0,  8,  4,  4,  2,  4,  2,  2,  1},
{0,  1,  0,  1,  0,  1,  0,  1,  4,  9,  2,  5,  2,  5,  1,  3},
{0,  0,  2,  1,  0,  0,  2,  1,  4,  2, 10,  5,  2,  1,  6,  3},
{0,  1,  1,  3,  0,  1,  1,  3,  2,  5,  5, 11,  1,  3,  3,  7},
{0,  0,  0,  0,  4,  2,  2,  1,  4,  2,  2,  1, 12,  6,  6,  3},
{0,  1,  0,  1,  2,  5,  1,  3,  2,  5,  1,  3,  6, 13,  3,  7},
{0,  0,  2,  1,  2,  1,  6,  3,  2,  1,  6,  3,  6,  3, 14,  7},
{0,  1,  1,  3,  1,  3,  3,  7,  1,  3,  3,  7,  3,  7,  7, 15}};

int shifts[16][16] = {
{4,  3,  3,  2,  3,  2,  2,  1,  3,  2,  2,  1,  2,  1,  1,  0},
{3,  4,  2,  3,  2,  3,  1,  2,  2,  3,  1,  2,  1,  2,  0,  1},
{3,  2,  4,  3,  2,  1,  3,  2,  2,  1,  3,  2,  1,  0,  2,  1},
{2,  3,  3,  4,  1,  2,  2,  3,  1,  2,  2,  3,  0,  1,  1,  2},
{3,  2,  2,  1,  4,  3,  3,  2,  2,  1,  1,  0,  3,  2,  2,  1},
{2,  3,  1,  2,  3,  4,  2,  3,  1,  2,  0,  1,  2,  3,  1,  2},
{2,  1,  3,  2,  3,  2,  4,  3,  1,  0,  2,  1,  2,  1,  3,  2},
{1,  2,  2,  3,  2,  3,  3,  4,  0,  1,  1,  2,  1,  2,  2,  3},
{3,  2,  2,  1,  2,  1,  1,  0,  4,  3,  3,  2,  3,  2,  2,  1},
{2,  3,  1,  2,  1,  2,  0,  1,  3,  4,  2,  3,  2,  3,  1,  2},
{2,  1,  3,  2,  1,  0,  2,  1,  3,  2,  4,  3,  2,  1,  3,  2},
{1,  2,  2,  3,  0,  1,  1,  2,  2,  3,  3,  4,  1,  2,  2,  3},
{2,  1,  1,  0,  3,  2,  2,  1,  3,  2,  2,  1,  4,  3,  3,  2},
{1,  2,  0,  1,  2,  3,  1,  2,  2,  3,  1,  2,  3,  4,  2,  3},
{1,  0,  2,  1,  2,  1,  3,  2,  2,  1,  3,  2,  3,  2,  4,  3},
{0,  1,  1,  2,  1,  2,  2,  3,  1,  2,  2,  3,  2,  3,  3,  4}};

int set[16][16] = {
{0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15},
{0,  0,  2,  2,  4,  4,  6,  6,  8,  8, 10, 10, 12, 12, 14, 14},
{0,  0,  1,  1,  4,  4,  5,  5,  8,  8,  9,  9, 12, 12, 13, 13},
{0,  0,  0,  0,  4,  4,  4,  4,  8,  8,  8,  8, 12, 12, 12, 12},
{0,  0,  1,  1,  2,  2,  3,  3,  8,  8,  9,  9, 10, 10, 11, 11},
{0,  0,  0,  0,  2,  2,  2,  2,  8,  8,  8,  8, 10, 10, 10, 10},
{0,  0,  0,  0,  1,  1,  1,  1,  8,  8,  8,  8,  9,  9,  9,  9},
{0,  0,  0,  0,  0,  0,  0,  0,  8,  8,  8,  8,  8,  8,  8,  8},
{0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5,  6,  6,  7,  7},
{0,  0,  0,  0,  2,  2,  2,  2,  4,  4,  4,  4,  6,  6,  6,  6},
{0,  0,  0,  0,  1,  1,  1,  1,  4,  4,  4,  4,  5,  5,  5,  5},
{0,  0,  0,  0,  0,  0,  0,  0,  4,  4,  4,  4,  4,  4,  4,  4},
{0,  0,  0,  0,  1,  1,  1,  1,  2,  2,  2,  2,  3,  3,  3,  3},
{0,  0,  0,  0,  0,  0,  0,  0,  2,  2,  2,  2,  2,  2,  2,  2},
{0,  0,  0,  0,  0,  0,  0,  0,  1,  1,  1,  1,  1,  1,  1,  1},
{0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0}};

unsigned long bitSpigot()
{
    static unsigned long       V;
    static int                 flag = 0;
    static unsigned long long  outBuff = 0;
    static int                 outBuffDepth = 0;
    static unsigned long long  bitQueue = 0;
    static int                 bitQueueDepth = 0;
    static unsigned long       mskQueue = 0;
    static int                 mskQueueDepth = 0;
    unsigned long              outVal;

    if (flag == 0)
    {
        flag = -1;
        V = Xn();
    }

    while (outBuffDepth < 32)
    {
        unsigned long  temp = 0;
        unsigned long  i, j, k, l;

        if (bitQueueDepth < 4)
        {
            bitQueue |= (((unsigned long long) Xn())
                     << (32 - bitQueueDepth));
            bitQueueDepth += 32;
        }
```

```
            if (mskQueueDepth == 0)
            {
                mskQueue = Yn();
                mskQueueDepth = 32;
            }

            V = (V >> 28) | (V << 4);       // rotates V.

            i = V & 0x0000000fL;

            mskQueue = (mskQueue >> 28)     // rotates mskQueue
                     | (mskQueue << 4);
            mskQueueDepth -= 4;             // bits are "spent"

            j = mskQueue & 0x0000000fL;

            k = (unsigned long) ((bitQueue & 0xf000000000000000L) >> 60);
            l = i ^ j;

            outBuff = (outBuff << shifts[i][j])
                    | ((unsigned long long) bits[i][j]);
            outBuffDepth += (unsigned long long) shifts[i][j];

            V = (V & (0xfffffff0L | l)) | set[l][i];

            bitQueue = bitQueue << shifts[i][j];
            bitQueueDepth -= shifts[i][j];
        }

        outVal = (unsigned long) (outQueue & 0xffffffffL);
        outBuff = outBuff >> 32;
        outBuffDepth = outBuffDepth - 32;

        return outVal;
    }
```

Table-based bit-wise shuffling

Note how the sampling has been revised. The low-order nybble of *V* is used after *V* has been rotated, rather than keeping explicit track of the current location in *V*. Also, as we place groups of up to four bits in the output buffer at a time, we will regularly push up to three bits above the region in the output buffer that will be used for the imminent output. Since we are free to sample as we wish in *V*, given the McLaren-Marsaglia model, this makes no objectionable difference, and improves the shuffling process, since such "over buffered" bits may remain in the output buffer for extended periods.

There are several things that may be done to increase the occurrence and effect of such over-buffering. By increasing the dimensions of the arrays used, and the associated number of bits operated on per cycle of the loop, and also by constructing first bytes, then final outputs from bytes, the over-buffering, and thus the shuffling effects can be increased.

## Some Cryptographic Considerations

One serious liability remains incompletely addressed at this point. The data units output by the composite generator, whether bits, bytes or other constructs, remain “raw” outputs from the <*X*> generator, though shuffled. However indistinguishable any one bit or byte is from any other of the same value may be, the obfuscation of the <*X*> output is purely a result of localized shuffling. In the case of bit shuffling, the probability of a bit’s appearance at successive points in the output stream is not smoothly decreasing. It displays regions of relatively higher probability which only tend to fade, but remain present, spaced at roughly $k$ / 2 intervals. This cannot be regarded as fully satisfactory, from a security standpoint. The redeeming qualities of the bit-wise McLaren-Marsaglia scheme in this regard are that the permutation is less limited in scope, being effected and varied over the full cycle of the <*Y*> generator while the DES permutations are limited to within each 64-bit cipher block, and it is at least pseudo-random as determined by the <*Y*> generator while the DES permutation is static. The fact remains that, if there are as yet unknown means to exploit the one-to-one mapping of bits from their original locations in the <*X*> sequence to their positions in the composite generator’s output that are faster than a brute force attack, and thus to “break” the generator, discovery of such means must be presumed to eventually occur.

By analogy, we can view the <*X*> sequence of a McLaren-Marsaglia generator as a plaintext to be encrypted, the <*Y*> sequence as the key used to perform that encryption, and the intervening mechanism being the cipher algorithm. As such, we can analyze some of the problems of the McLaren-Marsaglia approach using cryptographic concepts.

A basic objective of cipher design is to provide a mechanism that, whenever possible, forces an attacker to perform a brute force attack on the cipher key, when trying to crack a ciphertext. The size of the key-space and the degree to which an attacker is forced to perform either random or systematic trials of possible keys in that key-space play a substantial role in determining the ultimate strength of a cipher. Thus, it is necessary to understand the effect key size has on cipher strength, as well as how and when an attacker may conclude that a ciphertext has been successfully decrypted, and the key used in its encryption found. For this, the concept of “unicity distance” is useful, even though it was originally developed as a tool for examining the comparative strength of ciphers, to determine when one could reasonably conclude a cryptanalytic attack had been successful.

Unicity distance was defined by Claude Shannon in one of the most important papers ever written on cryptography [5]. In it he laid out the theoretical foundation of secure ciphers, and proved that the only perfect cipher systems are One-time Pads (OTP) or homologues of OTP. In this paper, he defined the unicity distance of a ciphertext as the ratio of the entropy of the key to the redundancy of the underlying plaintext. The definition of unicity distance is

$$U = \frac{H(keyspace)}{D},$$

where, *U* is the unicity distance, *keyspace* the size of the key-space, *H*(*keyspace*) the entropy of a randomly selected key, and *D* the redundancy of the plaintext.

A common illustration of the meaning of these terms, and of unicity distance, uses DES as an example. In DES, the effective key length is 56 bits. (It’s specification is in terms of eight 8-bit bytes, including a leading parity bit in each byte that is otherwise ignored.) If the key-space is assumed to contain no weak keys, there are $2^{56}$ possible keys, and the entropy of that key-space is $H(k) = lg(2^{56}) = 56$. In English plaintext, there are 26 letters. Ignoring punctuation, capitalization, white-space, etc., the amount of data carried in an individual character is $lg(26) \cong 4.7$. Analysis of textual material in English reveals that there are generally only about 1.5 bits worth of real information conveyed per character, so that D ≅ 4.7 – 1.5 = 3.2.  Thus, the unicity distance for English plaintext encrypted with DES is U ≅ 56 / 3.2 = 17.5.  This means that in order to determine whether a ciphertext of an English plaintext has been correctly decrypted using a particular key, an attacker ordinarily must decipher at least 17.5 characters worth of plaintext. As DES uses a 64-bit cipher block, this means that decryption of three blocks of a ciphertext that produces a meaningful English plaintext is generally sufficient to conclude that the correct key has been found.

If the <*Y*> sequence is assumed to be random, and we treat it as a key encrypting the <*X*>, while noting that the <*X*> sequence, if also random, has a redundancy of zero, we see that the entropy of <*Y*> is equal to its length in bits, and the limit of U as D approaches zero is infinity.  We would thus expect the encryption of the <*X*> to approach the strength of OTP, or equal it; effectively, the output is random.  This is not the case when the <*X*> and <*Y*> sequences are generated by PRNGs.

At best, we can view the seeds of the two generators as the collective key for the mutual encryption of the two sequences, and the redundancy of the "encrypted" material as being non-zero but small, if the PRNGs used are "good."  If we assume the redundancy of the output bit-stream is 1 per bit, and the entropy equals the initial state (seed) of the composite generator, then the unicity distance of the generator is equal to the size of the combined seeds plus any other random material used to initialize the McLaren-Marsaglia mechanism.  While the assumption that the bit-wise redundancy of the output equals 1 is purely a guess, it is a reasonable one for many types of PRNGs.  Since PRNGs are deterministic, every bit output by a PRNG is an indicator of its internal state, and thus every bit output reveals information about the internal state of the generator.  If a PRNG is designed such that every bit of the seed influences every bit of the output stream, each output reveals information about all bits of the internal state.  Viewed as a mapping, any PRNG maps one seed to one output sequence.  It is reasonable to believe that, for a “good” PRNG, the point at which the mapping can be identified requires at least as many bits of output as are contained in the seed.  Thus, we may well expect that the best way to increase the unicity distance of a PRNG is to increase the effective size of the seed.

A standard goal in the design of block ciphers is that, given any key, a change of any single bit in the plaintext enciphered using that key should result in a 0.5 probability for each bit in the resulting ciphertext block changing as well.  The same is sought for single-bit changes in keys. This diffusion of the effect of single bit changes, in both the plaintext and the key used, is part of what gives the better block ciphers their strength.  Understanding why this is desirable is relatively simple.

Consider a cipher that does not behave like this.  Rather, a single bit change in either the key or the plaintext will result in only a single bit change in the ciphertext.  As a result, it is possible to analyze groups of ciphertexts with regard to the differences, and thereby gain at least some information about the keys and underlying plaintexts associated with ciphertexts.  Where single bit differences are found, it is reasonable to suspect that either the plaintext or the key may differ by a single bit.  Thus, if one message can be cracked, the other should easily follow.  This is the basis of differential cryptanalysis, wherein patterns of differences resulting from bit-changes in keys and plaintexts are analyzed and used to help in breaking ciphertexts.  While far more complex patterns of changes occur in many block ciphers, when the probabilities for bit-level differences in ciphertexts, relative to single bit changes in plaintexts and keys, are not uniformly very near 0.5, the potential for an effective differential analysis attack exists.

From this, it should be evident that part of the problem with the common form of the McLaren-Marsaglia scheme, when viewed as a means of enciphering the <*X*> sequence by <*Y*> based shuffling lies in the fact that the bits in <*X*> are not altered, just displaced.  The same is true of all the indexed shuffling approaches down to the bit level, though considerable strength (in the cryptographic sense) may be obtained by finer-grain decomposition of the <*X*> sequence and increased *k*. Only when we get to the bit-matching-with-shuffling do we see individual output-bit probabilities changing based upon individual bit changes in the <*X*> and <*Y*> sequences, or the initial contents.

If a single bit in the bit-wise shuffling table is changed, the effect is to insert or delete a bit from the output sequence. While this may not appear to be much of a change, this shifting of the subsequent bits has the effect of altering, on a 0.5 probability basis, the values in all bit-positions that follow the insertion or deletion. The same is true of single bit changes in the <*X*> and <*Y*> sequences. Still, it would be better if, rather than changes associated with bit-positions, more pervasive changes were induced.

## Further Modification of the McLaren-Marsaglia Mechanism

In the table-based, bit-wise approach to we can see that there is no particular reason why we must select bits for output from *V* based on bit-matching except to preserve the permutation character of McLaren-Marsaglia. The number and values of bits selected via the *bits*[][] and *shifts*[][] arrays could as easily be arranged in a random manner. Likewise, the bits inserted into *V* from the *set*[][] array can be arbitrary. There is also no reason, apart from the space required, not to view the *bits*[][], *shifts*[][] and *set*[][] arrays as slices through larger, three-dimensional arrays, to view *V* as part of a more complex internal state, and to use the contents of *V* as indexing material for the third dimension for these arrays. The contents of these three dimensional arrays can then be filled with arbitrary values selected and arranged to obfuscate the individual indices used. With such an implementation, the tabular-bit-matching form of the McLaren-Marsaglia algorithm can be viewed as a special case of this more general form. The following code sample illustrates this approach.

```
#define    OBITS_MASK              0x007f
#define    SBITS_MASK              0x0380
#define    SBITS_SHFT              7
#define    NBITS_MASK              0xfc00
#define    NBITS_SHFT              10

// the following is a place-holder as the full array is large.

unsigned short SFrame[64][16][16];

// the following are place-holders for the fill and index generators

unsigned long Xn(void);
unsigned long Yn(void);

unsigned long bitBlendOpt()
{
    unsigned long             retVal = 0;

    static unsigned long long state = 0LL;
    static unsigned long long xQ = 0L;
    static int                xBits = 0;
    static unsigned long long yQ = 0L;
    static int                yBits = 0;
    static unsigned long long oQ = 0L;
    static int                oBits = 0L;

    // some temporary variables to hold

    unsigned long             tState = 0L;    // for 6 of 64 bits, state
    unsigned long             txQ = 0L;       // for 4 bits, fill buffer
    unsigned long             tyQ = 0L;       // for 4 bits, index buffer

    unsigned long             temp;           // for SFrame entry used
    unsigned long             tout;           // bits to insert in output
    unsigned long             tshft;          // # bits to be inserted
    unsigned long             tstat;          // new state bits

    if (state == 0LL)         // initialize the state if needed.
    {
        state  = (unsigned long long) Yn()
               ^ (unsigned long long) Xn();
        state  = state << 32;
        state |= (unsigned long long) Yn()
               | (unsigned long long) Xn();
    }

    // loop until we have enough output bits

    while (oBits < 32)
    {
        if (xBits < 4)    // ensure we have enough fill bits
        {
            xQ |= ((unsigned long long) Xn()) << xBits;
            xBits += 32;
        }

        if (yBits < 4)    // ensure we have enough index bits
        {
            yQ |= ((unsigned long long) Yn()) << yBits;
            yBits += 32;
        }
        // grab 6 state bits, plus 4 fill and 4 index bits

        tState = (unsigned long) (state & 0x3fLL);
        txQ    = (unsigned long) (xQ & 0xfLL);
        tyQ    = (unsigned long) (yQ & 0xfLL);

        temp  = (unsigned long) SFrame[tState][txQ][tyQ];
        tout  = temp & OBITS_MASK;
        tshft = (temp & SBITS_MASK) >> SBITS_SHFT;
        tstat = (temp & NBITS_MASK) >> NBITS_SHFT;

        // shift output queue to make room for new bits then append them

        oQ = (oQ << tshft) | (unsigned long long) tout;
        oBits += tshft;

        // note: only 5 state bits modified, and a shift of only 3, which
        // helps propagate bit diffusion

        state = (state & 0xffffffffffffffd0LL)
              ^ (unsigned long long) ((  SFrame[tState][txQ][tyQ]
                                       & NBITS_MASK) >> NBITS_SHFT);
        state = (state << 61) | (state >> 3);

        // shift 'expended' fill & index bits off the ends of queues

        xQ = xQ >> 4;           // note: could shift/sub min(4, outbits)
        xBits -= 4;             // rather than a fixed 4 bits

        yQ = yQ >> 4;
        yBits -= 4;
    }

    retVal = (unsigned long) (oQ & 0xffffffffLL);

    oQ = oQ >> 32;
    oBits -= 32;

    return retVal;
}
```

C code for the BitBlendOpt() function

Analysis of this scheme is difficult, since, with arbitrary or random sets of three values in the *SFrame.bits*[][][],*SFrame.shifts*[][][] and *SFrame.set*[][][] arrays (which are notional in the above code, and referred to as *bits*, *shifts* and *set* arrays in the following discussion) associated with the state, <*X*> and <*Y*> indices means that there is no necessary logical or mathematical relationship between any of the indices and the values selected using them. A simpler case illustrates the issues.

Consider a *bits* array that contains four-bit values, and a *shifts* array that contains a value of 4 in all locations, so that it may essentially be ignored. If we consider the array slice-wise, based on the first index (which will correspond to *tState* value above), and require that each column and each row contain one instance of each possible four-bit value, we can see that for any *txQ* value that might be used, any four-bit value may be obtained based upon the *tyQ* value.  This means that analysis of the output stream to determine the <*X*> and <*Y*> streams, and from them the states of the associated generators, will likely be difficult.  Within the bits array there will be 1024 occurrences of each nybble value.  Requiring that, for any *txQ*/*tyQ* pair, the list of values indexed by *tState* must contain equal numbers of each possible value (thus, four instances of each of the 16 values), it can be seen that there will be 1024 instances of each value in the *bits* array.  Given an output nybble, plus a guess regarding the associate *txQ* or *tyQ* value that generated it, there are 64 different combinations of *tState* and either *txQ* or *tyQ* that could have yielded that output nybble.

If we view a slice through the *bits* array based upon a given value of *tState*, we have a row/column permutation of the following array, the contents of which are hexadecimal.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 |
| 5 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 |
| 6 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 |
| 7 | 7 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 8 | 8 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 9 | 9 | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| A | A | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| B | B | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A |
| C | C | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B |
| D | D | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C |
| E | E | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D |
| F | F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E |

A slice through the state table in BitBlendOpt()

By a row/column permutation we mean that the rows may be re-arranged in any order, and the columns likewise rearranged, and the resulting array will retain the property that each row and column contains one each of all possible 4-bit values.

If we re-label the row and column indices with letters, we can recognize this array as essentially a form of Vigenère cipher tableau. This would ordinarily be regarded as very bad, as the standard Vigenère cipher is remarkably weak: cipher keys used in classical Vigenère ciphers are relatively short sequences of characters, yielding to straight-forward cryptanalysis that exploits the key length. However, there are also similarities between this array and the *SBox* structures in many block ciphers.

The classic Vigenère cipher suffers from the standard problems of many older stream ciphers. Encrypting plaintexts with significant redundancy, using short keys that are often words or phrases and the absence of diffusion in the resulting ciphertext allow multiple forms of attack. The present case is markedly different. We can view either the <*X*> or the <*Y*> sequence as the key, and the other as the plaintext to be encrypted with it. Assuming these generators are reasonably good, with long periods, the redundancy of the "plaintext" will be low, and the key-length long. When we add the further complication of the state data to the process, resulting in use of many distinct tableaus, we may expect the result to be more difficult to attack than a classical Vigenère cipher.

If we consider only the key length, initially, treating <*Y*> as the key, the entropy of the key will be related to the size of the elements of <*Y*> and its period. Again assuming that <*Y*> is random, with 32-bit elements and a $2^{32}$ period, $H(keyspace) = 2^{37}$. Using an estimate of $D = 1$ as before, for the redundancy of our pseudo-random <*X*>, we have $U = 2^{37}$ as an upper bound for a single tableau, and would be exact for <*Y*> if it were a random sequence of $2^{32}$ 32-bit values repeated. The actual unicity distance when using pseudo-random <*X*> and <*Y*> and a single tableau would be on the order of the size of the internal states of the two PRNGs, and, with the addition of the internal state of *BitBlendOpt*, would likely include some number of bits worth of that state to account for the multiple tableaus of the *SFrame*, which would, as per Kerckhoffs' principle, be presumed known by an attacker.

## Deterministic Aperiodicity

As one of the intrinsic problems with most PRNGs is that they are both deterministic and periodic, and the available means of generating aperiodic sequences are limited, it is worth considering the applicability of the revised McLaren-Marsaglia generators to the problem of providing aperiodic generators.  The standard computational source for aperiodic sequences is found in the realm of irrational numbers, while a more recent area of exploration has involved the geometry of quasi-crystals.  Both have their drawbacks.

Quasi-crystals are intriguing in that they display the characteristics of crystals without displaying the precise structure and symmetries of true crystals.  The inter-atomic distances within the structure vary in non-repeating ways that resemble in some ways the behavior of irrational numbers.  While we regard the issues surrounding use of computationally derived values based on quasi-crystal structures as beyond the necessary scope of this thesis, we note that the problems are similar to the use of irrational numbers: high-precision mathematics are required, and the process is computationally intensive.  Irrational numbers, and the subset of irrational numbers comprised of transcendental numbers, are more readily understood, and sufficient to the following discussion.

It is possible to generate a reasonably apparently random sequence by selecting a suitable irrational number, and applying a simple algebraic function to it that preserves the irrationality of the result.  Historically, the problem with this has been the computational resources required to produce any significant number of digits, and the easy with which errors can arise.  The history of the computation of π serves as an excellent example of these problems. While the number of digits that have been computed has passed the billion digit mark, this was a decidedly non-trivial feat. Further, until recently, it was generally necessary to compute all prior digits of such numbers in order to compute any specific digit. For high-volume applications, such expensive processes are not suitable.

Some developments in the means of computing at least some irrational numbers have significantly improved the situation with regard to the computational cost. Work published in 1997 by Bailey, Borwein and Plouffe [120] showed that a class of irrational numbers can be computed in polynomially logarithmic space and polynomial time. Members of this class are of the following form.

$$v = \sum_{k=1}^{\infty} \frac{p(k)}{b^{ck} q(k)}$$

where *p* and *q* are polynomials with integer coefficients, *c* a positive integer, and *b* the base.

Intriguingly, one of the members of this class is π, with the following form.

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left( \frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right),$$

where 16 is the base.

Even more interesting is that with this formulation it is possible to compute individual digits of , and other members of this class, without computing all prior, higher-order digits. The computation requirements for the *d*-th digit, base *b* for members of this class are given by Bailey, et al. [120], as follows.

$$space = \log^{O(1)}(d).$$

$$time = O(d \log^{O(1)}(d)).$$

While this represents a significant improvement over previous techniques, in terms of time and space, for very high values of *d*, the time and space required are still significant. Thus, while the approach must be regarded as far more interesting and useful for purposes of generating aperiodic sequences, means of efficiently using such sequences are desirable.

It can be seen that if we were to replace the <*X*> sequence in a standard McLaren-Marsaglia generator with successive segments of an irrational number, the resulting generator would produce an aperiodic output stream. This would not obviate the problem posed by Retter's attack, though. It would remain effective, since, given the irrational number used, an attacker would need only select *d* and proceed. Likewise, replacing the <*Y*> generator with an irrational number digit stream changes the output sequence, but not the problem. The <*X*> can still be solved for independently from <*Y*>, though the time and space requirements for the attack scale with the algorithm used to calculate the irrational number if the <*X*> sequence is based upon an irrational. In fact, there is no obvious benefit gained from selecting and using a second, unrelated, irrational number, so that both <*X*> and <*Y*> are aperiodic. All the benefits of aperiodicity to be had are obtained with use of just one irrational number in the context of one of the generators.

The revised forms of the McLaren-Marsaglia algorithm can be used, as well as the straight-forward bit-matching algorithm, providing either the <*X*> or <*Y*> generator with aperiodicity. In each case, the output becomes aperiodic, though the efficiency of the use of that irrational numbers digit sequence varies. With the straight-forward bit-matching scheme, only half the bits would be used, and so for very long sequences generated, the computational cost increases faster than if every bit is used. Thus, it appears that, unless the computational cost can be further reduced, or only short sequences need be generated, this is not a very practical approach.

The *BitMatchShuffle()* algorithm makes more efficient use of the <*X*> sequence than does either of the *BitMatchStream()* algorithms, but not of the <*Y*> sequence. Thus, *BitMatchShuffle()* is more appropriate for use in generating aperiodic sequences using an irrational sequence generating algorithm, but only as the <*X*> generator. The *BitSpigot()* form of this generator is the more efficient, and thus is preferable to *BitMatchShuffle(),* but with the same limitation. *BitBlendOpt()* can also yield good results, but since the <*X*> and <*Y*> generators are treated in an essentially identical manner, it makes no difference which is implemented via an irrational number technique, though, again, using that technique for both likely adds not improvement in terms of aperiodicity, just computational cost.

Another method of injecting aperiodicity via irrational number digit sequences is practical with the *BitBlendOpt()* scheme. If we added a mechanism for inserting some number of bits from the irrational digit stream into either the <*X*> or <*Y*> sequence on a periodic basis, a number of benefits accrue.

```
#define    OBITS_MASK             0x007f
#define    SBITS_MASK             0x0380
#define    SBITS_SHFT             7
#define    NBITS_MASK             0xfc00
#define    NBITS_SHFT             10
#define    INTERVAL               4

// the following is a place-holder as the full array is large.

unsigned short SFrame[64][16][16];

// the following are place-holders for the fill and index generators

unsigned long Xn(void);
unsigned long Yn(void);
unsigned char Ir(void);       // returns single bit from irrational seq.

unsigned long bitBlendAperiodic()
{
    unsigned long             retVal = 0;

    static unsigned long long state = 0LL;
    static unsigned long long xQ = 0L;
    static int                xBits = 0;
    static unsigned long long yQ = 0L;
    static int                yBits = 0;
    static unsigned long long oQ = 0L;
    static int                oBits = 0L;
    static int                intr = INTERVAL;
```

```
// some temporary variables to hold

unsigned long             tState = 0L;    // for 6 bits of state
unsigned long             txQ = 0L;       // for 4 bits of X buffer
unsigned long             tyQ = 0L;       // for 4 bits of Y buffer

unsigned long             temp;           // for SFrame entry used
unsigned long             tout;           // bits to insert in output
unsigned long             tshft;          // num bits to be inserted
unsigned long             tstat;          // new state bits

if (state == 0LL)         // initialize the state if needed.
{
    state  = (unsigned long long) Yn()
           ^ (unsigned long long) Xn();
    state  = state << 32;
    state |= (unsigned long long) Yn()
           | (unsigned long long) Xn();
}

// loop until we have enough output bits

while (oBits < 32)
{
    if (xBits < 4)    // ensure we have enough fill bits
    {
        xQ |= ((unsigned long long) Xn()) << xBits;
        xBits += 32;
    }

    if (yBits < 4)    // ensure we have enough index bits
    {
        yQ |= ((unsigned long long) Yn()) << yBits;
        yBits += 32;
    }

    if (intr == 0)
    {
        xQ |= (((unsigned long long) Ir()) << xBits);
        xBits++;
        intr = INTERVAL;
    }
    else intr--;

    // grab 6 state bits, plus 4 fill and 4 index bits

    tState = (unsigned long) (state & 0x3fLL);
    txQ    = (unsigned long) (xQ & 0xfLL);
    tyQ    = (unsigned long) (yQ & 0xfLL);

    temp  = (unsigned long) SFrame[tState][txQ][tyQ];
    tout  = temp & OBITS_MASK;
    tshft = (temp & SBITS_MASK) >> SBITS_SHFT;
    tstat = (temp & NBITS_MASK) >> NBITS_SHFT;

    // shift output queue to make room for new bits then append them

    oQ = (oQ << tshft) | (unsigned long long) tout;
    oBits += tshft;

    // note: only 5 state bits modified, and a shift of only 3, which
    // helps propagate bit diffusion
```

```
        state = (state & 0xfffffffffffffd0LL)
              ^ (unsigned long long) ((  SFrame[tState][txQ][tyQ]
                                       & NBITS_MASK) >> NBITS_SHFT);
        state = (state << 61) | (state >> 3);

        // shift 'expended' fill & index bits off the end of their queues

        xQ = xQ >> 4;          // note: could shift/sub min(4, outbits)
        xBits -= 4;            // rather than a fixed 4 bits

        yQ = yQ >> 4;
        yBits -= 4;
    }

    retVal = (unsigned long) (oQ & 0xffffffffLL);

    oQ = oQ >> 32;
    oBits -= 32;

    return retVal;
}
```

C code for bitBlendAperiodic()

Insertion of the extra bit into the <*X*> sequence will alter the alignment of the <*X*> and <*Y*> sequences relative to each other, and result in bit-aligned operation of the overall generator, rather than word-aligned operation. The two component generators will have their outputs word-aligned once per 32 bit-insertions, but the overall alignment will shift continually throughout the operation of the composite generator. Also, the inserted bits will have continuing effects upon the bit-values in *state*, and the period of the <*X*> sequence is artificially extended by the inserted bits. Finally, the output sequence that results is aperiodic.

Such an insertion technique can be applied to some of the other methods discussed. In the bit-matching schemes, buffering of the input streams can accommodate the inserted bits, but knowledge of the generators used, and a successful guess regarding the seeds used for the conventional generators, leaves less analysis to be performed than in *BitBlendAperiodic()*, since the produced sequence is largely a matter of single bit shifts of one generating sequence relative to the other, and the injected bits from the irrational sequence are immediately determined from the output sequence relative to the un-injected sequence component generator sequence.

The exceptions to simple bit-injection are the techniques that involve only dicing the <*X*> sequence into smaller multi-bit blocks and placing them into a shuffling array organized as a multiple of *R* entries. In those cases, the injection is simpler if done on the basis of identically sized blocks, rather than bit-wise. However, the lack of diffusion obtained via the *state* variable of the *BitBlendAperiodic()* technique means that the output stream remains largely a periodic sequence of raw outputs from the periodic PRNG used as the <*X*> generator, with irregularly inserted aperiodic components. It is difficult to regard this as a robust solution to the problem of aperiodicity.

It should be noted that they basic structure of the McLaren-Marsaglia variants need not be altered to incorporate such aperiodicity. The insertion of the irrational bit-stream material can be done within the component <*X*> or <*Y*> generators, since the fundamental structure of McLaren-Marsaglia and its variants is equally adaptable to any suitable component generators. If handled this way, a straight-forward XOR of a standard PRNG's output with the irrational digit stream might be considered, too. The result would, for all variants, be aperiodic, and tend to conceal the outputs from both the standard PRNG and the irrational bit-stream, just with any combiner function, but still suffers from the full cost of generating the irrational bit-stream.

# CHAPTER IV

# KEY AGREEMENT

Having explored some of the possible variants of the McLaren-Marsaglia generator scheme, we return to one of the initial questions asked: Can we replicate at least some of the hypothesized strengths of the Bennett-Brassard key agreement process without relying on quantum phenomena, or even on mathematically “hard” problems? We start with the assumption that the two parties share some information that is secret, held only by the two correspondents, and that at least one correspondent has a true, hardware-based random number generator. We will then describe a process, based in large part on some of the McLaren-Marsaglia variants described in the previous chapter, by which the correspondents can agree upon a set of bits, and discuss the associated security/confidentiality issues associated with this scheme.

## Themes and Goals

As discussed in Chapter I, the Bennett-Brassard quantum mechanics-based key agreement protocol is a major inspiration for the present effort, as is the problem posed by the Diffie-Hellman conjecture. The Diffie-Hellman conjecture is the assumption present in many proofs of security that the mathematical problem a particular algorithm is based on is difficult to solve, and cannot be reliably solved in a time-frame that would imply unacceptable risk in using the algorithm or protocol. Typical examples are the assumption that it is prohibitively difficult to factor very large integers into their prime factors, and that the discrete logarithm of a very large number in an arbitrary base is very difficult to calculate.

With regard to the Bennett-Brassard protocol (see Chapter I for a detailed description), problems include the difficulty and cost of providing a dedicated optical fiber connection between two remote points, let alone a network of such connections, the ability to generate single photons polarized to the desired orientations and to detect individual photons at the receiver's end of a fiber, loss of photons over long distances due to defects and impurities in the fiber, etc. Still, however theoretical parts of the scheme remain, a protocol that allows two parties to securely agree upon a string of random bits suitable for use as One-Time Pad (OTP) key material is both brilliant in conception and inspirational, particularly since the protocol places no reliance upon any form of Diffie-Hellman conjecture about a mathematical problem.

The inspiration derived from the Bennett-Brassard protocol is easily understood. Personal experience tends to demonstrate that for any problem, if there is at least one way to solve the problem, then there are likely others. Thus, the question is raised whether other secure means of agreeing upon a set of bits, without relying on a mathematical problem that is presumed to be hard or quantum phenomena, can be developed.

Also as discussed in Chapter I, "secure" is a relative term. While in a theoretical sense the Bennett-Brassard protocol is perfectly secure, when fully implemented, its practical implementation may be a different matter, particularly as the problem of reliably generating, transmitting and detecting single photons remains a serious challenge. One-Time Pads (OTP) are provably perfectly secure, yet of sufficient difficulty in practical use to warrant the continued use of systems that are less than perfect. Even systems designed by highly knowledgeable and experienced professionals can contain flaws that take years of analysis to identify and exploit. Thus, as a first effort, it is too much to expect a flawless system. Therefore, for the present effort, perfection will not be expected, but a very low probability of successfully guessing the state of the mechanisms, coupled with a degree of complexity in analysis that presents no obvious easily exploitable internal flaws, may be regarded as success.

The examination of variations based loosely on the McLaren-Marsaglia pseudo-random number generator (PRNG) served as a means to identify components that offer both flexibility and extendibility.  Two forms of one of the examined variations are used in the proposed protocols discussed and implemented here.

The bit_blend scheme, wherein the McLaren-Marsaglia mixing array is reinterpreted as a state vector for selection of an arbitrary mixing function for combining the inputs from the two component generators of the $<X_n>$ and $<Y_n>$ bit-sequences, is used in two distinct ways.  In the first, used in the *A*, *B* and *C* generators of the protocols described below, conventional PRNGs are used to provide the $<X_n>$ and $<Y_n>$ bit-sequences.  In the second way, the *D* generator discussed below, the $<X_n>$ and $<Y_n>$ sequences are derived from the random bit-string that is the sole communication between the two parties to the key agreement within the presented protocol.  The derivation of the bit-strings finally used as inputs to *D* is performed using the outputs of the *A*, *B* and *C* generators, in a manner intended to create a combinatorial problem of sufficient complexity to make cryptanalysis, in an effort to determine the initial or ending state of the process, difficult enough to warrant the appellation “secure” for the overall process.

Two additional processes are used in the overall process.  First, where the Bennett-Brassard protocol discards roughly half the initially transmitted random bits as a consequence of the guessing process in its first phase, here we apply the bit-matching process discussed in the previous chapter as a means of increasing the combinatorial complexity of cryptanalyzing the proposed protocols.  Second, we utilize a portion of the random bits transmitted to inject aperiodicity into the scheme, in an effort to preclude use of such tools as the Berlekamp-Massey algorithm to produce linear feedback shift register (LFSR) analogs to components of the scheme.

## The Basic Scheme: Walk-through

To start the process, the two parties to the key-agreement process, Alice and Bob, have identical sets of PRNGs, *A*, *B*, *C* and *D* (the mechanisms being public knowledge), and identical initial states for these generators and the overall mechanism (which are presumed to be a secret jointly held by Alice and Bob, and agreed upon by a mechanism outside the scope of this discussion). The structure of the component generators is of the *BitBlendOpt()* type previously discussed. Alice also has a true random number generator (RNG).

Alice and Bob decide they need a shared, secure, random bit-string to use in conducting a confidential exchange. They proceed as follows.

Alice generates a 32 bit long string of random bits, *r*, using her RNG. Using her saved copy of the initial state of the PRNG *A*, she generates a 32 bit long value using *A* and calls it *a*, then saves the new state of *A*. While keeping a copy of *r* for later, she XOR's the *a* and *r* bit strings together to produce bit string *axr*. Alice sends a copy of *axr* to Bob, and saves the new state of *A*. Notice that *a* acts as a Vernam cipher key with respect to *r*, or inversely that *r* acts as an OTP cipher key for *a*.

When Bob receives *axr* from Alice, he initializes his copy of the *A* with the identical copy of the initial state he shared with Alice, then generates his own copy of the *a*, and saves the new state of *A* for later, just as Alice did. Bob then XOR's his copy of *a* with the copy of *axr* he received from Alice to obtain a copy of *r*.

As Alice and Bob now both have identical copies of string *r*, they both take the lowest-order 15 bits of r and insert them into the <*X*> bit buffer of their respective initial states of the PRNG *B*.

Thus, the modified states of the two copies of *B* held by Alice and Bob are still identical. They then take the next lowest-order 14 bits of *r* and insert these bits into the <*X*> bit buffer of their copies of PRNG *C*. Again, as the modifications are identical, the resulting states are identical. Notice that both B and C now have had true random data injected into their states.

Using her RNG, Alice now generates a random 1024 bit long bit string and calls it *rp*, and saves a copy for later. Using her now modified initial state of generator *B*, she generates the 1024 bit long bit string *b*. She XOR's bit string *b* with bit string *rp* to produce bit string *bxrp*, which she sends to Bob. Again, we have that the string pair acts as cipher keys for each other. The string *b* is a Vernam cipher key with respect to *rp*, while *rp* is an OTP cipher key with respect to *b*.

Upon receiving *bxrp* from Alice, Bob uses his modified initial state of generator *B* to generate his own copy of *b*. Bob XOR's this copy of *b* with his copy of *bxrp* to obtain a copy of *rp*.

As Alice and Bob now share exact copies of the initial states of generators *C* (as modified above) and *D*, and have exact copies of *r* and *rp*, they both proceed as follows.

Both Alice and Bob initialize their copies of generator *C* with their shared, modified initial state for that generator, and generate a 1024 bit long bit string which they label *c*, then save the new state of *C* for later communications. They both create an empty bit string *m*, then perform a bit-by-bit comparison of *c* with *rp*. When corresponding bits in *c* and *rp* are equal, they append that bit value to bit string *m*, repeating the process for all bits in *c* and *rp*. At the end of this process, they will have identical copies of bit string *m*. They then truncate *m* to a multiple of eight bits in length, yielding a set of full bytes. Next, they take the highest-order three bits from *r* and use these as an integer value. They add four to this value, and count backwards that many bytes from the end of *m*. If there are still at least four bytes to the "right" of this position (they haven't backed up past the start of *m*), they remove four bytes from *m* starting at this position, and insert these bits to the <*X*> bit buffer of their saved new saved states of *A*, and each saves this again revised state for the next round. Notice again that random material has been injected into the state of *A*, just as such material was injected into the states of *B* and *C* earlier.

Both Alice and Bob will now use their identical copies of *m* as the source for the <*X*> and <*Y*> input sequences to the *D* generator, starting at index 0 for the <*X*> input and the other end for the <*Y*> input, stopping when the indices meet in the middle. This is truly random material, and thus the output from *D* will be random. They load their stored, identical initial states of *D* and proceed as described above, appending the output bytes to an initially empty bit string *k*, and saving the ending state of *D* for later communications.

Alice and Bob now have identical new states for *A*, *B*, *C* and *D*, and identical byte strings *k*. Thus, they are free to use the agreed-upon bit string *k* as they may choose, while being able to repeat the process (using freshly generated random bit strings *r*', *r*'', …, and *rp*', *rp*'', …) to generate additional random bit strings *k*', *k*'', ..., as needed. The new states of *A*, *B*, *C* and *D*, saved at the end of each invocation of the protocol, are not purely the products of deterministic processes upon a finite state, since random material obtained directly or derived from the *r* and *rp* bit strings was either injected into the input streams (in the cases of *A*, *B* and *C*, thereby randomly modifying their states during invocation of these generators) or provide the entirety of the input streams (as is the case with *D*, resulting in corresponding random alterations to the state of *D*). Thus, so long as the successive *r* and *rp* bit strings fed into the protocol are random, and the modified states of the mechanism's components are correctly saved between invocations, the resulting sequence of states will be aperiodic, as will the sequences generated by *A*, *B* and *C*. The output from *D* is, again, effectively random, given the random material fed to it as both input streams.

As discussed further below, the preprocessing of *r* and *rp* to generate *axr* and *bxrp* by Alice is not strictly necessary. An alternate version of the protocol eliminates this preprocessing by having

Alice generate the *axr* and *bxrp* random bit strings directly, and sending these to Bob, whereupon both Alice treats them in the same manner as described above by Bob.

## The Basic Protocol A: Formal Description

Let us assume that the correspondents, Alice and Bob, have agreed upon three PRNG algorithms of the *BitBlend()* type (designated *A*, *B* and *C*) and their initial seeds, are kept secret. They have also agreed upon a stream-based bit-matching algorithm *M*, and upon an algorithm based on *BitBlendOpt()*, referred to as *BitBlendRan()* (which will be described in more detail later) and designated *D*, with its initial state also secret. Alice also has a non-deterministic RNG, designated *R*. The exchange and agreement process is as follows in Protocol A.

**Protocol A**

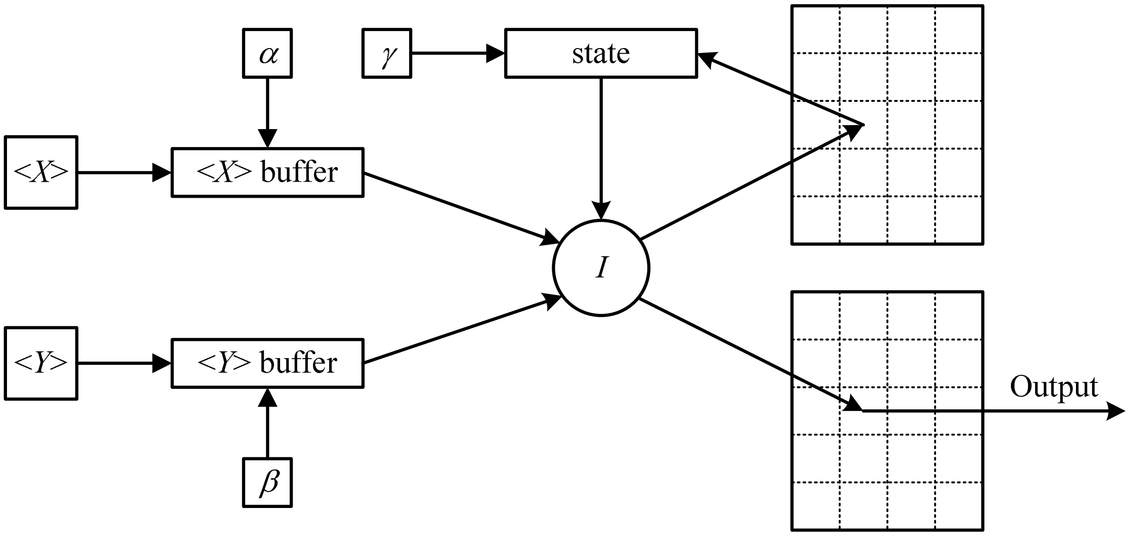
1. Alice generates a block of 32 bits (designated *r*) and a block of 1024 bits (designated *rp*) using *R*, her non-deterministic random number generator.
2. Alice pumps PRNG *A* for a 32-bit pseudo-random value *a*, and XORs this with *r*, generated in step 1 to produce the 32-bit block *axr*, then sends *axr* to Bob.
3. Alice inserts the lowest-order 15 bits from *r* (bits 17 through 31) into the *xQ* buffer of *B*.
4. Alice pumps *B* for a 1024-bit pseudo-random bit string *b*, XORs this with *rp* to produce the 1024-bit string *bxrp*, and sends this to Bob.
5. When Bob receives *axr*, he pumps his copy of *A* to obtain *a*, then XORs this with *axr* to obtain the bit block *r*.
6. Bob inserts the lowest-order 15 bits from *r* into the *xQ* buffer of his copy of *B*, then pumps his copy of *B* to produce his own copy of the 1024-bit string *b*.
7. When Bob receives *bxrp*, he XORs this with *b* to obtain *rp*.
8. Both Alice and Bob insert bits 3 through 16 (14 bits) of *r* into the *xQ* buffer of their respective copies of *C*, and pump these to obtain matching 1024-bit strings *c*.

9. Both Alice and Bob use their copies of *M* to do a bit-match selection of bits using *c* and *rp*, producing the bit-string *m*, which they truncate to a length of $l = \lfloor m/8 \rfloor$ bytes. (Note that this may be an odd number of bytes.)
10. If $l \geq 12$, Alice and Bob use bits 0 to 2 of *r* (as integer *o*) to select four bytes from *m* (viewed as a byte array) starting at $m[l - (o + 4)]$, remove them from *m* and insert them into the *xQ* buffers of their respective copies of *A*, else no action is taken.
11. Both Alice and Bob split their copies of *m* in half, and using the first half ($\lfloor l/2 \rfloor$ bytes) into *D* as the <*X*> bit-sequence for *D*, with nybbles flipped in order within bytes, and the second half ($\lceil l/2 \rceil$ bytes) in reverse nibble order as the <*Y*> bit-sequence for *D*. *D* is pumped until <*X*> is exhausted to yield the bit-block *k*, a block of agreed bits.
12. Alice and Bob repeat steps 1 through 11 until they have as many bits of agreed material as required.

### Observations Regarding Protocol A

While the walk-through of a prior section may be sufficient to understand this process, it is a complicated process, with many components that are not themselves simple. Thus some additional explanations of the various components used, and their interactions, may help, while more detailed analysis will follow in Chapter V.

The composite generators *A*, *B* and *C* were developed in Chapter IV specifically to provide means by which aperiodicity could be injected into what might otherwise have been purely periodic components of the scheme. A diagram, in generalized form, of these generators may be of use here, in order to clarify how the aperiodicity is injected via these three generators, and such a diagram is presented now.



Generalized structure of the aperiodic generators used

Ignoring for the moment the components *α*, *β* and *γ*, we observe that two component generators, <*X*> and <*Y*>, along with the mechanism state, provide input to an indexing function *I*, which selects entries in two tables. One table provides the output from an invocation of the mechanism, while the other is used to modify the state. If both <*X*> and <*Y*> are periodic, deterministic PRNGs, the overall mechanism will be periodic. From the discussion in prior chapters, we understand that this may be regarded as a weakness, since various tools, such as the Berlekamp-Massey algorithm, allow an attacker to eventually generate an LFSR that exactly replicates the output of the mechanism, given a sufficient sample of outputs. (We recognize, though, both that this LFSR may be extraordinarily long, and that we must anticipate use of far more advanced and powerful tools that would produce more useful results more quickly than Berlekamp-Massey.)

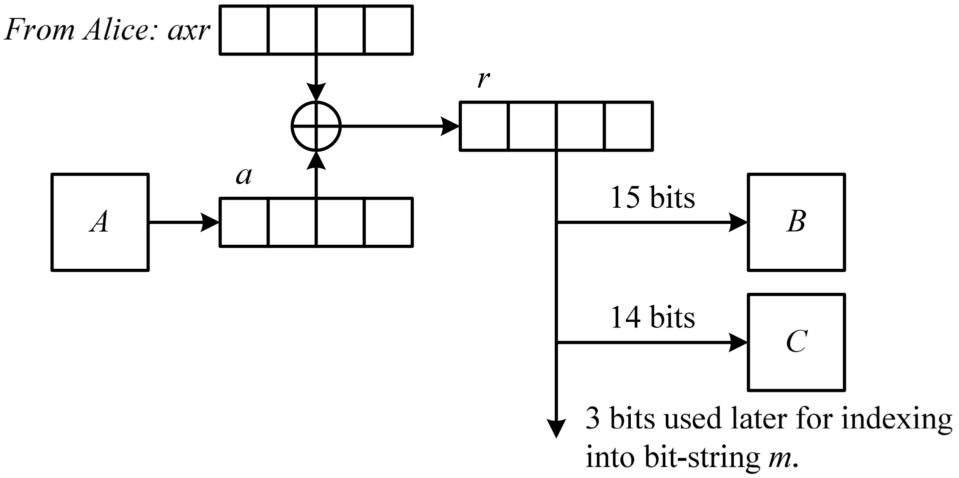
Due to the construction of these generators, there are three points at which we can easily inject additional material into the mechanism. The three points where this may be accomplished are indicated by the boxes labeled *α*, *β* and *γ*. In the cases of *α* and *β*, simply inserting bits into the buffers for the component PRNGs <*X*> and <*Y*> will result in overall aperiodicity for the mechanism, though we are also free to apply any kind of mixing function to the process of injection. However, it is simple, and quite effective in achieving aperiodicity, to simply inject bits at either $\alpha$ or $\beta$ on a regular basis.

While $\gamma$ indicates that we may also inject material into the *state* component, the means of injection is not into a stream of bits. Instead we must replace bits in *state*, or perform some form of hashing of the injected material with the state, in order to obtain the desired effect. As the goal is to simply inject aperiodicity, any one of the three locations has its merits, but we need use only one. The one selected in the discussions in Chapter III was at the point labeled $\alpha$, and merely insert the injected material into the bit-stream.

Selection of this point (into the <*X*> bit-stream at α) and this method for injecting material into the mechanism provides one additional useful feature to the mechanism. The alignment of bits between the <*X*> and <*Y*> inputs will be shifted relative to each other each time material is injected into (or we might say "appended to") the buffer holding outputs from <*X*>. While we will not discuss the benefits of this in detail, we note that it allows manipulation of the relative periods of <*X*> and <*Y*> with respect to each other.

The mechanism as described is used to implement the PRNGs *A*, *B* and *C* of the protocol. A further change is introduced in PRNG *D* of the protocol. In that generator, we dispense completely with the <*X*> and <*Y*> component PRNGs, and use random material derived from the bit-string *r* as inputs. Since this material is random, there is no need to inject any aperiodic material at $\alpha$, $\beta$ or $\gamma$, and therefore none is injected.

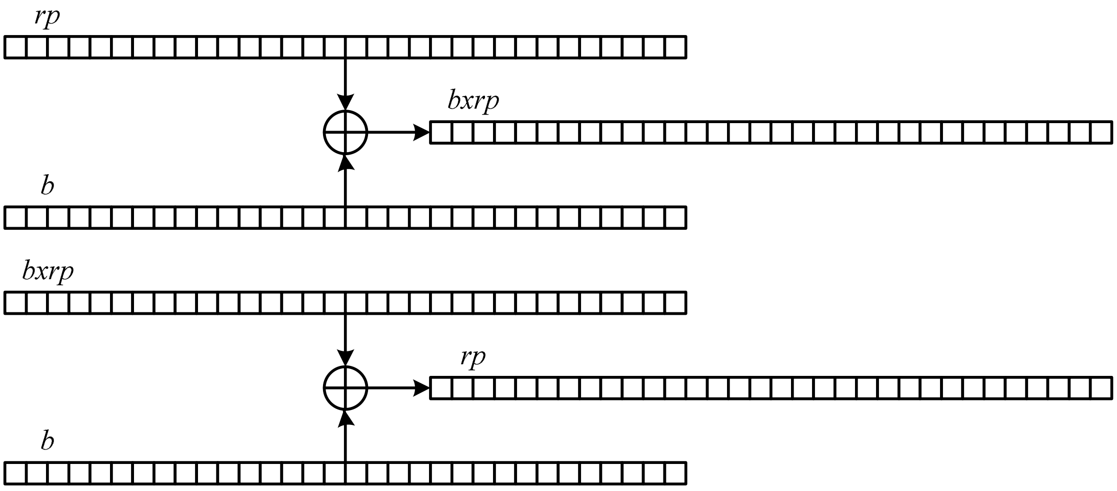
Returning now to the protocol itself, the random bit-string *r* is used to inject aperiodicity into the *B* and *C* PRNGs, and indirectly into the *A* PRNG, all three of which conform to the structure described above. But, as we wish to maintain good cryptographic strength, these bits cannot be communicated as plaintext. Thus, *A* is used to generate a Vernam cipher key *a*, which encrypts *r* as the ciphertext *axr*. And, as previously noted, we can also regard *a* as being encrypted by the random bit string *r*. Three bits of *r* are used to select 32 bits from *m* (provided *m* is long enough) to be used to inject aperiodicity into *A* in a subsequent iteration of the cycle. We can visualize this via the following diagram.



Injection of material from *axr* into PRNGs *B* and *C*.

In the above diagram, *a*, *r* and *axr* are shown as being composed of 4 bytes each, but we could as easily scale these string lengths to any desired length, to either scale up or scale down the mechanism, allocating bits for injection and indexing as required by the modified scale. The point is that *B* and *C* are made aperiodic by these injections, regardless of scale, since *r* is a random bit string, and thus aperiodic.

The processing of bit-strings *b* and *rp* by Alice to generate the bit-string *bxrp* is straight-forward, as is the processing of *b* and *bxrp* by Bob to extract the *rp* Alice generated using her non-deterministic RNG *R*. The second operation is merely undoes the first, so that both Alice and Bob have the same bit-string *rp*, with which to continue the process. This is illustrated by the following diagram.

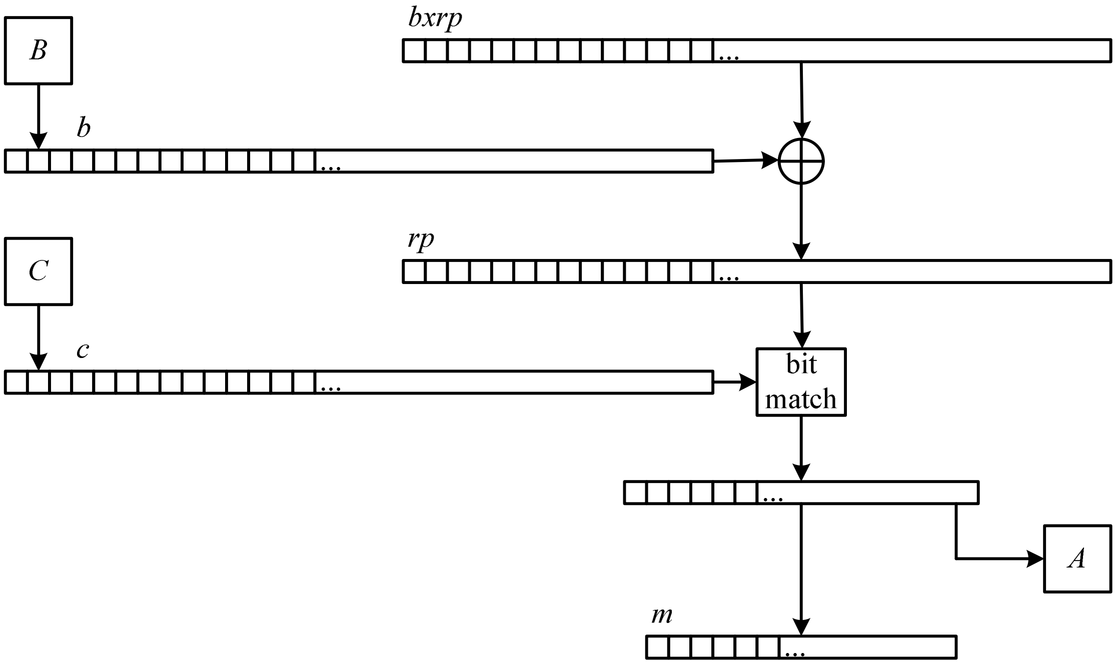


The relationship between *b, rp* and *bxrp*.

The above diagram is trivial, but should be recognized as a standard Vernam cipher. It is useful to recognize also that the trio of bit-strings, *b*, *rp* and *bxrp*, are such that an XOR of any two produces the third. The same is true of the bits-strings *a*, *r* and *axr*. These facts are the basis of the adaptation of Protocol A into Protocol B.

As will be discussed in the following section regarding Protocol B, the whole objective of Alice sending the bit-string *bxrp* to Bob is so they can both end up with matching bit-strings *rp*, which they can then further process to eventually agree upon bit-string *k*. Thus, it doesn't really matter whether Alice generates *rp* or *bxrp* using her hardware RNG. If she skips the pre-processing of *rp* to obtain *bxrp*, and instead generates a random *bxrp*, then follows Bob's steps in the procedure to generate *rp*, there is no practical difference in the result, thought he actual agreed bits will differ. The agreed bit-string will still be random and still agreed upon by both Alice and Bob. By the same reasoning, whether Alice generates a random bit-string *r*, or a random bit-string *axr* makes no difference in the result, so long as she produces the same bit-string *r* that Bob will generate from *axr*.

The further processing of the bits in bit-string *rp* gets more complicated to visualize. The bit-matching process used in the protocol discards some number of bits from *rp* to obtain bit-string *m*. Assuming PRNG *C* produces an essentially random bit-string *c*, the process will, on average, discard half the bits in *rp*. But, that is only an average. Since it is more convenient to deal with groups of bits, such as bytes (8-bit octets) or 32-bit quadlets (4 bytes), some number of bits, from 0 to 7, with be discarded from *m*, so as to yield a string of bytes. While we could trim *m* to an even multiple of 16 bits, at this point, planning as we are to use what is left byte-wise from each end, it is useful to remember that instead processing from the ends and discarding a possible "odd" middle byte adds a small factor of uncertainty to the process, and uncertainty is what we want to inflict upon any attacker. The following diagram illustrates the processing of *bxrp* to obtain bit-string *m*.



Processing of *bxrp* to obtain *m*.

As may be seen in the above diagram, the processing of the bits in *bxrp* uses the outputs from PRNGs *B* (bit-string *b*)and *C* (bit-string c) directly. The states of those two generators were altered as a result of the processing of bit-string *axr*, in the manner previously describe, by injection of the bits from bit-string/block *a*, as previously shown. Also, it can be seen that bits are extracted following the bit-matching process for use in altering the state of PRNG *A*. None of the bits so used are used for any other purpose.

As the bits extracted from *rp* are random (and therefore aperiodic), and selected after the bit-matching process applied using *M*, and match the amount of material in *A*'s <*X*> sequence required to generate the next *a* block, successive *a* blocks are an aperiodic sequence, and are in fact random. The *r* sequence is thus encrypted with a Vernam cipher that possesses at least some of the characteristics of OTP, as bit-string *a* has unbounded linear complexity. While there is a very small probability that *m* will not contain sufficient material to re-seed *A*, the probability is believed sufficiently small, and sufficiently difficult to detect in the routine operation of the scheme, as to be secure, despite the fact that it means that the *axr* encryption is not a true homologue of OTP. Part of the confidence in this lies in the fact that the distributions of probabilities for all possible lengths of *m*, given that *b* and *rp* are 1024 bits in length, means that there will likely be many thousands of iterations of the cycle before the backup PRNG *A* will be invoked. Outside the occurrence of that event, the *axr* encryption is very OTP-like, as it involves a random key-stream.

Assuming random behavior in the bit string *c*, the probabilities of the various possible lengths of *m* in bits is found via the binomial distribution. On average, *m* will contain 512 bits, ignoring truncation to byte bounds, and the probability that there will be 96 bits or fewer is on the order of $6.14 \times 10^{-175}$.

While the *B* PRNG/generator is not random, it is aperiodic due to the injection of the bits from the random bit string *r*, at the start of each cycle, and thus has an unbounded linear complexity. Further, the state of *B* at the point before generating *b*, but after insertion of bits from *r*, is effectively a randomly selected state out of $2^{15}$ possible states *B* could have been initialized in. Still the encryption of *rp* must be regarded as inadequate for its immediate use as the agreed bit sequence, at this point, since direct use of these bits would allow eventual cryptanalysis of the *b* bit string. (We choose to assume the worst: that, if we applied no additional layers of protection to the process, an attacker would eventually obtain samples of matching *r*, *rp* and *k* bit strings, a match between plaintext (*rp*) and ciphertext (*bxrp*) would reveal the Vernam cipher key (*b*).) Therefore, we apply the additional major steps of bit-matching to discard roughly half of the bits in *rp* (using the *C* generator and *M* bit-matcher to obtain *m*), extracting (in the vast majority of cases) 32 bits from the *m* sequence for use in the *A* generator (the bits extracted from *m* are not used further in deriving the agreed bit string *k* in the present cycle), and finally performing a bit-blending operation on the bits in *m*, using *D*, to arrive at an agreed bit-string *k*.

The bit-matching process, using the bit-string *c* that PRNG *C* generates, discards half of the bits in *rp*, on average. Unfortunately, the bit-matching process means that there is a high correlation between the resulting bits (*m*) and the output of generator *C*. All the remaining bits in *m* match bits in the *c* bit-sequence, largely in order. But, several factors obscure this fact from an attacker.

First, the attacker does not see the *rp* bit string in unencrypted, unprocessed form, since it is transmitted encrypted by bit string *b*. As previously observed, bit string *b* is, until the bit string *k* produced using it is known to the attacker, effectively encrypted with a OTP (since *rp* is random), even as *rp* is in turn encrypted by the Vernam cipher key *b*. This is believed to severely limit an attacker's ability to cryptanalyze it by forward analysis, potentially to brute force methods only, as per Shannon [5], one guess is as good as another, in the absence of corroborating evidence.

We can also view the situation with regard to bit-string *c* inversely.  Bits in *c* are selected at random by the bits in *rp*, or bits in string *rp* with random values are selected pseudo-randomly by bits in string *c*.  Further, as shown in the previous chapter, the number of bit-sequence pairs that can produce the same matched bits displays factorial growth as the number of bits subjected to the matching process increases.

Another complicating factor is the bits from bit string *r* injected into the *xQ* buffer of *C* (holding output from *C*'s <*X*> generator).  As these bits are random, they introduce aperiodicity into *C*, with the result that *C*, like *B*, has unbounded linear complexity.

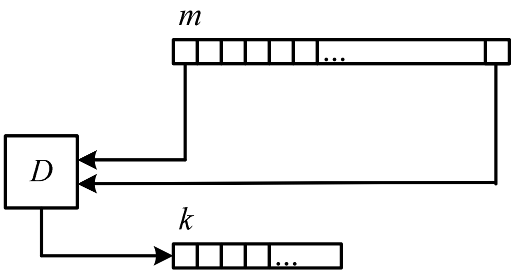
Finally, there are the matched bits either discarded as insufficient to construct a full byte or extracted for use in *A*.  If the number of matching bits between strings *c* and *rp* are not congruent to 0 mod 8, |*m*| mod 8 bits will be discarded at the end of the matching process as not being part of a complete byte.  If, after the extra bits at the end of *m* have been discarded and the re-seeding bits to be used with *A* have been removed, the length of the modified string *m* is an odd number of bytes long, a further eight bits will be discarded from the middle of the *m* string.  These are never seen outside the operation of the protocol, so that an attacker will not know whether any were thus discarded, nor are their precise origins within bit string *rp*.

Extraction of the bits to be injected into the *A* generator is performed using the highest order bits in *r* (bits 0 through 2).  Since this extraction occurs after the matching process, the bits extracted from *rp* are not consecutive in *rp*, or are at least extremely unlikely to be consecutive.  Coupled with the randomized point of extraction from *m*, an attacker is left with the problem of determining both the values of the bits extracted and their origins within *rp*, and further increases the spurious solutions that must eventually be recognized as such before a definitive cryptanalytic solution can be accepted.

An attacker may simply guess the values of bits used in the cross-/re-seeding processes, but this is a brute force attack. While a brute force attack with a complexity of $2^{64}$ cannot be regarded as particularly difficult, its multiplicative effect on any hypothesized attack on other parts of the scheme faster than brute force is likely to be significant.

One issue, previously mentioned, with regard to the extraction of the bits to be injected into *A* is that there is no guarantee that there will be enough bits in *m* to provide the required bits at the location specified by the bits from *r*. The fact that *A* has a defined but randomly re-seeded <*X*> generator means that the deficiencies will be made up by that generator, and further that the state of the <*X*> generator is at least marginally relevant to the cryptanalysis of the overall scheme, though far less so than other constituent generators. The entropy of that generator's seed and state are not wholly lost to the overall scheme, however marginal its effect.

The final step in the process is the bit-blending of the two halves of what remains of *m*. As the two halves are derived from an aperiodic stream of bits, the blending process will yield a bit string that is aperiodic and has an unbounded linear complexity. On average, the yield will be about 232 bits (or 29 bytes) for every 1056 bits of random material used. The following diagram completes the illustration of the overall process.



Generation of bit-string *k* from bit-string *m*, using generator/PRNG *D*

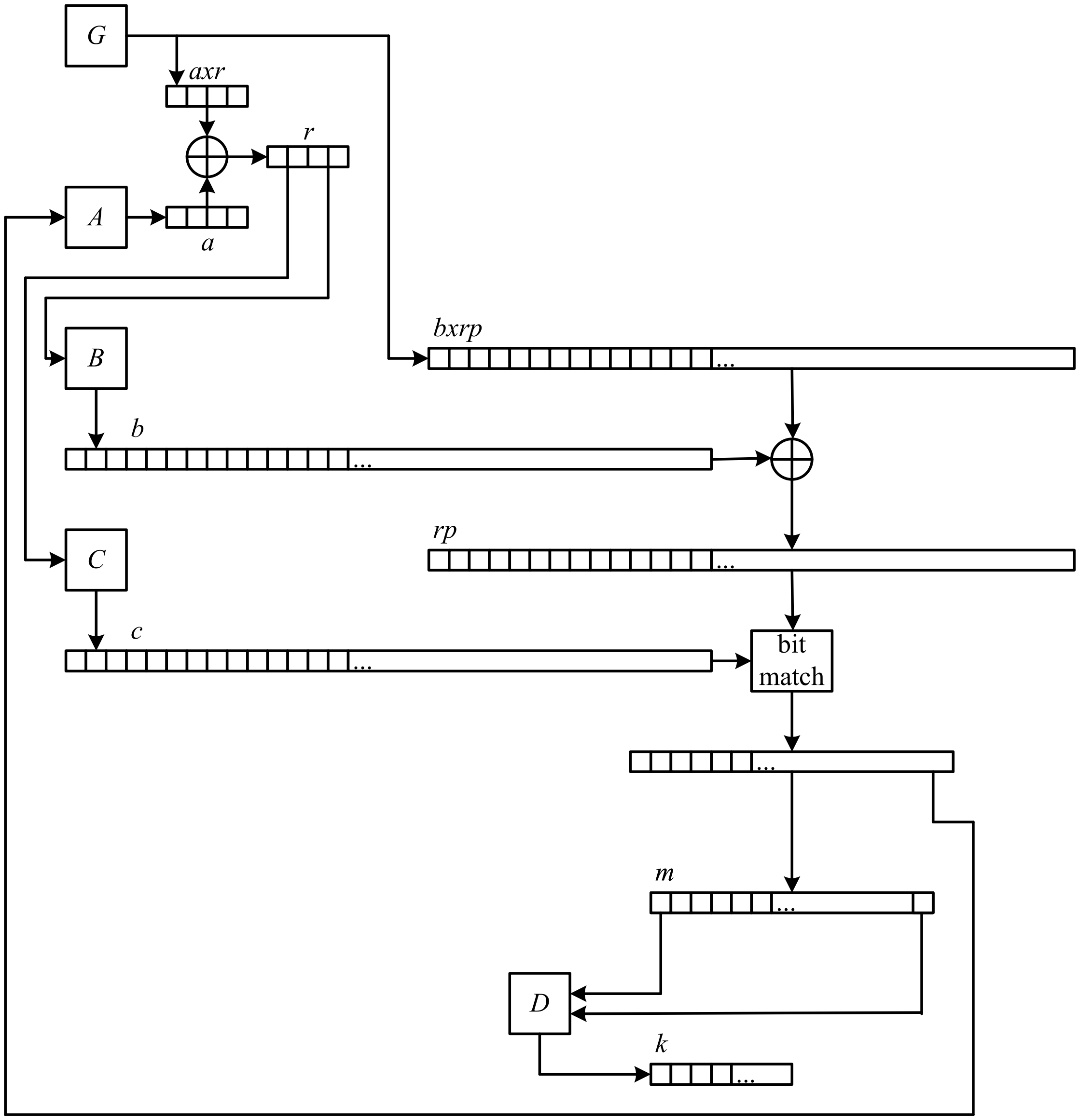
This last step uses the reduced bit-string *m*, which is the product of first discarding half of the bits in bit-string *rp*, then trimming to arrive at an even byte length. Now, due to the fact that the final reduction using generator *D* operates on bytes, a further byte of *m* may be discarded from the middle, if *m* is not an even number of bytes in length. And, we again note that, since the bits in the trimmed *m* are random, the output from *D* will also be random.

The entire process described in Protocol A is deterministic. Yet, because the data processed is random, the effect of the process is to map a pair of random bit-strings (*r* and *rp*) to another, shorter bit-string (*k*) while preserving the randomness of the bit-strings that served as input.

It is again observed that Protocol A can be modified to achieve the same result without the pre-processing performed by Alice in steps 1 through 4. This is a direct result of the fact that *r* and *rp* are random bit strings.

In any correct implementation of OTP cryptography, the strength of the system derives from the fact that a random bit value XOR'ed with a non-random bit value results in a random bit value. The ciphertext produced is an encryption of all possible plaintexts of the same length, with equal probability. XOR'ing the ciphertext with the plaintext produces the original random key. Changing the perspective slightly, if we treat the plaintext as a Vernam cipher key used to encrypt a random bit-string, XOR'ing the plaintext "key" with the ciphertext produces the random bit-string.

If *R* is used to generate *axr* and *bxrp* directly, both Alice and Bob can treat both in the same manner, starting with Bob's handling of *axr* in step 5 of Protocol A. This change results in the flow of operations illustrated in the following diagram, which integrates the pieces presented above.



Flow of operations and data in Protocol B

Formal description of Protocol B must be provided, and follows here.

## Protocol B

1. Alice generates a random bit string of bits *axr*, of 32 bits, and a random bit string *bxrp*, of 1024 bits, using random number generator *R*, and sends these to Bob.

2. Alice and Bob pump their respective copies of *A* to obtain *a*, then XOR this with *axr* to obtain the bit-string *r*.
3. They insert the lowest-order 15 bits from *r* into their respective *xQ* buffers of their copies of *B*.
4. Both pump their copies of *B* to obtain *b*, and XOR this with *bxrp* to obtain *rp*.
5. They insert bits 3 through 16 (14 bits) of *r* into the *xQ* buffer of their copies *C*, and pump these copies of *C* to obtain matching 1024-bit blocks *c*.
6. They use *M* to do a bit-match selection of bits using *c* and *rp*, producing *m*, which they truncate as in Protocol A.
7. Provided there are at least 12 bytes in *m*, they use bits 0 through 2 of *r*, as integer *o*, to select 32 bits, starting at *m*[*l* – (*o* + 4)]. These bits are removed from *m* and inserted into the *xQ* buffer of their respective copies of *A*. Otherwise, they use the *xQ* backup generator for *A* to satisfy the requirements for pumping that PRNG.
8. Both Alice and Bob split their copies of *m* in half as in Protocol A, and feed the first half into *D* as the <*X*> sequence stream, and the second half as the <*Y*> sequence stream. *D* is pumped until the <*X*> and <*Y*> sequences are exhausted (they meet in the middle of *m*) to yield the bit string *k*.
9. Alice and Bob repeat steps 1 through 8 until they have as many bits of agreed key material as they require.

Note that in step 1 of Protocol B it isn't necessary that Alice generate the random bits for *axr* and *bxrp*. "Generate" can be loosely interpreted as "selects," with complete freedom to use random bit strings from any sources she chooses. The sole requirement is that the sequences selected must be random.

As for the use made of the end product of the process (bit-string *k*), the Alice and Bob may make any use of it they wish that requires agreement on a set of random bits. This may be as session keys, message IDs, or anything else desired and appropriate according to the security requirements of the application, and the relative strength of the constituent <*X*> and <*Y*> sequence generators used in *A*, *B* and *C*. This includes use of *k* as a Vernam cipher key-stream, as a block cipher session key, or as updates to the "counter" when using block cipher counter mode. (See Appendix A regarding counter mode.) Such applications will be subject to the requirements and restrictions described in Chapter I, and according to the to-be-determined strength of the protocols described here. Discussion of the strength of these protocols is dealt with in Chapter V.

## Issues Not Covered or Briefly So

A number of issues regarding the above protocols are not dealt with in this document. Among these are the means by which the transmission of the *axr* and *bxrp* bit-sequences from Alice to Bob occurs, including how assurance of successful receipt may be obtained, and how the initial seeds or states for the various components come into possession of the correspondents.

As far distribution is concerned, we can simply assume distribution of the initial state, or a set of initial states allowing re-initialization at some prescribed intervals. While the volume of material that may be distributed and used as OTP key material is now much greater than in the past (on the order of terabytes, with distribution via a disk drive, at present) real-time video applications must still be recognized as capable of consuming that OTP material at a rate making such distribution impractical. Repeated such distributions create security problems in their own right, while single distributions are less risky. As the presented mechanism allows real-time replenishment of agreed random bit strings, if it proves sufficiently secure, it will tend to minimize that original material problem relative to the long-term operating costs.

Another issue not covered is the selection of the component generators to be used, beyond some basic observations. The protocols above are not inherently tied to any specific PRNGs in roles as the <*X*> and <*Y*> generators of *A*, *B* and *C*. Any PRNGs that satisfy the requirements of the target applications may be used.

The overall performance of the protocols is highly dependent upon the constituent PRNGs, and how those PRNGs are implemented. While the core mechanisms are relatively efficient, and their performance is discussed in Chapter V, the performance of an implementation of these protocols will tend to be dominated by the performance of the constituent PRNGs, particularly if these are slow. For example, use of software-based Blum-Blum-Shub (BBS) generators on a single processor will yield very slow performance relative to highly parallelized hardware implementations of efficient generators such is LFSRs. Such considerations are discussed only briefly in the following chapter.

Finally, though an implementation of Protocol B is provided in Appendix C (less the requisite RNG *R*, which must be a hardware solution), this is provided primarily as a test-bed for the overall scheme, and thus does not use particularly strong PRNGs as constituent generators. That implementation is used to verify correctness, not as an in any sense “mandatory” implementation, just as the explicit integer values used in the above protocol descriptions may be altered in any extrapolation of the overall scheme.

# CHAPTER V

## ANALYSIS AND CONCLUSION

In cryptography, there is no point to proposing an algorithm, protocol or anything else without offering some analysis of how that proposal addresses a cryptographic problem. Due to the character of cryptanalysis, which is as much an art as a science, no analysis by a tyro in the field can be regarded as conclusive, particularly as even the best efforts of a seasoned expert can miss salient points that may prove fatal to scheme [14][25]. Only when a rigorous formal proof is given can a conclusion be reached, rather than an inference.

Claude Shannon’s proof [5] that a One-Time Pad (OTP) provides perfect encryption is a rare exception to the general rule that cryptographic systems do not have rigorous proofs of strength. Many cryptographic algorithms for which any type of proof exists rely upon the Diffie-Hellman conjecture as a fundamental premise. If that conjecture can be shown to be false in any such instance, the proof is refuted, and the algorithm may well prove to be weak.

Shannon’s proof of OTP’s strength is a major reason for the structure of the present scheme. Rather than rely upon a conjecture that some of the various aspects of the scheme constitute “hard” problems, it is believed that incorporating a problem that is demonstrably impossible is a better approach and will lead to a stronger argument for the scheme’s strength, even in the absence of a formal proof of the security of the system as a whole.

The “hard” problems relied upon by many cryptographic systems are based can be solved, given unlimited time and resources. Such problems include the factoring of very large numbers, determining discrete logarithms, etc., as previously discussed. These have not been proven to be impractically hard. They are simply presumed to be so, based upon the present understanding of these problems. In contrast, OTP is provably impossible to break, given a correct implementation and proper operation.

Use of an OTP-like construct in the protocol does not imbue the protocol with perfection. In OTP ciphers, the attacker can never say with certainty whether a particular cryptogram used a particular OTP key if the key and the plaintext are destroyed as required by the implementation described by Shannon. In the present proposal, we must assume, as per a strong form of Kerckhoffs’ principle [3][4], that an attacker will eventually know both the *axr* and *bxrp* bit-strings transmitted as well as the resulting bit string *k*. (See Chapter IV for descriptions of these and other portions of the protocol.) The random bit sequence used to derive *k* is latent within any copy of *bxrp*, and is derivable from *bxrp* if the derived bit-string is also known. Likewise, the material latent within the *axr* bit-string may potentially be derived from the agreed bit-string. The cryptanalytic question is therefore whether an attacker possessing these substantial clues regarding the state (or key) to the agreement mechanism can derive that state from those bit strings in a way that compromises the scheme, given reasonably anticipatable resources and time.

Thus it may be seen that we cannot claim strength of mechanism simply from asserting that, like OTP, we start with random bit-strings *r* and *rp* (or *axr* and *bxrp*, in the case of Protocol B), and therefore are perfectly secure. Rather, we can say we have a firm foundation for parts of the scheme, but must show that this security is not fatally compromised by other elements of the system, and must justify claims of security in the remainder of the system, at least within the limits of the author’s knowledge and experience.

To claim at least some strength, we must show that the element of Shannon's proof that applies is the entropy continually injected by use of random sequences, and that, coupled with the combinatorial problems posed by the scheme, this entropy is not wasted, that the scale of the problem left an attacker remains sufficiently intractable, for a sufficiently long period between re-keying, that there is an acceptable level of risk associated with the scheme.

Note the distinction between re-seeding and re-keying. Re-seeding is the process by which we inject new, random material into the various composite generators as part of the normal operation of the mechanisms. Re-keying is the replacement of substantial portions of the overall state outside normal operation. Re-seeding is a continuing regenerative process that is intended to make a cryptanalyst's problem of identifying the mechanisms internal state more difficult by removing periodicity, and occurs within the operation of the protocol. Re-keying (at least with complete state replacement) forces the cryptanalyst to start over from scratch, and occurs outside the operation of the protocol.

Having said all this, it is important to reinforce one truth with regard to cryptography: it is exceedingly easy to be wrong. Two highly relevant quotes are worth offering here. The first is from David Kahn [14].

> "Few false ideas have more firmly gripped the minds of so many intelligent men than the one that, if they just tried, they could invent a cipher that no one could break."

The second quote is from Bruce Schneier [25], the first sentence of which has come to be known as "Schneier's Law." The latter two sentences are of particular relevance here.

> "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break. It is not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis. And the only way to prove that is to subject the algorithm to years of analysis by the best cryptographers around."

Given the time and resource constraints of a thesis, this protocol has not been subjected to the years of analysis by the seasoned cryptographers who might prove its ultimate worth or worthlessness.  It is the intention of the author that it will be submitted for far more extensive analysis, above and beyond anything the author is presently capable of.

What is believed of the presented protocol by the author is that it is composed of parts that have merits, as well as flaws, that the flaws of the individual components have been, to the best of the author’s ability, addressed by the other parts, but not that the scheme as a whole is therefore sound, cryptographically strong and secure as a result.  Rather, it is believed that it demonstrates some methods of worth, and that these may be further refined and developed.  Indeed, given the inherent difficulty of cryptography, and of designing good algorithms and protocols, the author will be pleased if any part of this protocol is carried forward into better, stronger solutions in the future, and thrilled if it is shown to be robust with minimal changes.  But, the present problem is to show that care and good thought has gone into an idea and its implementation, and that the result is worthy of further study.

## Apparent Randomness of the Produced Bit-Stream

The first consideration to be addressed is whether the scheme is capable of delivering what it is intended to provide: an agreed upon string of apparently random bits shared by two or more parties, wherein the apparent randomness is sufficient to afford some level of security from their use.  That two parties will, given identical initial states and algorithms, end the processes described with agreement upon strings of bits is here regarded as true, without formal proof.  The processes described here are deterministic.  Apart from re-keying, all randomness incorporated in the scheme is shared completely, leaving no window for divergence apart from system faults or external interference, both of which lie largely beyond the scope of this thesis.  (Re-keying is a separate process not dealt with in depth here.)  This leaves the question of what is meant by “apparently random.”

In the present instance, by “apparently random” we mean that, taken by itself, the agreed upon bit-string *k* produced is such that given any prefix of *k* we cannot predict with much greater than 0.5 probability of being correct the next bit in the sequence, and given any suffix of *k* we cannot infer with much greater than 0.5 probability of being correct the value of the bit that immediately preceded that suffix. In other words, we must show that we cannot distinguish *k* from a truly random sequence of bits. The demonstration of this is in two parts.

The first is via testing multiple output strings from the algorithm using an implementation of the BSI AIS 20 test suite. While numerous other test packages could have been used, the decision was to limit testing to a single, well-understood package. If this test is not passed, no amount of theoretical argument can substitute for the failure. The output sequences from 10 runs of 64 blocks all passed, giving reasonable assurance of at least minimal apparent randomness.

The second part of showing the result is apparently random, given random inputs, is based in theory. It can be summarized in a single statement: The resulting bit string *k* is random, provided the input strings *r* and *rp* (or *axr* and *bxrp*) are random, precisely because *k* is derived from them. This is a very unsatisfactory assertion without some support. An informal proof is as follows.

The *axr* and *bxrp* bit-strings may be regarded as either random bit strings in their own right (in the case of Protocol B) or as the product of the exclusive or (XOR) of a random strings of bits with pseudo-random bit-strings of equal length (in the case of Protocol A). As per Shannon [5], the XOR operation preserves entropy. Therefore *axr* and *bxrp* preserve entropy, or randomness, regardless of which protocol applies. The first step of the process, for both *axr* and *bxrp* is an XOR of the output from a PRNG with the subject string. As the initial *axr* and *bxrp* strings are random, the results preserve entropy and are putatively random. (In the case of Protocol A, at this point we have recovered *r* and *rp*, which were random in the first place.) In the next step dealing with the *rp* string, the output of yet another PRNG is used to select bits based upon value and position, yielding the *m* bit-string.  As the value of each individual bit in *rp* is random, and its selection or non-selection as output from this step is determined by the value of that bit, the result is again random.  Alternatively, we can view the process as using the bits in *rp* to randomly select bits in *b*, which is itself a random process and logically identical to using *b* to select bits in the random string *rp*.  In the third step of processing, now dealing with *m*, yet another composite PRNG is used, with the *m* bit-string used to provide its feeds.  As these bits in *m* are random, the output bit-string (*k*), which is the result of selecting four-bit values based upon those feeds, with equal probability for any bit in the output being a one or a zero, is itself random.

However random the resulting bit-string *k* is, given random *axr* and *bxrp*, the system is deterministic relative to both the state of the mechanisms and the inputs.  There is an important distinction to be drawn here.  So long as the result of a sequence of operations depends solely on the initial input and state, which are random, and any change to the input or state results in an uniform probability of change in the output (each bit in the output has a 0.5 probability of changing), the result of the process is apparently random, ***but the process itself is deterministic***.

One way to think of this is with regard to a periodic PRNG.  If the seed of such a PRNG is randomly selected, with uniform probability for all bits, we have in effect selected a random point in the output sequence of the PRNG via the random selection of the initial state.  Assuming a sufficiently long period with good apparent randomness, any portion of the PRNG's output sequence thus selected, taken in isolation, is effectively random, provided it is no longer than the seed, while the cycle generated by the PRNG is not random since it is determined by the algorithm.  The entropy of the output sequence declines as the output sequence increases in length past the length of the seed.  This is why cryptanalysis is possible for non-OTP systems.  Only in OTP and its homologues is cryptanalysis impossible, and only because entropy is preserved, never decreasing, throughout.  Therefore, we must understand the process embodied in the protocols and the implementation not as an encryption process, but as a synthesis or distillation process, whereby a pair of random input streams are used to generate bits that are apparently random in behavior.

## Confidentiality

The next question that must be addressed is whether the result affords useable confidentiality when the initial state of the mechanism is secret. This is far the more difficult question to answer. Again, by Kerckhoffs' criteria, we must presume that communications are monitored by an attacker, and thus the attacker will have copies of the *axr* and *bxrp* bit-strings. Using a strong interpretation of Kerckhoffs' principle, we assume not only that the *axr* and *bxrp* bit-strings are known by an attacker, but also the *k* bit-strings that result from processing *axr* and *bxrp* sequences, and that the real task of the attacker is to determine the internal state, or key, of the mechanism associated with a given *axr*/*bxrp*/*k* triplet. Once an attacker has determined the state, the subsequent output strings can be determined by the attacker upon receipt of sequential *axr* and *bxrp* strings. Therefore, as per Kerckhoffs, the confidentiality of the system must derive not from the secrecy of the scheme, but from the key, which is in this case the initial state of the mechanism, and the difficulty of deriving the key from the input and output bit-strings.

The most vexing problem in the analysis of any cryptographic scheme is that the types of attacks that may be developed in response to a new algorithm cannot be readily predicted. Known attacks can be analyzed with regard to the scheme being examined, but not unknown and yet-to-be-developed ones. There are many extant attacks in literature, though the applicability of any one to a particular scheme varies dramatically from extremely effective to wholly pointless. This fact has contributed significantly to the design of the scheme presented. The intent has been to rely only upon simple mechanisms that are relatively easy to analyze and as near devoid of reliance on any form of Diffie-Hellman conjecture, or even upon mathematical operations, as practical.

The most basic form of attack, brute force, must be discussed when analyzing any cryptographic system. Correlation issues must be considered when a scheme entails the interplay between component PRNGs. Periodicity, specifically with regard to the Berlekamp-Massey algorithm, must be addressed when dealing with deterministic systems of PRNGs. Differential attacks must also be considered, as non-uniform probabilities associated with the behavior of an algorithm, or resulting from the composition of the data structures used, can be effectively lethal to a system.

Some forms of analysis and attack are avoided by the fact that this is not a cipher system, *per se*. There is no encryption of a plaintext to form a ciphertext. The process is essentially a decryption process that derives the output string *k* from the random input strings, based upon the secret key, which is the initial state of the mechanism. In particular, the bit-matching process, in discarding roughly half of all bits in *rp*, is not reversible so that *rp* can be definitively derived from *k*, even though cryptanalysis may well allow eventual determination of the state that maps *r* and *rp* to *k*. The scheme provides means by which two correspondents may agree upon a string of apparently random bits in what is intended to be a secure manner. Ciphers are used to gain confidentiality through obfuscation of the plaintext encrypted. Plaintext generally incorporates some degree of redundancy in the message itself: the components of the plaintext relate to each other in ways that help to ensure the understandability of that plaintext by its intended recipient. Also, a cipher system must provide means, some inverse function, utilizing either a symmetric key or a member of a key set, to decipher the ciphertext and recover the original plaintext message. Here, we wish to preclude such inverse operations as far as is practical.

In certain senses, there is a “message” (*axr* and *bxrp* together, and ultimately *k*, which is derived from them) embedded within the material transmitted by Alice to Bob. There is also redundancy, in that a significant number of bits are discarded from *bxrp* (and derivatively from *rp*) as it is processed to determine *k*. These discarded bits are redundant in a different sense than extra information contained in ordinary plaintexts. Their contribution to the string *k* is their non-participation, obscuring the point of origin of the bits within *bxrp*, that are used to generate the output string *k*. Their non-participation also helps to determine the length of the *m* bit-string. This being the case, it should be possible to determine the unicity distance of the encrypted "message" *k*.

Given the definition $U = H(keyspace)/D,$ where *U* is unicity distance, *H*(*keyspace*) the entropy of the keyspace and *D* the redundancy of the plaintext in bits per character, we can observe certain facts. First, the "alphabet" of *k*, the putative plaintext, is binary digits. Second, *H*(*keyspace*) is non-zero for any keyspace that is non-empty if individual keys are selected randomly.

The third observation is more complex, but critical to the analysis. When *D* above is referred to as the redundancy of the message, the meanings of both "redundant" and "message" must be understood clearly. Obviously, the bit-string *k* is the message Alice wants Bob to receive, as well as possess herself. But, Alice does not select any specific *k*. Rather, in using random *r*/*rp* (or *axr*/*bxrp*), she randomly selects one of the possible outputs from the mechanism, given the current state. This means that there are putatively $\sum_{i=0}^{124}\left(2^8\right)^i$ distinct possible outputs from which Alice is randomly selecting for each *r*/*rp* block.

As regards redundancy in *k*, it is simplest to consider more typical messages. In human languages, particularly in their written form, much of the material conveyed, is in excess of the minimum required to convey meaning, but is present in part to reduce ambiguity and improve understandability. This gives us such things as the distinctions between the words "to," "too" and "two," as well as the difference between the definite article "the" and the indefinite articles "a" and "an." This is constructive redundancy in the sense that the excess information helps to confirm the meaning of the text by a human reader. It also provides "aid and comfort" to cryptanalysts by reducing the range of distinct, meaningful messages that may be encoded in a given number of characters in a “human” language.

In the case of the *k* bit-string discussed here, there is no constructive redundancy. There is no redundancy of any kind. Each bit is independent of the others. Changing any one bit changes the message, and depending on the use made of *k*, can completely alter the results. For example, if *k* is used as a session key for a symmetric cipher, such as AES, the ciphertext resulting from that single bit change will be substantially different from that produce without that change in *k*. So, if we consider *D* in the definition of unicity distance relative to the *k* bit-string, $D = 0$. Taking this view of *k*, the limit as *D* approaches zero in the above framing of unicity distance is infinity, for any non-zero constant *H*(*keyspace*).

The next question is then, “What is the value of *H*(*keyspace*)?” In part, the answer to that is simple. If the initial state is random, *H*(*keyspace*) = *n*, where *n* is the number of binary digits in the key (for binary keys) or $\log_2$(*keyspace*), making the problem determining what constitutes the key, and thus its length. That depends on the *r* bit-string, the implementations and states of the generators used as constituents of *B* and *C*, the state vectors of the *B*, *C* and *D* generators, as well as what of that material is secret, what known.

Note that the states PRNG *A*’s constituent generator are not considered in this, nor is the *State* vector of *A* itself. This requires explanation. As the probability that bit string *m* will be shorter than the re-seeding threshold is very low, 32 bits extracted from the *m* bit-string will usually be used in lieu of the output of one of *A*’s constituent generators (the <*X*> generator), the output from *A* (the bit-string *a*) is in fact random. Thus, bit string *axr* is random with regard to both bit strings *a* and *r*. By this we mean that, since *r* is random, and the <*X*> input string to PRNG *A* is also random (making the output from PRNG *A* random), and the bit-wise XOR of bit strings *a* and *r* preserves the entropy of both, bit string *axr* retains the entropy of both bit strings, and thus is as random as either.

Without some additional information regarding the *r* and the re-seeding 32-bt re-seeding bit string taken from *rp*, there appears to be no effective means whereby *axr* can be cryptanalyzed more rapidly than simply guessing what bit string *r* may be. Even if the *A* composite generator's <*X*> generator were invoked with some regularity, the problem of cryptanalysis appears to remain difficult, though aggregate state of PRNG *A* would start to contribute in a more conventional sense to the effective key length. Fortunately, invocation of *A*'s <*X*> generator has been found to be very rare.

From such considerations, it is believed important to any alternative implementation of the overall scheme presented that varies the lengths of the various bit-strings that the probability of not re-seeding *A* from *m* remain low, and that the number of re-seeding bits equal the length of *r*. This will likely mean that *A* does not play a direct role in *H*(*keyspace*), when considering the cryptanalysis of *bxrp* and *k*. An attacker need not cryptanalyze *axr*, only guess the effects on the states of the *B* and *C* generators, or, if the mechanism proves weaker than intended, derive the values obtained from *axr* based on cryptanalysis of *bxrp* and *k*.

This may be considered a flaw in the system, but it is believed that it helps *r* continue to inject entropy into the system with each cycle, at least until a solution for *B*, *C* and *D* has been found. At that point, with the re-seeding bits for *A* revealed before the next cycle, the process of breaking *A* can begin with good effect. It is therefore believed justifiable that, though the state of *A* is part of the key material for the overall system, it can and should excluded from determining the effective key length in determining the unicity distance with respect to *bxrp* and *k*, if for no other reason than an abundance of caution in that determination.

Unfortunately, such analysis is not very revealing with regard to the overall cryptographic strength of the agreement scheme. Unicity distance is an indication of how much material must be successfully decrypted in order to be certain that the key used is the correct key. With the assumption that an attacker will eventually obtain significant samples of the *axr*, *bxrp* and *k* bit-strings, the fact that the function that takes *axr* and *bxrp* to *k*, given some specific initial state, means that we must expect that, with unlimited time and resources, an attacker will eventually be able to determine the initial state from a sufficiently large set of these bit-strings. But, under the specified assumptions, this unbounded value indicates that, if all else is found to be reasonably secure, a certain amount of uncertainty remains for quite some time in any solution process.

Having raised the question of redundancy, and observed that there is redundancy in the *bxrp* bit-string transmitted by Alice to Bob, we must account for this relative to the Unicity distance in some manner. The problem with incorporating this redundancy in the calculation is that though we may regard *bxrp* as a message, it is ciphertext, not plaintext, and as much of it will usually be discarded, the redundancy is not constructive in the sense of making the "plaintext" string *k* more easily understandable. Therefore it believed that it plays no factor in the Unicity distance.

The distribution of lengths of *k* depends directly on four factors. The first is the selection process from *rp* using the *C* generator in the bit-matching process. Assuming the process is random, with a 0.5 probability for selection of any one bit, the resulting bit-string *m* will display lengths that conform to the binomial distribution. At this point, the second and third factors come into play. The process of selecting the re-seeding bits for *A* removes some number of bits (32 bits, in the present implementation) if the total length of *m* is above a threshold. This process requires (for the sake of simplicity in the presented implementation) that *m* be truncated to a length congruent to 0 mod 8. Thus, 32 bits, plus 0 to 7 bits are discarded or otherwise removed at this point. Finally, the bit blending that occurs as the final step requires (again for simplicity) that only an even number of bytes from the *m* bit-string be used in generating *k*. This results in "discarding" either 0 or 8 bits.  Any exact calculation of the average |k| must account for these factors, but we are left with the fact that it is only an average, and that we will still see a “binomial like” distribution of lengths, predominantly about the mean.

If we examine carefully the contribution discarded bits make to the security of the scheme, we return to the following equation, first introduced in Chapter III, which give the number of possible ordered bit-string pairs that could produce a specific bit-string of length *n*.

$$C = \sum_{i=n}^{\infty} \binom{i}{n} 2^{i-n}.$$

Now we are dealing with a specific instance.  We know the actual lengths of the compared input strings, and the length of the resulting matched-bit string, and so are not dealing with a summation over all possible input string lengths.  Therefore, we can calculate the exact number of ordered input string pairs that can produce the same result, ignoring for the moment that bits will be removed from *m*.

$$C_{rp,m} = \binom{|rp|}{|m|} 2^{|rp|-|m|}.$$

If there are matched bits that are omitted due to an incomplete byte at the end of the matching process, the result is a little more complicated, as we have the following.

$$C_{rp,m,o} = \binom{|rp|}{|m|+o} 2^{|rp|-|m|},$$

where *o* is the number of omitted matching bits, and *m* is now the matched bits the attacker “sees” as output from the matching process.  Initially, an attacker will know neither the number nor the values of the bits that matched but were discarded, though he may subsequently either guess them, or learn their values as a result of analysis. Therefore, an attacker will have to deal with a number of other possibilities, due to the additional bit omissions. The above equation now becomes, in most cases, the following, once we have also accounted for the 32 bits extracted for use in PRNG *A*.

$$C_{rp,2k+32,o} \cong \binom{|rp|}{2|k|+32+o} 2^{|rp|-2|k|+3},$$

where *k* is the agreed bits string, and *o* is an integer in the range [0, 15]. (Note that the “+3” in the exponent above accounts for the eight possible placements of the eight different placements of the 32 bits that are removed for use in PRNG *A*. Also, the calculated number of complementary string pairs is no-longer exact, since repetitive patterns in the region where the 32 bits extracted for use in PRNG *A* can produce the same results regardless of the exact segment extracted.) Now, an attacker does not initially know how many matches are not represented in bit-string *k*, plus the very probable 32 bits used in re-seeding *A*, and the maximum number has increased from 7 to 15. Initially, the attacker knows *k*, and thus its length. The number of possible *rp*/*c* bit-string pairs that could have created *k* is thus the sum of cases where *o* ranges from 0 to 15. As for the placement of the excluded bits, he initially knows roughly where 8 bits may have been removed from the middle of *m*, where any of the 0 to 7 “odd” bits were (at the end of the otherwise matching bits), and the 8 places where the 32 bits injected into *A* were originally in *m* (as a block from the last 8 bytes of *m*).

The overall probability distribution for |*k*| is messy, thanks to the threshold question. For lengths below the threshold for extraction of re-seeding bits, the tail function for the binomial distribution is useful. Because it really matters little where and whether a byte and from zero to seven bits are discarded for processing, we can sum the probabilities of individual lengths of m prior to discard, in blocks of 16 bit lengths. For example, the probability that *k* is an empty string is given by the tail of the binomial distribution from 0 to 15 bits over a 0 to 1024 bit domain. The probability that *k* is 8 bits in length is the tail probability from 0 to 31 bits minus the tail probability from 0 to 7 bits, etc, up to the threshold. At the threshold and above there will be an additional set of probabilities associated with four bytes removed for lengths greater than the threshold. Still these disjoint probability domains can be calculated in a reasonably straightforward manner, despite the overlap. It just takes time and high-precision computations, when |rp| is large.

Trials using the actual implementation presented produces information that, over very large numbers of blocks, may be expected to approximate the theoretical probability distribution. Such trials were run, with 1,000,000 1,056-bit blocks (bit strings *axr* and *bxrp* combined) processed in two separate runs. The average observed for these combined runs was 29.307 bytes per block processed.

Consider now the combinatorial problem of the system in generalized terms. For any given bit-string *k* produced by the mechanism with initial state *s*, we would expect that there would be, on average, |*r*||*rp*| / |*k*| separate *r/rp* bit-string pairs that would map to the *k* bit-string generated, since the mechanism embodies a many-to-one mapping. In other words, we can think of it as a function $f : G_2^{|r|+|rp|+|s|} \rightarrow G_2^k$, where *s* is the state of the mechanism at the start of a cycle, and thus |s| is the size of that state in binary digits, and $G_2$ is the set {0, 1}. For a fixed initial state *s*, the mapping function becomes $f' : G_2^{|r|+|rp|} \rightarrow G_2^k$. Further, since *k* varies freely with each cycle, given the results of two successive cycles concatenated without specification of the boundaries between the successive *k* bit-strings, it is not possible to definitively state where the boundaries between different *k* sequences lie without knowledge of the initial state, even when the applicable *axr* and *bxrp* bit-strings are known. Therefore, depending on how the *k* bits are used, it may not be possible to isolate an *axr*, *bxrp*, *k* triplet for analysis without knowledge of *s*.

This consideration does not lead to a claim of additional strength, but to recognition that we are in reality determining a "lower bound" on the cryptographic strength (or "the difficulty of reaching a cryptanalytic solution") of the proposed scheme, given the assumptions made regarding its use. The most important of these are that the initial state of the mechanism on its first invocation is random, as are the *r*/*rp* or *axr*/*bxrp* bit-string pairs used as inputs, depending on the protocol used. With the extreme interpretation of Kerckhoffs' criteria, and the resulting assumptions regarding analysis of the scheme, we believe a sounder argument for what strength is claimed can be made thereby. We therefore assume that, given a set of plausible solutions, and a known sequence of communications and products, the spurious solutions will over time be eliminated as inconsistent with subsequent products, resulting in increasing confidence in those that remain, despite the uncertainty imposed by an attacker not knowing the boundaries between successive *k* strings generated.

Restating the above for clarity, since in practical applications an attacker cannot expect to know the boundaries between successive *k* sequences relative to the corresponding *r* and *rp* sequences communicated, an attacker must continually guess at the boundaries, or determine the initial state for the cycle so as to determine the boundaries. The number of plausible solutions at the start of cryptanalysis will be significantly greater than $|axr||bxrp| / |k|$ due to the combination of this uncertainty and the random characters of *r* and *rp*. But, we choose not to claim additional strength for the scheme based upon this, in order to derive what we regard as a good, conservative conclusion. It is believed that the system is bounded by the single *axr*, *bxrp*, *k* triple, and that the system can be no weaker than the difficulty of solving for *a*, *b* and *c* bit-strings used (and thus the states *A*, *B* and *C*) plus the contents of the state vector of *D*, for a single *r*/*rp*/*k* triple, as carried forward through attacks on subsequent triples, barring some unfortunate and as yet unidentified flaw in the scheme or the implementation presented.

Returning at last to the question of $H$(*keystream*), and to reiterate the point, for purposes of determining the entropy of the keyspace with regard to cryptanalysis of the derivation of $k$ from $r$ and $rp$, we regard the key of the presented scheme to be comprised of the combined states the constituent generators of the $B$ and $C$ generators, plus the state vectors of $B$, $C$ and $D$ generators, plus $r$ from the current cycle of the mechanism.

While the constituent generators are explicitly given for the presented implementation, these are mere examples, and are easily replaced. Selection of LFSRs for their implementation is solely due to their linear complexity being well understood.

## Brute Force Time Requirements

While it is extraordinarily brazen to claim that a brute force attack is the only means by which a system may be successfully cryptanalyzed, no analysis of a cryptographic scheme can ignore the question of what such an attack would entail. If a brute force attack is not sufficiently expensive in terms of time and resources as to be impractical, given the scheme's application domain, the system must be regarded as insecure from the start.

What constitutes an inadequate problem from the perspective of someone considering a brute force attack changes with time due to improved tools and the resources available to an attacker. This can be seen in the history of the Data Encryption Standard (DES), and its demise as a viable block cipher in critical security applications [121]. DES was seen as potentially inadequate even during the adoption process [25], as the effective 56-bit key length was not perceived to constitute a long-term difficult problem. Reviewers anticipated the development of systems of sufficient power and flexibility to render such a short key ineffective. Thanks in part to the construction of the DES Cracker by the Electronic Frontiers Foundation (EFF) [121], this expectation was proven correct. While the full breadth and depth of such developments cannot be easily foreseen, some assumptions must be made for analysis to be meaningful.

At present, the potential of quantum computing is still largely unknown. An algorithm for quickly factoring very large numbers into their constituent primes has been proposed [26]. This should be regarded as a cautionary warning for cryptographic systems that rely on presumably very hard mathematical problems, and thus entail the Diffie-Hellman conjecture, even though there are arguments that the algorithm involves sufficient compounding error terms to render it ineffective. Even if such counter-arguments are true, it should be recognized that if the algorithm merely reduces the search space in factoring large numbers, it will materially weaken all systems that rely upon factorization being a hard problem. The present scheme starts with a One-Time Pad(OTP)-like approach. In OTP there is no mathematical relationship between plaintext, key and ciphertext that may be exploited directly. Absent such a potentially exploitable mathematical relationship, our assumption is that, while quantum computers may radically increase the power of computing systems, and thus decrease the time required to conduct trials in a brute force attack, there will be no algorithmic solution to the fundamental problem of the entropy of random data streams that collapse the computational complexity from exponential time (i.e. powers of 2, when dealing with collections random bit values, where brute force attacks lie) to polynomial time or better.

This is much like the Diffie-Hellman Conjecture used in the proofs of the strength of many cryptographic systems, implicitly or explicitly. The difference lies in the fact that while the associated mathematical problems of such schemes are presumed to be hard, OTP is provably impossible of solution when correctly implemented [5]. We therefore regard reliance upon the entropy of the inputs as reasonable cause to believe that quantum computers (which are as yet not fully realized) will speed the process of performing brute force attacks, possibly radically, but not provide a solution to the problem of the entropy in random key material or states.

Again, with the assumption that the mixing tables used in PRNGs *A*, *B*, *C* and *D*, the attacker has the *axr*, *bxrp* and *k* bit-strings for one complete cycle (including the bounds of this *k*) plus an arbitrary number of additional, contiguous samples of the three bit-strings (though not necessarily the bounds of the additional *k* bit-strings), and the taps used for the component generators, the state vectors in *A*, *B*, *C* and *D* may be treated as the key in the specific instance, when we temporarily ignore the states of the component generators. As such, we have as many bits of secret key as there are bits in these state vectors (256 bits), and successfully guessing these contents will "solve" the given instance of the scheme. The problem for the attacker is to determine the plausible keys that associate the known *axr* and *bxrp* bit-strings with the known *k* output bit-string, then eliminate the spurious keys (keys that appear to work, but are not in fact the key used) until the actual key is revealed.

Remembering the decision to deal with *r* as key material (for calculation purposes) rather than *A* in its totality, we have 224 bits of key material to consider, having replaced 64 bits with 32. Taken at face value, this is a reasonably long key, and a brute-force attack upon a random key of this length can be expected to take a very long time, unless attacked massively in parallel.

To illustrate the time required to solve for the key, let us assume that we have access to $2^{32}$ systems that can each test $2^{32}$ of these values per second. This means that it would take $2^{224}$ / $2^{64}$ = $2^{160}$ seconds to try all possible keys. There are 60 * 60 * 24 * 365 = 31,536,000 seconds in most years. This is less than $2^{25}$, the smallest integer power of two greater than the number of seconds in a year. Using $2^{25}$ as the number of seconds in a year, it would take $2^{160}$ / $2^{25}$ = $2^{135}$ years to try all possible keys. As the age of the universe is estimated to be about $2^{34}$ years, it would take over $2^{135}$ / $2^{34}$ = $2^{101}$ times that span to test all possible keys, given the above systems and assumptions.

In more practical terms, we would have a 0.5 probability of successfully guessing the key. Even if we increase the systems to $2^{64}$ and the number of trials per second by each to $2^{64}$, the result is still $2^{36}$ times the present age of the universe to reach a 0.5 probability of success.

If we now reincorporate the states of the constituent generators for *B* and *C*, and deal with those explicitly as LFSRs, those four constituent generators have linear complexities of $L_{\mathrm{B1}}$, $L_{\mathrm{B2}}$, $L_{\mathrm{C1}}$ and $L_{\mathrm{C2}}$, (assuming $L_B = L_{B1} + L_{B2}$, etc.) which may be summed and treated simply as *L*. Adding this complexity to the above, we see that a brute force attack will entail $2^L$ times more time to achieve a 0.5 probability of success. By selecting constituents such that *L* is at least 229, we can surpass the estimated number of atoms in the universe (excluding dark matter) times the present age of the universe as the time required to reach that 0.5 probability of solution.

The ability to easily expand the size of the keyspace by incorporating constituent generators of any desired linear complexity was deliberately designed into the scheme, so that an implementer can easily control the risk associated with the scheme by varying the linear complexity of the constituent generators, as well as the frequency of re-keying.

With regard to elimination of spurious keys, we should expect the following to describe the problem for a given *axr*/*bxrp*/*k* triplet.

$$Keys_{spurious} \approx 2^{H(keyspace)-|k|},$$

where $Keys_{spurious}$ is the approximate number of spurious keys, and *H*(*keyspace*) and *k* are as already discussed. While this in itself does not guarantee any advantage against an attacker (an attack might be found that efficiently eliminates spurious keys immediately), to the extent that an attack does not eliminate all spurious keys an attacker is left with the problem of eliminating them via additional operations. Thus, again, it is believed that the ability to aggressively scale up the keyspace by increasing the linear complexity of component generators will allow implementers to scale alternative implementations to the threats faced and the sensitivity of the use made of the agreed upon bit-strings.

The first problem with these calculations is that we do not know what technological and scientific advances may eventually come from quantum computing, from continuing developments in more conventional technologies, or from as yet unimagined technologies. Nor do we know that a brute force attack is the best form of attack. Indeed it is unlikely that this would be the case. What we can say with confidence is that they scheme is not trivially weak with respect to a brute force attack.

Two closely related questions need to be addressed: "Is it necessary to attack the full key at one time?" and "Can the key be attacked in parts?" Whether the key can be solved for in parts is related to the meet-in-the-middle attack, which is discussed later, but is a broader question than addressed by that attack alone. Here we are concerned not with an attack that separates the phases of the process, but attempts to attack the contents of the state vectors of the B, C and D PRNGs in "slices," since we use only a small portion of each state in processing nybble pairs from the <*X*> and <*Y*> bit-sequences in each phase. The potential also exists in the presented implementation due to the fact that the respective phases of the process interact through the passage of information from *bxrp* through those phases to the output string *k*, without all parts of the key being explicitly involved within all of the phases.

There is interaction between the *A*, *B* and *C* PRNGs via the "cross pollination" that uses the *r* bit-string to inject aperiodicity into the *B* and *C* PRNGs, and bits from *rp* (as filtered by the bit-matching process using *C*) to inject aperiodicity into PRNG *A*. We have previously recognized that *A* may be cryptanalyzed independently of *B* and *C*. Having removed PRNG *A* from consideration as intrinsic to security with regard to brute force attacks (though we retain its state as secret), we limit ourselves to considering the *B*, *C* and *D* PRNGs. We believe the answers to the two questions are yes and no, respectively – with regard to the *B*, *C* and *D* PRNGs, and with certain caveats. The reason lies in the operation of the *D* generator, and the order in which the results of the bit-matching process are fed into *D*.

Note that *D* is distinct in this scheme in that it has no component generators, and instead accepts the reduced *m* string as its feeds for both <*X*> and <*Y*>, operating from opposite ends of *m*. It is otherwise the same, using the common *dTable* and *sTable* pair to select both an output and a state update value. The *dTable* itself is constructed to ensure that, given random feed values from the <*X*> and <*Y*> sequences, there is an equal probability for all possible 4-bit nybble values to be produced. Also, the state update values obtained from *sTable* are 14 bits in length, with one and only one occurrence of each in the frame. It is this last point that is of particular interest here. At each point in the process of deriving *k* from *bxrp*, any single bit change in the result of a step is carried forward. A single bit change in *b* will have the effect of "flipping" a bit in *rp* and vice versa. Any single bit change in *c* will insert or delete a bit in *m*, as will any single bit change in *rp*. When *m* is fed into *D*, this carry-forward of bit deltas becomes important, as the shifting of the alignment of bits within bytes of *m* will be reflected in all succeeding bytes. With the tail of *m* being fed into *D* in reverse order, all such changes are automatically reflected in the state update value obtained from the *sTable*. All subsequent blending of *m* in *D* thus reflects ***all*** single bit changes in every stage of the process, while retaining equal probability of output on a per-bit basis, thereby achieving diffusion of effect similar to that sought in block ciphers. Only in cases where a bit delta is in a deleted tail of *m* (or not incorporated into *m* due to an incomplete byte at the end) will such a change have no immediate effect. Only in cases of the bit delta being in the possible incomplete byte at the end of processing *rp* will the effect be lost entirely. Otherwise, the effects of the single bit delta will be carried forward into the processing of all subsequent blocks.

A second relevant observation here is that there is vast freedom between *m* and *k*. Given the 64-bit state vector, we can safely observe that there are $2^{64}$ different possible mappings from *m* to *k* given a random state of PRNG *D*. On a per-nybble basis, any of the 256 possible combinations of two arbitrary nybbles from *m* can produce the same output nybble, and any output nybble in *k* can be the product of any pair of values of nybbles from *m*, in both cases with the specific combinations depending on the state vector bits used. Thus, it is believed that the nybble-wise approach to teasing out a solution for PRNG *D* (meaning the contents of its state vector) only produces the set of plausible solutions, which is effectively the same as attacking the D as a whole.

A third observation returns to the combinatorial problem presented by the bit-matching process that produced *m*, as discussed in the previous chapter. The number of different pairs of strings *rp* and *c* that could produce any given *m* bit-string is vast, and compounded by the deletions for re-seeding *A*, which may occur at any of eight starting points in *m*.

Finally we return to the fact that *rp* is random, and as such encrypts *b* perfectly … at least until the corresponding *k* bit-string is known.

The combination of these facts means that, any guess as to the state of *B* (including constituent generators) produces a result that is essentially meaningless without a correct guess with regard to the states of *C* and *D*. Whether *rp* (in Protocol A) or *bxrp* (in Protocol B) is the initial random bit-string doesn't matter. Shannon's analysis of OTP applies, and the corresponding bit-string can be viewed as encrypting any string of bits of equal length.

With a guess as to the full states of *B* and *C*, the result (*m*) remains essentially meaningless, without a plausible guess regarding the state vector of *D*, with the 64 bits of that vector defining one of $2^{64}$ possible mappings from *m* to *k*.

Thus the one severable point in the process of a brute force attack lies between the state of *A* and the combined states of *B*, *C* and D.

### Using the $\chi$-Square Test

One of the most ubiquitous classical tools in cryptanalysis, the $\chi$–square test, has broad applications beyond cryptography. In cryptanalysis, it provides a means of identifying deviations from the distribution that would be expected from a set of what would preferably be independent random variables. In the present implementation, and the scheme generalized by varying the lengths of the *r*, *rp*, re-seeding strings, and constituent generators, the whole of the result still depends upon the random values input, either as *r* and *rp* or *axr* and *bxrp*. As truly random input streams will not diverge from the expected essentially equi-distribution of output values over long periods, the test isn't relevant. It will only yield an indication of how closely a particular sub-sequence of random bits conformed to anticipated random behavior. Therefore, we believe it has no practical application in the cryptanalysis of the scheme.

To claim this, it must be shown in some manner that the results will not be biased in an exploitable way. While we can justifiably assert that the product of combining two strings, one random and the other pseudo-random, via bit-wise XOR is in turn random, when we perform a substitution, even when the substitution is based upon a random/pseudo-random string pair, we must show by some means that the resulting substitution produces equi-probable results. Unfortunately, this must be done by inspection, here.

If we examine the *dTable* used by *A*, *B*, *C* and *D* closely, we can see that the table as a whole is regularly structured, with an equal number of occurrences of each of the sixteen possible output nybbles present. Further, if we select any one of the three variables used to access the *dTable*, we see that the 2D slice through the 3D table thereby selected is also regularly structured, with equal numbers of occurrences of the possible output values. Selecting any two of the indexing variables results in a 1D column, row or line through the 3D frame, with equal numbers of each of the possible outputs. Finally, we can pick any three values for indexing, and see that if we change either of the four-bit indices we must change one of the other indices to obtain the same value. The “odd” case is that of the six-bit index obtained from the state vector. In this case, a change of from 1 to 15 in the value of this index requires one of the other indices to change in order to obtain the same value as with the original three indices. Though it is not obvious that this structure and behavior guarantees uniform distribution of output values, given uniform distributions of input values, it does. For any two random 4-bit indices into *dTable*, there is an equal probability for every possible 4-bit result.

## Meet-in-the-Middle

As mentioned earlier, an attack that might be effective, since the distinct steps in the scheme operate with distinct PRNGs and separate initial states or keys for those PRNGs, is the meet-in-the-middle attack. In this type of attack, the idea is to work from both ends, plaintext and ciphertext, simultaneously, working towards a common point between these ends. When that mid-point is reached from either end, the resulting partial solution can be compared against a catalog of partial solutions from the other end. If a matching partial solution match is found, the key produced from the two partial keys is a plausible solution. As the catalog of partial solutions from each end grows, the probability that a match, and thus a plausible full solution, will be found increases more rapidly than one might expect.

Where this attack is effective, the improvement in solution probability derives from the “birthday phenomenon.”

The meet-in-the-middle approach works in cases when there is a point in the encryption process where either different keys are used at either side of the point, or the key schedule is divided in a way that has a similar effect with respect to a single key. This attack is why double encryption (use of either two passes of the same cipher or two different ciphers in sequence, with separate keys for each pass) does not offer a significant improvement in the strength of encryptions, even when the combined keys are twice as long as the keys used by the individual ciphers alone.

It is believed that, provided the periods of the *A* and *B* generators are very long, starting from the *axr*/*bxrp* strings is not very productive, due to the fact that these bit-strings are analogous to an OTP cryptogram. In the case of PRNG *A*, the <*X*> bit-string is random, so that any possible value of length |*r*| is equally likely. If the period of PRNG *B* is $2^{2048}$ bits or greater (assuming no injection of random material from bit-string *r*), and is otherwise "well behaved" as per statistical tests of apparent randomness, we may reasonably expect all possible 1024 bit values to appear at some point in the generator's cycle, with a high probability that each will appear once in 1024-bit aligned blocks. Using 32 independent, parallel LFSRs (as is the case in the present implementation), this cycle length can be approached or passed by using LFSRs of 64 bits or longer. The present implementation supports LFSRs of this length, and can be easily expanded to support arbitrary length LFSRs. In such a situation, we know of no way to discern a plausible solution for *rp* (as derived from *bxrp* and *b*), and thus for *b*, since no matter what either bit-string was in reality, the other of the *bxrp* / *b* pair can still cause *rp* to take on any possible 1024-bit value.

If the periods of the *A* and *B* PRNGs are short, relative to the range of binary values the *r* and *rp* bit-strings may assume, the situation changes. For example, if the period of the *B* composite generator is less or equal to $2^{1024} - 1$ bits, there will be less than the full range of $2^{1024}$ unique 1024 bit values in the cycle *b* might otherwise have, at least with alignment to *rp* in repeated iterations of the cycle. As the period decreases, the number of possible values *b* may assume (viewed as a binary number) decreases accordingly. Thus, care must be taken to ensure that the period of the *B* generator is longer than $2^{|b|}$, where |*b*| is the length of *b* in bits.

The picture also changes if we consider cryptanalysis of multiple *bxrp* bit-strings. While the insertion of the 15 random bits into one of the component generators of B's contributions complicates matters by shifting the alignment of the sub-generators of that stream relative to the other component generator, the number of different points in the overall cycle that may be reached is limited to $2^{15}$ for each *bxrp* string processed.

Estimates of the complexity of the problem for an attacker will vary greatly, depending on the generators used to implement the *B* and *C* PRNGs and their periods, as well as the implementation of the *A* generator. Taken in isolation, though, a reasonable estimate may be arrived at. Consider the implementation of the *C* generator in Appendix C. As the sub-component LFSR generators' shift registers are all of 31 bits long, the periods individually are $2^{31} – 1$ bits. "Ganged" 32 across as they are, the equivalent of a generator with a period of $32(2^{31} – 1)$. While the linear complexity is straightforward to estimate at 32 * 31 = 992, which is shorter than |*rp*| in the presented implementation, it is still long enough that, with the regular injection of the re-seeding material, there will not be a sequence of twice that from the <*X*> side of the composite to apply the Berlekamp-Massey algorithm to. Further, as this is used in the bit matching process, the resulting *m* is random in any case.

As this stage is a many-to-one mapping process, working backwards through it is problematic. There will always be multiple plausible solutions. As seen in Chapter 3, the bit-matching process involves a factorial scaled problem, which is in this case compounded with intentional and incidental deletions of bits from the process. Given a guess regarding the contribution of the state vector of *C* to a given nybble of *m* we still must have good guess for the contributions of the constituent generators, and are then left with the problems of guessing how many bits of *m* this may have yielded, if any, and their relative position in *m*. In short, there is at best only marginal necessary synchronization between the value and position of a bit in *m*, and the sequential states *C* went through to produce it from *rp*. Thus, working forward from rp through the matching process is expected to be the far easier process, since the process is deterministic, once a guess is made regarding the state of *C*.

In order have a meet-in-the-middle attack we must now work backwards from *k*, with the assumption that the attacker knows *k*, and its length, as per the strong form of Kerckhoffs' criteria. Though we don't face the same sort of problem as is posed by trying to work backwards through the bit-matching process, we still have the complication of dealing with a many-to-one mapping process in reverse. In this case, we know where in *m* two of the indices used in obtaining a nybble of *k* came from, but we don't know what those indices were, nor which index came from which half of *m*.

Given any known output string *k*, the *m* string used to produce it had to be |*k*| / 4 bytes in length at minimum. From this, |*k*| / 4 nybble index pairs are extracted. Except when the two indices are equal, we again note that we do not know which half of *m* each came from. That means that, when trying to work backwards through the final stage, we have (15 / 16) bits of uncertainty for each nybble of *k*, or 15 bits of uncertainty for each eight bytes in *k*. We regard this as a "self-inflicted wound" of sorts in the cryptanalytic process on top of the fundamental problem of guessing the indices used in the first place. For that part of the problem, the key material in question is the internal state vector of *D*, which is 64 bits in length, six bits of which are used for indexing. The other eight bits used for indexing are from *m*, which is what an attacker is attempting to determine. This means the attacker is left to guess at the nybble pair that, along with the state material, was used to derive a nybble in *k*.

For every four bit nybble value in *k*, there are 1024 index triples that could have produced it, since there 16,384 entries in the *dTable*, as well as the *sTable*, and equal numbers of each nybble value in it. With a guess at the state vectors six bit contribution, there remain 16 different ways to obtain that nybble based upon the other two indices, which is equivalent to an added 4 bits of uncertainty per nybble of *k* on top of the six bits from the guess regarding the contents of the state vector. This adds up to 10.9375 bits of uncertainty for each nybble in *k*, when working backwards, or, for the approximate average case of 29 bytes of *k*, 634.375 bits worth of uncertainty. This exceeds the number of bits in the state vector of *D* by 570.375 bits. Thus, piecemeal guessing regarding the inputs to and state of *D* appears very inefficient. The degree to which such a process can be made more efficient, or exploiting the incompatibilities that will tend to arise with incompatible combinations of indices after the state vector has rotated back to its initial alignment, is yet to be determined.

An alternative to this approach is to guess the initial contents of the state vector of *D*, then determine the pairs of <*X*> and <*Y*> sequence nybbles required to produce the a given nybble of *k*. Since for any 6-bit fragment of *D*'s state vector used there are 16 plausible input nybble pairs that would produce the same result, we have $16^{58} = 2^{62}$ *m* bit-strings (ignoring the end-ordering discussed previously) that are consistent with any guess of the initial state vector contents of *D*, each yielding a different ending state. This appears to mean that the space requirements for storing candidate solutions will rapidly become unwieldy, since for all $2^{64}$ possible initial states of *D* to store the corresponding candidate strings would require space for $2^{122}$ such candidates, assuming 29 bytes in *k*. If we are dealing with an implementation where *A*, *B* and *C* have very long periods, the "middle" would appear to be problematic in its own right.

While we expect that a more efficient approach to this back-analysis of the last stage of the process to exist, it is quite possible that forward analysis (working from *bxrp* to *k*) rather than a meet-in-the-middle attack, will be substantially simpler, and more likely to produce results.

Still, a method not included in the present implementation that might well render a meet-in-the-middle attack to expensive is quite simple. We could, if we chose, add a step following the generation of the complete a, b, c and k bit-strings where the state vectors just used are XOR'ed with the state vector of the next generator to be used. Thus, once we had generated *a* and extracted *r*, we would XOR the A's state vector with that of *B*. After generation of *b*, we would likewise XOR *B*'s state vector into *C*'s, then *C*'s into *D*'s and finally *D*'s back into *A*'s. This would spread the effect of key material and input strings more widely throughout the whole process, likely denying an attacker the ability to separate the phases in a manner that would allow a meet-in-the-middle attack. But, while this might help improve the resistance of the scheme to attack, particularly a meet-in-the-middle attack, it is not inconceivable that it could introduces weaknesses. As we are exploring alternatives, and wish to examine the effectiveness of the components and phases as presented here, that idea is left for future research.

## Differential Cryptanalysis

This form of attack is based upon a careful examination of the behavior of ciphertexts based upon chosen plaintexts, following the composition process through the respective rounds of a cipher algorithm, using a given key. It is a form of chosen plaintext attack, which means the attacker finds some means to induce the target to encrypt plaintexts with desired properties using the same key.

An attacker starts by selecting a set pairs of plaintexts, all displaying fixed differences. When encrypting these plaintexts with the same key, the attacker examines the behavior of the encryption process, round by round. In this process, the attacker will be able to discern differences in the resulting ciphertexts, as well as commonalities, associated with the differences selected in the plaintexts, via which he can assign probabilities to certain keys for any pair of plaintexts possessing only those targeted differences. After some number of such message pairs, dependent upon the cipher used, the attacker will be able to determine the key used.

For this attack to work against the present scheme, we must first accept that we are dealing with an encryption process, and that *k* is in effect a ciphertext. For purposes of argument, this is not difficult to stipulate.  But, the attacker must be able to identify plaintexts and ciphertexts which pair-wise display specific differences, when encrypted using the same key.  Since the key changes for each cycle through the process, and Alice is presumed to have a true random number generator or access to reliable source of such, the attacker is faced with a first problem.  How does he get the requisite plaintext/ciphertext pairs, with the requisite fixed differences in the plaintexts, all using the same key?  That the attacker can and will is difficult to accept as a stipulation.  Being able to subvert use of Alice's RNG or source of randomness requires a level of trust by Alice, or access to Alice's equipment, that makes a differential attack seem a wasted effort.  Likewise, inducing Alice to start all exchanges with the same key/initial state is difficult to envision.  As Bob will have updated his key after any cycle, the next cycle will fail to produce agreed material, as synchronization between Alice's and Bob's copies of the mechanism and states will have been lost, giving both a sure indication that something is wrong.

### Man in the Middle

In this attack, and attacker places himself logically between Alice and Bob, and impersonates each to the other, thereby becoming the conduit by which the two communicate.  If done effectively, this attack can strip all confidentiality from the communications between Alice and Bob, as neither is aware that they are not communicating directly with each other.

As the present scheme assumes that Alice and Bob share identical copies of the initial state, any exchange between them that does not use that shared secret material results in different bit-strings being produced by each.  An attacker is not assumed in this attack to be privy to this information.  If he passes communicated information along unaltered he has accomplished nothing, he is become merely an eavesdropper.  If he alters the *axr* or *bxrp* bit-string, Bob will produce a different bit sequence than Alice.  When Alice and Bob try to communicate using their differing bit sequences, the result will be gibberish to both, making it clear that something is wrong, while the attacker remains in the dark about the messages encrypted using the divergent *k* bit-strings.

To be effective, the man in the middle must be able to get Alice and Bob to use initial states that he knows, whether this is a pair he creates for both, or separate pairs to share with Alice and Bob individually. Under such circumstances, virtually any system is broken, even systems based upon entangled pairs of sub-atomic particles, in the latter case. Thus, again, the problem from which the loss of security arises is far worse than a flaw in the scheme.

## Design Issues

While the implementation presented here relies upon several linear feedback shift registers (LFSR), there is nothing in the scheme that mandates that the component PRNGs of the composite PRNGs *A*, *B* and *C* must be of any particular type. However, as any linear recurrence can be replicated by an LFSR, it is deemed sufficient to base analysis of the state upon LFSRs, and upon the size of the state of the scheme as implemented with them.

The first consideration with regard to any implementation is whether equivalents to the *A*, *B* and *C* PRNGs may be implemented as LFSRs, regardless of the actual implementation of those PRNGs. In the absence of the injected aperiodicity for all three of these composite PRNGs, such analysis is straightforward. Despite the mixing mechanism (which may be viewed as a non-linear combiner) present in all three of the composite PRNGs and PRNG *D*, they would naturally produce linear recursions, being deterministic, and thus would be subject to analysis with the Berlekamp-Massey algorithm, yielding an LFSR replicating the behavior and output. With the injection of random bits into the state vectors at regular intervals, generation of a linear recurrence is avoided, but the issue must be dealt with in greater detail than that.

The present implementation, which is intended as an example and as an object for analysis, uses LSFRs exclusively as the component generators for *A*, *B* and *C*. The manner in which these components are combined varies by the lengths of the individual LFSRs. As per prior discussion, the linear complexity *L* of any of these (the taps selected for being a primitive polynomial) is its length, and the Berlekamp-Massey algorithm will yield an equivalent LFSR of length *L*, given 2*L* bits of sequential output. From this it should be apparent that the higher the linear complexity of the constituent LFSRs the better for the cryptographic strength of the system. Coupled with the discussion regarding meet-in-the-middle attacks, we favor linear complexities greater than 2048, and would implement the scheme thus for real-world applications.

As also noted in the discussion of meet-in-the-middle attacks, a feature considered but not implemented is successive XOR'ing of state vectors, from *A* to *B* to *C* to *D* and back to *A*, in order to distribute the effects of the re-seeding bits more broadly throughout the phases and the resulting agreed bits. This was not done in order to gain a better understanding of any flaws in the main operations of the scheme, once it has been submitted to a much broader audience. Identification of specific flaws in these phases will significantly assist in any subsequent development of the scheme, particularly with regard to correcting weaknesses in the overall scheme that may result from those flaws.

Other possible changes include increasing the lengths of the *axr* bit-string, and the number of bits extracted from the *m* bit-string for re-seeding the *A* generator. While this would reduce the eventual number of agreed bits per cycle, the increased number of bits in *r* and thus available to re-seed *B* and *C* would serve to increase the disruption in the output sequences from *B* and *C*. Additional bits might also be used to directly alter the state vector of *D*. Whether and how much such changes might improve the scheme are open questions, but it can be observed that number of bits injected, and their randomness, serve to select from among possible "jump targets" in the otherwise strictly periodic cycle of the composite generator with the new alignment of bits from <*X*> and <*Y*>. An increase in the number of bits injected will almost certainly improve the preservation of entropy between cycles processing successive *axr* and *bxrp* input string pairs.

Changing the point in the initial, unmodified *m* bit-string where the re-seeding bits for *A* are extracted, so that the tail of the full bytes of matched bits will have a more immediate effect, within the cycle in which they arise, rather than during the processing of the next blocks of input data, may be worthwhile.

Another idea that was considered but not incorporated is to retain the non-selected bits in *rp* during the bit-matching process between bit-strings *c* and *rp*.  This can be easily accomplished by adding one more table to the bit-matching process.  Thus if *n* bits from a nybble matched, 4 – *n* bits would be appended to an *m*' bit string.  Performing a bit-blending operation similar to the present operation of *D*, but using *m* and *m*', the longer of which would provide the re-seeding bits for *A* and the shorter (after removal of the bits for re-seeding *A*) determining the maximum length of the resulting *k*, would serve to increase the output ratio of agreed bits to input bits.  It would lose what is believed to be the very positive effect of the folding of the *m* bit-string as is presently performed in *D*.

A second idea for using the non-matched bits (*m*' as above) is to perform some hash function on these bits to obtain a number of bits that would be used to directly alter the state vectors of some or all of the *A*, *B*, *C* and *D* PRNGs.  Good hash functions can be slow, and as such lead to the exclusion of the idea from the presented scheme and implementation.  Direct alteration of the state vectors would increase the discontinuity between successive utilizations of the affected PRNGs, and likely help in maintaining entropy across multiple cycles.  Such a change has been deferred until an examination of suitable hash functions can be performed

While one of the minor objectives in the design of the presented protocols and implementation was a simple enough process to remain fast, while hopefully remaining reasonably cryptographically strong, the fact that the process is as fast as it is (see performance results in the following section) means that increasing the number of phases can be considered.

## Performance Results

No analysis of an algorithm can avoid certain empirical tests regarding performance and repeatability, and any algorithm intending to produce sequences with a high degree of apparent randomness must deal with the verification of the apparent randomness of the output.  Such testing has been performed on the implementation presented here.

In terms of performance, on a PC running Microsoft Windows 7 ©, Home Basic, with a 1.6 GHz Atom processor and 1 GB of RAM, 1000 runs, processing 1000 blocks per run, required five minutes, 21 seconds, including file I/O and system overhead in processing batch file commands, and produced an average of 29.254 bytes of output per block processed.

Another run totaling the processing of one million blocks was conducted on the same system, using different keys from those in the timing trial above, to again test the average number of outputs per run.  The average was found to be 29.360 bytes per run.

In terms of repeatability, multiple runs starting with the same key and input data were performed to determine whether the outputs remained consistent.  The manner of the testing was as follows.  A key file was generated, and an input file of 64 blocks produced. The algorithm was run on the input file and the results saved, along with the resulting updated key file.  The input file was split in half, and the algorithm run on the first half, using the previously generated initial key.  The results were saved, along with the updated key.  The algorithm was run on the second half using the updated key from processing the first half.  The results at the end of this pass were compared with the previously generated results for processing all 64 blocks, as well as the updated key file from processing all blocks.  These were found to match identically in content.  The input data was quartered, and processing repeated in like manner on the quarters, with comparisons of results and updated keys at mid and end points.  These were again found to match.  This process was repeated with cutting the input file into eighths and sixteenths, comparing intermediate results and keys where applicable, and all producing the expected identical results.

For ten runs each from the two above referenced systems, using the same initial keys and input files for corresponding runs, the results different systems were compared.  These were found to be identical.

Subjecting the outputs from ten runs, each on distinct input files of 64 blocks and distinct keys, to the BSI AIS 20 test suite produced passes for all ten files.

Two sets of five runs each on 64 blocks were performed, with matching input files pairs per run but different key files, and the five result file pairs XOR'ed, then the resulting files tested using the BSI AIS 20 test suite.  All five files passed.

Five runs each on two different sets of 64 block files, with identical keys pair-wise between batches, were performed and the results XOR'ed together and tested with the BSI AIS 20 test suite.  All five of these files passed.

From the above empirical testing, we conclude that the algorithm is consistent and repeatable, and with random keys and input streams produce good apparently random results, within the limits of the BSI AIS 20 test suite's limitations.

# REFERENCES

[1] Shapiro, C. and Varian, H.R., *Information Rules*, Harvard Business Press, 1999, ISBN 087584863X.

[2] Bishop, M. and Venkatramaanayya, S. S., *Introduction to Computer Security*, Pearson Education, Inc., 2005.

[3] Kerckoff, Auguste, “La cryptographie militaries,” *Journal des Sciences Militaires*, vol. IX, Jan. 1883, pages 5-83.

[4] Kerckoff, Auguste, “La cryptographie militaries,” *Journal des Sciences Militaires*, vol. IX, Feb. 1883, pages 161-191.

[5] Shannon, Claude. "Communication Theory of Secrecy Systems", *Bell System Technical Journal*, vol.28(4), 1949, pp. 656 – 715.

[6] Diffie, Whitfield, and Hellman, Martin E., “New Directions in Cryptography,” *IEEE Transactions on Information Theory*, vol. 22, Nov. 1976, pp. 644 – 654.

[7] Hellman, M. E., Diffie, W., and Merkle, R. C., “Cryptographic Apparatus and Method,” *U.S. Patent #4,200,770*, 29 April 1980.

[8] Hellman, M.E., Diffie, W., and Merkle, R.C., ““Cryptographic Apparatus and Method,” *Canada Patent #1,121,480*, 6 April 1982.

[9] den Boer, B., “Diffie-Hellman is as strong as discrete log for certain primes,” *Crypto 88, Lecture Notes in Computer Science 403*, Springer-Verlag, 1988.

[10] Boney, D., and Lipton, R. J., “Algorithms for black-box fields and their application to cryptography,” *Advances in Cryptography, CRYPTO 96, Lecture Notes in Computer Science 1070*, Springer-Verlag, 1996, pp. 283 – 297.

[11] Rivest, R., Shamir, A. and Adleman, L., “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” Communications of the ACM 21 (2), 1978, pp 120 – 126.

[12] Various Authors, “Trust and Security Challenges in Cyberspace,” from Brussels Workshop, 7-8 Dec 2000.

[13] Shor, Peter W., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM J. Comput.* 26 (5), 1997, pp. 1484–1509.

[14] Kahn, David, *The Codebreakers: The Comprehensive History of Secret Communications*, second edition, Scribner, 1996.

[15] Bennett, C. H., and Brassard, G., “Quantum Cryptography: Public key distribution and coin tossing,” *Proceedings of the IEEE International conference on Computers, Systems and Signal Processing*, Bangalore, 1984, p. 175.

[16] Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D. and Whiting, D.,”Improved Cryptanalysis of Rijndael,” Fast Software Encryption, 2000, pp. 213 – 230.

[17] Biryukom, A. and Khovratovich, D., “Related-key Cryptanalysis of the Full AES-192 and AES-256,” http://eprint.iacr.org/2009/317

[18] “Security Requirements for Cryptographic Modules,” Federal Information Processing Standard 140-2, National Institute of Standards and Technology, Dec. 2, 2002.

[19] Schindler, W., “Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators: AIS 20,” version 1, Bundesamt für Sicherheit in der Informationstechnik, Dec. 2, 1999.

[20] Pohlig, S.C., and Hellman, M.E., “An Improved Algorithm for Computing Logarithms in GF(*p*) and Its Cryptographic Significance,” *IEEE Transactions on Information Theory*, v. 24, n. 1, Jan 1978, pp. 106 – 111.

[21] Menezes, A. J., van Oorschot, P. C. and Vanstone, S. A., *Handbook of Applied Cryptography*, CRC Press, 1997.

[22] Merkle, R.C. and Hellman, M.E., “Hiding Information and Signatures in Trapdoor Knapsacks,” *IEEE Transactions on Information Theory*, v. 24, n 5, Sep. 1978, pp. 525 – 530.

[23] Hellman, M.E., “The Mathematics of Public-Key Cryptography,” *Scientific American*, v. 241, n. 8, Aug. 1979, pp. 146 – 157.

[24] Shamir, A., “A Fast Signature Scheme,” MIT Laboratory for Computer Science, Technical Memorandum, MIT/LCS/TM-107, Massachusetts Institute of Technology, July 1978.

[25] Schneier, B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition, John Wiley & Sons, Inc., 1996.

[26] Shor, P., "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," Proceedings 35th Annual Symposium on Foundations of Computer Science (1994), pp. 124 – 134.

[27] Hellman, M.E. and Pohlig, S.C., "Exponentiation Cryptographic Apparatus and Method," U.S. Patent #4,424,414, 3 Jan. 1984.

[28] Rabin, M.O., "Digital Signatures and Public-Key Functions as Intractable as Factorizations," MIT Laboratory for Computer Science, Technical Report, MIT/LCS/TR-212, Jan 1979.

[29] Williams, E.A., "A Modification fo the RSA Public-Key Encryption Procedure," *IEEE Transactions on Information Theory*, v IT-26, n. 6, Nov. 1980, pp. 726 – 729.

[30] ElGamal, T., "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 10 – 18.

[31] ElGamal, T., "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory*, v. IT-31, n.4, 1985, pp. 469 – 472.

[32] McEliece, R.J., "A Public-Key Crytosystem Based on Algebraic Coding Theory," Deep Space Network Progress Report 42 – 44, Jet Propulsion Laboratory, California Institute of Technology, 1978, pp. 114 – 116.

[33] Korzhik, V.I. and Turkin, A.I., “Cryptanalysis of McEliece’s Public-Key Cryptosystem,” *Advances in Cryptology – EUROCRYPT ’91 Proceedings*, Springer-Verlag, 1991, pp. 68 – 70.

[34] Bernstein, D.J., Lange, T. and Peters, C., "Attacking and Defending the McEliece Cryptosystem," *Proceedings of the 2nd International Workshop on Post-Quantum Cryptography*, Lecture Notes In Computer Science 5299, 2008, pp. 31 – 46.

[35] Chabaud, F, “On the Security of Some Cryptosystems Based on Error-Correcting Codes,” *Advances in Cryptology, EUROCRYPT ‘94 Proceedings*, Springer-Verlag, 1995, pp. 131 – 139.

[36] Koblitz, N., “Elliptic Curve Cryptosystems,” *Mathematics of Computation*, v. 48, n. 177, 1987, pp. 203 – 209.

[37] Miller, V.S., “Use of Elliptic Curves in Cryptography,” *Advances in Cryptology – CRYPTO ’85 Proceedings*, Springer-Verlag, 1986, pp. 417 – 426.

[38] Rosing, M., *Implementing Elliptic Curve Cryptography*, Manning Publications, 1998.

[39] Hankerson, D., Menezes, A.J. and Vanstone, S., *Guide to Elliptic Curve Cryptography*, Springer, 2004.

[40] Kravitz, D. and Reed, I., “Extensions of RSA Cryptostructure: A Galois Approach,” *Electronics Letters*, v. 18, n. 6, March 1982, pp. 255 – 256.

[41] Müller, W.B. and Nöbauer, W., “Some Remarks on Public-Key Cryptography,” *Studia Scientiarum Mathematicarum Hungarica*, v. 16, 1981, pp. 71-76.

[42] Müller, W.B. and Nöbauer, W., “Cryptanalysis of the Dickson Scheme,” *Andvances in Cryptology – EUROCRYPT ’85 Proceedings*, Springer-Verlag, 1986, pp. 50 – 61.

[43] Müller, W.B., “Polynomial Functions in Modern Cryptology,” *Contributions to General Algebra 3: Proceedings of the Vienna Conference*, Vienna: Hölder-Pichler-Tempsky, 1985, pp. 7 – 32.

[44] Lidl, R. and Müller, W.B., “Permutation Polynomials in RSA-Cryptosystems,” *Advances in Cryptology: Proceedings of Crypto 83*, Plenum Press, 1984, pp. 293 – 301.

[45] Smith, P., “LUC Public-Key Encryption,” *Dr. Dobb’s Journal*, v. 18, n. 1, Jan. 1993, pp 44 – 49.

[46] Smith, P. and Lennon, M., “LUC: A New Public Key System,” *Proceedings of the Ninth International Conference on Information Security, IFIP/sec 1993*, North Holland: Elsevier Science Publishers, 1993, pp. 91 – 111.

[47] Renji, T. and Shihua, C., “A Finite Automaton Public Key Cryptosystem and Digital Signatures,” *Chinese Journal of Computers*, v. 8, 1985, pp. 401 – 409.

[48] Renji, T. and Shihua, C., “Two Varieties of Finite Automaton Public Key Cryptosystems and Digital Signature,” *Journal of Computer Science and Technology*, v. 1, 1986, pp. 9 – 18.

[49] Renji, T. and Shihua, C., “An Implementation of Indentity-based Cryptosystems and Signature Schemes by Finite Automaton Public Key Cryptosystems,” *CHINACRYPT ’92*, Bejing Science Press, 1992, pp. 87 – 104.

[50] Renji, T. and Shihua, C., "Note on Finite Automaton Public Key Cryptosystems," *CHINACRYPT '94*, Xidian, China, 11 – 15 Nov. 1994, pp. 76 – 80.

[51] Bellovin, S.M. and Merritt, M., "Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks," *Proceedings of the1992 IEEE Symposium on Security and Privacy*, IEEE, May 1992, p. 72ff.

[52] Bellovin, S.M. and Merritt, M., "," *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ACM Press, 1993, pp. 244 – 250.

[53] Jablon, D., "Strong Password-Only Authenticated Key Exchange," *Computer Communication Review (ACM SIGCOMM) 26 (5)*, 1996, pp. 5 – 26.

[54] Jablon, D., "Extended Password Key Exchange Protocols Immune to Dictionary Attack," *Proceedings of the Sixth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE '97)*, IEEE Computer Society, 1997, pp. 248 – 255.

[55] Blum, M., Feldman, P. and Micali, S., "Non-Interactive Zero-Knowledge and Its Applications," *STOC '88 Proceedings of the twentieth annual ACM symposium on Theory of computing*, ACM, 1988, pp. 103 – 112.

[56] Rueppel, R.A., *Analysis and Design of Stream Ciphers*, Springer-Verlag, 1986.

[57] Massey, J.L., "Shift-Register Synthesis and BCH Decoding," *IEEE Transactions on Information Theory*, v. IT-15, n. 1, Jan. 1969, pp 122 – 127.

[58] Jansen, C.J.A, and Boekee, D.E., "The shortest feedback shift register that can generate a given sequence," *Advances in Cryptography – EUROCRYPT '89 (LNCS 435)*, Springer-Verlag, 1990, pp. 90 – 99.

[59] Ziv, J. and Lempel, A., "On the complexity of finite sequences," *IEEE Transactions on Information Theory*, 22, 1976, pp. 75 – 81.

[60] Geffe, P.R., "How to Protect Data With Ciphers That are Really Hard to Break," *Electronics*, v. 46, n. 1, Jan. 1973, pp. 99 – 101.

[61] Key, E.L., "An Analysis of the Structure and Complexity of Nonlinear Binary Sequence Generators," *IEEE Transactions on Information Theory*, v. IT-22, n. 6, Nov. 1976, pp. 732 – 736.

[62] Zeng, K.C., Yang, C.-H. and Rao, T.R.N., "On the Linear Consistency Test (LCT) in Cryptanalysis with Applications," *Advances in Cryptology – Crypto '89 Proceedings*, Springer-Verlag, 1990, pp. 164 – 174.

[63] Zeng, K.C., Yang, C.-H., Wei, D.-Y. and Rao, T.R.N., "Pseudorandom Bit Generators in Stream-Cipher Cryptography," *IEEE Computer*, v. 24, n. 2, Feb. 1991, pp. 8 – 17.

[64] Pless, V.S., "Encryption Schemes for Computer Confidentiality," *IEEE Transactions on Computing*, v. C-26, n. 11, Nov. 1977, pp. 1133 – 1136.

[65] Rubin, F, "Decrypting a Stream Cipher Based on J-K Flip-Flops," *IEEE Transactions on Computing*, v. C-28, n. 7, Jul. 1979, pp. 483 – 487.

[66] Siegenthaler, T., "Decrypting a Class of Stream Ciphers Using Ciphertext Only," *IEEE Transactions on Computing*, v. C-34, Jan. 1985, pp. 81 – 85.

[67] Jennings, S.M., “A Special Class of Binary Sequences,” Ph.D. Dissertation, University of London, 1980.

[68] Jennings, S.M., “Multiplexed Sequences: Some Properties of the Minimum Polynomial,” *Lecture Notes in Computer Science 149; Cryptography: Proceedings of the Workshop on Cryptography*, Springer-Verlag, 1983, pp. 189 – 206.

[69] Jennings, S.M., “Autocorrelation Function of the Multiplexed Sequence,” *IEEE Proceedings*, v 131, n. 2, Apr. 1984, pp. 169-172.

[70] Anderson, R.J., “Solving a Class of Stream Ciphers,” *Cryptologia*, v. 14, n. 3, Jul. 1990, pp. 285 – 288.

[71] Dawson, E. and Clark, A., “Cryptanalysis of Universal Logic Sequences,” *Advances in Cryptology – EUROCRYPT ’93 Proceedings*, Springer-Verlag, pre-print, publication date unknown.

[72] Beth, T., and Piper, F.C., “The Stop-and-Go Generator,” *Advances in Cryptography: Proceedings of EUROCRYPT 84*, Springer-Verlag, 1984, pp. 88 – 92.

[73] Gunther, C.G., “Alternating Step Generators Controlled by De Bruijn Sequences,” *Advances in Cryptology – EUROCRYPT ’87 Proceedings*, Springer-Verlag, 1988, pp. 5 – 14.

[74] Strobel, D., “Side Channel Analysis Attacks on Stream Ciphers,” Masters Thesis, Ruhr-Universität Bochum, Mar. 2009.

[75] Rueppel, R.A., “When Shift Registers Clock Themselves,” *Advances in Cryptology – EUROCRYPT ’87 Proceedings*, Spinger-Verlag, 1987, pp. 53 – 64.

[76] Chambers, W.G. and Gollman, D., “Generators for Sequences with Near-Maximal Linear Equivalence,” *IEEE Proceedings*, v. 135, pt. E, n. 1, Jan. 1988, pp. 67 – 69.

[77] Coppersmith, D. and Grossman, E., “Generators for Certain Alternating Groups of Applications to Cryptography,” *SIAM Journal on Applied Mathematics*, v. 29, n. 4, Dec. 1975, pp. 624 – 627.

[78] Coppersmith, D., Krawczyk, H. and Mansour, Y., “The Shrinking Generator,” *Advances in Cryptology – CRYPTO ’93 Proceedings*, Springer-Verlag, 1994, pp. 22 – 39.

[79] Meier, W., “On the Security of the IDEA Block Cipher,” *Advances in Cryptology – EUROCRYPT ’93 Proceedings*, Springer-Verlag, 1994, pp. 371 – 385.

[80] Massey, J.L. and Rueppel, R.A. “Linear Ciphers and Random Sequence Generators with Multiple Clocks,” *Advances in Cryptology: Proceedings of EUROCRYPT 84*, Springer-Verlag, 1985, pp. 74 – 87.

[81] Gollman, D., “Kaskadenschaltungen taktgesteuerter Schreiberegister als Pseudozufallszahlengeneratoren,” Ph.D. dissertation, Universitat Linz, 1983.

[82] Chambers, W.G. and Gollman, D., “Lock-In Effect in Cascades of Clock-Controlled Shift Registers,” *Advances in Cryptology – EUROCRYPT ’88 Proceedings*, Springer-Verlag, 1988, pp. 331 – 343.

[83] Bruer, J.O., “On Pseudo Random Sequences as Crypto Generators,” *Proceedings of the International Zurich Seminar on Digital Communications*, Switzerland, 1984.

[84] Rueppel, R.A., “Correlation Immunity and the Summation Combiner,” *Advances in Cryptology – EUROCRYPT ’85*, Springer-Verlag, 1986, pp. 260 – 272.

[85] Klapper, A. and Goresky, M., “2-adic Shift Registers,” *Fast Software Encryption, Cambridge Security Workshop Proceedings,* Springer-Verlag, 1994, pp. 174 – 178.

[86] Meier, W. and Staffelbach, O., “Correlation Properties of Combiners with Memory in Stream Ciphers,” *Advances in Cryptography – EUROCRYPT ’90 Poceedings*, Springer-Verlag, 1991, pp. 204 – 213.

[87] Meier, W. and Staffelbach, O., “Correlation Properties of Combiners with Memory in Stream Ciphers,” *Journal of Crytology*, v. 5, n. 1, 1992, pp. 67 – 86..

[88] Mihajlevic, M.J. and Golic, J.D., “Convergence of a Bayesian Iterative Error-Correction Procedure to a Noisy Shift Register Sequence,” *Advances in Cryptology – EUROCRYPT ’92 Proceedings*, Springer-Verlag, 1993, pp 124 – 137.

[89] Goresky, M. and Klapper, A., “Feedback Registes Based on Ramified Extension of the 2-adic Numbers,” *Advances in Cryptology – EUROCRYPT ’94 Proceedings*, Springer-Verlag, 1995, unknown.

[90] Klapper, A. and Goresky, M., “Feedback with Carry Shift Registers over Finite Fields,” *K.U. Leuven Workshop on Cryptographic Algorithms*, Springer-Verlag, 1995.

[91] Klapper, A. and Goresky, M., “Large Period Nearly de Bruijn FCSR Sequences,” *Advances in Cryptology – EUROCRYPT ’95 Proceedings*, Springer-Verlag, 1995, pp 263 – 273.

[92] Xu, S.B, He, D.K. and Wang, X.M., “An Implementation of the GSM General Data Encryption Algorithm A5,” *CHINACRYPT ’94*, Xadian, China, 11 – 15 Nov. 1994, pp. 287 – 291.

[93] Anderson, R.J., “On Fibonacci Keystream Generators,” *K.U. Leuven Workshop on Cryptographic Algorithms*, Springer-Verlag, unknown.

[94] Gueneysu, T., Kasper, T., Novotný, M., Paar, C. and Rupp, A., "Cryptanalysis with COPACOBANA". Transactions on Computers, 57, Nov. 2008, pp. 1498 – 1513.

[95] Barkan, E., Biham, E. and Keller, N., "Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication," *Crypto 2003*, 2003, pp. 600 – 616.

[96] Coppersmith, D. and Rogaway, P., “SOFTWARE EFFICIENT PSEUDORANDOM FUNCTION AND THE USE THEREOF FOR ENCRYPTION,” U.S. Patent 5,454,039, 26 Sept., 1995.

[97] Coppersmith, D. and Rogaway, P., “COMPUTER READABLE DEVICE IMPLEMENTING A SOFTWARE-EFFICIENT PSEUDORANDOM FUNCTION ENCRYPTION,” U.S. Patent 5,675,652, 7 Oct., 1997.

[98] Halevi, S., Coppersmith, D. and Jutla, C., “Scream: a software-efficient stream cipher,” IBM T.J. Watson Research Center, June 5, 2002.

[99] Handschuh, H. and Gilbert, H., “$\chi^2$ Cryptanalysis of the SEAL Encryption Algorithm,” *Fast Software Encryption, FSE ’97, LNCS*, 1997, pp. 1 – 12.

[100] Fluhrer, S.R., “Cryptanalysis of the SEAL 3.0 Psuedorandom Function Family,” *Lecture Notes in Computer Science, 2002*, vol. 2355/2002, 2002, pp. 333 – 334.

[101] Wheeler, D.J., “A Bulk Data Encryption Algorithm,” *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 127 – 134.

[102] Shamir, A., “On the Generation of Cryptographically Strong Pseudo-Random Sequences,” *Lecture Notes in Computer Science 62: 8th International Colloquium on Automata, Languages, and Programming*, Spring Verlag, 1981.

[103] Blum, M. and Micali, S., “How to Generate Cryptogrphically-Strong Sequences of Pseudo-Random Bits,” *SIAM Journal of Computing*, v. 13, n. 4, Nov. 1984, pp. 850 – 864.

[104] Alexi, W, Chor, B.-Z., Goldreich, O., and Schnorr, C.P., “RSA and Rabin Functions: Certain Parts Are as Hard as the Whole,” *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*, 1984, pp. 449 – 457/

[105] Alexi, W, Chor, B.-Z., Goldreich, O., and Schnorr, C.P., “RSA and Rabin Functions: Certain Parts Are as Hard as the Whole,” *SIAM Journal on Computing*, v. 17, n. 2, April 1988, pp. 194 – 209.

[106] Blum, L., Blum, M. and Shub, M., “A Simple Unpredictable Pseudo-Random Number Generator,” *SIAM Journal on Computering*, v. 15, n. 2, 1986, pp. 364 – 383.

[107] Lehmer, D., “Mathematical Methods in Large-Scale Computing Units,” *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machines*, Harvard University Press, 1951, pp. 141 – 146.

[108] Knuth, Donald E., *The Art of Computer Programming: Seminumerical Algorithms*, second edition, Addison-Wesley, 1981.

[109] Plumstead, J. Boyar, “Inferring a sequence generated by a linear congruence,” *Proceedings of the IEEE 23rd Annual Symposium on Foundations of Computer Science*, 1982, pp. 153 – 159.

[110] Plumstead, J. Boyar, “Inferring a sequence generated by a linear congruence,” *Advances in Cryptology – Proceedings of Crypto ‘82*, 1983, pp. 317 – 319.

[111] Lagarias, J.C. and Reed, J., “Unique Extrapolation of Polynomial Recurrences,” *SIAM Journal on Computing*, v. 17, n. 2, April 1988, pp. 342 – 362.

[112] Krawczyk, H., “How to Predict Congruential Generator,” *Advances in Cryptology – CRYTO ’89 Proceedings*, Springer-Verlag, 1990, pp. 138 – 153.

[113] Krawczyk, H., “How to Predict Congruential Generator,” *Journal of Algorithms*, v. 13, n. 4, December 1992, pp. 527 – 545.

[115] MacLaren M. D. and Marsaglia G., Uniform Random Number Generators, Journal of the Association for Computing Machinery, vol. 12, N 1, Jan. 1965.

[116] Retter, Charles T., “Cryptanalysis of a McLaren-Marsaglia System,” *Cryptologia*, volume 8, number 2, April 1984, pages 97-108.

[117] Retter, Charles T., “A Key-search Attack on McLaren-Marsaglia Systems,” Cryptologia, volume 9, number 2, April 1985, pages 114-130.

[118] Bays, C. and Durham, S.D., “Improving a Poor Random Number Generator,” *ACM Transactions on Mathematical Software*, v. 2, n. 1, March 1976, pp. 59 – 64.

[119] Tyanev, D., Petkova Y., Tyaneva A., "New Elements in the Method of McLaren-Marsaglia," *Conference Proceedings of TEHNONAV 2002*, 2002, p. 402 – 404.

[120] Bailey, D., Borwein, P. and Plouffe, S., “On the rapid computation of various polylogarithmic constants,”*Mathematics of Computation*, vol. 66, no. 218, April 1997, pp. 903 – 913.

[121] Gilmore, J., *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*, Electronic Frontiers Foundation, 1998.

[122] “DES MODES OF OPERATION,” Federal Information Processing Standard 81, National Institute of Standards and Technology, Dec. 2, 1980, revised Nov. 20, 1981.

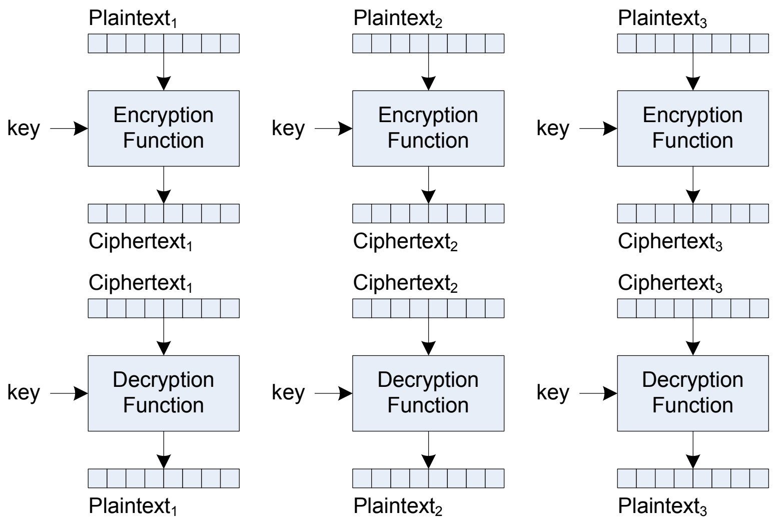
.

APPPENDICES

## APPENDIX A – BLOCK CIPHER MODES

For detailed discussions of block ciphers modes, see [122][21] and [25].

Symmetric block ciphers (referred to here as block ciphers) are typically defined in terms of a binary bit-block size *n*, a keyspace *K* (we are not concerned with the length of a key), an encryption function $E : V_n \times K \rightarrow V_n$ (where $V_n$ is a bit vector of *n* bits), and a decryption function $D : V_n \times K \rightarrow V_n$, such that $D(E(\mathrm{M}, k), k) = M = E(D(M, k), k)$, where *M* is an *n*-bit block, and $k \in K$. For a given key *k*, if any two plaintext blocks are equal, the resulting ciphertext blocks will be equal, and vice versa, since for that key *k* the encryption and decryption functions define one-to-one mappings between plaintext blocks and ciphertext blocks.

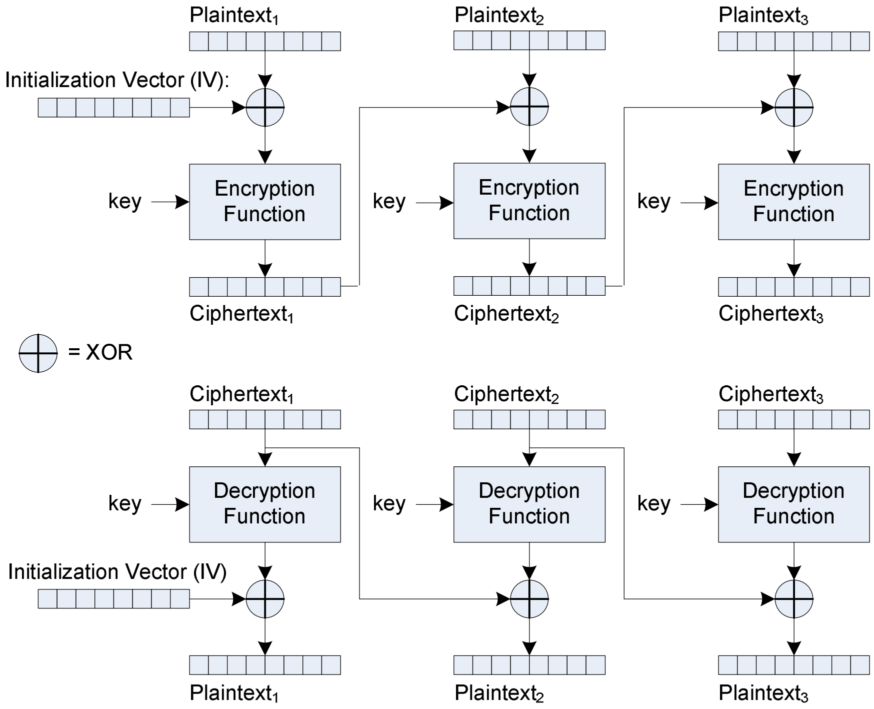


Electornic Codebook (ECB) Mode Encryption and Decryption

Encryption of multiple blocks in a single message (whether text, an image or anything else) using only the plaintext broken into *n*-bit blocks and the key *k* is said to be performed in ***electronic codebook*** (ECB) mode. While this is certainly a legitimate way to encrypt a message or a file, it is broadly agreed that it is not a good idea. One reason for this is the one-to-one mapping between plaintext and ciphertext. If two blocks in a long plaintext are the same, the resulting ciphertext blocks will be the same. This is particularly noticeable when encrypting certain types of diagrams and images. In many cases, the nature of the diagram will be clearly identifiable in the encrypted message or file when the ciphertext is viewed as if it were an image of the correct dimensions. And, there is worse that can happen. Suppose Mike is in a position to conduct a man-in-the-middle attack, and further is able to obtain the plaintexts corresponding to any ciphertext Alice wishes to send to Bob. It may well be possible for Mike to rearrange the ciphertext blocks, insert duplicates of some blocks and delete others in a manner that will allow him to completely change the meaning of the message Bob will decrypt, without having to know the key used by Alice to encrypt the plaintext. If Mike does not have access to the plaintexts, he can still garble the message, by inserting or deleting ciphertext blocks, as well as replacing blocks. In the former case, Bob may have no way of knowing that the message he decrypts is not what Alice intended, while in the latter case Alice and Bob may decide they are dealing with a man in the middle, or conclude that there is a problem with the communications medium or such.

To deal with some of the problems that may arise when using ECB mode encryption, it is common to superimpose operating modes on top of the encryption and decryption functions. Many operating modes have been developed, but some are more common than others. Four, including ECB mode, are defined in FIPS 81 [122].

One common mode is cipher-block chaining (CBC) mode. In this mode, encryption is initiated by XOR'ing the first plaintext block with an initialization vector (IV) of equal length. This IV may be secret, if the correspondents wish, but is typically not a secret. The result of this XOR operation is then encrypted using a session key, thus producing the first ciphertext block. This is then XOR’ed with the next plaintext block, the result of which is encrypted using the session key to produce the second ciphertext block. The process is repeated, with each ciphertext block being XOR’ed with the next plaintext block. The following diagram illustrates both the encryption process and the decryption process.



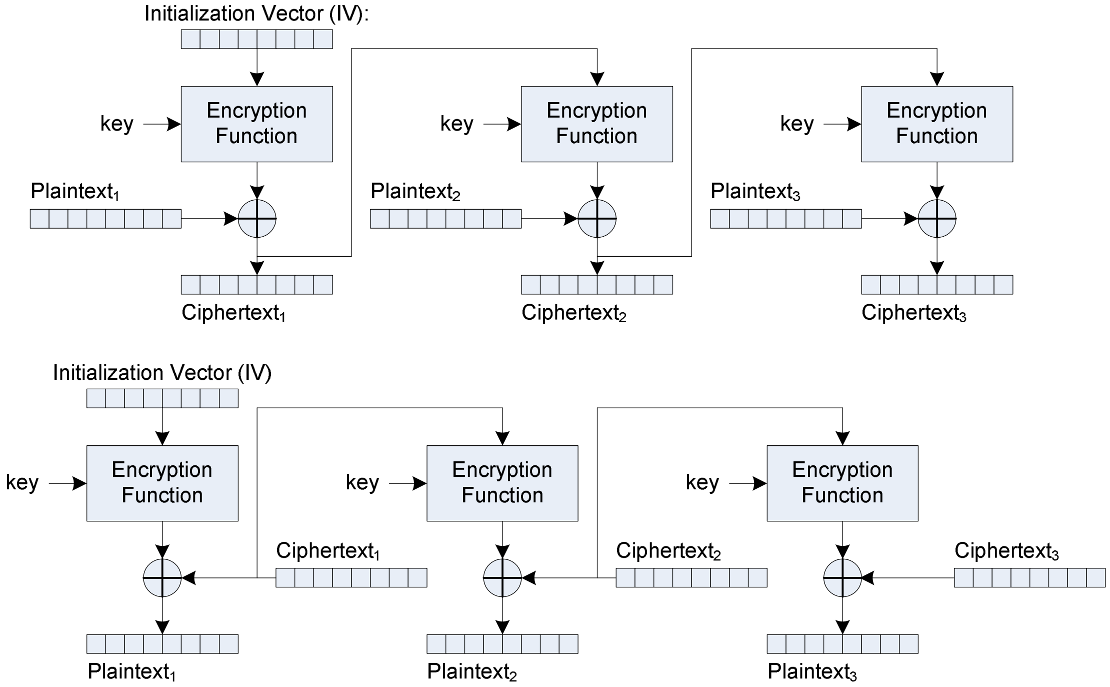
Cipher-block Chaining (CBC) Mode Encryption and Decryption

This approach has several advantages over ECB mode. First, each ciphertext block is the product of encrypting more than just a plaintext block, incorporating the previous ciphertext block into each block encryption. Thus, there is a greatly reduced risk that two plaintext blocks that are identical will produce the same ciphertext. Also, even if Mike has access to the corresponding plaintext, he cannot rearrange the ciphertext blocks to alter the meaning of the plaintext, since rearrangement will result in both the moved block and the one following it (in both its original and new locations) being indecipherable.

In CBC mode, even if a packet is garbled due to problems in a communications network, once "clean" ciphertext blocks start to arrive, decryption can continue, since the first undamaged ciphertext block received will allow successful decryption of the next ciphertext block, if it too is undamaged.

One drawback of CBC mode is that random read/write access in a CBC-mode encrypted file is problematic. Changes in any one ciphertext block are propagated through all succeeding blocks. If the plaintext of a single block must be changed, all subsequent blocks in the file must first be decrypted. They must then re-encrypted serially starting from the altered block's ciphertext. Also, bit errors in communication that result in framing errors are irrecoverable without trial and error in re-establishing the correct framing following the lots bits.

Another commonly used mode is cipher feedback (CFB) mode, which is illustrated in the following diagram.



Cipher Feedback (CFB) Mode Encryption and Decryption

Careful examination of CFB mode reveals that it converts a symmetric block cipher into a self-synchronized stream cipher. Note also that the decryption function is not used, since use of it would not produce identical results for XOR'ing with the current ciphertext block to yield the plaintext.

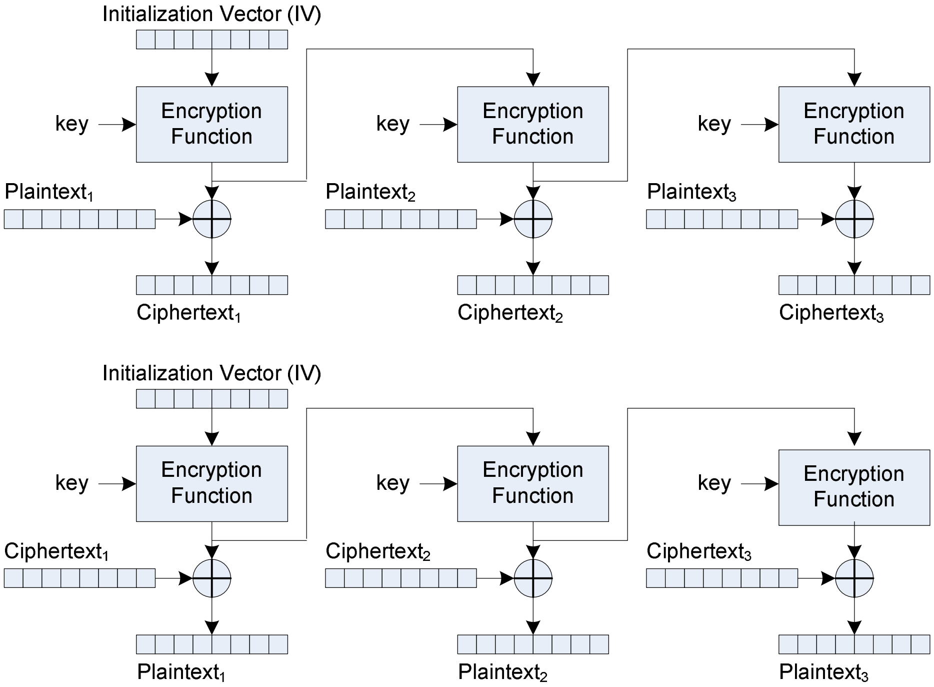
Like CBC mode, alteration of the decrypted plaintext via insertions, deletions and rearrangements are prevented byte CFB mode. But, CFB mode also allows real-time communication, since decryption of a ciphertext block can begin without having received the complete block. This is at the cost of losing the diffusion of plaintext bits within the resulting ciphertext blocks,

As with CBC mode, CFB mode is not suitable in cases where random read/write access to an encrypted file is required, as again, changes to any one ciphertext block will propagate through all succeeding blocks.

***Output feedback*** (OFB) mode converts a symmetric block cipher into a synchronous stream cipher running key (key-stream) generator, as can be seen in the diagram of the encryption and decryption processes on the next page. This is a straight-forward Vernam cipher.

As may be seen in that diagram, an IV is used, but the output for the encryption of the IV is used directly as portion of a synchronous stream cipher running key, and is also fed into the encryption function to produce the next block of running key bits/bytes. Again, this mode is not suitable where random read-write access within an encrypted file is required, but this time because one must generate the running key from the IV to the output block used to encrypt the block to be edited.

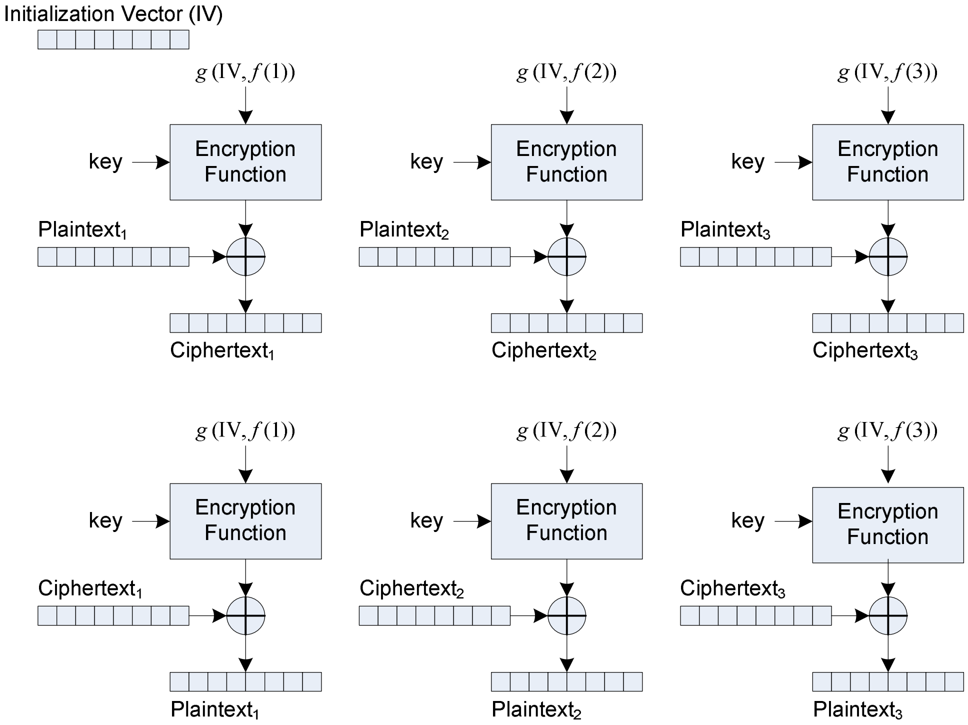
Notice that, like CFB mode, both encryption and decryption in OFP mode use the encryption function of the block cipher, since the processes must produce the same running key stream in order to maintain the symmetry of encryption and decryption. Further, and also as in CFB mode, there is no diffusion of the plaintext within the individual ciphertext blocks.



Output Feedback (OFB) Mode Encryption and Decryption

***Counter mode*** (either CTR or CM) is yet another mode that converts a block cipher into a synchronous stream cipher key generator. The result is again a straight-forward Vernam cipher. As may be seen in the diagram on the next page, the key is used to encrypt an IV that has been combined with a running count, here shown as the function $f(n)$, where $n$ is the block number in both the plaintext and the resulting ciphertext. Ordinarily $f(n) = n$. But, this is not necessarily the case. All that is required is that both Alice and Bob be able to easily generate the value $f(n)$, so as to maintain synchronization. The function $f$ can as easily produce a pseudo random sequence.

So long as a user can easily generate $f(n)$ for arbitrary $n$, without computing all prior values, CTR mode can be used to encrypt files requiring read-write access, though insertion or deletion of material still represents a problem. Also, there is considerable flexibility in how the IV (also sometimes called a nonce) and the counter value may be combined. If the sum of their lengths equals $n$, the block length, they can be concatenated. Or, they may be added, XOR'ed, subtracted, hashed, or any other agreed operation yielding the requisite number of bits. Thus, as in the diagram, we can simply view a function $g : V_o \times V_p \to V_n$, where *o* and *p* are the lengths of the IV and counter bit-blocks, respectively. We can then use *g*(IV, *f*(*n*)) to indicate the nonce used in generating the running key for block *n* of both the plaintext and ciphertext.



Counter (CTR or CM) Mode Encryption and Decryption

Both OFB and CTR modes allow pre-computation of running key sequences, and thus are highly suitable for real-time applications, so long as the average data rate of the encrypted traffic does not exceed the ability to compute the running key ahead of need.

For block ciphers, the use of the secret key depends upon the mode used. In the modes that are in fact stream ciphers, the same problem as with stream cipher keys pertains: reinitiating communications with the same key results in use of a running key that begins at the same point in its cycle as a prior initialization with that key. This is a very bad idea, as discussed in the context of Vernam ciphers.  If an attacker XOR's two message segments enciphered with the same running key, the result is the XOR of the two plaintexts.  It is important to remember, though, the distinction between the key used to initialize the cipher and the running key the cipher algorithm produces.  It is the re-initiation with the same key that is the problem, not continued communication using the same running key, but continued from the last state of the cipher algorithm.

For "pure" block cipher modes such as ECB and CBC modes (and not CFB, OFB, CTR or other modes that are effectively stream ciphers), a key may be re-used, provided one is cautious about such re-use.  For example, if two plaintexts contain identical first blocks and are encrypted with the same key the first ciphertexts blocks with also be identical.  This will continue for CBC as long as the successive plaintext blocks are identical, but will end with the first non-matching plaintext blocks.  For ECB the situation is, as previously stated, much worse, as identical plaintext blocks encrypt identically, regardless of position, when using the same key.

For OFB and CTR, all identical blocks in the same positions in the plaintexts will encrypt identically, when using the same key.  The situation is more complex with CFB, as the encryption of any block depends upon the prior block.  Thus, the second and successive of two or more consecutive identical plaintext blocks (in content and placement) will encrypt identically, when using the same key.

Excluding ECB, it is possible to generalize the observations above regarding CFB, OFB and CTR to block cipher modes that produce stream ciphers: key re-use is not a good idea, just as is the case with Vernam ciphers generally.  But, this must be understood in terms of re-use in the form of re-initialization of the cipher state.  Provided the cycle of the running key generated using a given cipher key is not exhausted, continued use of the running key from the last state reached is not a problem, in and of itself.

## *Appendix B: Test Code*

The following pages contain the C language source code for the Bundesamt für Sicherheit in der Informationstechnik BSI AIS 20 test of random behavior. The file is a “free standing” source file in that it requires no additional header files than the standard C library headers specified in the source file itself.

```
//=============================================================================
// BSI_test_suite.cpp
//
//   This file contains the source code for an implementation of the required
// tests enumerated and described in the following document.
//
//     Bundesamt für Sicherheit in der Informationstechnik (BSI):
//     Application Notes and Interpretation of the Scheme (AIS),
//     AIS 20, Version 1,
//     2 December, 1999.
//
//   This test suite is essentially stripped of most error detection code, so
// as to improve the clarity of the source code.  Equally, efficiency is some-
// what sacrificed for clarity.  While this set of tests is stated to be man-
// datory by the BSI, it should not be taken as a definitive test suite.  Other
// organizations, standards and specifications exist and differ from this set
// of test.  Many have good rationales.  Still, as a mandatory set, these tests
// serve as a sound basis for evaluating the apparent randomness of any pseudo-
// random number generator.
//
//   Compilation and linking of this file, and the naming of the resultant
// executable is operating system and user dependent.  As such, the user is
// left to perform such actions.  However, this file has been compiled and run
// on both Windows and Unix systems, and functioned correctly in all cases.
// Thus, no significant difficulties are foreseen in creating the executable,
// and ensuring that it does in fact execute.
```

```
//
//   To use the executable generated by compilation of this file, at a command
// line prompt, in a directory from which the executable can be executed, enter
// the following command line.
//
//     <executable_name> <input_file_name> <output_file_name>
//
//   Where <executable_name is the name of the executable file as per the user
// determination, <input_file_name> is the name (including any required path
// specification) of the input file of random outputs from a generator, con-
// taining at least 20,000 bits, or 2,500 bytes, in binary format, with big-
// endian ordering, and <output_file_name> is the name (including any required
// path specification) of the file that will contain the results, in ASCII
// text characters, of the tests on the input file.
//   Elegence would dictate that separate tests should be broken out into
// separate functions, but it is regarded as unnecessary for this simple a
// program.

//-----------------------------------------------------------------------------
// Being a very simple program, we need little by way of IO libraries and
// functions.

#include <stdio.h>

// We need to define a few constants symbolically for readability, etc.
```

```
#define         BIT_COUNT       20000   // the required number of bits to test
#define         NYBBLE_COUNT    5000    // BIT_COUNT / 4
#define         BYTE_COUNT      2500    // BIT_COUNT / 8
#define         BITS_BYTES      8       // number of bits per byte

//=============================================================================
// int main (int argc, char *argv[])
//   This is where all the work is done.

int main (int argc, char *argv[])
{
    int         i;
    int         j;
    int         k;
    int         one_cnt = 0;
    int         zero_cnt = 0;
    char        c;
    char        ac[20000];
    FILE        *ipf;
    FILE        *opf;

    // To start, we have to convert the file containing the binary stream of
    // generator outputs to a more manageable format for conducting the BSI
    // required tests.  We can perform the mono-bit test concurrently, and
    // do so here.  Technically, we don't need to count zeroes, but we do so
    // as a check against the count of ones.  As previously noted, error
// detection is almost non-existent.  The input file is assumed to be
// of the correct length.

ipf = fopen(argv[1], "rb");

for (i = 0, j = 0; i < BYTE_COUNT; i++, j += 8)
{
    // get a byte

    c = getc(ipf);

    // Cycle through the bits, converting to char values.
    // We assume a big-endian orientation throughout.

    for (k = 0; k < BITS_BYTES; k++, c = c << 1)
    {
        if ((c & 0x80) == 0x80)
        {
            ac[(j + k)] = 0x01;
            one_cnt++;
        }
        else
        {
            ac[(j + k)] = 0x00;
            zero_cnt++;
        }
    }
}

// The mono-bit test has been completed, so write the results to the
// output file.

fprintf(opf, "Test 1 : Mono-bit test : %d ones and %d zeroes.\n",
             one_cnt, zero_cnt);
fprintf(opf, "         Passing range is 9654 < # ones < 10346.\n");

if ((9654 < one_cnt) && (one_cnt < 10346))
    fprintf(opf, "         Test passed\n\n");
else
    fprintf(opf, "         Test FAILED\n\n");

// Now, perform the bit-run test.  Both the bit-run and long-run tests can
// be performed concurrently, which we do here.  For readability, the run
// count array entry 0 is unused, allowing the number of runs of 1 to 5
// bits to be counted in entries with the corresponding index.  Entry 6
// will contain the count of all runs of 6 or more bits, and entry 7 the
// number of runs of 34 bits and longer.

int         run;
int         run_cnt[8] = {0, 0, 0, 0, 0, 0, 0, 0};
char        current;
```

```
for (i = 1, run = 1, current = ac[0]; i < BIT_COUNT; i++)
{
    if (ac[i] == current)
        run++;
    else
    {
        if (run <= 5)
            run_cnt[run]++;
        else
        {
            run_cnt[6]++;

            if (run >= 34)
                run_cnt[7]++;
        }

        run = 1;
        current = ac[i];
    }
}

// Whatever the last bit was, it is part of a run, so we need to account
// for it.

if (run <= 5)
    run_cnt[run]++;
```

```
else
{
    run_cnt[6]++;

    if (run >= 34)
        run_cnt[7]++;
}

// Now we write the results of the run test.

fprintf(opf, "Test 3 : Run test --\n");
fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              1           %4d        2267     2733",
             run_cnt[1]);
if ((2267 <= run_cnt[1]) && (run_cnt[1] <= 2733))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");

fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              2           %4d        1079     1421",
             run_cnt[2]);
if ((1079 <= run_cnt[2]) && (run_cnt[2] <= 1421))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");
fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              3           %4d         502      748",
             run_cnt[3]);
if ((502 <= run_cnt[3]) && (run_cnt[3] <= 748))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");

fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              4           %4d         233      402",
             run_cnt[4]);
if ((233 <= run_cnt[4]) && (run_cnt[4] <= 402))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");

fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              5           %4d          90       223",
             run_cnt[5]);
if ((90 <= run_cnt[1]) && (run_cnt[5] <= 223))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");

fprintf(opf, "         Run length    occurences    upper    lower\n");
fprintf(opf, "              6+          %6d          90       223",
             run_cnt[6]);
if ((90 <= run_cnt[6]) && (run_cnt[6] <= 223))
    fprintf(opf, "passed\n");
else
    fprintf(opf, "FAILED\n");

// The long-run test fails if there are any runs of 34 bits or longer.

fprintf(opf, "Long run tests: %d 34+ bit runs occurred.\n", run_cnt[7]);
if (run_cnt[7] == 0)
    fprintf(opf, "                Test passed.\n");
else
    fprintf(opf, "                Test FAILED.\n");

// The poker test examines nybble values, checking for any skewing in the
// distribution of values.

int         nybble;
int         values[16] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
double      chi = 0.0;

for (i = 0; i < NYBBLE_COUNT; i += 4)
{
    nybble = (ac[i] << 3) + (ac[i + 1] << 2) + (ac[i + 2] << 1) + ac[i + 3];
    values[nybble]++;
}

for (i = 0; i < 16; i++)
    chi = ((double) values[i]) * ((double) values[i]);

chi = (chi * (16.0 / 5000.0)) - 5000.0;

// Write the poker test results.

fprintf(opf, "Test 2 : Poker test : chi = %f.\n", chi);
fprintf(opf, "         Valid range is 1.03 < chi < 57.4.\n");

if ((1.03 < chi) && (chi < 57.4))
    fprintf(opf, "         Test passed\n\n");
else
    fprintf(opf, "         Test FAILED\n\n");

// The autocorrelation test is the most time-consuming, as it involves 5000
// iternations of an inner loop, for each of 5000 iternations of its outer
// loop, each with 5000 iterations of the inner loop, or 25,000,000 passes
// through the inner loop.  Each iteration of the outer loop constitutes a
// test pass, and failure of any of the 'outer' iternations represents a
// failure of the test as a whole.

int tau;
int sum;
```

```
    int tau_flag = 0;

    for (tau = 1; tau <= 5000; tau++)
    {
        sum = 0;

        for (i = 0; i < 5000; i++)
        {
            sum += (ac[i] ^ ac[(i + tau)]);
        }

        if ((2326 < sum) && (sum < 2674))
        {
            tau_flag = -1;
            fprintf(opf, "Autocorrelation test FAILED for tau = %d\n", tau);
        }
    }

    if (tau_flag == 0)
        fprintf(opf, "Autocorrelation test passed.\n");

    fclose(opf);

    return 0;
}
```

## Appendix C: Key Agreement Code.

The following pages contain the C language code for an implementation of the key agreement Protocol B.  The code is broken into three files: agree.c, polyLFSR.h and tables.h.

```
//=============================================================================
// agree.c
//
//     This file is the base for the key agreement scheme of Protocol B.  It is
// designed to process a sequence of 1056 bit blocks, stored as binary data in
// in input file, to produce a result string, based upon the state stored in a
// second input file.  The output string of bits, plus the ending state of the
// generator, are written to files.  The usage is as follows.
//
//     agree n <in> <out> <in_state> <out_state>
//
// where
//     n           is the number of 1056 bit / 132 byte blocks to be processed
//     <in>        is the name of the file containing those blocks
//     <out>       is the name of the file to write agreed bits/bytes to
//     <in_state>  is the initial state file
//     <out_state> is the end state file
//
//     The executable uses material from tables.h and polyLFSR.h.  Therefore,
// to build the executable, place those two files in the same directory with
// agree.c, and use the following command on a Unix or Linux system.
//
//     gcc -o agree agree.c
//

#include <stdio.h>
```

```
#include <stdlib.h>

#include "tables.h"
#include "polyLFSR.h"

#define             ASIZE          4
#define             BSIZE          128

// The following declarations lay out the memory required to support the four
// composite generators: A, B, C and D.  The first three of these are "full"
// generators, in that they have two separate input generators (these are their
// "constituent" generators) used as the <Xn> and <Yn> sequence sources in the
// McLaren-Marsaglia generators they are loosely based on.  Thus there are bit
// queues (?xQ and ?yQ) for each, along with a counter to keep track of how
// many unused bits remain in these queues.  The D generator is the exception.
// It is identical in operation to the other three except that the <Xn> and
// <Yn> generators are replaced with the matched bit sequence generaged from rp
// and the output from C.
// Implementation in C is strightforward, but a C++ implementation, wherein
// each of the generators could be implemented as instances of appropriate
// classes, would be simple, too.  Implementation of these memory declarations
// could also have been done using struct typedefs, but the intent is to expose
// as much of the structure as possible as clearly as possible.
// Note that while it is necessary to keep track of the number of bits in the
// <Xn> and <Yn> buffers, this is not the case with the current position in
// the state vector, as it is rotated left with each nybble generated, and thus
// is auto-aligned.

unsigned long long  AxQ     = 0LL;  // the <Xn> buffer
int                 AxQbits = 0;    // the number of bits therein
unsigned long long  AyQ     = 0LL;  // the <Yn> buffer
int                 AyQbits = 0;    // the number of bits in that buffer
unsigned long long  Astate  = 0LL;  // the state of the A composite generator

unsigned long long  BxQ     = 0LL;
int                 BxQbits = 0;
unsigned long long  ByQ     = 0LL;
int                 ByQbits = 0;
unsigned long long  Bstate  = 0LL;

unsigned long long  CxQ     = 0LL;
int                 CxQbits = 0;
unsigned long long  CyQ     = 0LL;
int                 CyQbits = 0;
unsigned long long  Cstate  = 0LL;

unsigned long long  Dstate = 0LL;

// The following data declarations are in a sense excessive.  The b_array and
// c_array are never used at the same time, and thus could easily be merged so
// as to reuse the space.  or even omitted, if we chose to implement the rp
// extraction and bit matching slightly differently.  We are erring on the side
// of clerity and ease of debugging in all such cases.
// The data areas themselves are self-explanatory.  work_array[] is used to
// hold axr at first, then bxrp.  As bxrp is long, and subject to much
// processing, each of the B and C output strings used are stored, as are other
// working arrays.  The r_array and a_array are used to process axr, and are
// handled similarly, though processing is in only one "layer."

unsigned char       work_array[BSIZE];   // for storing axr, bxrp and rp

unsigned char       a_array[ASIZE];      // the output from A
unsigned char       r_array[ASIZE];      //

unsigned char       b_array[BSIZE];      // the output from B
unsigned char       c_array[BSIZE];      // the output from C
unsigned char       match_array[BSIZE];  // matched bits after application of
                                         // matching process on rp and c
unsigned char       k_array[BSIZE];      // the final output string


//===============================================================================
// void ping(int i)
//
//     This function is used in debugging, and provides a very simple way to
// show progress.  It can be ignored ...

void ping(int i)
```

```
{
    printf("PING: %d\n", i);
    return;
}

//==============================================================================
// void usage()
//
//     This function simply displays the usage data to stdout.

void usage()
{
    printf("Usage:\n    agree n <in> <out> <in_state> <out_state>\n");
    printf("where\n    n - the number blocks to be processed\n");
    printf("    <in> - is the name of the input file\n");
    printf("    <out> - is the name of the output file\n");
    printf("    <in_state>  is the initial state file\n");
    printf("    <out_state> is the end state file\n");

    return;
}

//==============================================================================
// int loadState(FILE *fp)
//
//     This function extracts the state information for the A, B, C and D
// generators, including the bit buffers for the A, B and C generators.  As
// some compilers have problems with reading long long using any of the scanf()
// functions, the unsigned long long items are stored in the file as pairs of
// unsigned longs in hexadecimal format.

int loadState(FILE *fp)
{
    int retCode = 0;
    unsigned long a, b;

    fscanf(fp, "%lx %lx %x\n", &a, &b, &AxQbits);
    AxQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx %x\n", &a, &b, &AyQbits);
    AyQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx\n", &a, &b);
    Astate = (((unsigned long long) a) << 32) | ((unsigned long long) b);

    fscanf(fp, "%lx %lx %x\n", &a, &b, &BxQbits);
    BxQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx %x\n", &a, &b, &ByQbits);
    ByQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx\n", &a, &b);
    Bstate = (((unsigned long long) a) << 32) | ((unsigned long long) b);

    fscanf(fp, "%lx %lx %x\n", &a, &b, &CxQbits);
    CxQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx %x\n", &a, &b, &CyQbits);
    CyQ = (((unsigned long long) a) << 32) | ((unsigned long long) b);
    fscanf(fp, "%lx %lx\n", &a, &b);
    Cstate = (((unsigned long long) a) << 32) | ((unsigned long long) b);

    fscanf(fp, "%lx %lx\n", &a, &b);
    Dstate = (((unsigned long long) a) << 32) | ((unsigned long long) b);

    return retCode;
}

//=============================================================================
// int intialize(FILE *fp)

int initialize(FILE * fp)
{
    int retCode;

    if ((retCode = loadState(fp)) == 0)
    {
        if ((retCode = loadGenerators(fp)) != 0)
            printf("Could not load generators.\n");
    }
    else
        printf("Could not load state.\n");
    return retCode;
}

//=============================================================================
// int get_data(FILE *fp, unsigned char *a, unsigned char *b)
//
//     This function gets a block of data for processing, and assumes certain
// facts regarding the structure of the data to be read and stored.  It assumes
// that the target arrays are of sufficient size, and that there are two such
// targets, with sizes ASIZE and BSIZE.
//     The data to be reach is assumed to be binary in nature, not ASCII text,
// though there is nothing to preclude that.
//     The return value indicates whether an error occurred in reading the data
// via a value of 0 returned for no errors, and -1 for an error.  Error testing
// is limited to an EOF.  Note that the j counter/index will be 128 only if the
// expected number of bytes have been read.

int get_data(FILE *fp, unsigned char *a, unsigned char *b)
{
    int i = 0;              // counter/index for the first array
    int j = 0;              // counter/index for the second array
    int c;                  // target for getc(fp), which returns an int.

    // For ASIZE bytes, read from the input file into the a[] array, bailing if
    // an EOF is encountered.
```

```
while (i < ASIZE)
{
    if ((c = fgetc(fp)) == EOF) break;

    *(a + i) = (unsigned char) c;
    i++;
}

// only if the first ASIZE bytes were successfully read, read BSIZE bytes
// into the b[] array, again bailing if an EOF is encountered.

if (i == ASIZE)
{
    while (j < BSIZE)
    {
        if ((c = fgetc(fp)) == EOF) break;

        *(b + j) = (unsigned char) c;
        j++;
    }
}

// return the correct return value/code: 0 if the data was read correctly,
// or -1 if not.

return ((j == BSIZE) ? 0 : -1);
```

```
}

//================================================================================
// unsigned long myRand(unsigned long       (*rX) (void),
//                      unsigned long       (*rY) (void),
//                      unsigned long long   *xQ,
//                      int                  *xQbits,
//                      unsigned long long   *yQ,
//                      int                  *yQbits,
//                      unsigned long long   *state)
//
//     This function "pumps" the composite generator defined by the parameters
// passed, it is designed so that the <Xn> and <Yn> may be of any type.  The
// function maintains the xQ, yQ, and state.  The value generated is returned.
//     The dTable and sTable arrays used are assumed to be within scope and
// defined as required.  (see tables.h)

unsigned long myRand(unsigned long       (* rX)(void),
                     unsigned long       (* rY)(void),
                     unsigned long long   *xQ,
                     int                  *xQbits,
                     unsigned long long   *yQ,
                     int                  *yQbits,
                     unsigned long long   *state)
{
    int             i = 0;      // the counter
int             jj, kk, ll; // indices for accessing dTable/sTable entries
unsigned long   v;          // a temporary value returned by a PRNG
unsigned long   outVal = 0; // where we put what is generated and returned

while (i < 8)
{
    if (*xQbits <= 4)
    {
        v = (*rX)();
        *xQ = *xQ | (((unsigned long long) v) << *xQbits);
        *xQbits += 32;
    }

    if (*yQbits <= 4)
    {
        v = (*rY)();
        *yQ = *yQ | (((unsigned long long) v) << *yQbits);
        *yQbits += 32;
    }

    jj = (int) (*state & 0x3FLL);
    *state = (*state >> 4) | (*state << 60);

    kk = (int) (*xQ & 0x0000000FLL);
    *xQ = (*xQ >> 4);
    *xQbits = *xQbits - 4;
```

```
        ll = (int) (*yQ & 0x0000000FLL);
        *yQ = (*yQ >> 4);
        *yQbits = *yQbits - 4;

        outVal = (outVal << 4) | ((unsigned long) (dTable[jj][kk][ll] & 0x0F));
        *state ^= ((unsigned long long) sTable[jj][kk][ll]) & 0x3FFFLL;

        i++;
    }

    return outVal;
}

//=============================================================================
// int fill_a()
//
//     This function fills the a array, using the A composite generator.  We
// perform a 'manual' insertion from the unsigned long A produces in order to
// ensure that implementations on Bit-Endian and Little-Endian processors still
// behave the same.

int fill_a()
{
    unsigned long val;
    val = myRand(genAx, genAy, &AxQ, &AxQbits, &AyQ, &AyQbits, &Astate);

    a_array[0] = (unsigned char) ((val >> 24) & 0xFFL);
    a_array[1] = (unsigned char) ((val >> 16) & 0xFFL);
    a_array[2] = (unsigned char) ((val >>  8) & 0xFFL);
    a_array[3] = (unsigned char) ( val        & 0xFFL);

    return 0;
}

//=============================================================================
// int fill_bc()
//
//     This function uses the B and C composite generators to fill the contents
// of the b and c byte arrays, which will be used to first perform a Vernam
// decipher of the bxrp bit sequence, then to do the bit matching process
// between rp and c.

int fill_bc()
{
    unsigned long ii;
    unsigned long val;

    // we deal with blocks of four bytes in a "Big-endian" manner, manually, to
    // ensure compatibility between different processors with different
    // -endianism.
    for (ii = 0; ii < 128; ii += 4)
    {
        val = myRand(&genBx, &genBy, &BxQ, &BxQbits, &ByQ, &ByQbits, &Bstate);

        b_array[ii]     = (unsigned char) ((val >> 24) & 0xFFL);
        b_array[ii + 1] = (unsigned char) ((val >> 16) & 0xFFL);
        b_array[ii + 2] = (unsigned char) ((val >>  8) & 0xFFL);
        b_array[ii + 3] = (unsigned char) ( val        & 0xFFL);
    }

    for (ii = 0; ii < 128; ii += 4)
    {
        val = myRand(&genCx, &genCy, &CxQ, &CxQbits, &CyQ, &CyQbits, &Cstate);

        c_array[ii]     = (unsigned char) ((val >> 24) & 0xFFL);
        c_array[ii + 1] = (unsigned char) ((val >> 16) & 0xFFL);
        c_array[ii + 2] = (unsigned char) ((val >>  8) & 0xFFL);
        c_array[ii + 3] = (unsigned char) ( val        & 0xFFL);
    }

    return 0;
}

//==============================================================================
// int skew_bc(unsigned char *aa)
```

```
//
//     This function inserts the requisite bits from the characcter string at
// aa into the <Xn> bit buffers for the B and C composite generators.  The low
// order 15 bits are inserted into B's buffer, and the next higher 14 bits into
// C's buffer.  The net effect, if the contents of aa are random, is to make
// B and C aperiodic.  In addition, the outputs of the <Xn> and <Yn> generators
// of the two are shifted relative to each other, so that only once every 32
// cycles (in the case of B), or every 16 (in the case of C), do the save bit
// generators (the individual LFSRs) align pair-wise, between <Xn> and <Yn>.
// at each such occurrence, the bits from each are "out of phase" relative to
// their previous alignment.  The highest order three bits are returned, and
// are used in subsequent aperiodization of A.

int skew_bc(unsigned char *aa)
{
    unsigned long long ii = 0LL;

    ii = (((unsigned long long) aa[0]) << 24)
       | (((unsigned long long) aa[1]) << 16)
       | (((unsigned long long) aa[2]) <<  8)
       |   (unsigned long long) aa[3];

    BxQ |= ((((unsigned long long) ii) && 0x07FFFLL) << BxQbits);
    BxQbits += 15;

    CxQ |= (((((unsigned long long) ii) >> 15) && 0x03FFFLL) << CxQbits);
    CxQbits += 14;

    return (int) ((aa[0] >> 29) & 0x00000007);
}

//=============================================================================
//int skew_a(unsigned char *ca,
//           int            n,
//           int            bias)
//
//     This function inserts 32 bits from the matched bit array into the AxQ
// bit buffer, ensuring the desired aperiodicity of the A generator.  If there
// are not enough bits,

int skew_a(unsigned char *ca,
           int            n,
           int            bias)
{
    int ii = 0;
    int jj = 0;
    unsigned long skew = 0;

    // look back from the end of the matched bit buffer by 4 + bias, which
    // effectively means we'll extract bits starting at ca[n - (4 + bias)]
    // ... if at all.
    ii = n - (4 + bias);

    if (ii > 4)     // only extract and skew if there are at least 96 bits
    {
        skew = (unsigned long) ca[ii];
        skew = (skew << 8) + (unsigned long) ca[ii + 1];
        skew = (skew << 8) + (unsigned long) ca[ii + 2];
        skew = (skew << 8) + (unsigned long) ca[ii + 3];

        for (jj = ii; jj < (n - 4); jj++)
        {
            ca[jj] = ca[(jj + 4)];
        }

        AxQ = AxQ | (((unsigned long long) skew) << AxQbits);
        AxQbits += 32;
    }

    return ((ii > 4) ? (n - 4) : n);
}

//=============================================================================
// void vernam_crypt(unsigned char me[], ke[], unsigned int n)
//
//     This function performs a simple Vernam cipher encrypt/decrypt.  Which is
// performed is determined by the contents of me[].  If it contains a plaintext
// the operation is encryption.  If it is a ciphertext, it is a decryption.
//     Whereas typical Vernam ciphers (other than OTP) would use the PRNG key-
// generator directly, use of arrays allows more generality, including OTP
// encryption and decryption.
//     No error checking is performed on the length of the key (ke[]) and
// message (me[]).

void vernam_crypt(unsigned char *me,    // message to be [en|de]crypted.
                  unsigned char *ke,    // key to be used.
                  unsigned int  n)      // number of bytes in the input arrays.
{
    int             ii;                 // an index.

    // [En|De]cryption is a simple, incremental XOR of the two arrays, with the
    // resulting [plain|cipher]text placed in me[].

    for (ii = 0; ii < n; ii++)
    {
        *(me + ii) ^= *(ke + ii);
    }

    return;
}

//=============================================================================
// int crp_match(unsigned char rp], c[], ma[], int n)
```

```
//
//     This function performs a bit-match process between the c[] and rp[]
// arrays to reduce the number of bits to be used in subsequent steps.
//     As rp[] will contain the random bitstream to be massaged into the agreed
// random bits and c[] the output of a PRNG, the process may be viewed altern-
// atively as either randomly selecting pseudo-random bits (in c[]), or as
// pseudo-randomly selecting random bits (in rp[]).  In either view, the result
// is placed in ma[], and may be regarded as random.
//     The match[][] array is structured to contain the number of bits that
// match in the two nybbles used as indices into the array in the low-order
// nybble of the entries, while the high nybble of each entry ccntains the
// matching bit values packed to the right.

int crp_match(unsigned char rp[],       // random byte stream.
              unsigned char c[],        // byte stream from PRNG C().
              unsigned char ma[],       // output array of matching bits/bytes.
              unsigned int  n)          // number of bytes in c[] and rp[].
{
    // Match data array contains the number of bits that match in the indices
    // of each entry, in the low-order nybble, and the bits that match, packed
    // to the right, in the high-order bits.

    static unsigned char   match[16][16] = {
        {0x04, 0x03, 0x03, 0x02, 0x03, 0x02, 0x02, 0x01,
         0x03, 0x02, 0x02, 0x01, 0x02, 0x01, 0x01, 0x00},
        {0x03, 0x14, 0x02, 0x13, 0x02, 0x13, 0x01, 0x12,
 0x02, 0x13, 0x01, 0x12, 0x01, 0x12, 0x00, 0x11},
{0x03, 0x02, 0x24, 0x13, 0x02, 0x01, 0x23, 0x12,
 0x02, 0x01, 0x23, 0x12, 0x01, 0x00, 0x22, 0x11},
{0x02, 0x13, 0x13, 0x34, 0x01, 0x12, 0x12, 0x33,
 0x01, 0x12, 0x12, 0x33, 0x00, 0x11, 0x11, 0x32},
{0x03, 0x02, 0x02, 0x01, 0x44, 0x23, 0x23, 0x12,
 0x02, 0x01, 0x01, 0x00, 0x43, 0x22, 0x22, 0x11},
{0x02, 0x13, 0x01, 0x12, 0x23, 0x54, 0x12, 0x33,
 0x01, 0x12, 0x00, 0x11, 0x22, 0x53, 0x11, 0x32},
{0x02, 0x01, 0x23, 0x12, 0x23, 0x12, 0x64, 0x33,
 0x01, 0x00, 0x22, 0x11, 0x22, 0x11, 0x63, 0x32},
{0x01, 0x12, 0x12, 0x33, 0x12, 0x33, 0x33, 0x74,
 0x00, 0x11, 0x11, 0x32, 0x11, 0x32, 0x32, 0x73},
{0x03, 0x02, 0x02, 0x01, 0x02, 0x01, 0x01, 0x00,
 0x84, 0x43, 0x43, 0x22, 0x43, 0x22, 0x22, 0x11},
{0x02, 0x13, 0x01, 0x12, 0x01, 0x12, 0x00, 0x11,
 0x43, 0x94, 0x22, 0x53, 0x22, 0x53, 0x11, 0x32},
{0x02, 0x01, 0x23, 0x12, 0x01, 0x00, 0x22, 0x11,
 0x43, 0x22, 0xA4, 0x53, 0x22, 0x11, 0x63, 0x32},
{0x01, 0x12, 0x12, 0x33, 0x00, 0x11, 0x11, 0x32,
 0x22, 0x00, 0x53, 0xB4, 0x11, 0x32, 0x32, 0x73},
{0x02, 0x01, 0x01, 0x00, 0x43, 0x22, 0x22, 0x11,
 0x43, 0x22, 0x22, 0x11, 0xC4, 0x63, 0x63, 0x32},
{0x01, 0x12, 0x00, 0x11, 0x22, 0x22, 0x11, 0x32,
 0x22, 0x53, 0x11, 0x32, 0x63, 0xD4, 0x32, 0x73},
{0x01, 0x00, 0x22, 0x11, 0x22, 0x11, 0x63, 0x32,
         0x22, 0x11, 0x63, 0x32, 0x63, 0x32, 0xE4, 0x73},
        {0x00, 0x11, 0x11, 0x32, 0x11, 0x32, 0x32, 0x73,
         0x11, 0x32, 0x32, 0x73, 0x32, 0x73, 0x73, 0xF4}
    };

    int             ii;                 // an index into the input arrays.
    int             bits = 0;           // number of matching bits buffered.
    int             cnt = 0;            // number of matching bytes.
    unsigned int    jj;                 // index into match[] from c[].
    unsigned int    kk;                 // index into match[] from m[].
    unsigned int    ll;                 // number of matched bits, this test.
    unsigned int    mm;                 // matched bits, this try.
    unsigned int    out = 0;            // buffer for bits not yet grouped.

    for (ii = 0; ii < n; ii++)
    {
        // Do a bit-match selection between the bytes in c and those in rp.
        // note that can do the nybble-wise matching either low-nybble-first
        // or high-nybble-first.  The ordering used here is low-nybble-first,
        // which achieves a limited mixing of the output bits.  This in itself
        // is not significant, but coupled with other swizzling operations it
        // increases the uncertainty of relationship between an output bit from
        // the aggregate process and any given bit in the input, which is the
        // bxrp array.
        // For each nybble, extract corresponding bits from the current bytes
        // of c and rp, then use these values of indices into the match table.
```

```
// The byte extracted contains the number of matching bits in the two
// indices in the low-order nybble, and the values of the matching bits
// packed to the right in the high-order nybble.  The bits obtained are
// appended to the out buffer, and the number of bits in the buffer
// is updated.

jj = (unsigned int) (c[ii] & 0x0F);
kk = (unsigned int) (rp[ii] & 0x0F);
ll = (unsigned int) (match[jj][kk] & 0x0F);
mm = (unsigned int) ((match[jj][kk] >> 4) & 0x0F);
out = ((out << ll) | mm);
bits += ll;

jj = (unsigned int) ((c[ii] >> 4) & 0x0F);
kk = (unsigned int) ((rp[ii] >> 4) & 0x0F);
ll = (unsigned int) (match[jj][kk] & 0x0F);
mm = (unsigned int) ((match[jj][kk] >> 4) & 0x0F);
out = ((out << ll) | mm);
bits += ll;

// If there are enough bits for a byte in the buffer, copy these to the
// target array (ma), increment the byte count, update the bit count,
// and shift the buffer to account for the bits removed.
// Note that if any bits are pushed into the buffer further than the
// right-most eight bits, such bits will remain in the buffer.  Thus,
// additional scrambling will occur if such bits are eventually placed
```

```
        // in the output array.  Such bits may in fact be discarded, along with
        // any other bits that were insufficient to form a byte when teh last
        // nybble has been porcessed.

        if (bits >= 8)
        {
            ma[cnt] = (unsigned char) (out & 0xFF);
            out = (out >> 8);
            bits = bits - 8;
            cnt++;
        }
    }

    // Return the full count of bytes generated through the bit-matching.

    return cnt;
}

//=============================================================================
// int mk_blend(unsigned long long *s, unsigned char m[], k[], int n)
//
//     This function performs the blending operation between the opposing ends
// of the matched bits array.  The process starts at the ends and works towards
// the middle.

int mk_blend(unsigned long long  *s,        // state for the mixing process.
             unsigned char *m,              // matched bits from prior step.
             unsigned char *k,              // the output array.
             unsigned int   n)              // number of bytes in m[].
{
    int             ii = 0;                 // 'left' index into m[].
    int             jj = n - 1;             // 'right' index into m[].
    int             aa;                     // first index into t[] and d[].
    int             bb;                     // second index into t[] and d[].
    int             cc;                     // third index into t[] and d[].
    unsigned long   temp;                   // temp for modifying state *s.

    // We only deal with even numbers of bytes.  If m[] contains an odd number
    // of bytes, the end-test for the operational loop will stop before it
    // process the middle byte, since m[] is processed from the ends towards
    // the middle, and the end-test is satisfied if ii == jj.

    for (; ii < jj; ii++, jj--)
    {
        // Extract the needed indices for the first nybble

        aa = (m[ii] >> 4) & 0x0F;
        bb = m[jj] & 0x0F;
        cc = *s & 0x3F;
        k[ii]  = (dTable[aa][bb][cc] & 0x0F);

        // adjust the state vector
        temp = (((unsigned long long) sTable[aa][bb][cc]) & 0x3FFFLL);
        *s  = ((*s >>  4) & 0x0FFFFFFFFFFFFFFFLL)
            | ((*s << 28) & 0xF000000000000000LL);
        *s ^= temp;

        // extract the indices for the second nybble

        aa = m[ii] & 0x0F;
        bb = (m[jj] >> 4) & 0x0F;
        cc = *s & 0x3F;
        k[ii] |= ((dTable[aa][bb][cc] & 0x0F) << 4);

        // adjust the state vector again

        temp = (((unsigned long long) sTable[aa][bb][cc]) & 0x3FFFLL);
        *s  = ((*s >>  4) & 0x0FFFFFFFFFFFFFFFLL)
            | ((*s << 28) & 0xF000000000000000LL);
        *s ^= temp;
    }

    return (n >> 1);
}

//==============================================================================
// void write_output(FILE *fp, unsigned char *k, int n)
```

```
//
//     This function writes teh agreed bits (as binary bytes) to the output
// file.  It is very simple, and requires very little explanation.

void write_output(FILE *fp, unsigned char *k, int n)
{
    int ii;

    for (ii = 0; ii < n; ii++)
    {
        putc(*(k + ii), fp);
    }

    return;
}

//=============================================================================
// int saveState(FILE *fp)
//
//     This function deals with saving the states of the composite generators,
// with regard to the buffered bits and the bit counts, as well as the state
// vectors, but uses the saveGenerators() function in polyLFSR to save the
// shift registers themselves.

int saveState(FILE *fp)
{
```

```
fprintf(fp, "%x %x %x\n", (unsigned long) ((AxQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( AxQ        & 0xFFFFFFFFL),
                          AxQbits);
fprintf(fp, "%x %x %x\n", (unsigned long) ((AyQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( AyQ        & 0xFFFFFFFFL),
                          AyQbits);
fprintf(fp, "%x %x\n",    (unsigned long) ((Astate >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( Astate        & 0xFFFFFFFFL));

fprintf(fp, "%x %x %x\n", (unsigned long) ((BxQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( BxQ        & 0xFFFFFFFFL),
                          BxQbits);
fprintf(fp, "%x %x %x\n", (unsigned long) ((AyQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( ByQ        & 0xFFFFFFFFL),
                          ByQbits);
fprintf(fp, "%x %x\n",    (unsigned long) ((Bstate >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( Bstate        & 0xFFFFFFFFL));

fprintf(fp, "%x %x %x\n", (unsigned long) ((CxQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( CxQ        & 0xFFFFFFFFL),
                          CxQbits);
fprintf(fp, "%x %x %x\n", (unsigned long) ((CyQ >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( CyQ        & 0xFFFFFFFFL),
                          CyQbits);
fprintf(fp, "%x %x\n",    (unsigned long) ((Cstate >> 32) & 0xFFFFFFFFL),
                          (unsigned long) ( Cstate        & 0xFFFFFFFFL));
```

```
    fprintf(fp, "%x %x\n",    (unsigned long) ((Dstate >> 32) & 0xFFFFFFFFL),
                              (unsigned long) ( Dstate        & 0xFFFFFFFFL));

    saveGenerators(fp);

    return 0;
}

//=============================================================================
// int main(int argc, char *argv[])
//
//     This is the driving function of the process.  Five command line argu-
// ments are required to run the program, as described above in usage().
//     The basic operation of main is performed in a loop which tests for an
// end of the input file by reading an input line, as well as testing for the
// exhaustion of the count passed in command line.

int main(int argc, char *argv[])
{
    int             n = 0;              // Byte counts.
    int             blocks = 0;         // Number of blocks.
    int             bias = 0;           // Distance in bytes from end, matched
                                        //   bits to extract A skew bits.
    FILE            *ipf;               // Pointer to material input file.
    FILE            *opf;               // Pointer to agreed key output file.
FILE            *isf;               // Pointer to input state file.
FILE            *osf;               // Pointer to output state file.

// need to set up file io and the number of blocks to process

if (argc != 6)
{
    printf("Invalid command line.\n");
    usage();
    goto myExit;
}

blocks = atoi(argv[1]);

ipf = fopen(argv[2], "rb");
opf = fopen(argv[3], "wb");
isf = fopen(argv[4], "r");
osf = fopen(argv[5], "w");

if    ((ipf == (FILE *) NULL) || (opf == (FILE *) NULL)
   ||  (isf == (FILE *) NULL) || (osf == (FILE *) NULL))
{
    printf("One of the required files could not be opened.\n");
    usage();
    goto myLongExit;
}
```

```
    // must have the initial state, so load it and intialize the process to
    // that agreed state.

    if (initialize(isf) != 0)
    {
        goto myLongExit;
    }

    while ((get_data(ipf, r_array, work_array) == 0) && (blocks-- > 0))
    {
//        printf("axr = %02x %02x %02x %02x\n", r_array[0], r_array[1], r_array[2], r_array[3]);

        fill_a();

//        printf("a = %02x %02x %02x %02x\n", a_array[0], a_array[1], a_array[2], a_array[3]);

        vernam_crypt(r_array, a_array, 4);

//        printf("r = %02x %02x %02x %02x\n", r_array[0], r_array[1], r_array[2], r_array[3]);

        bias = skew_bc(work_array);

        fill_bc();

        vernam_crypt(work_array, b_array, 128);
        n = crp_match(work_array, c_array, match_array, 128);
        n = skew_a(work_array, n, bias);

        mk_blend(&Dstate, match_array, k_array, n);

        write_output(opf, k_array, (n / 2));
    }

    // since we intend to use the revised state of the agreement process in
    // subsequent agreement processes, we must save the revised state.  IF we
    // were to take security seriously, we might consider using a suitable tool
    // to destroy the key file initially loaded, but as this is a didactic
    // implementation, and we may also be concerned about verifying the data
    // received was correct, this is not done here.

    saveState(osf);

    // regardless of whether the process was completed, we will be polite and
    // close any open files.

myLongExit:
    if (isf != (FILE *) NULL) fclose(isf);
    if (osf != (FILE *) NULL) fclose(osf);
    if (ipf != (FILE *) NULL) fclose(ipf);
    if (opf != (FILE *) NULL) fclose(opf);
```

```
myExit:
    return 0;
}
```

```
//=============================================================================
// polyLFSR.h
//
//     Contained herein are components necessary for the loading and saving of
// the overall state of the application, and specifically what may be regarded
// as the 'key', which changes
//     This file contains the definitions of the Galois configuration LFSRs
// used as the constituent PRNGs of the A, B and C generators in Protocols
// A and B.  As implemented, these are "banks" of 32 LFSRs, all of the Galois
// configuration, which can be operated in parallel, providing significantly
// more bits per cycle than via a single LFSR.  LFSRs of lengths up to 64 are
// supported by the code, though only the g2 and g5 generators below use long
// polynomials/shift registers.
//     Several changes may be made to improve the efficiency of the generators
// with regard to loading and saving state and operation.  Carrying a
// 'length' value, so that we avoid unnecessary reads and writes, as well as
// not continuing the update process past the end of the polynomial, for
// example, would help overall efficiency.
//     While the polynomials are defined herein, and are thus 'static', there
// is no requirement that this be so for the protocols supported.  The poly-
// nomials could as easily be regarded as "key material" to be determined upon
// initial establishment of the ends of the exchange (meaning giving the
// initial states to Alice and Bob).  This would involve an expansion of the
// saved state and additional initialization code, but does not represent a
// significant problem.
// set the maximum size of the various tables used by the ganged Galois config-
// uration LFSRs.

#define             MAX_BITS        64

// For each of the six ganged Galois LFSR sets, the taps are provided for
// verification of the polynomials.

// 31,3           31,6            31,9,5,1        31,10,6,5,2,1
// 35,11,9,7,6,1  35,11,10,6,5,1  35,11,10,7,6,4  35,11,10,9,6,4
// 35,10,8,7,4,3  35,10,9,1       35,11,6,5       35,11,8,5
// 35,9,6,2       35,10,4,3       35,10,7,3       35,10,8,7,4,3
// 36,9,7,1       37,9,7,6,5,1    36,12,8,4,2,1   36,13,9,6
// 37,6,4,1       37,9,2,1        37,10,5,4       37,11,6,1
// 38,6,5,1       38,9,8,5,4,1    38,11,5,2       38,11,6,4
// 39,4           39,9,8,5,4,1    39,10,9,5,2,1   39,11,9,2,1

// Three tables are used for each ganged set of LFSRs.  The first will contain
// the actual bits in the virtualized shift registers, the second the bits
// corresponding to the taps of the LFSRs, and the third a set of masks used
// to remove extraneous 1's from the first table at initialization.  This
// removal should not be necessary, in normal operation of the generators, but
// which the state file is creeated their use simplifies the process of
// ensuring that there are no extraneous bits present, and thereby helps in
// verification of the correctness of the processes involved.
// Start of the g0 specification

unsigned long       g0Bits[MAX_BITS];
unsigned long       g0Taps[MAX_BITS] = {
         0x00000000ul, 0x3C40EDC7ul, 0x10082223ul, 0x80870000ul, // 00 - 03
         0x03852A5Cul, 0x343042E6ul, 0x5F285980ul, 0x0A83C000ul, // 04 - 07
         0x00912044ul, 0x2948D477ul, 0x17C70202ul, 0x0F300134ul, // 08 - 11
         0x00002000ul, 0x00001000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 16 - 19
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 20 - 23
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0xF0000000ul, // 28 - 31
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x0FFF0000ul, // 32 - 35
         0x0000F000ul, 0x00000F00ul, 0x000000F0ul, 0x0000000Ful, // 36 - 39
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned long       g0Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0x0FFFFFFFul, 0x0FFFFFFFul, 0x0FFFFFFFul, 0x0FFFFFFFul,
         0x0000FFFFul, 0x00000FFFul, 0x000000FFul, 0x0000000Ful,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned int        g0Here = 0;

// Start of the g1 specification

// 32,9,5,3            32,11,5,2      32,11,10,7,6,2  32,12,11,6,5,3
// 33,8,5,1            33,9,8,3       33,10,7,3       33,11,5,2
// 34,9,5,1            34,10,8,5      34,11,7,2       34,12,8,7,5,1
// 35,2                35,9,8,5       35,10,9,1       35,11,10,9
// 36,9,4,3            36,10,6,1      36,11           36,12,6,5
// 37,9,2,1            37,10,6,5,2,1  37,11,9,7,6,2   37,11,10,9,5,4
// 38,9,3,2            38,10,9,6,5,4  38,11,8,5       38,12,9,8,7,4
// 39,10,7,6,5,3,2,1  39,10,9,5,4,3  39,11,7,1       39,11,9,5,4,3,2,1
```

```
unsigned long       g1Bits[MAX_BITS];
unsigned long       g1Taps[MAX_BITS] = {
         0x00000000ul, 0x08924C0Bul, 0x61280E89ul, 0x9600808Dul, // 00 - 03
         0x00008155ul, 0xC9D4156Dul, 0x30005648ul, 0x2230021Aul, // 04 - 07
         0x0C940030ul, 0x84878BD5ul, 0x2243454Cul, 0x71212323ul, // 08 - 11
         0x10101010ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 16 - 19
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 20 - 23
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 28 - 31
         0xF0000000ul, 0x0F000000ul, 0x00F00000ul, 0x000F0000ul, // 32 - 35
         0x0000F000ul, 0x00000F00ul, 0x000000F0ul, 0x0000000Ful, // 36 - 39
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 40 - 43
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 44 - 47
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 48 - 51
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 52 - 55
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 56 - 59
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul  // 60 - 63
         };
unsigned long       g1Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0x0FFFFFFFul, 0x00FFFFFFul, 0x000FFFFFul,
         0x0000FFFFul, 0x00000FFFul, 0x000000FFul, 0x0000000Ful,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned int        g1Here = 0;

// Start of the g2 specification

// 32,22,2,1    33,20         34,27,2,1    35,33
// 36,25        37,5,4,3,2,1  38,6,5,1     39,35
// 40,38,21,19  41,38         42,41,20,19  43,42,38,37
// 44,43,18,17  45,44,42,41   46,45,26,25  47,42
// 48,47,21,20  49,40         50,49,24,23  51,50,36,35
// 52,49        53,52,38,37   54,53,18,17  55,31
// 56,55,35,34  57,50         58,39        59,58,38,37
// 60,59        61,60,46,45   62,61,6,5    63,62

unsigned long       g2Bits[MAX_BITS];
unsigned long       g2Taps[MAX_BITS] = {
         0x00000000ul, 0xA6000000ul, 0xA4000000ul, 0x04000000ul, // 00 - 03
         0x04000000ul, 0x06000002ul, 0x02000002ul, 0x00000000ul, // 04 - 07
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 08 - 11
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00080200ul, 0x00080200ul, 0x00A00000ul, // 16 - 19
         0x40208000ul, 0x00808000ul, 0x80000000ul, 0x00002000ul, // 20 - 23
         0x00002000ul, 0x08020000ul, 0x00020000ul, 0x20000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000100ul, // 28 - 31
         0x80000000ul, 0x50000000ul, 0x20000080ul, 0x11001080ul, // 32 - 35
         0x08001000ul, 0x04000410ul, 0x02D00410ul, 0x01100020ul, // 36 - 39
         0x00804000ul, 0x00640000ul, 0x00350000ul, 0x00180000ul, // 40 - 43
         0x000C0000ul, 0x00060004ul, 0x00020004ul, 0x00018000ul, // 44 - 47
         0x00008000ul, 0x00006800ul, 0x00003040ul, 0x00001000ul, // 48 - 51
         0x00000C00ul, 0x00000600ul, 0x00000200ul, 0x00000180ul, // 52 - 55
         0x00000080ul, 0x00000040ul, 0x00000030ul, 0x00000010ul, // 56 - 59
         0x0000000Cul, 0x00000006ul, 0x00000003ul, 0x00000001ul  // 60 - 63
         };
unsigned long       g2Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0x7FFFFFFFul, 0x3FFFFFFFul, 0x1FFFFFFFul,
         0x0FFFFFFFul, 0x07FFFFFFul, 0x03FFFFFFul, 0x01FFFFFFul,
         0x00FFFFFFul, 0x007FFFFFul, 0x003FFFFFul, 0x001FFFFFul,
         0x000FFFFFul, 0x0007FFFFul, 0x0003FFFFul, 0x0001FFFFul,
         0x0000FFFFul, 0x00007FFFul, 0x00003FFFul, 0x00001FFFul,
         0x00000FFFul, 0x000007FFul, 0x000003FFul, 0x000001FFul,
         0x000000FFul, 0x0000007Ful, 0x0000003Ful, 0x0000001Ful,
         0x0000000Ful, 0x00000007ul, 0x00000003ul, 0x00000001ul
         };
unsigned int        g2Here = 0;

// Start of the g3 specification

// 31,8,6,2       31,9,3,1       31,9,5,1       31,9,8,4
// 31,10,5,3,2,1  31,10,7,5,3,1  31,10,7,6,5,2  31,10,9,1
// 31,10,9,7,4,2  31,10,9,8,5,3  31,10,9,8,6,3  31,10,9,8,7,1
// 31,11,2,1      31,11,7,5,4,3  31,11,7,6,5,3  31,11,9,1
// 31,11,9,6,5,4  31,11,9,7      31,11,4,3      31,10,9,6,3,2
// 31,10,8,7,2,1  31,10,8,6,5,4  31,10,8,5,4,3  31,10,6,5,2,1
// 31,9,8,7,4,1   31,9,8,4,3,2   31,9,7,6,4,1   31,9,5,4
// 31,8,7,5       31,8,5,3,2,1   31,6,4,2       31,3

unsigned long       g3Bits[MAX_BITS];
unsigned long       g3Taps[MAX_BITS] = {
         0x00000000ul, 0x6D1909A4ul, 0x8A881946ul, 0x4C663245ul, // 00 - 03
         0x1084A6F2ul, 0x2E46871Cul, 0x82229522ul, 0x069648A8ul, // 04 - 07
         0x90700ECCul, 0x71F1D0F0ul, 0x0FF01F00ul, 0x000FE000ul, // 08 - 11
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 16 - 19
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 20 - 23
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0xFFFFFFFFul, // 28 - 31
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned long       g3Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned int        g3Here = 0;

// Start of the g4 specification

// 31,3            31,6            31,9,5,1        31,10,6,5,2,1
// 35,11,9,7,6,1  35,11,10,6,5,1  35,11,10,7,6,4  35,11,10,9,6,4
// 35,10,8,7,4,3  35,10,9,1       35,11,6,5       35,11,8,5
// 35,9,6,2       35,10,4,3       35,10,7,3       35,10,8,7,4,3
// 36,9,7,1       37,9,7,6,5,1    36,12,8,4,2,1   36,13,9,6
// 37,6,4,1       37,9,2,1        37,10,5,4       37,11,6,1
// 38,6,5,1       38,9,8,5,4,1    38,11,5,2       38,11,6,4
// 39,4           39,9,8,5,4,1    39,10,9,5,2,1   39,11,9,2,1

unsigned long       g4Bits[MAX_BITS];
unsigned long       g4Taps[MAX_BITS] = {
         0x00000000ul, 0x3C40EDC7ul, 0x10082223ul, 0x80870000ul, // 00 - 03
         0x03852A5Cul, 0x343042E6ul, 0x5F285980ul, 0x0A83C000ul, // 04 - 07
         0x00912044ul, 0x2948D477ul, 0x17C70202ul, 0x0F300134ul, // 08 - 11
         0x00002000ul, 0x00001000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 16 - 19
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 20 - 23
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0xF0000000ul, // 28 - 31
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x0FFF0000ul, // 32 - 35
         0x0000F000ul, 0x00000F00ul, 0x000000F0ul, 0x0000000Ful, // 36 - 39
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned long       g4Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0x0FFFFFFFul, 0x0FFFFFFFul, 0x0FFFFFFFul, 0x0FFFFFFFul,
         0x0000FFFFul, 0x00000FFFul, 0x000000FFul, 0x0000000Ful,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul,
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul
         };
unsigned int        g4Here = 0;

// Start of the g5 specification

// 32,22,2,1    33,20         34,27,2,1    35,33
// 36,25        37,5,4,3,2,1  38,6,5,1     39,35
// 40,38,21,19  41,38         42,41,20,19  43,42,38,37
// 44,43,18,17  45,44,42,41   46,45,26,25  47,42
// 48,47,21,20  49,40         50,49,24,23  51,50,36,35
// 52,49        53,52,38,37   54,53,18,17  55,31
// 56,55,35,34  57,50         58,39        59,58,38,37
// 60,59        61,60,46,45   62,61,6,5    63,62

unsigned long       g5Bits[MAX_BITS];
unsigned long       g5Taps[MAX_BITS] = {
         0x00000000ul, 0xA6000000ul, 0xA4000000ul, 0x04000000ul, // 00 - 03
         0x04000000ul, 0x06000002ul, 0x02000002ul, 0x00000000ul, // 04 - 07
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 08 - 11
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000000ul, // 12 - 15
         0x00000000ul, 0x00080200ul, 0x00080200ul, 0x00A00000ul, // 16 - 19
         0x40208000ul, 0x00808000ul, 0x80000000ul, 0x00002000ul, // 20 - 23
         0x00002000ul, 0x08020000ul, 0x00020000ul, 0x20000000ul, // 24 - 27
         0x00000000ul, 0x00000000ul, 0x00000000ul, 0x00000100ul, // 28 - 31
         0x80000000ul, 0x50000000ul, 0x20000080ul, 0x11001080ul, // 32 - 35
         0x08001000ul, 0x04000410ul, 0x02D00410ul, 0x01100020ul, // 36 - 39
         0x00804000ul, 0x00640000ul, 0x00350000ul, 0x00180000ul, // 40 - 43
         0x000C0000ul, 0x00060004ul, 0x00020004ul, 0x00018000ul, // 44 - 47
         0x00008000ul, 0x00006800ul, 0x00003040ul, 0x00001000ul, // 48 - 51
         0x00000C00ul, 0x00000600ul, 0x00000200ul, 0x00000180ul, // 52 - 55
         0x00000080ul, 0x00000040ul, 0x00000030ul, 0x00000010ul, // 56 - 59
         0x0000000Cul, 0x00000006ul, 0x00000003ul, 0x00000001ul  // 60 - 63
         };
unsigned long       g5Mask[MAX_BITS] = {
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul, 0xFFFFFFFFul,
         0xFFFFFFFFul, 0x7FFFFFFFul, 0x3FFFFFFFul, 0x1FFFFFFFul,
         0x0FFFFFFFul, 0x07FFFFFFul, 0x03FFFFFFul, 0x01FFFFFFul,
         0x00FFFFFFul, 0x007FFFFFul, 0x003FFFFFul, 0x001FFFFFul,
         0x000FFFFFul, 0x0007FFFFul, 0x0003FFFFul, 0x0001FFFFul,
         0x0000FFFFul, 0x00007FFFul, 0x00003FFFul, 0x00001FFFul,
         0x00000FFFul, 0x000007FFul, 0x000003FFul, 0x000001FFul,
         0x000000FFul, 0x0000007Ful, 0x0000003Ful, 0x0000001Ful,
         0x0000000Ful, 0x00000007ul, 0x00000003ul, 0x00000001ul
         };
unsigned int        g5Here = 0;

//=============================================================================
// unsigned long galois32(unsigned long *bits,
//                        unsigned long *taps,
//                        unsigned int  *here)
//     This function "operates" the ganged LFSRs of each six generators above,
// when passed the appropriate pointers.  The '*bits' pointer indicates where
// the contents of the virtualized shift registers may be found, the '*taps'
// pointer where the tap masks are to be found, and '*here' points to where the
// current index to the "lowest order" bits of the LFSRs may be found.
//     As the operation rotates throw the bits storage area, rather than doing
// an actual shift of the bits in the "registers", the cost of moving all that
// data is avoided.
//     It was decided NOT to provide a struct typedef corresponding to the
// ganged generators, primarily to avoid the confusion that sometimes arises
// during a cursory reay of code that relies heavily upon such typedefs.

unsigned long galois32(unsigned long *bits,
                       unsigned long *taps,
                       unsigned int  *here)
{
```

```
unsigned int    i;      // a counter
unsigned long   out;    // stores the output value(s)

out = bits[*here];      // get the output for this cycle

// loop through the bit fields that correspond to the bit positions in the
// individual LFSRs.  again, this could be made more efficient by using the
// actual length of the longest of the LFSRs, which would necessarily be
// passed as another parameter, unless we were to create a struct typedef
// corresponding to the component generators.

for (i = 1; i < MAX_BITS; i++)
{
    bits[((*here + i) % MAX_BITS)] ^= (out & taps[i]);
}

// clear the just used output bit field to avoid extraneous (and thus
// erroneous) feedback into the generator, and update the current location
// in the LFSRs.

bits[*here] = 0UL;
*here = (*here + 1) % MAX_BITS;

// return the output bits.

return out;
```

```
}

//==============================================================================
// unsigned long genAx()
// unsigned long genAy()
// unsigned long genBx()
// unsigned long genBy()
// unsigned long genCx()
// unsigned long genCy()
//
//     These six functions are provided as "hooks" to the appropriate component
// PRNGs for the A, B and C composite generators.  The reason for their use is
// to simplify the calls to the component generators within agree.c, where they
// are passed, via pointers to them, to the code that pumps the composite
// generators.  Such references and calls can be confusing enough without the
// added complication of passing parameters to the functions referenced via
// pointers.

unsigned long genAx()
{
    return galois32(g0Bits, g0Taps, &g0Here);
}

unsigned long genAy()
{
    return galois32(g1Bits, g1Taps, &g1Here);
```

```
}

unsigned long genBx()
{
    return galois32(g2Bits, g2Taps, &g2Here);
}

unsigned long genBy()
{
    return galois32(g3Bits, g3Taps, &g3Here);
}

unsigned long genCx()
{
    return galois32(g4Bits, g4Taps, &g4Here);
}

unsigned long genCy()
{
    return galois32(g5Bits, g5Taps, &g5Here);
}

//=============================================================================
// int loadGenerators(FILE *fp)
//     This function performs a straightforward load of the state stored in
// the file indicated by '*fp'.  The data in that file is stored as six columns
// of unsigned longs in hexadecimal format, with each column representing one
// of the six component generators to be restored.  As the data is stored with
// the lowest order bit fields in position zero, no accounting need be kept of
// where in the operation of the generators processing was halted prior to
// storing the state.
//     Note that it would be simple to insert values into the file to indicate
// how many bits the longest LFSR contains, and to load the generators appro-
// priately.  A malloc() could then be used to allocate as much space as needed
// for the generators.  In a more general implementation, this would be fully
// appropriate, but would unnecessarily complicate the present example.  The
// same types of observations can be made regarding the taps and masks.  Again,
// to simplify the present example, such generalizations were not implemented.

int loadGenerators(FILE *fp)
{
    int             i;              // a counter/index
    int             retCode = 0;    // return code for detecting faults

    // loop through the lines of the state file, filling in the bit field array
    // that is the current state of the LFSRs.

    for (i = 0; i < MAX_BITS; i++)
    {
        // read one value from each of the six columns of data

        fscanf(fp, "%lx %lx %lx %lx %lx %lx",
               &g0Bits[i], &g1Bits[i], &g2Bits[i],
               &g3Bits[i], &g4Bits[i], &g5Bits[i]);

        // mask out any stray ones, thereby removing bits that might cause
        // erroneous feedback in any of the LFSRs.  this is redundant, but
        // it is better to be safe.

        g0Bits[i] &= g0Mask[i];
        g1Bits[i] &= g1Mask[i];
        g2Bits[i] &= g2Mask[i];
        g3Bits[i] &= g3Mask[i];
        g4Bits[i] &= g4Mask[i];
        g5Bits[i] &= g5Mask[i];
    }

    g0Here = 0;
    g1Here = 0;
    g2Here = 0;
    g3Here = 0;
    g4Here = 0;
    g5Here = 0;

    return retCode;
}

//==============================================================================
// int saveGenerators(FILE *fp)
//     This function saves the current state of the six constituent ganged
// LFSRs.  The file writing process is performed in such a way that it is not
// necessary to know the alignment of the LFSRs as they rotated (rather than
// being shifted) through the 'g#Bits[]' storage area.
//     As described above, the ganged LFSRs are stored as columns in the file
// indicated by '*fp', as ASCII hexadecimal unsigned longs.
//     The process here is generalized to account for the fact that the g0 and
// g1 ganged generators will operate at a different pace from the other four.
// In particular, as the g0 generator will be very infrequently pumped, and the
// g1 generator only once per cycle, while the other four will be pumped 32
// times per cycle, means that in most cases there will be an increasing dis-
// parity between the current location within the g0 and g1 generators.  The
// disparity between the g2 through g5 generators is more irregular.
//     As the g2 through g5 generators will be pumped the same number of times
// per cycle, barring incorporation explicitly stated LSFR lengths, i2 through
// i5 COULD be condensed into a single index, but have not been in anticipation
// of the incorporation of said explicit LFSR lengths, along with other planned
// revisions.  The loss of efficiency is deemed inconsequential, and preserves
// the opportunity to use the code in other applications where the number of
// iterations may vary among all of the generators.

int saveGenerators(FILE *fp)
{
    int             i, i0, i1, i2, i3, i4, i5;  // a counter and indices
    int             retCode = 0;                // for capturing errors
```

```
// set the initial values of the indices.

i0 = g0Here; i1 = g1Here; i2 = g2Here;
i3 = g3Here; i4 = g4Here; i5 = g5Here;

// loop through the entire bits table, writing the corresponding positions
// (relative to current locations) as a row in the output file.

for (i = 0; i < MAX_BITS; i++)
{
    fprintf(fp, "%lx %lx %lx %lx %lx %lx\n",
            g0Bits[i0], g1Bits[i1], g2Bits[i2],
            g3Bits[i3], g4Bits[i4], g5Bits[i5]);

    // update the indices.

    i0 = (i0 + 1) % MAX_BITS;
    i1 = (i1 + 1) % MAX_BITS;
    i2 = (i2 + 1) % MAX_BITS;
    i3 = (i3 + 1) % MAX_BITS;
    i4 = (i4 + 1) % MAX_BITS;
    i5 = (i5 + 1) % MAX_BITS;
}

return retCode;
```

```
}
```

```
//=============================================================================
// tables.h
//
//     This file contains the dTable and sTable 3D arrays used by the specific
// implementation of the bitBlend function in its two forms in agree.c.  The
// dTable is constructed so that any given pair of <X> and <Y> sequence nybble
// inputs will display an equal probability of yielding any possible 4-bit
// output value, with the actual value determined by the third index, obtained
// from the state bit-field for the generator used.  The sTable is constructed
// so that every combination of index values produces a unique 14-bit modifier
// for use in updating the state bit-field of the generator used.

unsigned char dTable[64][16][16] = {
      { {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
         0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
        {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
         0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
        {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
         0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
        {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
         0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
        {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
         0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
        {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
         0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e}
},
{ {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}
},
{ {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00}
},
{ {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01}
},
{ {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02}
},
{ {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03}
},
{ {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04}
},
{ {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05}
},
{ {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06}
},
{ {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}
},
{ {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}
},
{ {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09}
},
{ {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a}
```

```
},
{ {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b}
},
{ {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c}
},
{ {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d}
},
{ {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
{0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
{0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
{0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
    {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
     0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01}
  },
  { {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
     0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
    {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
     0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
    {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
     0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
    {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
     0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
    {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
     0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
    {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
     0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
    {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
     0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
    {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
     0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
    {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
     0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
    {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
     0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
    {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
     0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
    {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00}
},
{ {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f}
},
{ {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e}
},
{ {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
{0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
{0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
{0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
{0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
{0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d}
},
{ {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c}
},
{ {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
    {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
     0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
    {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
     0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
    {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
     0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
    {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
     0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
    {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
     0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
    {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
     0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
    {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
     0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
    {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
     0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
    {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
     0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b}
  },
  { {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
     0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
    {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
     0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
    {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
     0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
    {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a}
},
{ {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
    0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
    0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
    0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
    0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
    0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
    0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
    0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
    0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
    0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
    0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
    0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
    0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09}
},
{ {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
       0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
      {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
       0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
      {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
       0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
      {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
       0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
      {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
       0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
      {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
       0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
      {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
       0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08}
    },
    { {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
       0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
      {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
       0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
      {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
       0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
      {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
       0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
      {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
       0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
      {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
       0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07}
},
{ {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
{0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
{0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06}
},
{ {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05}
},
{ {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04}
},
{ {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03}
},
{ {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
{0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
{0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
{0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02}
},
{ {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
    {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
     0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
    {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
     0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
    {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
     0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
    {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
     0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
    {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
     0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00}
  },
  { {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
     0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
    {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
    {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
     0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
    {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
     0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
    {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
     0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
    {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
     0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
    {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
     0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
    {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01}
},
{ {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02}
},
{ {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03}
},
{ {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04}
},
{ {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
    0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
   {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
    0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
   {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
    0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
   {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
   {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
    0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
   {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
    0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
   {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
   {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
   {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
    0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
   {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
    0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05}
 },
 { {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
    0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
   {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
    0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
   {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
    0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06}
```

```
},
{ {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07}
},
{ {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}
},
{ {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09}
},
{ {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
{0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
{0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
{0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
{0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a}
},
{ {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b}
},
{ {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c}
},
{ {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d}
},
{ {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
{0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
{0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
{0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
{0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
{0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
{0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
{0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
{0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
{0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
{0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
{0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
{0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
{0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f},
  {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e}
},
{ {0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06,
   0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e},
  {0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05,
   0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d},
  {0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04,
   0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c},
  {0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03,
   0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b},
  {0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02,
   0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a},
  {0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01,
   0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09},
  {0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00,
   0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08},
  {0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
   0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07},
  {0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e,
   0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06},
  {0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d,
   0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05},
  {0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
   0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03, 0x04},
  {0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b,
   0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02, 0x03},
  {0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a,
   0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01, 0x02},
  {0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
   0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00, 0x01},
  {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
   0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x00},
  {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
   0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}
},
{ {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01}
},
{ {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00}
},
{ {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f}
},
{ {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e}
},
{ {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d}
},
{ {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
{0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
{0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
{0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
{0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
{0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c}
},
{ {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
    {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
     0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
    {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
     0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
    {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
     0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
    {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
     0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b}
  },
  { {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
     0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
    {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
     0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
    {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
     0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
    {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
     0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
    {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
     0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
    {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
     0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
    {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
     0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
    {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
     0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
    {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a}
},
{ {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
    {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
     0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
    {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
     0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
    {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
     0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
    {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
     0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
    {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
     0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
    {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
     0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
    {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
     0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
    {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
     0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
    {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
     0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
    {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
     0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
    {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
     0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09}
  },
  { {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
     0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
    {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
{0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
{0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
{0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
{0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
{0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08}
},
{ {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07}
},
{ {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06}
},
{ {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
  {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05}
},
{ {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
{0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
{0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
{0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
{0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
{0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
{0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
{0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
{0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
{0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
{0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
{0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
{0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
{0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04}
},
{ {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02},
  {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
   0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
  {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
   0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
  {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
   0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
  {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
   0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
  {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
   0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
  {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
   0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
  {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
   0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
  {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
   0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
  {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
   0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03}
},
{ {0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09,
   0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01},
  {0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08,
   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00},
  {0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07,
   0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f},
  {0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06,
   0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e},
  {0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05,
   0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d},
  {0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04,
   0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c},
  {0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03,
         0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b},
        {0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02,
         0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a},
        {0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01,
         0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09},
        {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
         0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08},
        {0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f,
         0x0e, 0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07},
        {0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e,
         0x0d, 0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06},
        {0x04, 0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d,
         0x0c, 0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05},
        {0x03, 0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c,
         0x0b, 0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04},
        {0x02, 0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b,
         0x0a, 0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03},
        {0x01, 0x00, 0x0f, 0x0e, 0x0d, 0x0c, 0x0b, 0x0a,
         0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02}
      }
    };

unsigned short sTable[64][16][16] = {
      { {0x11ae, 0x1f51, 0x030f, 0x084f, 0x2d5b, 0x3c6f, 0x1ac5, 0x2e5c,
         0x296a, 0x0a9b, 0x0aa7, 0x1dca, 0x1fac, 0x1d51, 0x2cc1, 0x0b2e},
        {0x1659, 0x0a3e, 0x3b16, 0x2bbb, 0x16fc, 0x16f7, 0x32e2, 0x37ff,
 0x10a8, 0x3abf, 0x25f8, 0x147f, 0x351c, 0x2272, 0x2646, 0x2d30},
{0x27b5, 0x148b, 0x0032, 0x35f0, 0x18da, 0x17c3, 0x1152, 0x2852,
 0x0d77, 0x1824, 0x2695, 0x0bcf, 0x0bba, 0x0b1f, 0x050e, 0x3cb8},
{0x06c2, 0x0f5f, 0x2a45, 0x3cf6, 0x1292, 0x0aeb, 0x01d7, 0x3eaf,
 0x2845, 0x3a7a, 0x2387, 0x0c78, 0x07be, 0x07ef, 0x0129, 0x30d4},
{0x2b76, 0x2b0d, 0x344a, 0x1b40, 0x348c, 0x3f7c, 0x214f, 0x135e,
 0x351e, 0x39d5, 0x1654, 0x1274, 0x3745, 0x1ef0, 0x115c, 0x2482},
{0x3ccf, 0x3cc9, 0x03ab, 0x3f1f, 0x0325, 0x334a, 0x059b, 0x3416,
 0x3ca1, 0x24a9, 0x2d77, 0x0189, 0x0354, 0x34c3, 0x1691, 0x061b},
{0x2a53, 0x3937, 0x249f, 0x34eb, 0x1783, 0x2540, 0x3a5d, 0x12cf,
 0x17d7, 0x182e, 0x2cb1, 0x1491, 0x2284, 0x00aa, 0x2b09, 0x104f},
{0x0e9c, 0x3a03, 0x346a, 0x2b60, 0x37c7, 0x10a7, 0x3db4, 0x39bf,
 0x0087, 0x268d, 0x25ce, 0x2ebd, 0x147a, 0x0c5f, 0x025b, 0x2b8d},
{0x18df, 0x2cc7, 0x2d10, 0x1e49, 0x2da8, 0x1ae3, 0x2896, 0x1b25,
 0x13f2, 0x136e, 0x1fae, 0x27f5, 0x0af0, 0x1571, 0x1a00, 0x2c3e},
{0x27c1, 0x0f32, 0x0551, 0x2efd, 0x3f4f, 0x09fe, 0x380f, 0x0fe1,
 0x0bff, 0x168d, 0x0451, 0x0669, 0x0173, 0x1130, 0x082b, 0x0635},
{0x37e0, 0x220e, 0x39b1, 0x03c4, 0x02f7, 0x13a5, 0x3e2d, 0x3dda,
 0x2778, 0x301a, 0x1de7, 0x024f, 0x14a0, 0x2ce8, 0x3651, 0x338c},
{0x1798, 0x25d9, 0x0d76, 0x2f72, 0x0854, 0x3115, 0x16d8, 0x3f91,
 0x0eae, 0x1f09, 0x318c, 0x2002, 0x2218, 0x19bb, 0x1a5a, 0x3748},
{0x32dd, 0x1474, 0x2071, 0x2247, 0x3be3, 0x3cd9, 0x2c51, 0x3160,
 0x236f, 0x1b80, 0x2614, 0x2cf5, 0x3e99, 0x1035, 0x398a, 0x33cf},
{0x3947, 0x0ba4, 0x3fec, 0x2ba4, 0x0a5c, 0x3c4c, 0x1b29, 0x1913,
 0x3654, 0x2ae3, 0x21e8, 0x2114, 0x311b, 0x0ac7, 0x1e54, 0x19fb},
{0x0525, 0x069e, 0x1086, 0x3d0b, 0x2c63, 0x2a40, 0x3aa4, 0x3e6b,
   0x1730, 0x2c2e, 0x24ed, 0x05e4, 0x14e3, 0x29f7, 0x08ba, 0x1722},
  {0x3a72, 0x00ab, 0x2fb6, 0x2381, 0x0f25, 0x14b9, 0x1289, 0x2e8e,
   0x027c, 0x0ccd, 0x3439, 0x332f, 0x1874, 0x1ed8, 0x3ca0, 0x0624}
},
{ {0x1daa, 0x2dca, 0x3bdf, 0x3ce6, 0x1679, 0x10e9, 0x35ee, 0x01da,
   0x0498, 0x1165, 0x028c, 0x2196, 0x174a, 0x2953, 0x204f, 0x3513},
  {0x3f1d, 0x2023, 0x2ec2, 0x3339, 0x2233, 0x3370, 0x25e2, 0x10f5,
   0x2075, 0x3fcc, 0x1fde, 0x1ed9, 0x1e2f, 0x1f5e, 0x3abd, 0x0449},
  {0x338e, 0x01c0, 0x3b2e, 0x04c1, 0x3035, 0x1270, 0x0e7c, 0x0e83,
   0x24a7, 0x277f, 0x3adc, 0x31ef, 0x015d, 0x0e8b, 0x2c5f, 0x0178},
  {0x3754, 0x2b11, 0x34b7, 0x0738, 0x29fb, 0x1342, 0x1ca7, 0x37eb,
   0x0ae0, 0x260e, 0x3563, 0x1c7b, 0x383a, 0x0dd0, 0x0366, 0x3c3d},
  {0x17a8, 0x2f91, 0x14f4, 0x1cf4, 0x0218, 0x357f, 0x120d, 0x072d,
   0x1c0c, 0x2b81, 0x22c1, 0x30fe, 0x0742, 0x065a, 0x27b3, 0x197b},
  {0x2035, 0x38fa, 0x2ad6, 0x02ed, 0x2827, 0x02c4, 0x1bb5, 0x3da5,
   0x346b, 0x13e4, 0x0e46, 0x1f5b, 0x2e83, 0x29a5, 0x0f9f, 0x1863},
  {0x2560, 0x069f, 0x2d8c, 0x3884, 0x1f36, 0x2df0, 0x034a, 0x02aa,
   0x0088, 0x3835, 0x30f5, 0x2be9, 0x11a6, 0x3eeb, 0x1719, 0x1708},
  {0x08ec, 0x0a7b, 0x1673, 0x2d67, 0x181b, 0x3172, 0x3cd6, 0x1c25,
   0x1a4c, 0x3f11, 0x2587, 0x2678, 0x00b8, 0x0623, 0x1ead, 0x00b1},
  {0x170e, 0x1eb9, 0x1d10, 0x0c5a, 0x3a19, 0x0cc3, 0x11fa, 0x2f81,
   0x3733, 0x2a0b, 0x15c6, 0x122f, 0x18a7, 0x00de, 0x0184, 0x3dbd},
  {0x0bb0, 0x0a99, 0x0dd9, 0x2ce6, 0x1c8a, 0x3735, 0x0a5f, 0x1a83,
   0x3fc3, 0x1c33, 0x3725, 0x272a, 0x1ecf, 0x2322, 0x0133, 0x134d},
  {0x2ae0, 0x0829, 0x187e, 0x37cd, 0x30de, 0x346d, 0x1b03, 0x028b,
   0x004f, 0x1077, 0x08f0, 0x1f39, 0x1244, 0x392a, 0x0b12, 0x37b8},
  {0x3b4f, 0x1803, 0x25f6, 0x3a2c, 0x35dd, 0x0fcf, 0x0fc5, 0x1c02,
   0x1973, 0x256e, 0x3bf7, 0x1e60, 0x0603, 0x29d6, 0x3683, 0x1f1f},
  {0x3ee4, 0x3517, 0x1fd1, 0x08e2, 0x210c, 0x28b7, 0x00d2, 0x0f2e,
   0x2324, 0x3458, 0x39d1, 0x1a13, 0x08aa, 0x00b3, 0x28e7, 0x2634},
  {0x330e, 0x1663, 0x13f7, 0x10b3, 0x3e6a, 0x16d3, 0x0625, 0x1178,
   0x269f, 0x22bd, 0x3028, 0x27bc, 0x2e5a, 0x2bb2, 0x2e3c, 0x1340},
  {0x1764, 0x26da, 0x3ffc, 0x1699, 0x02d5, 0x1696, 0x3eac, 0x3b59,
   0x0b8b, 0x0296, 0x3953, 0x056c, 0x3a91, 0x3f9e, 0x166a, 0x26a8},
  {0x19f1, 0x2940, 0x22b6, 0x01a0, 0x23b0, 0x3c29, 0x1a82, 0x1187,
   0x0a2d, 0x04a1, 0x2d40, 0x3fd0, 0x0900, 0x36e2, 0x2a91, 0x3c49}
},
{ {0x3e0d, 0x33f3, 0x222a, 0x1ee4, 0x0cdb, 0x17ab, 0x2a8e, 0x0ab6,
   0x2564, 0x148a, 0x2a7e, 0x386b, 0x3636, 0x3cd5, 0x3024, 0x3b23},
  {0x3220, 0x3990, 0x2ce2, 0x3071, 0x1a5e, 0x1593, 0x26c6, 0x3b85,
   0x34c8, 0x32ce, 0x2a74, 0x39c0, 0x070d, 0x25e7, 0x1229, 0x08df},
  {0x22dc, 0x30e2, 0x3b83, 0x0ee3, 0x1c09, 0x3a06, 0x04f2, 0x0605,
   0x2d8b, 0x2c5b, 0x2fde, 0x0c37, 0x0ad2, 0x362a, 0x0ba1, 0x18c8},
  {0x240e, 0x062f, 0x17c6, 0x0671, 0x2901, 0x1b51, 0x0b63, 0x35a9,
   0x1804, 0x08b2, 0x33f5, 0x376c, 0x0d33, 0x074a, 0x180d, 0x2b32},
  {0x22c0, 0x0048, 0x1a1c, 0x1add, 0x3c7a, 0x01ed, 0x026a, 0x319c,
   0x1967, 0x204b, 0x0ee4, 0x2236, 0x1246, 0x0593, 0x08a5, 0x2eb9},
  {0x2c22, 0x0fe3, 0x3df7, 0x1d77, 0x332e, 0x2d1e, 0x0cc1, 0x2e81,
   0x127e, 0x1d40, 0x0937, 0x077b, 0x1781, 0x33da, 0x3d7c, 0x132d},
  {0x0a77, 0x11e6, 0x0176, 0x0bd0, 0x0156, 0x152c, 0x1c29, 0x02c5,
   0x10b8, 0x2279, 0x0ab5, 0x0389, 0x0884, 0x1eb5, 0x2472, 0x253e},
  {0x3d9a, 0x0e43, 0x2d91, 0x2f8f, 0x06bb, 0x3378, 0x3bfe, 0x1421,
     0x2a9c, 0x0bdb, 0x0d7e, 0x37ef, 0x3ccd, 0x35a7, 0x3e5e, 0x34b9},
    {0x2684, 0x2f57, 0x3061, 0x3ac9, 0x08a9, 0x379b, 0x212b, 0x37e4,
     0x01ba, 0x0ffc, 0x271d, 0x3ab9, 0x0482, 0x2ef0, 0x3ff3, 0x13c2},
    {0x06be, 0x004c, 0x227a, 0x3635, 0x3ef9, 0x0843, 0x0f34, 0x01eb,
     0x28e9, 0x1920, 0x1fbf, 0x3d4c, 0x3b8d, 0x22cf, 0x1d50, 0x3433},
    {0x3e87, 0x0a01, 0x32c1, 0x21ee, 0x38ae, 0x239f, 0x2908, 0x32fa,
     0x3e70, 0x328b, 0x1260, 0x2fc4, 0x04d1, 0x34ca, 0x0bd5, 0x105b},
    {0x21fe, 0x2e6d, 0x3cf5, 0x0bb7, 0x160a, 0x222b, 0x3f0b, 0x07ce,
     0x296f, 0x2446, 0x1428, 0x241f, 0x016b, 0x318a, 0x18f6, 0x204c},
    {0x0cda, 0x2a24, 0x2019, 0x20db, 0x140c, 0x1e1b, 0x1c05, 0x2d20,
     0x3692, 0x298e, 0x3432, 0x3597, 0x28f0, 0x3b4e, 0x1f8a, 0x126d},
    {0x3b70, 0x0963, 0x1172, 0x2549, 0x0447, 0x33c8, 0x38a4, 0x01a3,
     0x10e6, 0x1174, 0x0770, 0x13f5, 0x188b, 0x3182, 0x1a19, 0x0334},
    {0x12ec, 0x3c1f, 0x0b1d, 0x12d0, 0x009b, 0x184c, 0x0013, 0x2296,
     0x0081, 0x0338, 0x3bd8, 0x08a6, 0x039c, 0x3b01, 0x125c, 0x0e01},
    {0x147e, 0x3e82, 0x1286, 0x142a, 0x01e7, 0x3c64, 0x1ca1, 0x0e53,
     0x3541, 0x375b, 0x0300, 0x2099, 0x023e, 0x1b7d, 0x13f6, 0x084d}
  },
  { {0x0d79, 0x1f2d, 0x3cb2, 0x02d9, 0x0fff, 0x28a6, 0x1da1, 0x3c09,
     0x0df8, 0x11a0, 0x247e, 0x2d35, 0x14bf, 0x209c, 0x28e5, 0x177c},
    {0x03e7, 0x3b22, 0x031e, 0x1fce, 0x0192, 0x39dc, 0x11b3, 0x310b,
     0x30d0, 0x3b8e, 0x0c03, 0x22fc, 0x0e2c, 0x2820, 0x2115, 0x1222},
    {0x048b, 0x1161, 0x0901, 0x0139, 0x232e, 0x1aa1, 0x0615, 0x0830,
     0x2dc3, 0x29d5, 0x1fa6, 0x37a6, 0x11eb, 0x0abc, 0x25d0, 0x3b9d},
    {0x32df, 0x3f99, 0x1276, 0x1f13, 0x3c7d, 0x18ba, 0x2996, 0x2815,
     0x2032, 0x0f0e, 0x2358, 0x3554, 0x362d, 0x3509, 0x39da, 0x03fa},
  {0x3a3a, 0x0b2c, 0x1525, 0x3319, 0x3cfe, 0x345d, 0x016d, 0x3d24,
   0x3b8b, 0x0231, 0x1bd1, 0x2dc8, 0x3246, 0x1df3, 0x3acd, 0x3e3c},
  {0x2e8a, 0x02e3, 0x2a37, 0x3790, 0x18cc, 0x3ab0, 0x2572, 0x2990,
   0x272c, 0x2b89, 0x3de6, 0x3ffd, 0x1c54, 0x2e44, 0x359a, 0x2aa6},
  {0x1420, 0x200f, 0x0910, 0x3958, 0x0bec, 0x0aa6, 0x1a0e, 0x233c,
   0x0faa, 0x39fa, 0x1224, 0x1d9e, 0x3b84, 0x0977, 0x28eb, 0x1364},
  {0x1ef7, 0x3b64, 0x29f1, 0x2f9a, 0x1821, 0x2c7b, 0x2310, 0x30e0,
   0x3c9a, 0x04da, 0x15ce, 0x1a41, 0x31dd, 0x1e4a, 0x337a, 0x3948},
  {0x3210, 0x0cca, 0x1d03, 0x0d8b, 0x3cd8, 0x18e6, 0x314f, 0x1aab,
   0x3333, 0x0571, 0x1dfe, 0x11a7, 0x0596, 0x17d4, 0x3bbe, 0x1071},
  {0x39e7, 0x2420, 0x09a1, 0x2258, 0x38d7, 0x11ec, 0x07d6, 0x2e41,
   0x3fd8, 0x0f76, 0x223e, 0x33de, 0x023d, 0x1c41, 0x34fa, 0x1bf2},
  {0x10e0, 0x1b10, 0x3180, 0x3476, 0x23df, 0x3343, 0x23de, 0x1681,
   0x2c62, 0x2f56, 0x199f, 0x1969, 0x0278, 0x1882, 0x2462, 0x1563},
  {0x32c8, 0x31b4, 0x04d2, 0x06bc, 0x283b, 0x3a9f, 0x2a68, 0x27c4,
   0x265b, 0x1597, 0x32ec, 0x0981, 0x0e97, 0x3ad1, 0x18c2, 0x1f7a},
  {0x3e92, 0x26d9, 0x3c42, 0x2c8a, 0x1438, 0x20a5, 0x33b3, 0x13ef,
   0x2184, 0x2b2f, 0x0f1c, 0x28b1, 0x3e4e, 0x2b08, 0x0c2f, 0x3891},
  {0x071e, 0x3856, 0x29c6, 0x2b6b, 0x141a, 0x1a61, 0x2110, 0x28c8,
   0x0124, 0x1cb4, 0x1732, 0x0193, 0x3cbe, 0x2b67, 0x0873, 0x04a6},
  {0x3849, 0x0051, 0x0ea0, 0x3eba, 0x0d9f, 0x35b8, 0x1f08, 0x2209,
   0x12da, 0x056e, 0x2db4, 0x3f72, 0x2874, 0x20c5, 0x3ecd, 0x2d6d},
  {0x206f, 0x0945, 0x0839, 0x0037, 0x1d6b, 0x2a5a, 0x0b88, 0x322a,
   0x1c17, 0x1258, 0x169a, 0x0045, 0x3c47, 0x1af1, 0x15c0, 0x1f4e}
},
{ {0x261a, 0x3b6d, 0x033d, 0x35d9, 0x0b09, 0x330a, 0x1b8b, 0x2da5,
 0x1b6f, 0x06a0, 0x1818, 0x3b4d, 0x20eb, 0x174e, 0x3f69, 0x3eb5},
{0x3641, 0x1d41, 0x133f, 0x29c9, 0x26d3, 0x2012, 0x3588, 0x335f,
 0x2961, 0x05ec, 0x3cea, 0x06e0, 0x01ff, 0x1b91, 0x14f8, 0x2559},
{0x0a60, 0x28fc, 0x0e91, 0x1848, 0x28bd, 0x3f36, 0x0758, 0x254e,
 0x0f9d, 0x0cc8, 0x1182, 0x1dfd, 0x17de, 0x3fb7, 0x35fa, 0x1b9d},
{0x3134, 0x04bb, 0x20b2, 0x173b, 0x059d, 0x39d9, 0x30ec, 0x2a1b,
 0x16ce, 0x1892, 0x2fab, 0x2f6d, 0x33bc, 0x2784, 0x2cda, 0x01bd},
{0x0009, 0x253f, 0x14f0, 0x26fa, 0x0595, 0x0e7f, 0x00a1, 0x38d5,
 0x1b30, 0x1476, 0x390e, 0x0b6a, 0x1e28, 0x0e65, 0x116f, 0x19fd},
{0x29d7, 0x09ea, 0x3025, 0x11e7, 0x3a71, 0x33f0, 0x1a50, 0x0e2e,
 0x3586, 0x0812, 0x3a8d, 0x1009, 0x1d18, 0x1105, 0x3274, 0x0e38},
{0x0720, 0x0065, 0x104d, 0x123c, 0x26dd, 0x07d9, 0x0613, 0x1fc7,
 0x217d, 0x109b, 0x269a, 0x0b17, 0x0e02, 0x18a8, 0x1fa3, 0x185d},
{0x350d, 0x1b65, 0x0c9b, 0x2c64, 0x2e0a, 0x2a72, 0x1f34, 0x02af,
 0x003b, 0x163c, 0x1271, 0x0d48, 0x2663, 0x2f93, 0x132e, 0x04cd},
{0x38d6, 0x3949, 0x1fed, 0x21df, 0x1435, 0x0364, 0x2318, 0x0ec0,
 0x06f9, 0x0633, 0x0e20, 0x108c, 0x213b, 0x1934, 0x2aad, 0x16bd},
{0x3396, 0x309b, 0x0096, 0x3713, 0x0579, 0x34a4, 0x0a4f, 0x0871,
 0x3f8e, 0x0930, 0x2d0a, 0x3fe7, 0x0344, 0x2ba3, 0x13e9, 0x39a4},
{0x03e0, 0x2960, 0x3bf9, 0x0ff7, 0x39bb, 0x34dc, 0x3c60, 0x12d9,
 0x34f8, 0x1d00, 0x1570, 0x3915, 0x1502, 0x264f, 0x02d6, 0x3969},
{0x2c5d, 0x12f0, 0x1eec, 0x2622, 0x1adc, 0x0dbb, 0x340e, 0x3a86,
 0x134b, 0x1ec9, 0x3938, 0x3e9c, 0x3259, 0x1c60, 0x0666, 0x3e18},
{0x2207, 0x066d, 0x30ac, 0x3ae6, 0x2d97, 0x0c73, 0x08e3, 0x1802,
 0x1ffe, 0x38c5, 0x1f80, 0x114b, 0x23fc, 0x3f28, 0x1d88, 0x0e74},
{0x2a47, 0x368f, 0x11f4, 0x1f20, 0x25c4, 0x3cba, 0x3522, 0x07b6,
   0x3dc8, 0x272b, 0x2dcb, 0x3ae1, 0x351a, 0x0827, 0x382e, 0x2c37},
  {0x3138, 0x3fb2, 0x3db1, 0x086d, 0x386c, 0x0284, 0x0960, 0x1d78,
   0x0a98, 0x2ab9, 0x20b5, 0x2a8c, 0x203b, 0x1fc4, 0x03ea, 0x15b0},
  {0x0dcb, 0x3b9c, 0x22ab, 0x1190, 0x2045, 0x07c3, 0x19c8, 0x3be7,
   0x2e58, 0x278b, 0x1def, 0x32e5, 0x2d70, 0x3795, 0x1c4f, 0x1723}
},
{ {0x2dce, 0x0788, 0x091f, 0x3af2, 0x1dff, 0x26ea, 0x36a3, 0x1859,
   0x0f0b, 0x0885, 0x00e5, 0x3f21, 0x03a6, 0x113e, 0x0916, 0x1531},
  {0x1711, 0x1c94, 0x396e, 0x0809, 0x1b19, 0x229a, 0x08cf, 0x20e0,
   0x18e4, 0x2f2b, 0x35bb, 0x07eb, 0x1504, 0x05d2, 0x3c73, 0x0cd8},
  {0x08e0, 0x2771, 0x04d5, 0x3c35, 0x3482, 0x0822, 0x1112, 0x1f05,
   0x0a3b, 0x30be, 0x1ad2, 0x2f66, 0x2cbe, 0x0d90, 0x251b, 0x2773},
  {0x1c90, 0x1669, 0x34d8, 0x3301, 0x1aa8, 0x17ba, 0x37bd, 0x2876,
   0x2470, 0x1e5a, 0x280e, 0x0710, 0x3429, 0x04ed, 0x2ef9, 0x24bc},
  {0x0e9e, 0x34f5, 0x12eb, 0x0fe2, 0x34b2, 0x14fd, 0x2acd, 0x16e5,
   0x0359, 0x0532, 0x0e2d, 0x290a, 0x10e3, 0x0a35, 0x0ed6, 0x3e02},
  {0x2cf0, 0x1fdc, 0x1a2d, 0x04c9, 0x014c, 0x0ced, 0x1f12, 0x26f0,
   0x13b0, 0x37e8, 0x2a3a, 0x11e0, 0x0d45, 0x310f, 0x29d1, 0x0462},
  {0x0a75, 0x0813, 0x24c9, 0x3626, 0x1a53, 0x1fd9, 0x186a, 0x0fbc,
   0x04ac, 0x1a89, 0x25f7, 0x1993, 0x3b8a, 0x113a, 0x3d6a, 0x3ee1},
  {0x3ebb, 0x0d82, 0x0f2c, 0x39ec, 0x19b9, 0x0b6b, 0x0a97, 0x046f,
   0x141f, 0x08e9, 0x3bf3, 0x1396, 0x22cd, 0x054e, 0x3529, 0x2a8d},
  {0x245f, 0x0665, 0x1017, 0x2259, 0x39c3, 0x3038, 0x2fbb, 0x044f,
   0x360f, 0x0232, 0x318f, 0x1962, 0x0502, 0x38ee, 0x2da9, 0x110b},
  {0x3eec, 0x0ad9, 0x38dc, 0x10d0, 0x0f7c, 0x380a, 0x2b62, 0x2329,
   0x0aa9, 0x31c8, 0x2629, 0x3766, 0x388c, 0x1e16, 0x0c7a, 0x081b},
  {0x03f9, 0x1c32, 0x2c7f, 0x3601, 0x2eac, 0x2ea6, 0x1ec5, 0x3451,
   0x21c6, 0x3206, 0x0cdc, 0x193b, 0x0e41, 0x1a60, 0x14e1, 0x00b0},
  {0x1c89, 0x0b70, 0x2216, 0x2d93, 0x0070, 0x040a, 0x365d, 0x1204,
   0x0375, 0x2392, 0x10c5, 0x061f, 0x32e9, 0x3d0a, 0x2ed3, 0x0d6e},
  {0x0074, 0x3f27, 0x1cb9, 0x1b98, 0x2e26, 0x22f3, 0x1e7d, 0x0cb7,
   0x251f, 0x3cac, 0x00be, 0x1c36, 0x237e, 0x3d17, 0x0182, 0x3515},
  {0x3317, 0x0953, 0x3c71, 0x167e, 0x095e, 0x0fc0, 0x0a6b, 0x0d75,
   0x3e7a, 0x286f, 0x0fb0, 0x2c21, 0x2be0, 0x30f2, 0x1063, 0x1249},
  {0x3630, 0x37d6, 0x3716, 0x2a35, 0x056f, 0x2106, 0x0752, 0x01f2,
   0x2375, 0x31ed, 0x19b7, 0x0391, 0x08b9, 0x1896, 0x2193, 0x2f70},
  {0x0af7, 0x0a59, 0x0f20, 0x211a, 0x2f2e, 0x11d1, 0x165a, 0x0bbf,
   0x1d05, 0x14ac, 0x2240, 0x09ff, 0x0b4a, 0x07f1, 0x1236, 0x06e9}
},
{ {0x11ac, 0x39e6, 0x1016, 0x1bd9, 0x0b6d, 0x2804, 0x09b9, 0x255a,
   0x3ce7, 0x3787, 0x17dd, 0x3f04, 0x04ef, 0x068a, 0x038e, 0x0aa2},
  {0x1f03, 0x0103, 0x32d8, 0x30b6, 0x11b0, 0x2167, 0x0e1a, 0x32d5,
   0x249e, 0x311a, 0x111f, 0x0322, 0x29f6, 0x3c43, 0x2205, 0x12d6},
  {0x3d5c, 0x2f59, 0x28e4, 0x2bdb, 0x3cb9, 0x2199, 0x0a4a, 0x19b3,
   0x0e05, 0x2b0c, 0x2b1b, 0x1c7f, 0x17c4, 0x30e9, 0x2533, 0x3c31},
  {0x1fba, 0x3297, 0x1445, 0x00fd, 0x130a, 0x0f16, 0x08fe, 0x07e0,
   0x2503, 0x228f, 0x13bf, 0x3b5c, 0x36f4, 0x377f, 0x2869, 0x20e4},
  {0x3d97, 0x1a9a, 0x2f08, 0x125b, 0x0d8e, 0x1bc0, 0x034c, 0x002a,
   0x1150, 0x2c4b, 0x0406, 0x26c8, 0x020c, 0x205b, 0x139e, 0x2d3f},
  {0x1b37, 0x3566, 0x099f, 0x036c, 0x2ddb, 0x0456, 0x3758, 0x3deb,
   0x0f03, 0x0f14, 0x0f19, 0x1cd1, 0x26f8, 0x3d57, 0x02f9, 0x03a7},
  {0x21cd, 0x39f5, 0x3348, 0x16f9, 0x1a2f, 0x0a71, 0x3572, 0x371b,
   0x12dc, 0x365f, 0x192c, 0x12bc, 0x0b84, 0x2277, 0x1a9c, 0x3d7d},
  {0x2a21, 0x29bf, 0x3846, 0x0d1e, 0x3d1f, 0x080d, 0x150d, 0x0ef4,
   0x1ed3, 0x096d, 0x3b39, 0x2fb8, 0x01f5, 0x377a, 0x137a, 0x1ed6},
  {0x070c, 0x22d5, 0x336d, 0x154d, 0x234c, 0x3570, 0x39e2, 0x06af,
   0x3373, 0x03bc, 0x36fe, 0x26c0, 0x3834, 0x251d, 0x3ddf, 0x3cbd},
  {0x0f78, 0x32eb, 0x10c9, 0x3489, 0x202b, 0x366c, 0x054d, 0x044c,
   0x1403, 0x2f55, 0x393e, 0x3ba4, 0x0a4d, 0x0007, 0x21f8, 0x1bed},
  {0x0b31, 0x03e6, 0x03f3, 0x0e1f, 0x0c5b, 0x06a2, 0x1ba1, 0x38d2,
   0x09db, 0x03fd, 0x355e, 0x0400, 0x0d52, 0x34df, 0x1aeb, 0x3b1d},
  {0x2db6, 0x1703, 0x1879, 0x06f7, 0x3168, 0x2c95, 0x0250, 0x2ab5,
   0x3b52, 0x069b, 0x2728, 0x2281, 0x1f1d, 0x045b, 0x2ef3, 0x23b8},
  {0x2840, 0x1c81, 0x27c7, 0x1393, 0x1bc1, 0x3627, 0x0e13, 0x2c80,
   0x1f0b, 0x33d8, 0x33be, 0x2366, 0x1295, 0x2595, 0x1bc2, 0x2293},
  {0x0d11, 0x3c38, 0x131a, 0x22fd, 0x1d0f, 0x0f15, 0x053d, 0x2915,
   0x1949, 0x309c, 0x097c, 0x05a8, 0x2ed4, 0x3dd6, 0x06d7, 0x18ad},
  {0x2d44, 0x3050, 0x36cf, 0x1079, 0x16cb, 0x39ab, 0x1bbb, 0x26e7,
   0x2bcf, 0x28db, 0x23f8, 0x0011, 0x0ca6, 0x1ddb, 0x1e7e, 0x3f6b},
  {0x3be1, 0x1b16, 0x21b6, 0x2fa9, 0x2775, 0x1ab2, 0x2aa0, 0x3858,
   0x3a84, 0x035b, 0x1004, 0x2f0a, 0x2450, 0x2d57, 0x2b18, 0x343c}
},
{ {0x2068, 0x1094, 0x3c3f, 0x2de3, 0x0561, 0x2fda, 0x1923, 0x319d,
   0x2b56, 0x228b, 0x0130, 0x02a0, 0x2e0d, 0x1a57, 0x3824, 0x0b9e},
  {0x1537, 0x23b9, 0x3e88, 0x0f70, 0x2bf8, 0x3ac6, 0x3cc4, 0x171b,
   0x2f16, 0x19df, 0x3aea, 0x0b2b, 0x0a39, 0x001a, 0x2a60, 0x3b51},
  {0x1dfb, 0x3153, 0x3971, 0x046e, 0x3f4d, 0x1cba, 0x2017, 0x13d8,
   0x382f, 0x2556, 0x0e03, 0x129e, 0x3662, 0x10af, 0x1c9c, 0x0664},
{0x0a93, 0x0c17, 0x1c12, 0x2c69, 0x368d, 0x237b, 0x2673, 0x0145,
 0x0897, 0x37a5, 0x276d, 0x356c, 0x0a2c, 0x2d22, 0x3b65, 0x010c},
{0x1147, 0x1d23, 0x39f0, 0x0682, 0x1310, 0x089a, 0x23ac, 0x09b8,
 0x1158, 0x1f4b, 0x0a43, 0x3005, 0x0320, 0x23e8, 0x01e3, 0x1d55},
{0x0ecd, 0x22a3, 0x270e, 0x3dbb, 0x1ba6, 0x396c, 0x14a2, 0x259e,
 0x1040, 0x2619, 0x2cfb, 0x1252, 0x2854, 0x0dc2, 0x0c28, 0x1b56},
{0x0750, 0x18f5, 0x2e4f, 0x2998, 0x17ec, 0x2615, 0x33d2, 0x0d5a,
 0x0fb8, 0x01fa, 0x2b9e, 0x1cf3, 0x3756, 0x39d8, 0x1dac, 0x01f1},
{0x15d8, 0x2a9a, 0x204e, 0x06eb, 0x15e9, 0x24d7, 0x05f4, 0x20f5,
 0x3354, 0x1232, 0x1e01, 0x24d4, 0x2e11, 0x1171, 0x3a25, 0x2031},
{0x343a, 0x2c73, 0x3da1, 0x0859, 0x36ed, 0x24f3, 0x3232, 0x2cb0,
 0x1639, 0x215e, 0x0536, 0x193f, 0x09ee, 0x08da, 0x137f, 0x2c03},
{0x3db0, 0x2b19, 0x130f, 0x0895, 0x1e77, 0x036f, 0x3ac7, 0x3bc0,
 0x2d48, 0x3ae9, 0x3183, 0x2345, 0x0801, 0x388f, 0x00fa, 0x2478},
{0x1ce0, 0x2a08, 0x037e, 0x32da, 0x14ec, 0x0988, 0x3302, 0x3e85,
 0x3431, 0x2cc4, 0x285e, 0x1a85, 0x2826, 0x1237, 0x386d, 0x1bb8},
{0x12a8, 0x2c3a, 0x1bc7, 0x2309, 0x2d09, 0x01cf, 0x2c29, 0x3ff9,
 0x11df, 0x3ad6, 0x25a4, 0x0068, 0x3fe3, 0x344f, 0x3a8f, 0x051a},
{0x0ce0, 0x12b9, 0x33db, 0x0eb8, 0x25e6, 0x0b6e, 0x0c33, 0x3ce5,
 0x0f3f, 0x1e30, 0x28a1, 0x0a56, 0x358c, 0x107b, 0x3780, 0x3d4d},
{0x04a7, 0x0a45, 0x0cea, 0x33b2, 0x0a24, 0x3e29, 0x0033, 0x2f64,
 0x3ebf, 0x2436, 0x2624, 0x13e1, 0x1c2c, 0x2fc7, 0x3dba, 0x36d9},
{0x06e4, 0x2793, 0x19a4, 0x0658, 0x3bb5, 0x1f5f, 0x1667, 0x3bb0,
 0x0938, 0x0117, 0x1b61, 0x00f6, 0x2121, 0x0f8d, 0x1c4b, 0x3d04},
{0x0afb, 0x3ca5, 0x18c5, 0x0b9d, 0x1788, 0x3e8f, 0x03fc, 0x2abf,
 0x1698, 0x21a9, 0x1b34, 0x1241, 0x2034, 0x13c1, 0x2578, 0x25a9}
```

```
},
{ {0x03ae, 0x19bc, 0x3c97, 0x23cf, 0x0d9b, 0x30a6, 0x3793, 0x0dfb,
   0x2d52, 0x2ff4, 0x1901, 0x3fea, 0x3f64, 0x0f9b, 0x3667, 0x313e},
  {0x3689, 0x23ba, 0x3364, 0x1a66, 0x0095, 0x0c8e, 0x1e66, 0x18d5,
   0x3c37, 0x3562, 0x225a, 0x34dd, 0x292c, 0x1287, 0x3ff4, 0x3f62},
  {0x3603, 0x345e, 0x2fdf, 0x1508, 0x1350, 0x1894, 0x2b65, 0x21cb,
   0x2c45, 0x13eb, 0x320f, 0x1e6e, 0x0531, 0x37f8, 0x16c2, 0x2802},
  {0x17cc, 0x2e18, 0x3f88, 0x315b, 0x3775, 0x1bba, 0x36d0, 0x3462,
   0x01a1, 0x0fef, 0x1a17, 0x27b2, 0x2571, 0x0168, 0x2c9f, 0x1f70},
  {0x1d36, 0x2ae2, 0x1c19, 0x0ebf, 0x0c8c, 0x2ccd, 0x2ac5, 0x3311,
   0x14da, 0x240c, 0x00bc, 0x2e31, 0x0108, 0x0a17, 0x2e8b, 0x25fd},
  {0x2010, 0x020f, 0x3b14, 0x27b4, 0x10b0, 0x2829, 0x13ba, 0x1595,
   0x1308, 0x3b30, 0x3f10, 0x32c5, 0x3332, 0x14e0, 0x13e7, 0x39f1},
  {0x1486, 0x0941, 0x3580, 0x0ac0, 0x1217, 0x1049, 0x26d2, 0x34c5,
   0x2c6e, 0x065c, 0x30ef, 0x0ef1, 0x3568, 0x0b11, 0x27b0, 0x2c30},
  {0x2bac, 0x0c5c, 0x1bae, 0x13c6, 0x147c, 0x12c0, 0x21c7, 0x231e,
   0x255b, 0x2554, 0x151e, 0x2b0b, 0x15fa, 0x0281, 0x1592, 0x33fb},
  {0x07a6, 0x1f73, 0x197f, 0x257e, 0x094c, 0x02b5, 0x1a65, 0x0177,
   0x24c3, 0x19ab, 0x3c91, 0x3116, 0x38e4, 0x2caf, 0x2401, 0x2362},
  {0x1e90, 0x0a9a, 0x1155, 0x31c7, 0x3170, 0x1813, 0x0e2f, 0x2f76,
   0x0a61, 0x1e2d, 0x0714, 0x1c08, 0x283c, 0x2bd4, 0x0ed4, 0x36bb},
  {0x0fe0, 0x2cbf, 0x31a1, 0x0706, 0x255e, 0x1dd7, 0x1197, 0x29e1,
   0x2a4a, 0x1099, 0x086b, 0x2951, 0x1484, 0x3819, 0x047e, 0x041d},
  {0x3537, 0x279f, 0x2364, 0x2404, 0x30ae, 0x33f9, 0x02c2, 0x32ea,
   0x38c9, 0x23ed, 0x0987, 0x3a31, 0x2212, 0x3386, 0x1d6f, 0x3201},
  {0x2a05, 0x3253, 0x2a26, 0x12df, 0x3042, 0x1059, 0x2173, 0x2fe5,
   0x2eff, 0x13af, 0x303b, 0x1cfc, 0x3244, 0x0397, 0x152e, 0x0bfb},
  {0x3822, 0x28ae, 0x15f3, 0x12e7, 0x12c6, 0x0c25, 0x0ae7, 0x3217,
   0x2a84, 0x3a33, 0x0fa0, 0x2dee, 0x032e, 0x0da2, 0x0fc2, 0x0c13},
  {0x280a, 0x2267, 0x2df5, 0x0d5f, 0x046c, 0x1022, 0x2437, 0x0547,
   0x2bc6, 0x3208, 0x1c59, 0x1fc8, 0x3fa0, 0x313d, 0x0fea, 0x1977},
  {0x18ef, 0x1b38, 0x0746, 0x15a9, 0x3327, 0x39b3, 0x145e, 0x129d,
   0x1b68, 0x1262, 0x0358, 0x3174, 0x1e7a, 0x3e16, 0x096a, 0x1b84}
},
{ {0x0b00, 0x035a, 0x1c6e, 0x1744, 0x17c8, 0x0bab, 0x3dee, 0x31b2,
   0x292d, 0x19c5, 0x16b2, 0x23f4, 0x1754, 0x2ede, 0x2dc1, 0x0ea4},
  {0x2122, 0x08e8, 0x0e5c, 0x3155, 0x247d, 0x1a15, 0x3687, 0x0df5,
   0x0a38, 0x09af, 0x0ff6, 0x03a1, 0x04fb, 0x0f4e, 0x1d74, 0x1d7e},
  {0x01c7, 0x124f, 0x3622, 0x3e3b, 0x0a52, 0x1494, 0x28a9, 0x0e9f,
   0x238f, 0x105e, 0x0cf8, 0x3047, 0x30c4, 0x2e38, 0x01e4, 0x2b85},
  {0x1de5, 0x0d35, 0x1210, 0x1478, 0x3ad9, 0x28de, 0x2f17, 0x225d,
   0x2bc5, 0x3e61, 0x0e94, 0x18fa, 0x2685, 0x0648, 0x2379, 0x2bff},
  {0x01d4, 0x15b3, 0x1e96, 0x3f58, 0x1370, 0x3722, 0x0776, 0x3952,
   0x258d, 0x3222, 0x0cc0, 0x2131, 0x32f4, 0x1564, 0x0cd4, 0x0857},
  {0x2a28, 0x1858, 0x1d21, 0x1940, 0x0e07, 0x29b3, 0x08b3, 0x299c,
   0x0b74, 0x1d26, 0x0d31, 0x0f77, 0x337f, 0x3f8b, 0x0e4e, 0x038a},
  {0x3c54, 0x185a, 0x300b, 0x3e63, 0x0eff, 0x3173, 0x3582, 0x1b2e,
   0x107f, 0x3471, 0x0aa3, 0x270c, 0x2d84, 0x3181, 0x270b, 0x262f},
  {0x2c40, 0x1ad1, 0x088e, 0x2435, 0x2cce, 0x2b0f, 0x3902, 0x2c31,
   0x2c81, 0x22db, 0x38a0, 0x27af, 0x3956, 0x19ff, 0x0249, 0x24b0},
  {0x1ed2, 0x248a, 0x2adf, 0x2e59, 0x3749, 0x0a64, 0x20c1, 0x2eb7,
   0x2ca1, 0x0372, 0x167a, 0x2cff, 0x038f, 0x2800, 0x07ca, 0x0c4d},
  {0x1380, 0x31a3, 0x141b, 0x3128, 0x08fa, 0x1415, 0x372b, 0x0f90,
   0x3bd6, 0x22c3, 0x0326, 0x2fe7, 0x0935, 0x06ec, 0x1702, 0x34ed},
  {0x2463, 0x3e8b, 0x199a, 0x342f, 0x2542, 0x2b22, 0x3c1c, 0x1167,
   0x3278, 0x0a1e, 0x0e5e, 0x1efc, 0x0e76, 0x2e5d, 0x2f21, 0x280c},
  {0x1689, 0x3003, 0x2924, 0x0097, 0x2d69, 0x0d1b, 0x268b, 0x169c,
   0x38c4, 0x246b, 0x2129, 0x1f9a, 0x0fed, 0x0465, 0x2b4e, 0x0f40},
  {0x1bfb, 0x0546, 0x0bfe, 0x0d7d, 0x2da4, 0x2635, 0x02eb, 0x0be7,
   0x1dd9, 0x23ca, 0x0cbe, 0x2d45, 0x0ace, 0x0d41, 0x26c5, 0x3d20},
  {0x1adf, 0x2ee6, 0x2b1e, 0x2527, 0x3955, 0x0c96, 0x219d, 0x3ea1,
   0x3b1c, 0x06c0, 0x136d, 0x39e1, 0x376a, 0x36c3, 0x0523, 0x1006},
  {0x3d54, 0x03e8, 0x3277, 0x3cf1, 0x2903, 0x1239, 0x0cb2, 0x0d19,
   0x3031, 0x1c26, 0x17d1, 0x1911, 0x34d1, 0x0bd6, 0x02d3, 0x34ba},
  {0x2f79, 0x2a12, 0x2eee, 0x12c7, 0x11a2, 0x234a, 0x0072, 0x0e0a,
   0x3d8d, 0x395b, 0x2d9d, 0x3841, 0x0634, 0x3e54, 0x3b9e, 0x1c61}
},
{ {0x0084, 0x267b, 0x296d, 0x2cee, 0x35cc, 0x2720, 0x1843, 0x2378,
   0x1eae, 0x1ee8, 0x2d18, 0x33d0, 0x0e6c, 0x3357, 0x2632, 0x2e2b},
  {0x35b4, 0x2c42, 0x04dc, 0x08ad, 0x19ba, 0x0172, 0x2523, 0x1185,
   0x1847, 0x1288, 0x0511, 0x27d6, 0x3eca, 0x0bc0, 0x1e79, 0x0dc6},
  {0x0dfc, 0x0308, 0x0905, 0x0505, 0x0e9a, 0x0290, 0x1033, 0x1e05,
   0x15f4, 0x066c, 0x3a66, 0x0a23, 0x3cbf, 0x1771, 0x2204, 0x26ee},
  {0x10b1, 0x3e64, 0x0560, 0x161c, 0x26d8, 0x221b, 0x1cf8, 0x0f4a,
   0x132c, 0x191b, 0x2c72, 0x126f, 0x0728, 0x0f9a, 0x1eda, 0x2c2b},
  {0x3ac0, 0x0411, 0x16a6, 0x2966, 0x2580, 0x007f, 0x2b30, 0x1c2b,
   0x1373, 0x3995, 0x0e9b, 0x3a5c, 0x0c31, 0x2f32, 0x0311, 0x1b70},
  {0x2476, 0x2cbd, 0x1117, 0x2243, 0x1068, 0x0bce, 0x2d59, 0x363b,
   0x2ee7, 0x0545, 0x3e00, 0x1fe0, 0x35d2, 0x3b7c, 0x033e, 0x3518},
  {0x0711, 0x1a1b, 0x1506, 0x2e57, 0x0d42, 0x2e6f, 0x0cd2, 0x1f0f,
   0x0089, 0x36ab, 0x370d, 0x2356, 0x0f81, 0x397b, 0x31ae, 0x2394},
  {0x1850, 0x2643, 0x2174, 0x10cf, 0x1bde, 0x3bc8, 0x2d79, 0x1740,
   0x30cb, 0x31a7, 0x0195, 0x2b48, 0x0692, 0x2f7a, 0x1b9a, 0x37a4},
  {0x2b90, 0x3da0, 0x3853, 0x17ce, 0x2433, 0x3d3f, 0x38de, 0x1925,
   0x21f9, 0x09bc, 0x1228, 0x1f87, 0x266f, 0x1e69, 0x006b, 0x1974},
  {0x34b0, 0x261c, 0x33e4, 0x29f3, 0x34e2, 0x0c8b, 0x3a61, 0x317c,
   0x2591, 0x2535, 0x366b, 0x27a0, 0x1bc9, 0x38d9, 0x2a51, 0x264e},
  {0x2aa7, 0x3e07, 0x22ae, 0x3604, 0x04de, 0x3c96, 0x04df, 0x3be9,
   0x23e7, 0x2977, 0x1439, 0x0f51, 0x0ce8, 0x27a3, 0x2731, 0x18ce},
  {0x0ce4, 0x2303, 0x0f61, 0x05a0, 0x1964, 0x1d62, 0x326a, 0x1829,
   0x1da5, 0x1947, 0x2494, 0x2e79, 0x3fc5, 0x196b, 0x1ccb, 0x264d},
  {0x16ee, 0x11dd, 0x1026, 0x0fa1, 0x1f69, 0x10aa, 0x2082, 0x318e,
   0x09c2, 0x26d1, 0x19fa, 0x238c, 0x1b9e, 0x3442, 0x2f83, 0x3fcb},
  {0x3c75, 0x2567, 0x2227, 0x3d49, 0x0d05, 0x331a, 0x1dcb, 0x3f66,
   0x14bc, 0x36a8, 0x07f8, 0x1a1d, 0x0415, 0x1869, 0x33e5, 0x12fe},
  {0x2a4c, 0x2e60, 0x1e3a, 0x3638, 0x3923, 0x24f8, 0x2e46, 0x304e,
   0x2fef, 0x3eef, 0x1faf, 0x0c11, 0x19cc, 0x2daf, 0x063c, 0x3aaa},
  {0x11d0, 0x382c, 0x1dd1, 0x0ba7, 0x279e, 0x2bef, 0x291c, 0x2ff5,
   0x334f, 0x2f7e, 0x0a82, 0x36c1, 0x23d6, 0x1ac7, 0x1d68, 0x38b2}
},
{ {0x3ef3, 0x1828, 0x2e87, 0x35d1, 0x2baf, 0x140e, 0x04b2, 0x3178,
   0x1a6c, 0x127b, 0x3f7d, 0x10c3, 0x2691, 0x3f24, 0x1a96, 0x2938},
  {0x2d9a, 0x080a, 0x0980, 0x0104, 0x1a81, 0x1dd6, 0x3684, 0x28f2,
   0x11f8, 0x2aac, 0x3655, 0x13e6, 0x1037, 0x3321, 0x217e, 0x115e},
{0x03c9, 0x346e, 0x1bdf, 0x01bc, 0x0f11, 0x16c8, 0x3771, 0x1d47,
 0x2e47, 0x3833, 0x1314, 0x1b73, 0x16c4, 0x0972, 0x2e6c, 0x231a},
{0x0d2c, 0x04c7, 0x25df, 0x0bac, 0x27de, 0x144f, 0x1e81, 0x1c04,
 0x3f1a, 0x2842, 0x0374, 0x3a5f, 0x121c, 0x183b, 0x0508, 0x1fcd},
{0x2e3b, 0x07cf, 0x387a, 0x2e73, 0x1176, 0x3717, 0x111e, 0x0f65,
 0x2583, 0x0833, 0x026e, 0x082d, 0x14bd, 0x3efd, 0x3d75, 0x190b},
{0x0cf0, 0x0d87, 0x1293, 0x3867, 0x2c9b, 0x32c9, 0x087c, 0x16b9,
 0x2262, 0x1941, 0x186e, 0x1d97, 0x2954, 0x2266, 0x2ca4, 0x399f},
{0x3f70, 0x3122, 0x2263, 0x1378, 0x06ac, 0x1e2c, 0x2b58, 0x1758,
 0x3c83, 0x2252, 0x07ec, 0x13ad, 0x0994, 0x1647, 0x2f6a, 0x25bf},
{0x3ba1, 0x1943, 0x2736, 0x04db, 0x0f33, 0x110f, 0x1f02, 0x3659,
 0x3927, 0x0ec9, 0x3131, 0x1d1f, 0x32b4, 0x2666, 0x25bc, 0x3d4e},
{0x2008, 0x3353, 0x315c, 0x30af, 0x1ad3, 0x0741, 0x0d04, 0x1ce3,
 0x1fa2, 0x180f, 0x04cc, 0x056d, 0x0cb0, 0x1294, 0x3af9, 0x3add},
{0x34b6, 0x3007, 0x2f11, 0x01db, 0x11f3, 0x2004, 0x1988, 0x366e,
 0x1d6e, 0x217c, 0x1ea3, 0x2669, 0x02d0, 0x2cfa, 0x0b69, 0x1aed},
{0x325b, 0x2adb, 0x1ebb, 0x357e, 0x20ac, 0x1f4a, 0x26e5, 0x1fd8,
 0x2c74, 0x266b, 0x3673, 0x0a7d, 0x1044, 0x2612, 0x11a8, 0x12ce},
{0x0653, 0x098a, 0x1bce, 0x0730, 0x2afa, 0x13f9, 0x24d8, 0x1611,
 0x2d43, 0x102b, 0x0704, 0x340b, 0x2223, 0x1ac3, 0x09f6, 0x194d},
{0x2841, 0x2894, 0x1b64, 0x021a, 0x3803, 0x035c, 0x3a99, 0x326e,
 0x2c6b, 0x1ebc, 0x2a48, 0x3455, 0x2553, 0x1b4e, 0x3ec2, 0x2de6},
{0x2ceb, 0x1cb0, 0x1675, 0x2a85, 0x18f4, 0x0423, 0x21e3, 0x1c57,
 0x1a3d, 0x2af9, 0x2ea8, 0x0557, 0x001d, 0x259b, 0x3ea8, 0x33b1},
{0x2aec, 0x15c7, 0x153b, 0x3500, 0x24d2, 0x28fe, 0x3d36, 0x0253,
 0x0377, 0x2c83, 0x059e, 0x215d, 0x11b9, 0x08db, 0x28ea, 0x3744},
  {0x05bf, 0x12c3, 0x20b6, 0x28be, 0x3c9c, 0x3cc8, 0x0621, 0x04b1,
   0x0ae8, 0x2441, 0x27d4, 0x316b, 0x1ee6, 0x0083, 0x248f, 0x2132}
},
{ {0x1918, 0x2b9f, 0x349a, 0x3fb9, 0x2626, 0x02be, 0x361f, 0x3b1f,
   0x3d00, 0x218b, 0x1170, 0x28f9, 0x3e0b, 0x054b, 0x2b59, 0x353d},
  {0x0330, 0x3c05, 0x207e, 0x38b9, 0x32b2, 0x27d9, 0x0c27, 0x0b8f,
   0x0126, 0x00bb, 0x0b91, 0x3bc4, 0x2a94, 0x2bf9, 0x2893, 0x3703},
  {0x11a9, 0x3e37, 0x0329, 0x2108, 0x1642, 0x1ba3, 0x0828, 0x28e8,
   0x0c0e, 0x3865, 0x2552, 0x200e, 0x020a, 0x3c61, 0x2653, 0x1ae4},
  {0x1a56, 0x029c, 0x25fb, 0x2697, 0x3096, 0x2451, 0x1c37, 0x2276,
   0x1786, 0x13ea, 0x2d7d, 0x2b0e, 0x2fb0, 0x22be, 0x1700, 0x2f31},
  {0x3da6, 0x224c, 0x1548, 0x20b8, 0x339f, 0x138d, 0x3472, 0x0e0d,
   0x2892, 0x0e7e, 0x14e6, 0x0d57, 0x31d7, 0x12a7, 0x3a07, 0x14b1},
  {0x0098, 0x2d86, 0x157d, 0x3086, 0x27e8, 0x31a0, 0x1c16, 0x02cd,
   0x2f88, 0x0811, 0x20ff, 0x055c, 0x3438, 0x345b, 0x23bd, 0x08f6},
  {0x35b6, 0x0f71, 0x1b2f, 0x0d94, 0x2a99, 0x3efc, 0x0bb8, 0x1a27,
   0x2ac8, 0x1cab, 0x1a88, 0x304d, 0x1119, 0x2145, 0x25f2, 0x3124},
  {0x35d8, 0x0c2d, 0x2979, 0x22b3, 0x361c, 0x0476, 0x1c2e, 0x0d70,
   0x0a7f, 0x09ac, 0x3f8c, 0x14f9, 0x1e22, 0x33a7, 0x389c, 0x08d0},
  {0x1d80, 0x2976, 0x3c4a, 0x028a, 0x0559, 0x2d1c, 0x3b12, 0x01d3,
   0x096f, 0x231b, 0x18d8, 0x19c2, 0x0f18, 0x2467, 0x20ca, 0x0f08},
  {0x2c88, 0x3a16, 0x1ff9, 0x372a, 0x10b7, 0x0826, 0x24c6, 0x35ff,
   0x2c48, 0x1517, 0x3a5e, 0x3f44, 0x1bee, 0x0f85, 0x2e99, 0x222c},
  {0x1be0, 0x2524, 0x1f84, 0x3957, 0x0a1a, 0x3665, 0x1032, 0x1c38,
   0x0420, 0x1b5b, 0x275e, 0x3551, 0x3844, 0x1137, 0x112a, 0x3147},
  {0x207a, 0x1d4e, 0x1a28, 0x1736, 0x17b0, 0x0606, 0x2053, 0x0f7e,
     0x2aa8, 0x3fe2, 0x39b6, 0x1481, 0x28d4, 0x14b7, 0x3142, 0x32ba},
    {0x2c04, 0x3250, 0x152f, 0x25a2, 0x2570, 0x2eaa, 0x1e08, 0x1de9,
     0x2166, 0x384b, 0x25be, 0x2b1f, 0x1fe6, 0x28b9, 0x22b7, 0x35bf},
    {0x083f, 0x3a09, 0x2b98, 0x25aa, 0x2265, 0x383c, 0x2ae5, 0x1b32,
     0x091a, 0x1944, 0x23e2, 0x1cc7, 0x20d4, 0x179c, 0x2574, 0x1379},
    {0x04ee, 0x3e5f, 0x21d0, 0x2aeb, 0x2bb4, 0x1e5f, 0x0afd, 0x32f8,
     0x3f3f, 0x2688, 0x241e, 0x13a0, 0x1311, 0x0df7, 0x3aec, 0x29dd},
    {0x351b, 0x34c4, 0x05b3, 0x39c5, 0x35c2, 0x0147, 0x2988, 0x2344,
     0x2649, 0x1058, 0x1431, 0x1141, 0x2860, 0x26cb, 0x2e9f, 0x090d}
  },
  { {0x295f, 0x2722, 0x19b8, 0x391c, 0x2e08, 0x060c, 0x0a3d, 0x385f,
     0x0cb3, 0x06b9, 0x2260, 0x23a1, 0x08c8, 0x1388, 0x2411, 0x1d94},
    {0x0e19, 0x1cbb, 0x070b, 0x207d, 0x2ad1, 0x3747, 0x2546, 0x17d5,
     0x0d30, 0x3449, 0x1e15, 0x3ddb, 0x3c10, 0x0e18, 0x3895, 0x3786},
    {0x2eef, 0x2f62, 0x369d, 0x328c, 0x0632, 0x15f6, 0x2f23, 0x3705,
     0x06b6, 0x0331, 0x0f28, 0x25c1, 0x3a15, 0x118e, 0x18aa, 0x1e9a},
    {0x3777, 0x37d0, 0x1076, 0x0af8, 0x2b8c, 0x1819, 0x35c3, 0x073a,
     0x2428, 0x321b, 0x2dc9, 0x2186, 0x2712, 0x25f4, 0x2964, 0x05cd},
    {0x1510, 0x2d2b, 0x02f8, 0x3893, 0x1ec8, 0x3c5a, 0x0b39, 0x23a4,
     0x0f97, 0x1211, 0x05c7, 0x3d5f, 0x38a7, 0x2c96, 0x3d8a, 0x09d5},
    {0x2ddc, 0x1a0c, 0x0c57, 0x0681, 0x3c79, 0x226c, 0x2e12, 0x164d,
     0x16dd, 0x213d, 0x1be7, 0x0396, 0x02a9, 0x0fcb, 0x1b6b, 0x0b5d},
    {0x15b7, 0x05e2, 0x1e29, 0x356e, 0x3ec0, 0x0821, 0x3e3e, 0x3334,
     0x370f, 0x1b42, 0x1939, 0x2b91, 0x3c82, 0x0777, 0x156a, 0x235d},
    {0x17df, 0x2b8a, 0x3039, 0x2da6, 0x0488, 0x38fb, 0x12f6, 0x2e33,
     0x0e08, 0x2edb, 0x2f95, 0x1945, 0x2d4a, 0x352f, 0x2ea1, 0x2fae},
   {0x1cd3, 0x27b6, 0x14d6, 0x000e, 0x3648, 0x163f, 0x3b61, 0x1de1,
    0x1372, 0x0e0b, 0x0ea7, 0x2311, 0x0906, 0x03b6, 0x11b5, 0x0e10},
   {0x3337, 0x291d, 0x0b96, 0x3945, 0x35fd, 0x2e6e, 0x151f, 0x0d7c,
    0x055d, 0x16b0, 0x3ea2, 0x2135, 0x0b2a, 0x0469, 0x3737, 0x3b06},
   {0x2165, 0x36f2, 0x007e, 0x2f7d, 0x3aaf, 0x2105, 0x196f, 0x1ea1,
    0x011d, 0x0f75, 0x3288, 0x1e51, 0x3644, 0x0939, 0x283f, 0x1915},
   {0x1b23, 0x1c2a, 0x0636, 0x26a3, 0x242c, 0x13de, 0x26db, 0x042f,
    0x0256, 0x201a, 0x398e, 0x1a7c, 0x26a7, 0x3711, 0x2ad4, 0x1b2c},
   {0x1b96, 0x1b33, 0x34b5, 0x1363, 0x3672, 0x12ff, 0x3b97, 0x19f9,
    0x3c00, 0x037f, 0x385a, 0x30fa, 0x19be, 0x290c, 0x3cf0, 0x1dd4},
   {0x1dae, 0x29c5, 0x0856, 0x3b62, 0x0fb4, 0x02ee, 0x077e, 0x302d,
    0x3379, 0x3448, 0x38e7, 0x0cf7, 0x2627, 0x1666, 0x26bc, 0x2ba7},
   {0x2bde, 0x029a, 0x128b, 0x12c8, 0x08c3, 0x352c, 0x2f03, 0x3381,
    0x3b38, 0x28c2, 0x171e, 0x13f8, 0x03d1, 0x2e6b, 0x393b, 0x27b8},
   {0x24c0, 0x2faf, 0x356a, 0x0b62, 0x2313, 0x289f, 0x3c02, 0x2ecf,
    0x387f, 0x30bd, 0x3800, 0x3a59, 0x0aa4, 0x33cc, 0x21ed, 0x2f14}
 },
 { {0x29b1, 0x1791, 0x35bc, 0x3850, 0x1c64, 0x3ee7, 0x1db2, 0x0631,
    0x16f2, 0x3f41, 0x308a, 0x2d5e, 0x343b, 0x1ebe, 0x3f7b, 0x19d6},
   {0x2a71, 0x3ed4, 0x1e9d, 0x1a0a, 0x0267, 0x3101, 0x1443, 0x0cfb,
    0x1e27, 0x26bb, 0x0804, 0x02fa, 0x354e, 0x2982, 0x355d, 0x2352},
   {0x0836, 0x26c1, 0x175e, 0x332a, 0x2d0c, 0x26b1, 0x3193, 0x3982,
    0x0b26, 0x1931, 0x14af, 0x21d1, 0x12f8, 0x1ffa, 0x1db8, 0x3d2f},
   {0x203a, 0x31b6, 0x3544, 0x1159, 0x172f, 0x16da, 0x027e, 0x1e50,
    0x2abd, 0x0573, 0x0f91, 0x0101, 0x31cd, 0x216e, 0x157c, 0x0370},
   {0x3933, 0x19da, 0x1606, 0x057e, 0x16fb, 0x1714, 0x2e10, 0x3240,
   0x1be8, 0x0f5e, 0x0387, 0x197d, 0x1961, 0x3bcf, 0x2249, 0x12a6},
  {0x0d2b, 0x38bb, 0x1109, 0x0b79, 0x12aa, 0x21f4, 0x1223, 0x3b50,
   0x048f, 0x3e9a, 0x2130, 0x32fc, 0x1bb9, 0x1b47, 0x119e, 0x2b9a},
  {0x2be2, 0x1313, 0x3872, 0x0846, 0x3d15, 0x05dc, 0x2e0c, 0x385b,
   0x3270, 0x2054, 0x15cb, 0x2343, 0x2c84, 0x1c46, 0x0369, 0x1607},
  {0x261b, 0x1f3b, 0x0b14, 0x2bab, 0x1343, 0x3009, 0x1836, 0x05f5,
   0x0c71, 0x0deb, 0x3f4b, 0x35a8, 0x0bb4, 0x19a7, 0x2e78, 0x1516},
  {0x00cb, 0x2ffd, 0x290b, 0x3ea6, 0x0684, 0x093f, 0x37fb, 0x1fbe,
   0x06f4, 0x010b, 0x23c5, 0x2be5, 0x1d7d, 0x36f6, 0x358b, 0x0da7},
  {0x1bc3, 0x2052, 0x3faf, 0x346c, 0x0148, 0x1a7b, 0x342e, 0x3dfb,
   0x1d01, 0x32cc, 0x2e2e, 0x1423, 0x29ef, 0x1ddd, 0x25fa, 0x3a7c},
  {0x1436, 0x12a9, 0x214d, 0x1f4c, 0x03df, 0x3be2, 0x1805, 0x0ce5,
   0x0a1d, 0x2e84, 0x1360, 0x36cb, 0x36ff, 0x11b4, 0x38f5, 0x2014},
  {0x1aa7, 0x1838, 0x0769, 0x281e, 0x0d06, 0x3e59, 0x00e8, 0x1a20,
   0x17b1, 0x381f, 0x29ee, 0x3a81, 0x27fe, 0x0e29, 0x0030, 0x0874},
  {0x1e9f, 0x0d29, 0x01cc, 0x1524, 0x134e, 0x04f7, 0x1ac9, 0x2a18,
   0x3f0e, 0x108f, 0x1853, 0x0ff2, 0x2f09, 0x0f1d, 0x2222, 0x1d20},
  {0x1080, 0x3f15, 0x1a94, 0x3697, 0x1230, 0x1c00, 0x01b4, 0x0c2b,
   0x00f1, 0x1eb6, 0x2a09, 0x38e1, 0x1a69, 0x14c7, 0x11f0, 0x0234},
  {0x1bb1, 0x2138, 0x13c0, 0x2e85, 0x0f10, 0x00fb, 0x3c15, 0x2943,
   0x00d3, 0x3c2b, 0x21f7, 0x1811, 0x373a, 0x1234, 0x1ea9, 0x22d0},
  {0x20ce, 0x1905, 0x2c56, 0x010e, 0x0912, 0x181e, 0x1f93, 0x2a65,
   0x3aa7, 0x1dbd, 0x1fcf, 0x0204, 0x2aaf, 0x2eaf, 0x309f, 0x32f9}
},
{ {0x3f4e, 0x329e, 0x17cd, 0x1d33, 0x2931, 0x060d, 0x2787, 0x3aca,
   0x263b, 0x25d5, 0x1a71, 0x23d5, 0x2464, 0x3b6b, 0x2e7d, 0x07db},
{0x134c, 0x2a23, 0x3816, 0x1770, 0x13fc, 0x3dcc, 0x1627, 0x35c9,
 0x2d8f, 0x3875, 0x27ae, 0x2d75, 0x10d7, 0x1acf, 0x2d11, 0x17f3},
{0x1a8c, 0x173f, 0x0515, 0x3bee, 0x19cb, 0x02ac, 0x1f77, 0x1ef6,
 0x08ca, 0x3b66, 0x017d, 0x2662, 0x0881, 0x30b7, 0x3f43, 0x30c1},
{0x3bf0, 0x1184, 0x0cd5, 0x2dbd, 0x03be, 0x0541, 0x2f0c, 0x2739,
 0x3ebe, 0x3c6b, 0x0e1b, 0x1aa3, 0x30f9, 0x0fb1, 0x2652, 0x1ecb},
{0x209b, 0x0bb1, 0x097a, 0x3ecc, 0x3b43, 0x3fe1, 0x074f, 0x19f0,
 0x2958, 0x0677, 0x3598, 0x0eeb, 0x3ea9, 0x1085, 0x10d5, 0x2c75},
{0x20e2, 0x3254, 0x1b59, 0x156d, 0x1334, 0x3ca3, 0x1075, 0x0ec8,
 0x101f, 0x0399, 0x050c, 0x2170, 0x3fe9, 0x3827, 0x2bd0, 0x2a2d},
{0x23cc, 0x0a49, 0x2dbc, 0x3c48, 0x0838, 0x19c9, 0x0bd2, 0x39df,
 0x3c80, 0x1e36, 0x1250, 0x35a2, 0x247b, 0x3aa3, 0x3212, 0x2bd5},
{0x1750, 0x338b, 0x2c94, 0x3bd4, 0x1b60, 0x1bbf, 0x103c, 0x352d,
 0x06e8, 0x14db, 0x1a42, 0x0bd3, 0x12f7, 0x20e7, 0x0aad, 0x0df0},
{0x1792, 0x0c5d, 0x0a8a, 0x15ca, 0x10ac, 0x2701, 0x3f23, 0x1825,
 0x1b1e, 0x3b4c, 0x22bf, 0x1031, 0x0091, 0x176c, 0x39ca, 0x2638},
{0x393c, 0x3c8f, 0x3f06, 0x2b84, 0x2949, 0x2291, 0x364f, 0x308f,
 0x2373, 0x3ff8, 0x0dc7, 0x062a, 0x260b, 0x3d50, 0x31f1, 0x14b4},
{0x1df4, 0x1acd, 0x10c6, 0x30db, 0x09ba, 0x3355, 0x2273, 0x370b,
 0x03db, 0x0425, 0x005e, 0x1965, 0x3397, 0x10d6, 0x1b01, 0x324e},
{0x049f, 0x33ab, 0x0fce, 0x3de9, 0x3bd5, 0x3918, 0x3eaa, 0x00a3,
 0x371a, 0x05a6, 0x2e29, 0x2d81, 0x2444, 0x3829, 0x3a37, 0x2f49},
{0x3340, 0x08f2, 0x3144, 0x1d19, 0x01ce, 0x220b, 0x06f2, 0x0b0f,
 0x3f4a, 0x15c2, 0x3291, 0x0e45, 0x06ca, 0x2897, 0x1951, 0x13ec},
{0x35d0, 0x096e, 0x2e56, 0x0b02, 0x0a3c, 0x3ac3, 0x32bf, 0x030a,
 0x0c44, 0x3edf, 0x202e, 0x1c97, 0x34ef, 0x0426, 0x3634, 0x31b7},
  {0x180c, 0x3f93, 0x3dc2, 0x14e9, 0x3b92, 0x375f, 0x0552, 0x1345,
   0x2ee9, 0x3218, 0x34f6, 0x160c, 0x2376, 0x3881, 0x34ea, 0x2438},
  {0x2cf4, 0x0f3c, 0x365b, 0x1399, 0x1341, 0x3a01, 0x30d1, 0x3bfa,
   0x10ec, 0x28e2, 0x289d, 0x08d9, 0x1140, 0x0ee6, 0x098e, 0x19ec}
},
{ {0x1556, 0x3df3, 0x17d9, 0x3f90, 0x175a, 0x067d, 0x0c2e, 0x11a5,
   0x15bb, 0x2b6d, 0x2a66, 0x0eb5, 0x243f, 0x030b, 0x12c5, 0x086a},
  {0x3960, 0x2746, 0x234f, 0x3f50, 0x15eb, 0x0ce9, 0x0fbb, 0x2de4,
   0x271c, 0x006f, 0x1aae, 0x1787, 0x0433, 0x196c, 0x3490, 0x206d},
  {0x349c, 0x09de, 0x2e74, 0x09da, 0x3176, 0x24fb, 0x0ceb, 0x1005,
   0x3823, 0x0ac8, 0x39de, 0x0a09, 0x1567, 0x1b95, 0x2aff, 0x098c},
  {0x3848, 0x3eb2, 0x11bc, 0x1497, 0x2794, 0x176f, 0x25f0, 0x141e,
   0x1619, 0x0eb4, 0x19ed, 0x1a07, 0x34f4, 0x2b63, 0x2974, 0x1ecd},
  {0x0e4d, 0x3ab3, 0x035f, 0x25d7, 0x0d4a, 0x0066, 0x3238, 0x049a,
   0x1fe3, 0x1c20, 0x140d, 0x2180, 0x1424, 0x161f, 0x075e, 0x37fd},
  {0x172d, 0x19e3, 0x2098, 0x0d39, 0x3363, 0x1cb3, 0x0de4, 0x0729,
   0x326c, 0x33c0, 0x0da0, 0x2561, 0x2dea, 0x274e, 0x22f1, 0x260c},
  {0x09b2, 0x0a88, 0x3dbc, 0x0f8b, 0x0d47, 0x11cf, 0x25ab, 0x283d,
   0x0fc6, 0x2de1, 0x0976, 0x17e9, 0x21c5, 0x2077, 0x1365, 0x3389},
  {0x304a, 0x14d8, 0x2d08, 0x0d3e, 0x368b, 0x1f76, 0x29d4, 0x1455,
   0x2deb, 0x03f1, 0x03d0, 0x08ac, 0x1af0, 0x2286, 0x035e, 0x2ed0},
  {0x0656, 0x177d, 0x1275, 0x31fb, 0x07d4, 0x0fa3, 0x1446, 0x0dfe,
   0x20b0, 0x1a3c, 0x35b3, 0x02f1, 0x3967, 0x017c, 0x2087, 0x05f2},
  {0x0512, 0x3125, 0x108a, 0x064f, 0x007c, 0x350e, 0x3150, 0x3b80,
   0x0c86, 0x1f55, 0x29f4, 0x13f4, 0x2fc3, 0x3298, 0x3a73, 0x0940},
  {0x0ad0, 0x3418, 0x0c34, 0x1c01, 0x2675, 0x0675, 0x324f, 0x27e4,
   0x170c, 0x0e28, 0x335e, 0x2623, 0x1edb, 0x2705, 0x0f53, 0x2142},
  {0x3ca9, 0x3696, 0x22f6, 0x28cf, 0x27f8, 0x04e1, 0x20cf, 0x2922,
   0x36dd, 0x1518, 0x3940, 0x0aea, 0x3b91, 0x0ca2, 0x0a5a, 0x09f1},
  {0x1218, 0x3e4f, 0x0226, 0x37f2, 0x1d13, 0x1381, 0x1abc, 0x0bd8,
   0x0763, 0x32e3, 0x2f2d, 0x3fb1, 0x2b3a, 0x3185, 0x298a, 0x377d},
  {0x0570, 0x3755, 0x0cee, 0x143b, 0x1b8d, 0x32f6, 0x0102, 0x0bf8,
   0x1e2e, 0x2530, 0x0609, 0x1ee1, 0x02f2, 0x3117, 0x36ba, 0x1928},
  {0x0c97, 0x0baa, 0x0f06, 0x3f92, 0x1c6c, 0x16d6, 0x1441, 0x0d4d,
   0x1301, 0x3f18, 0x0c23, 0x0974, 0x3677, 0x19f7, 0x0868, 0x205d},
  {0x1da8, 0x061d, 0x1e45, 0x024a, 0x2353, 0x2811, 0x3a5a, 0x250c,
   0x1d9a, 0x17e1, 0x0724, 0x297e, 0x2b34, 0x0378, 0x32cd, 0x2b42}
},
{ {0x3c25, 0x30f3, 0x0b87, 0x3890, 0x2dcd, 0x2738, 0x062c, 0x378b,
   0x1298, 0x1d9b, 0x3699, 0x0903, 0x33d5, 0x3ca4, 0x1c28, 0x1e04},
  {0x3b31, 0x1023, 0x2e15, 0x2113, 0x24cd, 0x3a8e, 0x37b1, 0x3f5f,
   0x1997, 0x2c39, 0x18d4, 0x1742, 0x1a33, 0x3a85, 0x3c21, 0x0fdc},
  {0x3110, 0x0adc, 0x0d17, 0x371f, 0x12db, 0x1227, 0x3440, 0x310a,
   0x1c0b, 0x3c1a, 0x1dde, 0x0ecf, 0x0fb3, 0x0ebe, 0x02a7, 0x38d0},
  {0x1ab9, 0x3815, 0x2b2b, 0x2add, 0x385c, 0x11b6, 0x1442, 0x3e84,
   0x03d6, 0x18dd, 0x320c, 0x075c, 0x23a9, 0x35e7, 0x2647, 0x0235},
  {0x2d53, 0x2398, 0x0b1b, 0x01b8, 0x29fc, 0x09b5, 0x249d, 0x3c1e,
   0x0e4a, 0x14ff, 0x0641, 0x2b68, 0x0bef, 0x0440, 0x3cf8, 0x399c},
  {0x19a2, 0x2e28, 0x0e8e, 0x2b2a, 0x3656, 0x2bc9, 0x2c10, 0x2ecb,
   0x3d72, 0x088a, 0x1f3e, 0x372e, 0x139b, 0x19fe, 0x023a, 0x1622},
  {0x2feb, 0x2c13, 0x2b96, 0x07f9, 0x2102, 0x0ee0, 0x0c70, 0x2047,
   0x1e56, 0x35b1, 0x23e3, 0x0409, 0x1110, 0x3a75, 0x0611, 0x34e5},
  {0x11f2, 0x3615, 0x1936, 0x0abd, 0x016c, 0x307a, 0x1102, 0x3c52,
   0x3d12, 0x3adb, 0x08ce, 0x16a1, 0x24b4, 0x2ab1, 0x038b, 0x0413},
  {0x1fbc, 0x151b, 0x3812, 0x1e71, 0x3852, 0x15e6, 0x3492, 0x123a,
   0x170a, 0x3939, 0x2af6, 0x306b, 0x03e4, 0x1ff1, 0x0a84, 0x29ad},
  {0x1588, 0x3f8d, 0x26b3, 0x018d, 0x337c, 0x116d, 0x062b, 0x30a9,
   0x1eb2, 0x106d, 0x072e, 0x28ac, 0x2314, 0x1afa, 0x3904, 0x1323},
  {0x2d1f, 0x3360, 0x2af4, 0x26b2, 0x15c8, 0x008e, 0x38c7, 0x021d,
   0x1807, 0x242e, 0x265e, 0x1757, 0x3589, 0x1142, 0x0925, 0x35d7},
  {0x2ea5, 0x2ca3, 0x08c9, 0x3a0b, 0x0248, 0x0164, 0x3e65, 0x09e4,
   0x262c, 0x1767, 0x3ed5, 0x2cc6, 0x3e77, 0x025d, 0x233b, 0x03d3},
  {0x22ff, 0x0582, 0x3e9b, 0x0319, 0x3862, 0x1d65, 0x0e35, 0x2cd7,
   0x1646, 0x3731, 0x36c7, 0x3356, 0x0e1e, 0x3473, 0x3d29, 0x32af},
  {0x3870, 0x3c8d, 0x3341, 0x06e6, 0x1e10, 0x3df8, 0x1772, 0x0237,
   0x0020, 0x1a5b, 0x3ceb, 0x16f4, 0x3184, 0x2dc7, 0x3d7e, 0x09ad},
  {0x17a0, 0x2593, 0x0575, 0x39e0, 0x2215, 0x1d5f, 0x0caf, 0x3911,
   0x1c21, 0x027a, 0x0db1, 0x3ffb, 0x0797, 0x2a30, 0x27a4, 0x230a},
  {0x0a0f, 0x2c58, 0x1fc1, 0x2477, 0x11c0, 0x2fd3, 0x2d0e, 0x3cdb,
   0x021b, 0x3547, 0x21f0, 0x3bdb, 0x24ef, 0x092b, 0x313c, 0x38a9}
},
{ {0x3831, 0x38a8, 0x09cd, 0x3674, 0x0494, 0x3002, 0x2386, 0x2e62,
   0x0d12, 0x1545, 0x2ac0, 0x2c60, 0x183c, 0x28cb, 0x1990, 0x0386},
  {0x1c0d, 0x1f9c, 0x15c9, 0x32e8, 0x360d, 0x0691, 0x28a4, 0x0af5,
   0x24e8, 0x26d5, 0x00ae, 0x37e7, 0x3491, 0x12a3, 0x3719, 0x08b7},
  {0x1904, 0x3c8a, 0x38e9, 0x2f74, 0x242b, 0x1e91, 0x2812, 0x3ca2,
   0x330b, 0x25b1, 0x3055, 0x0584, 0x135a, 0x0aca, 0x34d0, 0x1e97},
  {0x20d0, 0x1057, 0x2efa, 0x0771, 0x2b5c, 0x3b02, 0x0587, 0x0150,
   0x1c69, 0x0edf, 0x0c2c, 0x2e53, 0x1af4, 0x2886, 0x0637, 0x1f6c},
  {0x1369, 0x185b, 0x13d0, 0x14d4, 0x2229, 0x2dff, 0x3629, 0x0de0,
   0x1f14, 0x3caa, 0x3585, 0x0437, 0x26ad, 0x2d47, 0x19e0, 0x25ba},
  {0x2cab, 0x24c4, 0x2dad, 0x3612, 0x3e76, 0x3c85, 0x31a9, 0x3921,
   0x01ea, 0x1600, 0x0d2a, 0x0d9a, 0x3fed, 0x270f, 0x2f3c, 0x032d},
  {0x1120, 0x2c7e, 0x3ed2, 0x201d, 0x0c98, 0x053f, 0x1072, 0x33bf,
   0x31e4, 0x3c78, 0x3d25, 0x20f9, 0x2ca7, 0x0017, 0x05f3, 0x053b},
  {0x3493, 0x3e43, 0x0a85, 0x2ef4, 0x0d49, 0x0b3d, 0x1a01, 0x1fdf,
   0x1fe8, 0x0f1e, 0x233a, 0x05eb, 0x2713, 0x04fe, 0x3a90, 0x20bf},
  {0x32a6, 0x3c39, 0x2cc3, 0x0d84, 0x090b, 0x17d8, 0x102c, 0x104c,
   0x2e2a, 0x27b9, 0x1a4a, 0x39ef, 0x34fd, 0x3425, 0x0bd1, 0x184f},
  {0x2c67, 0x0410, 0x0a26, 0x2e77, 0x30a2, 0x3aed, 0x1f9d, 0x3ec4,
   0x2b15, 0x26c3, 0x3a2e, 0x3f60, 0x0ce6, 0x2543, 0x2907, 0x1c72},
  {0x1efd, 0x371c, 0x3783, 0x09f3, 0x27ad, 0x2210, 0x2192, 0x0b3a,
   0x0538, 0x278d, 0x0023, 0x2751, 0x3242, 0x1045, 0x1106, 0x160e},
  {0x00c9, 0x1001, 0x0f8a, 0x0aae, 0x015b, 0x39d7, 0x188d, 0x2bb3,
   0x192a, 0x03c8, 0x388e, 0x0baf, 0x1a77, 0x2299, 0x1559, 0x3064},
  {0x388d, 0x3408, 0x3efe, 0x1129, 0x17f6, 0x26aa, 0x1ff2, 0x3d6f,
   0x211b, 0x0304, 0x11a1, 0x3394, 0x2e76, 0x1bcc, 0x39c4, 0x277d},
  {0x194f, 0x2ad5, 0x324a, 0x0967, 0x012f, 0x254d, 0x1b83, 0x0d15,
   0x3878, 0x28fd, 0x33a6, 0x084a, 0x0049, 0x2c0a, 0x3bd3, 0x1a25},
  {0x16de, 0x1e3d, 0x139f, 0x2b4f, 0x3189, 0x1ea2, 0x28ca, 0x2285,
   0x3b41, 0x19ac, 0x161e, 0x2506, 0x312b, 0x1c06, 0x2a07, 0x283a},
  {0x0a87, 0x37c3, 0x1878, 0x13a9, 0x0d2f, 0x0b01, 0x0c93, 0x2b6a,
   0x266e, 0x10f0, 0x3a43, 0x333e, 0x0a12, 0x37d1, 0x0f67, 0x226b}
},
{ {0x306a, 0x3e6c, 0x0e92, 0x0dca, 0x0120, 0x2b83, 0x2c02, 0x0487,
   0x05b8, 0x13bb, 0x288a, 0x3d08, 0x0170, 0x19d5, 0x0921, 0x223b},
  {0x2830, 0x1047, 0x264c, 0x1a54, 0x2502, 0x2def, 0x3584, 0x3136,
   0x0585, 0x0dd2, 0x33ae, 0x343f, 0x21ef, 0x017e, 0x2b7a, 0x3a70},
  {0x2f07, 0x2374, 0x19c0, 0x11ad, 0x2f27, 0x1d7a, 0x2d29, 0x1c4e,
   0x36f8, 0x38fd, 0x1d71, 0x37a2, 0x2ec4, 0x37e6, 0x172b, 0x2f02},
  {0x21aa, 0x3483, 0x23ef, 0x204a, 0x239a, 0x12bb, 0x2ae8, 0x39f7,
   0x214c, 0x2796, 0x0038, 0x3d61, 0x0df4, 0x1ad6, 0x1382, 0x37c2},
  {0x05c0, 0x0215, 0x21a6, 0x2af5, 0x2b8f, 0x3011, 0x1972, 0x2765,
   0x3f74, 0x034b, 0x35f6, 0x1f31, 0x1775, 0x0ada, 0x21bc, 0x16fd},
  {0x24f0, 0x2d3a, 0x28d6, 0x3ef0, 0x105a, 0x160f, 0x1055, 0x37bf,
   0x34f7, 0x3d5e, 0x2360, 0x1890, 0x09f0, 0x1a47, 0x2536, 0x1e4d},
  {0x19ea, 0x0daf, 0x18dc, 0x2df9, 0x3bec, 0x0ff8, 0x3806, 0x12e1,
   0x21eb, 0x01ab, 0x3fbe, 0x2d32, 0x17e8, 0x1948, 0x2ece, 0x2cdb},
  {0x2f1e, 0x3910, 0x34d5, 0x2eb3, 0x039b, 0x1513, 0x0886, 0x04f5,
   0x3c7f, 0x0619, 0x2a90, 0x1316, 0x2e23, 0x0e51, 0x136f, 0x1219},
  {0x2edf, 0x1073, 0x068e, 0x2350, 0x06a5, 0x0b78, 0x3973, 0x227e,
   0x2ab4, 0x33eb, 0x2f5d, 0x2cf8, 0x24eb, 0x162a, 0x2a10, 0x027f},
  {0x3a53, 0x2fa3, 0x3ea3, 0x05df, 0x197c, 0x1586, 0x1e8f, 0x30e5,
   0x23ce, 0x27dc, 0x1412, 0x3441, 0x0914, 0x1919, 0x24fa, 0x3d11},
  {0x015c, 0x18ec, 0x1b36, 0x01f7, 0x3fc7, 0x1bfe, 0x0258, 0x2576,
   0x2532, 0x1ad9, 0x2bc0, 0x0ca5, 0x218e, 0x2c3d, 0x310e, 0x1447},
  {0x3e56, 0x1cc6, 0x1d2f, 0x21f6, 0x352b, 0x109c, 0x0645, 0x01b5,
   0x0825, 0x057a, 0x2239, 0x189e, 0x2923, 0x15c4, 0x18f3, 0x0442},
  {0x2711, 0x30dc, 0x2b27, 0x263c, 0x0517, 0x241a, 0x2133, 0x2b16,
   0x2ed8, 0x02d4, 0x1ab8, 0x0031, 0x1c75, 0x130b, 0x0167, 0x367b},
    {0x071b, 0x3372, 0x29db, 0x3928, 0x059a, 0x0b7d, 0x2e42, 0x3f9f,
     0x20e8, 0x1b41, 0x23f3, 0x0427, 0x08bf, 0x0c32, 0x1a58, 0x3cde},
    {0x29bc, 0x1902, 0x1029, 0x2779, 0x2547, 0x2a03, 0x38cf, 0x1895,
     0x0743, 0x21c9, 0x091e, 0x2e1f, 0x37e9, 0x32f7, 0x024e, 0x34f0},
    {0x1f1b, 0x2a8a, 0x1281, 0x2b12, 0x066e, 0x2b40, 0x3e06, 0x322c,
     0x2efe, 0x1df1, 0x284e, 0x2833, 0x05c6, 0x1898, 0x0a28, 0x0e09}
  },
  { {0x1eca, 0x395f, 0x0a55, 0x1aa4, 0x291a, 0x0510, 0x29f8, 0x3ab8,
     0x395a, 0x2b93, 0x0110, 0x1b7c, 0x2b9b, 0x3157, 0x1eb3, 0x369b},
    {0x05bc, 0x21a7, 0x183f, 0x16ff, 0x3c6c, 0x2dc4, 0x0f88, 0x01ee,
     0x0e87, 0x06df, 0x25e3, 0x12a5, 0x3b54, 0x0e71, 0x0384, 0x2c6d},
    {0x24f1, 0x12de, 0x03b7, 0x2e1d, 0x28d9, 0x0dad, 0x0e12, 0x21a1,
     0x336b, 0x2cfe, 0x30fb, 0x3204, 0x0a96, 0x27d0, 0x3b0f, 0x03cc},
    {0x02c7, 0x0bf5, 0x0445, 0x3215, 0x128a, 0x23c8, 0x1430, 0x2ffe,
     0x0928, 0x3d27, 0x2bec, 0x0484, 0x0e84, 0x3453, 0x135c, 0x1367},
    {0x38c0, 0x0e17, 0x3811, 0x3034, 0x2b1a, 0x366a, 0x3f59, 0x2500,
     0x09d6, 0x364b, 0x391e, 0x1dc7, 0x1cbd, 0x2e17, 0x0d8a, 0x3789},
    {0x3b67, 0x362b, 0x33d6, 0x3ee3, 0x04b0, 0x0ff1, 0x28f4, 0x28df,
     0x2b54, 0x0c82, 0x0274, 0x00a8, 0x2957, 0x216c, 0x214b, 0x2d62},
    {0x3720, 0x341e, 0x183e, 0x1dfa, 0x119a, 0x2afb, 0x0e58, 0x3be8,
     0x394a, 0x1a45, 0x2fca, 0x150f, 0x2445, 0x1832, 0x17e2, 0x07a0},
    {0x089e, 0x3e7d, 0x3080, 0x0029, 0x2a41, 0x159c, 0x2a02, 0x0285,
     0x0d71, 0x1bf1, 0x2ac2, 0x3ef7, 0x1cf6, 0x05ea, 0x0a6d, 0x2a17},
    {0x3e24, 0x0887, 0x3723, 0x050d, 0x303d, 0x3b3f, 0x18c1, 0x1b99,
     0x23a3, 0x1371, 0x13cf, 0x1cb2, 0x1ce4, 0x221e, 0x3187, 0x2748},
    {0x2f5e, 0x33a0, 0x321d, 0x2c9d, 0x05da, 0x3020, 0x0280, 0x2027,
   0x3721, 0x279d, 0x2189, 0x1e3f, 0x3c14, 0x3292, 0x12f2, 0x1a92},
  {0x0c9e, 0x3390, 0x063a, 0x3ef5, 0x2055, 0x1a30, 0x06d2, 0x1585,
   0x02a4, 0x26a0, 0x0402, 0x0282, 0x1a87, 0x2f3b, 0x040d, 0x0fba},
  {0x06ee, 0x3dac, 0x3545, 0x2b7c, 0x25a5, 0x21b3, 0x29d8, 0x0da3,
   0x15e4, 0x0ff0, 0x33cd, 0x3d1e, 0x3a74, 0x13fb, 0x2150, 0x1d8f},
  {0x3fd6, 0x16e9, 0x07fe, 0x1cea, 0x21a8, 0x3a4e, 0x3f56, 0x250f,
   0x2369, 0x17fe, 0x18af, 0x34be, 0x1e3c, 0x1c5f, 0x097d, 0x06fb},
  {0x2b28, 0x325c, 0x1bf6, 0x1a5d, 0x2cca, 0x2a33, 0x3cf3, 0x2e27,
   0x348e, 0x2b00, 0x2788, 0x2a4b, 0x1916, 0x21c8, 0x2a06, 0x0917},
  {0x30a0, 0x1ca3, 0x2f50, 0x16ed, 0x049b, 0x36e8, 0x08b5, 0x1202,
   0x2038, 0x0c21, 0x1ba8, 0x2d27, 0x3328, 0x0c01, 0x33ea, 0x01dd},
  {0x084c, 0x2d9b, 0x10b5, 0x39b9, 0x2590, 0x3f38, 0x2c8c, 0x06e3,
   0x0d37, 0x0d5b, 0x32d2, 0x1760, 0x3f2b, 0x2789, 0x2935, 0x167c}
},
{ {0x07e6, 0x1552, 0x0ae6, 0x3e8d, 0x0844, 0x1abe, 0x02b2, 0x3c4f,
   0x0834, 0x1598, 0x3a7e, 0x2cd9, 0x1e8e, 0x2457, 0x313b, 0x003d},
  {0x0e30, 0x123e, 0x1e21, 0x102f, 0x12be, 0x0690, 0x2220, 0x094f,
   0x31f5, 0x1520, 0x0afc, 0x23dd, 0x3594, 0x2eb8, 0x117a, 0x302a},
  {0x0a48, 0x20ef, 0x22f4, 0x02d1, 0x252c, 0x0798, 0x3530, 0x38b8,
   0x085f, 0x1538, 0x2c1d, 0x287d, 0x3b34, 0x2084, 0x3a36, 0x0e3e},
  {0x0c90, 0x3b88, 0x0312, 0x3669, 0x1bcf, 0x3565, 0x0d92, 0x2e35,
   0x1840, 0x08b4, 0x09c3, 0x3e57, 0x22fe, 0x0e8a, 0x3468, 0x3db8},
  {0x003c, 0x2dd3, 0x30d3, 0x0251, 0x2a32, 0x1960, 0x354b, 0x0ef6,
   0x0004, 0x294b, 0x2046, 0x0c6d, 0x1991, 0x2117, 0x3f6a, 0x3b21},
  {0x0005, 0x28b8, 0x3752, 0x223f, 0x3366, 0x2490, 0x0667, 0x15ff,
   0x0810, 0x219c, 0x02c9, 0x120b, 0x246e, 0x0888, 0x0c6f, 0x1c2d},
  {0x3d34, 0x2a54, 0x042c, 0x0205, 0x271b, 0x13ca, 0x070a, 0x2d61,
   0x2413, 0x13cb, 0x0548, 0x0aac, 0x22c9, 0x3ecb, 0x0419, 0x06b2},
  {0x2cc9, 0x148c, 0x1789, 0x376f, 0x2a7f, 0x03e5, 0x223c, 0x2e43,
   0x38e8, 0x3c6e, 0x0b50, 0x2510, 0x0d14, 0x1b46, 0x2a3c, 0x061a},
  {0x3b46, 0x3def, 0x1613, 0x1259, 0x0735, 0x1515, 0x1303, 0x159a,
   0x2195, 0x0ef5, 0x33fe, 0x1074, 0x336f, 0x23c3, 0x17c5, 0x2070},
  {0x1624, 0x366d, 0x2a27, 0x3bb3, 0x2f36, 0x1dc4, 0x327d, 0x18ed,
   0x1950, 0x0764, 0x1e9e, 0x3ccb, 0x22ad, 0x3d07, 0x3108, 0x0f94},
  {0x2a44, 0x0ffb, 0x02c6, 0x1e0e, 0x0796, 0x3a18, 0x0ed9, 0x064d,
   0x031d, 0x03d9, 0x243b, 0x38d4, 0x3943, 0x353f, 0x2a2a, 0x277e},
  {0x042e, 0x1891, 0x1b7a, 0x2b38, 0x3549, 0x3a87, 0x3798, 0x06d9,
   0x3dc1, 0x1cbf, 0x2f38, 0x1f81, 0x0323, 0x09ab, 0x36cc, 0x2866},
  {0x3de0, 0x0c85, 0x0d28, 0x1d45, 0x0d6d, 0x35de, 0x1f04, 0x3043,
   0x2723, 0x3e09, 0x0187, 0x294a, 0x0ca4, 0x200c, 0x39b2, 0x2936},
  {0x3dea, 0x1263, 0x0287, 0x19a3, 0x38c3, 0x16b7, 0x2e21, 0x23ea,
   0x24ba, 0x1d83, 0x3c0d, 0x1cc5, 0x0562, 0x39c7, 0x10ff, 0x352e},
  {0x29e8, 0x26a5, 0x3bc1, 0x3bdc, 0x1265, 0x1deb, 0x338d, 0x16a7,
   0x2245, 0x06f6, 0x0815, 0x2b88, 0x13d3, 0x37a9, 0x25ec, 0x3325},
  {0x143d, 0x138e, 0x0d0f, 0x1ae1, 0x3a9c, 0x16f8, 0x1682, 0x13ab,
   0x1fa4, 0x2c25, 0x3c7e, 0x0f57, 0x2a34, 0x0e47, 0x2539, 0x23e5}
},
{ {0x1568, 0x29aa, 0x28e0, 0x2c1f, 0x2237, 0x26c4, 0x35b2, 0x23ae,
   0x3691, 0x1fe9, 0x3ec1, 0x397c, 0x0964, 0x0c51, 0x07c0, 0x1daf},
  {0x03a8, 0x0a2f, 0x3279, 0x1817, 0x37be, 0x1942, 0x14e2, 0x0f74,
   0x35c1, 0x1bd5, 0x0cae, 0x27f7, 0x0f52, 0x34fe, 0x27c9, 0x3b99},
  {0x1e4e, 0x2cd6, 0x2537, 0x38a1, 0x02fe, 0x1839, 0x3412, 0x2e71,
 0x3a30, 0x14f3, 0x1da7, 0x1427, 0x07c4, 0x26e3, 0x3d35, 0x1522},
{0x09d3, 0x0c48, 0x0958, 0x303f, 0x0bbb, 0x0d96, 0x05d5, 0x3be6,
 0x03f6, 0x3d1b, 0x3f02, 0x2065, 0x39e5, 0x2846, 0x2eb1, 0x1728},
{0x1ec0, 0x0ee2, 0x3105, 0x18f8, 0x055b, 0x13e5, 0x0e6a, 0x252a,
 0x21c4, 0x20fa, 0x128c, 0x2c0b, 0x2079, 0x2ccf, 0x11ea, 0x05ae},
{0x0e0c, 0x2a62, 0x097f, 0x208b, 0x2679, 0x2548, 0x1fef, 0x1af3,
 0x33ac, 0x0d7a, 0x1833, 0x1761, 0x179b, 0x0c0b, 0x01e0, 0x0f2d},
{0x2706, 0x1975, 0x0842, 0x0dde, 0x3c34, 0x1d06, 0x1c34, 0x240f,
 0x312f, 0x1b1c, 0x301f, 0x1490, 0x00dc, 0x3e30, 0x3d53, 0x3e32},
{0x005d, 0x350c, 0x1648, 0x1437, 0x2692, 0x299d, 0x0e40, 0x0ed7,
 0x0151, 0x36b0, 0x08f5, 0x22ed, 0x0132, 0x07a5, 0x3c7b, 0x0d8d},
{0x13f0, 0x1115, 0x0558, 0x235c, 0x2cdd, 0x213f, 0x191a, 0x21ca,
 0x17fd, 0x3768, 0x0f1b, 0x0e8d, 0x229e, 0x330d, 0x37de, 0x14e4},
{0x2e6a, 0x08a3, 0x17f0, 0x29c7, 0x2168, 0x3d81, 0x0e78, 0x1a36,
 0x17b3, 0x383b, 0x3fc0, 0x1672, 0x1c7c, 0x1c55, 0x23b7, 0x1467},
{0x0869, 0x1347, 0x1f0a, 0x147b, 0x1392, 0x0be4, 0x034d, 0x19e6,
 0x1499, 0x2458, 0x1a39, 0x3351, 0x356b, 0x2f9c, 0x193c, 0x0907},
{0x3497, 0x2f44, 0x14f6, 0x3374, 0x07e9, 0x0b4f, 0x0fda, 0x3001,
 0x3706, 0x3880, 0x0f1f, 0x133b, 0x2074, 0x0e00, 0x2cec, 0x07b8},
{0x0404, 0x2efc, 0x3912, 0x39fd, 0x12ca, 0x2fd7, 0x052c, 0x26a4,
 0x07d8, 0x19db, 0x3eb7, 0x2e24, 0x31fd, 0x3ff7, 0x178a, 0x24a6},
{0x2b53, 0x0de2, 0x03ff, 0x1e4b, 0x3358, 0x3cb1, 0x2952, 0x3062,
 0x2708, 0x37e1, 0x07a7, 0x288f, 0x3512, 0x0773, 0x177b, 0x1472},
{0x2d65, 0x2ea9, 0x26e9, 0x244b, 0x1bcd, 0x2921, 0x2e91, 0x386e,
 0x0638, 0x1101, 0x009e, 0x33bd, 0x1704, 0x1a93, 0x1a84, 0x2809},
{0x2a43, 0x2cef, 0x3526, 0x075f, 0x0363, 0x1406, 0x0982, 0x36ac,
   0x297d, 0x35a1, 0x0376, 0x312e, 0x3661, 0x2cf6, 0x09f4, 0x08b6}
},
{ {0x3f67, 0x21bd, 0x2887, 0x0bcc, 0x02c8, 0x07bc, 0x093b, 0x067f,
   0x3686, 0x1e62, 0x3503, 0x164c, 0x1f41, 0x136b, 0x188f, 0x19ae},
  {0x3b40, 0x27ef, 0x067e, 0x0fab, 0x1307, 0x0a42, 0x3c4b, 0x0046,
   0x095d, 0x2155, 0x3fae, 0x099a, 0x23c7, 0x33ba, 0x1caf, 0x32be},
  {0x0d62, 0x1c18, 0x35e9, 0x29cf, 0x323f, 0x3e1c, 0x23cd, 0x28b2,
   0x1eb1, 0x3f55, 0x3778, 0x2cd1, 0x3162, 0x3ce1, 0x21e0, 0x0bf2},
  {0x3613, 0x239d, 0x3f95, 0x027d, 0x3be5, 0x37f0, 0x0219, 0x15e0,
   0x3eb9, 0x0663, 0x1398, 0x236b, 0x00ef, 0x2ade, 0x36c0, 0x3303},
  {0x246a, 0x19b2, 0x1c1b, 0x214a, 0x20c7, 0x177f, 0x369c, 0x212a,
   0x1671, 0x3186, 0x04d8, 0x188e, 0x003e, 0x32ee, 0x3192, 0x21af},
  {0x2d36, 0x3704, 0x36d2, 0x38ba, 0x0694, 0x184e, 0x285a, 0x27e5,
   0x1f56, 0x273c, 0x2ca8, 0x0a22, 0x3129, 0x32b7, 0x2861, 0x13dc},
  {0x26f9, 0x2858, 0x2981, 0x127f, 0x0808, 0x181a, 0x07f6, 0x3f7a,
   0x3fb8, 0x1ac2, 0x2179, 0x3788, 0x07e5, 0x2416, 0x07ae, 0x129c},
  {0x30f8, 0x3059, 0x0286, 0x12ab, 0x052e, 0x2514, 0x1268, 0x1d5e,
   0x3323, 0x3ba7, 0x1751, 0x165d, 0x1282, 0x1929, 0x3bf5, 0x2d85},
  {0x0e80, 0x029f, 0x0475, 0x13d5, 0x1f19, 0x0345, 0x3959, 0x13d7,
   0x0d3c, 0x0e73, 0x36d7, 0x2589, 0x07fd, 0x1ef1, 0x17ad, 0x28dc},
  {0x12e0, 0x0c83, 0x2319, 0x1a55, 0x25cf, 0x3533, 0x3dc4, 0x33f7,
   0x03c0, 0x0b46, 0x13c7, 0x3463, 0x1514, 0x2668, 0x1a98, 0x014e},
  {0x0a02, 0x0a8c, 0x03ca, 0x0381, 0x3fd1, 0x3b6a, 0x35b9, 0x0003,
   0x3f5a, 0x0f37, 0x2e69, 0x2f2a, 0x0f96, 0x3a0d, 0x1693, 0x36b9},
  {0x1a34, 0x15a2, 0x2faa, 0x37a8, 0x0115, 0x2dcf, 0x3cfc, 0x2e00,
   0x03a3, 0x0016, 0x378e, 0x3b17, 0x1374, 0x1384, 0x0ac3, 0x16bf},
    {0x0b40, 0x1845, 0x18b2, 0x16af, 0x2264, 0x0288, 0x075d, 0x2ce5,
     0x1459, 0x23e9, 0x1aaf, 0x04f3, 0x195d, 0x10d3, 0x2613, 0x2275},
    {0x2a70, 0x1577, 0x1f85, 0x1173, 0x1b5a, 0x098b, 0x3e22, 0x0c15,
     0x1180, 0x0975, 0x0eb9, 0x20d6, 0x08cd, 0x3842, 0x0a7c, 0x298f},
    {0x1fbb, 0x215b, 0x2b6c, 0x06a1, 0x31d5, 0x1da4, 0x2a58, 0x2f04,
     0x076e, 0x15b4, 0x1dc2, 0x20a7, 0x2e92, 0x1bb0, 0x1309, 0x1fad},
    {0x28d0, 0x2f69, 0x3362, 0x2fbd, 0x398d, 0x2aae, 0x1e37, 0x01a4,
     0x3e25, 0x1b06, 0x37ae, 0x22c2, 0x05bd, 0x38ac, 0x2418, 0x0dec}
  },
  { {0x0800, 0x3257, 0x197a, 0x04cb, 0x2518, 0x3d7f, 0x32c2, 0x07f4,
     0x3f16, 0x3e40, 0x2326, 0x22af, 0x136c, 0x3e9f, 0x0ae2, 0x09a3},
    {0x01e1, 0x3523, 0x2208, 0x115b, 0x302e, 0x22c8, 0x0877, 0x19c7,
     0x146a, 0x0542, 0x0501, 0x101c, 0x0c55, 0x0e21, 0x2a7a, 0x0b37},
    {0x0660, 0x0050, 0x3072, 0x00b5, 0x2bf5, 0x0a13, 0x35db, 0x2805,
     0x3f1b, 0x0b94, 0x1e43, 0x3f32, 0x3e21, 0x2c28, 0x0458, 0x26b6},
    {0x079c, 0x0e89, 0x207b, 0x02bb, 0x2a3d, 0x22a5, 0x07c8, 0x12af,
     0x0221, 0x3a47, 0x0a6f, 0x3996, 0x14ef, 0x0317, 0x068b, 0x01c6},
    {0x0f0d, 0x02b6, 0x277a, 0x2dde, 0x0be5, 0x0805, 0x25b9, 0x2665,
     0x0109, 0x024b, 0x20e1, 0x3bf4, 0x0594, 0x26c2, 0x2569, 0x2b2e},
    {0x0bf4, 0x19cf, 0x028d, 0x37c9, 0x347d, 0x129b, 0x3fce, 0x0414,
     0x2a6d, 0x09e7, 0x035d, 0x0de8, 0x0a36, 0x2b7b, 0x2d51, 0x0008},
    {0x0f41, 0x3411, 0x2ded, 0x2391, 0x1b20, 0x0ade, 0x0c66, 0x2df8,
     0x007b, 0x3576, 0x2a1e, 0x1c91, 0x10d1, 0x1cbe, 0x03dd, 0x3a83},
    {0x2ab8, 0x216a, 0x1e06, 0x01fe, 0x36fd, 0x3874, 0x36dc, 0x35ae,
     0x0272, 0x341a, 0x15fd, 0x0332, 0x0c74, 0x0bfc, 0x259a, 0x1466},
    {0x0c02, 0x2112, 0x23a5, 0x18b0, 0x394c, 0x3fad, 0x1532, 0x3668,
   0x01e6, 0x10c2, 0x1dea, 0x3f65, 0x023c, 0x1897, 0x04ce, 0x030d},
  {0x1f4f, 0x1afb, 0x18ea, 0x2f2c, 0x18ca, 0x0355, 0x1a8b, 0x33cb,
   0x3aab, 0x1f61, 0x2dfe, 0x2913, 0x0f8e, 0x1ca0, 0x16cc, 0x06f0},
  {0x2085, 0x220c, 0x22c4, 0x0fe4, 0x1b31, 0x059f, 0x28d3, 0x2185,
   0x3864, 0x3d6b, 0x0b5e, 0x0a2b, 0x3342, 0x22ea, 0x2f41, 0x193a},
  {0x1695, 0x320a, 0x0790, 0x0e9d, 0x16dc, 0x392e, 0x3329, 0x3015,
   0x0421, 0x0fd1, 0x14fe, 0x1992, 0x297f, 0x0ec3, 0x2a38, 0x0f46},
  {0x0c35, 0x3316, 0x1fc2, 0x342a, 0x05ee, 0x03f4, 0x2f6f, 0x0efe,
   0x37aa, 0x3710, 0x172c, 0x1e6a, 0x2a46, 0x3de7, 0x1256, 0x0afa},
  {0x3d8b, 0x17f9, 0x32bb, 0x0749, 0x15dd, 0x121a, 0x3c03, 0x21d9,
   0x3d39, 0x04ba, 0x0dea, 0x22ef, 0x3bb7, 0x0040, 0x20ba, 0x396a},
  {0x1ca6, 0x0edd, 0x1886, 0x01aa, 0x2b3f, 0x3eea, 0x206a, 0x12cc,
   0x018c, 0x0b3c, 0x0c30, 0x3903, 0x18c7, 0x0ead, 0x28f1, 0x0217},
  {0x1bd0, 0x3a67, 0x37b6, 0x1394, 0x2cc0, 0x3556, 0x0b80, 0x1ff0,
   0x0b68, 0x3a41, 0x26ef, 0x1d64, 0x3148, 0x3ee5, 0x0f04, 0x2d49}
},
{ {0x1e12, 0x08f3, 0x3107, 0x2660, 0x18c0, 0x34e9, 0x2407, 0x3a4f,
   0x0759, 0x1fec, 0x0a07, 0x1471, 0x246d, 0x12b1, 0x090a, 0x0d66},
  {0x08c5, 0x12e8, 0x137c, 0x106f, 0x16ba, 0x3741, 0x0bda, 0x0ce7,
   0x0a08, 0x1c43, 0x05d3, 0x2ba8, 0x099d, 0x0463, 0x0d13, 0x3048},
  {0x3d0c, 0x0b97, 0x28da, 0x0e66, 0x0524, 0x31c6, 0x1ee0, 0x05d6,
   0x0670, 0x3695, 0x2af1, 0x29de, 0x1b90, 0x0078, 0x1ea6, 0x0c0a},
  {0x2df2, 0x0e2b, 0x17db, 0x3718, 0x2f10, 0x1e4f, 0x3fe5, 0x0d3f,
   0x0761, 0x161b, 0x3b87, 0x392d, 0x33b8, 0x20bb, 0x0e0e, 0x243d},
  {0x20fc, 0x3169, 0x13a6, 0x0ccc, 0x30a7, 0x0f00, 0x0faf, 0x3c3c,
   0x05e6, 0x3802, 0x2b33, 0x39ed, 0x1f24, 0x1bef, 0x3b0a, 0x15b8},
  {0x39b4, 0x1132, 0x1ba5, 0x3430, 0x0ebc, 0x3ebc, 0x0348, 0x32ad,
   0x20fe, 0x0212, 0x0b58, 0x2f73, 0x3a9e, 0x3069, 0x1907, 0x3af5},
  {0x2a22, 0x1870, 0x2f8b, 0x3eda, 0x1c39, 0x2388, 0x09d2, 0x250d,
   0x2dab, 0x2e8f, 0x30c7, 0x0716, 0x132a, 0x0db3, 0x2471, 0x1ef2},
  {0x1550, 0x2bd8, 0x24fd, 0x3805, 0x3fac, 0x3e2f, 0x2bf1, 0x08e6,
   0x012b, 0x2758, 0x087e, 0x1c3c, 0x3fa1, 0x1ccf, 0x3b00, 0x198f},
  {0x3620, 0x0478, 0x3ac8, 0x3359, 0x1bb7, 0x2639, 0x1c73, 0x2c34,
   0x3f78, 0x2972, 0x3ffe, 0x2934, 0x151c, 0x0514, 0x022d, 0x1e09},
  {0x3406, 0x09bb, 0x18db, 0x3d89, 0x2ce0, 0x29d9, 0x2aaa, 0x16ae,
   0x1bdc, 0x2a5c, 0x34d4, 0x2d21, 0x1661, 0x0257, 0x1eef, 0x22d9},
  {0x0ddf, 0x261e, 0x3352, 0x122d, 0x3dd9, 0x0ee8, 0x2d89, 0x1485,
   0x38f2, 0x1720, 0x0755, 0x0067, 0x1b24, 0x158b, 0x000d, 0x1fc6},
  {0x0ce1, 0x0b66, 0x18ff, 0x2f71, 0x20d8, 0x12b7, 0x25c2, 0x298d,
   0x28bc, 0x3bea, 0x1c7a, 0x337d, 0x2f3d, 0x2b80, 0x226e, 0x39d3},
  {0x14ce, 0x0951, 0x1914, 0x0893, 0x0100, 0x1749, 0x3e44, 0x2e94,
   0x240d, 0x1e92, 0x2341, 0x0ffe, 0x0018, 0x3427, 0x0202, 0x3bcd},
  {0x1070, 0x1c9a, 0x3d6c, 0x0b1c, 0x3d4b, 0x002b, 0x1039, 0x2405,
   0x04fd, 0x1d0a, 0x0f80, 0x0c65, 0x2280, 0x0cbd, 0x2f20, 0x3a44},
  {0x1b17, 0x0b32, 0x0a74, 0x2af2, 0x3a46, 0x22b0, 0x1946, 0x3a11,
   0x1f38, 0x3bc2, 0x3d78, 0x28a3, 0x00c6, 0x3d51, 0x37ca, 0x2bf4},
  {0x086f, 0x3a1e, 0x39c1, 0x2683, 0x18ee, 0x06f1, 0x3154, 0x30d5,
   0x2c27, 0x3282, 0x1a4e, 0x30fd, 0x29ed, 0x21b8, 0x2f80, 0x20cc}
},
{ {0x1d9f, 0x0ab4, 0x245b, 0x282d, 0x25cc, 0x005f, 0x2f8a, 0x3614,
   0x2601, 0x1d58, 0x3f73, 0x0310, 0x0f17, 0x1329, 0x3bab, 0x144e},
  {0x2c1a, 0x2b3d, 0x2278, 0x3709, 0x04fc, 0x2acf, 0x2f90, 0x2ed5,
 0x0f58, 0x14d9, 0x3640, 0x331d, 0x25da, 0x382b, 0x181d, 0x3a6e},
{0x0b42, 0x1860, 0x2f6e, 0x067c, 0x1c1c, 0x302c, 0x2126, 0x2cb2,
 0x2596, 0x2020, 0x28d1, 0x3de2, 0x399e, 0x33d9, 0x304c, 0x3d9d},
{0x1c6d, 0x1535, 0x31d2, 0x0f60, 0x1725, 0x26b7, 0x25f3, 0x1b8f,
 0x152d, 0x3e2e, 0x13b3, 0x08f4, 0x390a, 0x0214, 0x1636, 0x1af8},
{0x3b7b, 0x3261, 0x0b0b, 0x1fc9, 0x0ad5, 0x0680, 0x114e, 0x3bba,
 0x21ba, 0x36a0, 0x11ca, 0x1d56, 0x069c, 0x25ed, 0x36e0, 0x3bfd},
{0x2261, 0x2752, 0x16d4, 0x3013, 0x163d, 0x3198, 0x32dc, 0x204d,
 0x17ae, 0x10f9, 0x3b37, 0x2361, 0x1bd3, 0x0f54, 0x0d3a, 0x0e67},
{0x27b1, 0x3608, 0x2b4c, 0x3229, 0x3b5a, 0x2813, 0x1631, 0x1c58,
 0x18b8, 0x26ab, 0x12ad, 0x1851, 0x359c, 0x381e, 0x3951, 0x2187},
{0x100a, 0x333b, 0x040e, 0x2c89, 0x1881, 0x2618, 0x187c, 0x3cf9,
 0x1434, 0x0446, 0x056b, 0x0fe9, 0x2fb4, 0x281f, 0x0b9a, 0x2608},
{0x1cda, 0x35cf, 0x33b0, 0x0b90, 0x2d4e, 0x0e61, 0x09c0, 0x3dff,
 0x0802, 0x1f21, 0x32fe, 0x0f89, 0x26f7, 0x1cd7, 0x2a5e, 0x05a5},
{0x06b0, 0x1ff3, 0x13ee, 0x34ac, 0x1440, 0x3f33, 0x1a62, 0x2be3,
 0x121b, 0x2cd0, 0x122e, 0x0d0c, 0x1958, 0x2fa2, 0x08b0, 0x3112},
{0x22a1, 0x3036, 0x3d68, 0x0ebd, 0x1dbc, 0x2d6a, 0x23fb, 0x0b47,
 0x278a, 0x079b, 0x0dae, 0x3fa3, 0x09a8, 0x3b04, 0x3ae4, 0x083d},
{0x21b5, 0x2f54, 0x0cd6, 0x224b, 0x1cef, 0x2a11, 0x335a, 0x20b4,
 0x198b, 0x2483, 0x0899, 0x2242, 0x3ed1, 0x348a, 0x0196, 0x2daa},
{0x3c66, 0x18ae, 0x0026, 0x1fa1, 0x3c04, 0x1d60, 0x24e4, 0x0879,
 0x3a3f, 0x2afd, 0x3e34, 0x1ecc, 0x289b, 0x0ea1, 0x2c7a, 0x1034},
{0x11aa, 0x00ac, 0x374b, 0x3e0f, 0x00f7, 0x3ef6, 0x1fb3, 0x3304,
 0x3445, 0x011c, 0x0eee, 0x2ef8, 0x196a, 0x19d3, 0x03d2, 0x11ee},
{0x0ef9, 0x1d0e, 0x2eed, 0x38f8, 0x2986, 0x23a0, 0x36e4, 0x3318,
   0x143c, 0x310c, 0x3894, 0x1cdd, 0x05b2, 0x3950, 0x3b9f, 0x22fb},
  {0x19e9, 0x0f2a, 0x1db6, 0x2363, 0x03ce, 0x06b4, 0x28c0, 0x09a2,
   0x28b4, 0x3383, 0x0d4e, 0x0620, 0x2231, 0x3166, 0x3e4d, 0x130e}
},
{ {0x2100, 0x0aa1, 0x1fe7, 0x2fed, 0x04f8, 0x0581, 0x1c4c, 0x2408,
   0x12e6, 0x3417, 0x18d1, 0x3e6d, 0x3826, 0x2de5, 0x21b2, 0x17a6},
  {0x388b, 0x0e23, 0x050a, 0x0e55, 0x1d75, 0x0403, 0x13e2, 0x256b,
   0x2fc8, 0x328a, 0x3653, 0x3ba2, 0x06ff, 0x24ac, 0x04fa, 0x1542},
  {0x28d8, 0x20c3, 0x0ca1, 0x0d59, 0x3f31, 0x2dd6, 0x2945, 0x0105,
   0x111b, 0x312a, 0x1f53, 0x0289, 0x3481, 0x38ea, 0x3c18, 0x05c9},
  {0x3e90, 0x1628, 0x0717, 0x08d4, 0x276a, 0x0140, 0x3a5b, 0x1a91,
   0x2f15, 0x02e6, 0x07dd, 0x12a0, 0x29e7, 0x33e7, 0x0c06, 0x1a64},
  {0x0554, 0x165b, 0x1c47, 0x39aa, 0x3d71, 0x36b3, 0x0e04, 0x1acb,
   0x2ead, 0x3fd2, 0x0dd7, 0x15d0, 0x25f1, 0x1f1a, 0x28c6, 0x0cdf},
  {0x047a, 0x1269, 0x0c7f, 0x078b, 0x1687, 0x1212, 0x21ac, 0x17f5,
   0x005b, 0x3089, 0x0969, 0x218f, 0x384d, 0x2424, 0x3edc, 0x073e},
  {0x3326, 0x2e1c, 0x0d95, 0x05f9, 0x0688, 0x244e, 0x0166, 0x1f95,
   0x2c01, 0x1995, 0x069a, 0x3591, 0x18e8, 0x2dbf, 0x0497, 0x3ae3},
  {0x1226, 0x224e, 0x38f4, 0x2529, 0x0be8, 0x2bc8, 0x0336, 0x2968,
   0x3c95, 0x38db, 0x257b, 0x28b6, 0x22b4, 0x11e2, 0x0f83, 0x0c8d},
  {0x1b39, 0x2e4b, 0x122b, 0x2545, 0x0a1c, 0x2550, 0x1a32, 0x0b7e,
   0x226f, 0x2d0b, 0x245c, 0x085a, 0x312c, 0x260a, 0x083c, 0x2bbd},
  {0x2cae, 0x14f5, 0x221c, 0x3a34, 0x174f, 0x3f53, 0x34af, 0x156e,
   0x33fa, 0x394b, 0x27cb, 0x25a8, 0x1e1d, 0x00e1, 0x11e8, 0x21a2},
  {0x1f7b, 0x2ad7, 0x1db9, 0x1aa9, 0x22b2, 0x0f6e, 0x16a3, 0x2568,
   0x0878, 0x0c81, 0x32b6, 0x022c, 0x0044, 0x298c, 0x001e, 0x1c6a},
  {0x1c27, 0x0d64, 0x3286, 0x1d2a, 0x1463, 0x211f, 0x0eb6, 0x1fa9,
   0x3647, 0x394f, 0x1264, 0x305a, 0x1f74, 0x2d72, 0x17e7, 0x18a9},
  {0x3b18, 0x2ad3, 0x2b8e, 0x2c77, 0x332b, 0x2819, 0x05c5, 0x20f4,
   0x0b8d, 0x359e, 0x20e9, 0x1bda, 0x3736, 0x12ac, 0x05e3, 0x03a9},
  {0x1e47, 0x07f7, 0x0aff, 0x14df, 0x1156, 0x093d, 0x1a6f, 0x0eaf,
   0x0d54, 0x2fc2, 0x1bea, 0x1ed0, 0x200a, 0x2495, 0x27ec, 0x314e},
  {0x361d, 0x279a, 0x1b4b, 0x23d1, 0x195e, 0x30e6, 0x1f1c, 0x2e02,
   0x0225, 0x2658, 0x26f4, 0x2d8e, 0x2097, 0x2410, 0x35ea, 0x3d1a},
  {0x3409, 0x1495, 0x2fcd, 0x22e9, 0x1bd6, 0x0d5e, 0x080f, 0x1f60,
   0x2703, 0x0ce3, 0x25b0, 0x1304, 0x34a8, 0x2123, 0x381a, 0x0862}
},
{ {0x1400, 0x21f3, 0x36d6, 0x1bbe, 0x18e3, 0x19bf, 0x300c, 0x2a5b,
   0x3fbc, 0x0de1, 0x2a56, 0x1a7e, 0x3093, 0x0957, 0x34ad, 0x374a},
  {0x316c, 0x048e, 0x0fbf, 0x25af, 0x0e8c, 0x0ad6, 0x12f1, 0x33c7,
   0x1d39, 0x30bb, 0x3eae, 0x083e, 0x2cde, 0x3fc8, 0x2f4c, 0x04ff},
  {0x0dd4, 0x0ec2, 0x262a, 0x0043, 0x1b78, 0x0c07, 0x18de, 0x041e,
   0x07b5, 0x0ca0, 0x3284, 0x3838, 0x0203, 0x0e88, 0x090e, 0x365e},
  {0x0df3, 0x02d7, 0x2d76, 0x322b, 0x0849, 0x1e57, 0x2621, 0x3814,
   0x1be6, 0x289c, 0x3d0e, 0x019a, 0x2128, 0x170d, 0x1d98, 0x0dd1},
  {0x3c2d, 0x1cb5, 0x1b7e, 0x15f7, 0x065f, 0x1087, 0x21ce, 0x3f20,
   0x2a13, 0x3ec6, 0x2431, 0x2a31, 0x0241, 0x0a83, 0x21d3, 0x2cac},
  {0x0eb1, 0x3de3, 0x21ec, 0x02e0, 0x31f4, 0x31db, 0x2d16, 0x2048,
   0x22a8, 0x33a1, 0x3b6e, 0x35b5, 0x0041, 0x3925, 0x322f, 0x0123},
  {0x244c, 0x2ca0, 0x2b06, 0x38f9, 0x3d38, 0x29b2, 0x13b6, 0x1f6e,
   0x3a96, 0x004e, 0x09a7, 0x2891, 0x3bcc, 0x165f, 0x2e3a, 0x227f},
  {0x3962, 0x3c06, 0x2ecc, 0x390c, 0x03c3, 0x10d8, 0x1d6d, 0x14b5,
   0x2513, 0x25b5, 0x39ce, 0x0436, 0x18d9, 0x23a7, 0x217b, 0x18f1},
  {0x1640, 0x222d, 0x3534, 0x146c, 0x3269, 0x25fc, 0x1d8a, 0x1e5c,
   0x0127, 0x1bc8, 0x3197, 0x26f1, 0x287a, 0x15b9, 0x0e8f, 0x1ebd},
  {0x2fa1, 0x2f68, 0x0328, 0x2c3b, 0x002f, 0x3a55, 0x18d2, 0x3e97,
   0x2c46, 0x0c72, 0x1f0e, 0x385d, 0x1b3d, 0x11c8, 0x39ad, 0x0e72},
  {0x0be0, 0x29a7, 0x0dcc, 0x3d16, 0x3e89, 0x119f, 0x255d, 0x3d98,
   0x2d4c, 0x1b79, 0x07fa, 0x2492, 0x3afd, 0x17f1, 0x1e6d, 0x013a},
  {0x00ad, 0x157b, 0x2ab2, 0x2b24, 0x1ab1, 0x2e9c, 0x11be, 0x0cb5,
   0x36b5, 0x258b, 0x0ff4, 0x0481, 0x1854, 0x2160, 0x0c4a, 0x056a},
  {0x0b0a, 0x190e, 0x2ffc, 0x2ff6, 0x2fe4, 0x2a2c, 0x383e, 0x0d46,
   0x146d, 0x181c, 0x15be, 0x0337, 0x31a4, 0x22c6, 0x112d, 0x14c3},
  {0x0392, 0x0ef8, 0x0eec, 0x3365, 0x1d79, 0x0be9, 0x0da9, 0x0b15,
   0x38cb, 0x282a, 0x3bff, 0x21b7, 0x21e6, 0x01fb, 0x13bc, 0x3781},
  {0x0047, 0x2fd9, 0x0c19, 0x2fe9, 0x2383, 0x1900, 0x0c62, 0x3145,
   0x3c11, 0x3b2a, 0x0566, 0x1091, 0x1752, 0x38d3, 0x09d4, 0x33c9},
  {0x057b, 0x2fe6, 0x2de9, 0x08ee, 0x09aa, 0x3e10, 0x3267, 0x2f85,
   0x367a, 0x189c, 0x3ada, 0x3b72, 0x1290, 0x1d27, 0x0504, 0x1b55}
},
{ {0x2677, 0x0506, 0x35c0, 0x348b, 0x3ac4, 0x05ef, 0x08c4, 0x041b,
   0x0e42, 0x2be8, 0x1ee5, 0x1cf7, 0x2e64, 0x18cf, 0x0d97, 0x1de4},
  {0x32a9, 0x1d1d, 0x131e, 0x0157, 0x1dfc, 0x12b5, 0x2bfd, 0x3632,
   0x3e47, 0x21c2, 0x1083, 0x2039, 0x2f58, 0x2b25, 0x18f7, 0x0db9},
  {0x0a62, 0x0867, 0x196e, 0x2f39, 0x3d3c, 0x1b0b, 0x1e8a, 0x2838,
   0x0c7b, 0x1233, 0x3dd0, 0x1ed1, 0x09f7, 0x209e, 0x095c, 0x169d},
  {0x114a, 0x2c98, 0x0076, 0x3743, 0x10e1, 0x3a26, 0x350f, 0x37ba,
   0x2474, 0x1356, 0x28a7, 0x3e01, 0x0ae3, 0x01ef, 0x0bf1, 0x35cd},
  {0x201f, 0x1fa0, 0x1dd5, 0x2d99, 0x2c8b, 0x3ff0, 0x333f, 0x259d,
   0x0ebb, 0x3dab, 0x3e41, 0x249a, 0x1331, 0x1eb8, 0x0dfa, 0x2724},
  {0x3395, 0x04d7, 0x2f4e, 0x2c06, 0x1242, 0x28af, 0x36a2, 0x2799,
   0x356f, 0x3213, 0x3010, 0x17b7, 0x3d77, 0x1d3c, 0x3cb0, 0x1d59},
  {0x31d6, 0x1013, 0x10fd, 0x0942, 0x20f7, 0x07df, 0x0d50, 0x160d,
   0x3cec, 0x26d7, 0x0301, 0x08a1, 0x1339, 0x1d61, 0x04f1, 0x135d},
  {0x1e61, 0x3b36, 0x1a9f, 0x118a, 0x1107, 0x2c4e, 0x2fb3, 0x36e6,
   0x10e8, 0x1279, 0x0eed, 0x2011, 0x2904, 0x3b8f, 0x248c, 0x328d},
  {0x3c20, 0x0f27, 0x0d65, 0x0499, 0x3c3b, 0x3a0f, 0x1432, 0x3b63,
   0x2582, 0x0e4f, 0x1dbf, 0x0757, 0x0b93, 0x0eba, 0x0abb, 0x3f2d},
  {0x0feb, 0x1175, 0x2a6e, 0x2e51, 0x1d5c, 0x37cb, 0x2f53, 0x0a44,
   0x3f6f, 0x0f7f, 0x072b, 0x0ef7, 0x2d37, 0x1808, 0x2d68, 0x1064},
  {0x33b5, 0x2baa, 0x1ac6, 0x2a57, 0x17a4, 0x1410, 0x2774, 0x1cd8,
   0x2e2d, 0x06fc, 0x3b03, 0x1856, 0x03cb, 0x2674, 0x23f0, 0x101d},
  {0x1753, 0x127a, 0x2f45, 0x0b19, 0x05d8, 0x2224, 0x15f1, 0x3fb5,
   0x0fa4, 0x3ade, 0x3bc7, 0x33c2, 0x0574, 0x36c8, 0x1e70, 0x212d},
  {0x2863, 0x2d0d, 0x1926, 0x1153, 0x28e1, 0x37a0, 0x2443, 0x30f0,
   0x2620, 0x2f46, 0x08be, 0x1547, 0x1198, 0x3415, 0x2359, 0x1e72},
  {0x1c70, 0x37f3, 0x11fd, 0x087a, 0x092a, 0x15fb, 0x39c6, 0x1db4,
   0x261f, 0x0c18, 0x34f3, 0x3d43, 0x1f32, 0x269b, 0x038d, 0x1ade},
  {0x10ce, 0x04e9, 0x01a7, 0x0199, 0x11d4, 0x3fa8, 0x1b6e, 0x2ace,
   0x363c, 0x0e70, 0x11da, 0x0521, 0x3675, 0x0beb, 0x3df6, 0x1220},
  {0x175b, 0x2729, 0x22e3, 0x3516, 0x28ec, 0x0731, 0x3b53, 0x3226,
   0x20a8, 0x05e5, 0x049e, 0x1477, 0x0d53, 0x1589, 0x1a80, 0x0a6e}
},
{ {0x3cc0, 0x06c6, 0x3adf, 0x194e, 0x191c, 0x3d76, 0x0015, 0x163a,
 0x0a25, 0x1ce9, 0x0b72, 0x1021, 0x0b6c, 0x2d73, 0x24d6, 0x27be},
{0x36d3, 0x220f, 0x04a9, 0x14cf, 0x27cc, 0x0754, 0x0e82, 0x3514,
 0x1ddc, 0x16bb, 0x3b0b, 0x1c65, 0x2aea, 0x00ba, 0x33ce, 0x1fc0},
{0x1fd7, 0x0f86, 0x2da0, 0x292a, 0x37c5, 0x3e1f, 0x11c3, 0x1d31,
 0x1996, 0x03e3, 0x2292, 0x30b4, 0x06d5, 0x2dfd, 0x3baa, 0x3db7},
{0x0a92, 0x0b56, 0x00d8, 0x0169, 0x0c79, 0x1d4f, 0x21c3, 0x3cb7,
 0x08bc, 0x0db4, 0x1e58, 0x3245, 0x290e, 0x213a, 0x2566, 0x3c3e},
{0x3714, 0x00f8, 0x08fb, 0x3b69, 0x3cae, 0x07ff, 0x1181, 0x0c1e,
 0x0495, 0x16be, 0x12fa, 0x391a, 0x1c0f, 0x1ba7, 0x2b26, 0x1534},
{0x3997, 0x0e69, 0x1a0b, 0x2dba, 0x17b9, 0x1baf, 0x2e13, 0x254a,
 0x0588, 0x0787, 0x03ac, 0x0569, 0x3c19, 0x2021, 0x3599, 0x272d},
{0x1766, 0x3e73, 0x1717, 0x1216, 0x1335, 0x112f, 0x109e, 0x373c,
 0x24b8, 0x32ab, 0x38ce, 0x2415, 0x3abe, 0x2a77, 0x00cf, 0x0064},
{0x0837, 0x011f, 0x1efb, 0x1bfa, 0x2246, 0x0056, 0x2f4a, 0x106b,
 0x36b6, 0x34b8, 0x32e6, 0x1709, 0x2656, 0x1a9e, 0x2a20, 0x383f},
{0x2b70, 0x0e36, 0x0f66, 0x180e, 0x2999, 0x2061, 0x1cfb, 0x37d7,
 0x0472, 0x062d, 0x340c, 0x0cf1, 0x167f, 0x2655, 0x1ae0, 0x1fb0},
{0x2698, 0x0712, 0x1052, 0x02cc, 0x3fc9, 0x009d, 0x324c, 0x16f5,
 0x3919, 0x21e5, 0x1e34, 0x1f25, 0x2158, 0x3b15, 0x1ca2, 0x1090},
{0x3e11, 0x1f44, 0x0e22, 0x0f42, 0x0077, 0x06fe, 0x1c95, 0x1385,
 0x2178, 0x047b, 0x1966, 0x0b51, 0x2402, 0x2440, 0x05d0, 0x3e71},
{0x1e20, 0x266c, 0x2e40, 0x2962, 0x142e, 0x1733, 0x22d7, 0x1543,
 0x2f5b, 0x1326, 0x149c, 0x05b0, 0x14fc, 0x3e5a, 0x145a, 0x20fd},
{0x2f18, 0x3e33, 0x32b1, 0x2b3e, 0x3e60, 0x064a, 0x1eac, 0x36e7,
 0x05a9, 0x3914, 0x08d7, 0x3564, 0x0a9f, 0x1008, 0x3c01, 0x2213},
{0x244f, 0x0bcd, 0x1a4f, 0x12b6, 0x244d, 0x1b0f, 0x2577, 0x315e,
     0x3040, 0x0ffa, 0x3964, 0x0e6e, 0x278c, 0x3926, 0x3839, 0x319b},
    {0x2d33, 0x20dd, 0x0a86, 0x24f6, 0x0d21, 0x361e, 0x2da3, 0x1d2b,
     0x256d, 0x1b87, 0x2916, 0x208c, 0x02c3, 0x0da8, 0x1cd0, 0x3fdd},
    {0x2f8e, 0x23da, 0x20ae, 0x0ca7, 0x036d, 0x15af, 0x1954, 0x300a,
     0x24af, 0x2b4b, 0x3e39, 0x0678, 0x2cdc, 0x0327, 0x0709, 0x0997}
  },
  { {0x26a2, 0x3100, 0x3f54, 0x16d9, 0x2d66, 0x17b5, 0x11d2, 0x2389,
     0x22b9, 0x2a8b, 0x081d, 0x3fe6, 0x1464, 0x2fe2, 0x0a50, 0x10b9},
    {0x1c80, 0x1a23, 0x38aa, 0x3609, 0x3605, 0x01f9, 0x3e0c, 0x2ee8,
     0x08eb, 0x09f9, 0x3371, 0x319a, 0x22c5, 0x0479, 0x19e4, 0x1a75},
    {0x05ff, 0x314c, 0x1629, 0x0c40, 0x241b, 0x311f, 0x0f56, 0x1910,
     0x14a3, 0x1d28, 0x347a, 0x0544, 0x12c4, 0x276b, 0x3498, 0x04e4},
    {0x0a90, 0x269c, 0x250a, 0x0d2d, 0x0a76, 0x225f, 0x03f0, 0x3307,
     0x0be3, 0x0be1, 0x3eab, 0x085c, 0x1616, 0x3046, 0x0b64, 0x0860},
    {0x29c0, 0x11d6, 0x3d93, 0x131f, 0x10bf, 0x3b86, 0x15c3, 0x2d71,
     0x36c2, 0x1125, 0x0a4b, 0x1f01, 0x353b, 0x0927, 0x2083, 0x0e49},
    {0x3e93, 0x21db, 0x0c77, 0x07c6, 0x354d, 0x1815, 0x1d2e, 0x175c,
     0x3888, 0x19b6, 0x35cb, 0x39c2, 0x0e3f, 0x1b4c, 0x2c38, 0x2d3e},
    {0x100d, 0x3367, 0x34bf, 0x11f9, 0x21bb, 0x3a69, 0x1f5d, 0x212c,
     0x17b8, 0x29c3, 0x33f6, 0x1177, 0x1b69, 0x3866, 0x1867, 0x3ff6},
    {0x3cff, 0x3398, 0x0721, 0x28f7, 0x3eed, 0x09d8, 0x247f, 0x131c,
     0x3991, 0x295b, 0x2e04, 0x2e8c, 0x2eb4, 0x3920, 0x1814, 0x2d2c},
    {0x339e, 0x1c79, 0x0266, 0x2e32, 0x3054, 0x3b42, 0x377c, 0x1674,
     0x0f26, 0x240b, 0x06d8, 0x3ff1, 0x1a18, 0x35ad, 0x3f71, 0x2412},
    {0x2504, 0x1b27, 0x16c7, 0x236c, 0x08ab, 0x0b6f, 0x1ece, 0x1777,
     0x2e0f, 0x3bca, 0x10f4, 0x0d74, 0x2937, 0x3ba8, 0x1580, 0x2880},
  {0x2234, 0x3bb8, 0x3a7d, 0x25ae, 0x1a22, 0x3a4b, 0x262b, 0x347b,
   0x3a60, 0x2873, 0x00df, 0x058e, 0x02fc, 0x27ca, 0x26fb, 0x2a2e},
  {0x2342, 0x2f86, 0x0ec4, 0x2f05, 0x255c, 0x3249, 0x3c2f, 0x201e,
   0x2dd8, 0x0491, 0x36a4, 0x1d81, 0x2b74, 0x257f, 0x0f4f, 0x16e4},
  {0x2933, 0x3133, 0x2b5a, 0x16c3, 0x2b5e, 0x094a, 0x2ec7, 0x31cf,
   0x1abb, 0x20cb, 0x2ebe, 0x216d, 0x2d7c, 0x09d7, 0x24fe, 0x3820},
  {0x2146, 0x3507, 0x0700, 0x04bf, 0x1933, 0x006e, 0x2ff9, 0x1c30,
   0x27cf, 0x3300, 0x3797, 0x0a67, 0x1f17, 0x3c2a, 0x237a, 0x32a2},
  {0x16f1, 0x2664, 0x0d16, 0x17ea, 0x09a5, 0x3044, 0x3552, 0x33e3,
   0x0550, 0x1e0b, 0x344b, 0x01ad, 0x340d, 0x05e1, 0x1721, 0x168f},
  {0x1e87, 0x17f2, 0x1cb6, 0x17e6, 0x04dd, 0x1b48, 0x1482, 0x2e1e,
   0x3f6d, 0x339d, 0x0c4e, 0x2e0e, 0x1f2f, 0x19d1, 0x041a, 0x2e25}
},
{ {0x0bd9, 0x14eb, 0x1e4c, 0x3a9d, 0x139d, 0x09fd, 0x0f55, 0x0c42,
   0x0f98, 0x1060, 0x250e, 0x07d1, 0x31fa, 0x3405, 0x3b0d, 0x35c5},
  {0x17a2, 0x1dd3, 0x05ba, 0x22de, 0x3d55, 0x3bbd, 0x0b2f, 0x3e8c,
   0x1fca, 0x02de, 0x015f, 0x2586, 0x09e1, 0x16f3, 0x147d, 0x0eea},
  {0x09e3, 0x0d80, 0x276c, 0x3560, 0x330f, 0x263d, 0x35e6, 0x1927,
   0x307c, 0x1397, 0x0949, 0x0a46, 0x05c4, 0x13b7, 0x2cf1, 0x32d6},
  {0x1a2b, 0x0aa0, 0x0c3e, 0x0a5d, 0x06da, 0x2423, 0x2afe, 0x1f35,
   0x2290, 0x13c8, 0x3976, 0x3421, 0x3d46, 0x32e7, 0x333a, 0x3863},
  {0x0e68, 0x1cfa, 0x3c53, 0x0699, 0x1743, 0x2c1e, 0x18c6, 0x33ad,
   0x1866, 0x1ae2, 0x32e4, 0x01a8, 0x09dd, 0x264a, 0x06ce, 0x0b53},
  {0x3b11, 0x3eb1, 0x36e1, 0x26c9, 0x0467, 0x0b5a, 0x34fb, 0x0471,
   0x2782, 0x1046, 0x350b, 0x1561, 0x2354, 0x186c, 0x3f3a, 0x22a7},
  {0x1ceb, 0x23d9, 0x2870, 0x221a, 0x35f9, 0x0292, 0x2b7e, 0x3cc5,
   0x2bcc, 0x32c6, 0x0f63, 0x0dfd, 0x04d3, 0x28c1, 0x0662, 0x1df6},
  {0x36ef, 0x00f2, 0x02c0, 0x2429, 0x1e74, 0x000f, 0x2f8d, 0x3813,
   0x3377, 0x1afc, 0x37b7, 0x15d7, 0x1fb2, 0x2e16, 0x0e11, 0x01d1},
  {0x0870, 0x2bca, 0x3c1b, 0x16aa, 0x3a56, 0x0bbe, 0x0324, 0x1214,
   0x2910, 0x0cad, 0x2ce1, 0x395e, 0x19d0, 0x034e, 0x1560, 0x1c7d},
  {0x3a0a, 0x2944, 0x21b1, 0x31b8, 0x07bd, 0x1841, 0x045c, 0x091b,
   0x12c2, 0x14c5, 0x1a6b, 0x2837, 0x0e1d, 0x00eb, 0x0e7b, 0x07bb},
  {0x1868, 0x3759, 0x33c6, 0x3b5d, 0x0fac, 0x3051, 0x0086, 0x1842,
   0x17e5, 0x24e9, 0x235e, 0x0b35, 0x3f3b, 0x2b44, 0x25b6, 0x0d0e},
  {0x1c24, 0x0503, 0x027b, 0x07c2, 0x1c85, 0x1092, 0x27f9, 0x3ae2,
   0x0600, 0x1062, 0x1c9b, 0x185f, 0x321c, 0x29e9, 0x0d40, 0x1c1d},
  {0x101a, 0x3676, 0x24c1, 0x0019, 0x320b, 0x2f42, 0x0a66, 0x37e5,
   0x20a4, 0x1581, 0x081c, 0x178d, 0x2c91, 0x1376, 0x0793, 0x2f6b},
  {0x3860, 0x0a9d, 0x02e2, 0x1d4c, 0x1d86, 0x00bf, 0x14a7, 0x3524,
   0x0d22, 0x32fb, 0x3e50, 0x0ee1, 0x05ca, 0x02e5, 0x082e, 0x3312},
  {0x00e0, 0x22ec, 0x1df7, 0x1d3f, 0x39bc, 0x05e8, 0x2396, 0x3143,
   0x13ae, 0x04c3, 0x3d70, 0x1d11, 0x3f2f, 0x103d, 0x08d2, 0x1b86},
  {0x03c2, 0x3cca, 0x2ff3, 0x1da9, 0x0bf6, 0x00d5, 0x1f8c, 0x1575,
   0x2e9a, 0x211d, 0x038c, 0x3384, 0x266d, 0x12ed, 0x0c87, 0x2648}
},
{ {0x0db5, 0x2630, 0x3cdc, 0x0c4c, 0x0acc, 0x18bf, 0x3f47, 0x34a5,
   0x2853, 0x3dfc, 0x39bd, 0x0ef2, 0x00e7, 0x0962, 0x164a, 0x141d},
  {0x3faa, 0x3cc2, 0x02cf, 0x19a5, 0x1980, 0x0971, 0x22a0, 0x39af,
   0x1d1e, 0x1ee3, 0x3ad0, 0x2562, 0x0ef0, 0x1755, 0x146b, 0x1ac1},
  {0x3ece, 0x1066, 0x2d46, 0x3847, 0x092c, 0x168c, 0x332d, 0x2ef1,
   0x04b3, 0x0883, 0x0534, 0x3423, 0x21a5, 0x359f, 0x0855, 0x100c},
{0x1500, 0x2148, 0x25db, 0x0c1a, 0x3dec, 0x2f99, 0x1d42, 0x23a8,
 0x2028, 0x0194, 0x3b79, 0x2bd3, 0x38fe, 0x3077, 0x3c99, 0x01cd},
{0x1a72, 0x3b94, 0x1ef4, 0x1fe1, 0x3beb, 0x27d8, 0x1557, 0x0c1b,
 0x3420, 0x2f84, 0x1b3e, 0x3540, 0x2544, 0x14b6, 0x1201, 0x12e3},
{0x24ce, 0x1144, 0x2950, 0x2e50, 0x279b, 0x1010, 0x1151, 0x05de,
 0x06e5, 0x02f5, 0x0931, 0x15e1, 0x1f90, 0x3446, 0x00ca, 0x0273},
{0x0ac9, 0x1aef, 0x10fa, 0x37f9, 0x2b69, 0x24f2, 0x075a, 0x1b0c,
 0x1dcd, 0x3127, 0x02ef, 0x3e42, 0x09b7, 0x3a1f, 0x116a, 0x10ba},
{0x2d50, 0x31e8, 0x37d2, 0x1729, 0x21d4, 0x1aea, 0x2d34, 0x130d,
 0x34e3, 0x10d9, 0x0c56, 0x0f0a, 0x1eaa, 0x31e2, 0x2641, 0x1cc2},
{0x0c80, 0x3205, 0x0539, 0x3e45, 0x1dc5, 0x1b11, 0x391d, 0x1735,
 0x1a3b, 0x339c, 0x0473, 0x2610, 0x3225, 0x106a, 0x0fca, 0x0f92},
{0x0876, 0x1a9d, 0x1d9d, 0x140b, 0x2c79, 0x326d, 0x16b6, 0x094e,
 0x11d3, 0x148e, 0x3dae, 0x0f09, 0x26c7, 0x2777, 0x3111, 0x16ad},
{0x3bf8, 0x18d3, 0x03e2, 0x1ed4, 0x29b8, 0x31dc, 0x19b5, 0x1617,
 0x1fd6, 0x24ec, 0x0055, 0x39fc, 0x35c7, 0x11af, 0x2ee2, 0x3264},
{0x0e4c, 0x3d52, 0x3c12, 0x025e, 0x303c, 0x1fd5, 0x0737, 0x31bf,
 0x11ef, 0x0ffd, 0x0727, 0x242a, 0x265a, 0x1f67, 0x152b, 0x008b},
{0x1a7d, 0x37d4, 0x3633, 0x25d8, 0x1de2, 0x0966, 0x2821, 0x07f3,
 0x3ed7, 0x1284, 0x14be, 0x0c12, 0x32a1, 0x0d67, 0x053a, 0x24e2},
{0x31c3, 0x1ea7, 0x14d7, 0x1b08, 0x0bde, 0x2e2f, 0x389b, 0x0e6f,
 0x2036, 0x106c, 0x1df9, 0x3f7e, 0x23b4, 0x249b, 0x3dd7, 0x2a25},
{0x3617, 0x1864, 0x06c5, 0x0161, 0x3f6c, 0x072a, 0x3e69, 0x2f29,
 0x3738, 0x1253, 0x1146, 0x39dd, 0x15b2, 0x3cf7, 0x20c0, 0x3120},
{0x00b2, 0x21c0, 0x2db3, 0x010a, 0x23e0, 0x30ba, 0x0ea5, 0x2268,
 0x26e6, 0x3480, 0x20f1, 0x2508, 0x0e34, 0x3810, 0x2a1a, 0x3dd3}
```

```
},
{ {0x06bd, 0x0486, 0x23db, 0x08b8, 0x3906, 0x0ff3, 0x050f, 0x14ae,
   0x0f84, 0x2e7f, 0x1a5c, 0x2096, 0x311d, 0x0080, 0x1e38, 0x0d08},
  {0x1dba, 0x190c, 0x3c40, 0x0a06, 0x052b, 0x345a, 0x2372, 0x21de,
   0x1b52, 0x07f0, 0x18e2, 0x33e9, 0x1166, 0x0a81, 0x34e4, 0x1135},
  {0x160b, 0x2afc, 0x228a, 0x3f77, 0x2cf2, 0x1c7e, 0x1451, 0x3422,
   0x2603, 0x2da2, 0x295d, 0x3657, 0x02e1, 0x39b7, 0x07aa, 0x159d},
  {0x32a7, 0x1e5d, 0x3f76, 0x31e1, 0x0955, 0x0f4c, 0x128e, 0x0535,
   0x292b, 0x026b, 0x300d, 0x3d01, 0x2434, 0x2cfc, 0x1126, 0x30ca},
  {0x3ae5, 0x07a2, 0x1a52, 0x2c99, 0x3b13, 0x087f, 0x138f, 0x1cfd,
   0x0c9d, 0x004b, 0x374c, 0x27bf, 0x3f94, 0x0f9c, 0x1020, 0x11ba},
  {0x095a, 0x1a08, 0x2b1d, 0x3d86, 0x3be0, 0x2bb5, 0x2865, 0x1aca,
   0x054c, 0x3bb4, 0x02b8, 0x3023, 0x0696, 0x0073, 0x1cde, 0x1a68},
  {0x2872, 0x0f13, 0x1d52, 0x38cc, 0x38da, 0x1dbb, 0x3e67, 0x2493,
   0x37cf, 0x3fef, 0x16b5, 0x1411, 0x2c19, 0x2c09, 0x1f82, 0x3095},
  {0x0c9f, 0x1b4d, 0x1f88, 0x0255, 0x205e, 0x260f, 0x1b72, 0x0490,
   0x1454, 0x0646, 0x2bce, 0x3d8c, 0x07b4, 0x3f85, 0x3c26, 0x318d},
  {0x3265, 0x0d73, 0x00f4, 0x3c0c, 0x2aef, 0x3e1e, 0x23cb, 0x046a,
   0x1148, 0x1b76, 0x28a2, 0x0071, 0x3f49, 0x100b, 0x2526, 0x3cd0},
  {0x0ea8, 0x1f71, 0x17af, 0x019b, 0x12cb, 0x26ff, 0x3262, 0x1de8,
   0x18b1, 0x2909, 0x37a1, 0x1c52, 0x1f40, 0x080b, 0x3ba3, 0x1f3d},
  {0x085e, 0x1346, 0x329f, 0x0bc1, 0x001c, 0x0a04, 0x0341, 0x2ecd,
   0x02a6, 0x1036, 0x1885, 0x0bc6, 0x3fcd, 0x37ee, 0x17a5, 0x194a},
  {0x280f, 0x3075, 0x38f6, 0x1e48, 0x3af1, 0x3407, 0x1c3a, 0x1aa6,
   0x0ca8, 0x2f67, 0x03f8, 0x0483, 0x3b33, 0x1797, 0x0fb7, 0x26ec},
  {0x01fc, 0x2a82, 0x1826, 0x2a6f, 0x1ec3, 0x3088, 0x3966, 0x00c1,
   0x3146, 0x0c36, 0x04f4, 0x0260, 0x2b9d, 0x2948, 0x1f47, 0x195b},
  {0x2900, 0x0685, 0x3fdb, 0x3c89, 0x14fa, 0x0405, 0x2f22, 0x019e,
   0x0b08, 0x18fb, 0x3aa1, 0x34e1, 0x121f, 0x3d84, 0x2ed7, 0x1ce6},
  {0x3892, 0x20c4, 0x019d, 0x2a9e, 0x275b, 0x0cb8, 0x0e52, 0x1c45,
   0x192f, 0x23c4, 0x3f97, 0x30c3, 0x0647, 0x280b, 0x3308, 0x0bdd},
  {0x3200, 0x27c3, 0x1715, 0x1cd9, 0x150e, 0x1f8f, 0x2d82, 0x3b75,
   0x02e4, 0x01b2, 0x0cc2, 0x20f2, 0x0540, 0x1a7f, 0x2b14, 0x201c}
},
{ {0x3900, 0x0c88, 0x1d17, 0x3ad3, 0x2397, 0x2d78, 0x118d, 0x1d2c,
   0x15bf, 0x2817, 0x1d34, 0x0832, 0x3698, 0x17fa, 0x384e, 0x0ab8},
  {0x12bd, 0x1e76, 0x1d8c, 0x18c9, 0x2e06, 0x1b57, 0x218d, 0x1612,
   0x0583, 0x103f, 0x229d, 0x1c74, 0x0753, 0x2ae9, 0x3843, 0x3bcb},
  {0x3760, 0x3796, 0x3707, 0x17ac, 0x21be, 0x23a6, 0x31f2, 0x2bc3,
   0x0dbf, 0x3543, 0x3c2e, 0x2eab, 0x3a95, 0x3ae0, 0x06b5, 0x019f},
  {0x20f8, 0x28a0, 0x3276, 0x1c96, 0x31a5, 0x3acc, 0x2ee4, 0x3499,
   0x02a8, 0x33dc, 0x3637, 0x3a20, 0x2f7b, 0x28c9, 0x1db0, 0x3239},
  {0x0acf, 0x0ec5, 0x0fdf, 0x1f99, 0x2e98, 0x10e4, 0x2654, 0x28ab,
   0x299e, 0x2835, 0x3fd7, 0x3e9e, 0x1837, 0x0ddc, 0x2bb6, 0x0961},
  {0x38f7, 0x136a, 0x370e, 0x1409, 0x3772, 0x3977, 0x37c0, 0x050b,
   0x1fb5, 0x3021, 0x1d3a, 0x0e95, 0x308b, 0x3cc7, 0x008a, 0x30e1},
  {0x2645, 0x34bd, 0x1ea0, 0x1f1e, 0x102d, 0x0ad1, 0x162b, 0x196d,
   0x32f3, 0x2c2a, 0x2eae, 0x2bdd, 0x037b, 0x3385, 0x0923, 0x04ec},
  {0x2fc1, 0x2cea, 0x1abd, 0x04eb, 0x3234, 0x1d08, 0x2e1b, 0x17e0,
   0x10a1, 0x2095, 0x0553, 0x0e57, 0x3e78, 0x1ad0, 0x3c4d, 0x24a8},
  {0x341c, 0x082a, 0x0ae4, 0x3221, 0x0984, 0x3afa, 0x0c3c, 0x1fe4,
   0x2e3e, 0x0276, 0x3dfe, 0x305c, 0x2147, 0x3af8, 0x06aa, 0x1162},
    {0x2672, 0x295e, 0x1f7f, 0x34e8, 0x2657, 0x2c65, 0x27aa, 0x1f10,
     0x3e1a, 0x0f3b, 0x233e, 0x1302, 0x185e, 0x0707, 0x3f29, 0x36c6},
    {0x128f, 0x0e37, 0x0112, 0x0a2a, 0x08e5, 0x3c5e, 0x0492, 0x39d2,
     0x2078, 0x067a, 0x1ff6, 0x3d18, 0x1844, 0x2625, 0x2081, 0x026d},
    {0x3aa8, 0x0d81, 0x2bf6, 0x346f, 0x0672, 0x3869, 0x23c2, 0x1e39,
     0x0353, 0x35e8, 0x0d58, 0x33b9, 0x10d4, 0x10f3, 0x183d, 0x15a4},
    {0x1c8e, 0x3821, 0x27ab, 0x282c, 0x2a86, 0x0530, 0x00a6, 0x10e5,
     0x1470, 0x2ccb, 0x2c3c, 0x099c, 0x173d, 0x34c1, 0x03aa, 0x0000},
    {0x1449, 0x0333, 0x026f, 0x1cdf, 0x299f, 0x2ec8, 0x340f, 0x3ad5,
     0x1d09, 0x1b92, 0x3b89, 0x15f8, 0x0b86, 0x03c7, 0x3519, 0x26de},
    {0x2ad8, 0x0022, 0x1dd0, 0x13b5, 0x256c, 0x3bb1, 0x2565, 0x1c48,
     0x247a, 0x3fbf, 0x076f, 0x1e88, 0x0cab, 0x17f8, 0x3616, 0x3edd},
    {0x1c1a, 0x0b3f, 0x2914, 0x36a9, 0x164b, 0x2154, 0x1710, 0x1426,
     0x3e05, 0x2acb, 0x39e3, 0x3975, 0x3434, 0x3e6e, 0x3984, 0x20f6}
  },
  { {0x0a53, 0x1f86, 0x0863, 0x0461, 0x1906, 0x0bb9, 0x16f6, 0x2d31,
     0x0da6, 0x22cb, 0x273e, 0x326b, 0x1987, 0x1a1e, 0x10f8, 0x0848},
    {0x1a49, 0x3607, 0x30eb, 0x30da, 0x0208, 0x1fb8, 0x049d, 0x2cd5,
     0x1b28, 0x3808, 0x3ce4, 0x269e, 0x3294, 0x0fd3, 0x21fa, 0x2b43},
    {0x10a9, 0x059c, 0x1d9c, 0x3f9b, 0x3454, 0x3000, 0x3929, 0x0c05,
     0x314d, 0x2153, 0x3fd3, 0x35e3, 0x3a6a, 0x2202, 0x037c, 0x378f},
    {0x0990, 0x0d02, 0x2ebc, 0x1ea5, 0x0f3d, 0x273b, 0x30bc, 0x2b35,
     0x1e35, 0x00c0, 0x23eb, 0x2ac7, 0x05e9, 0x031a, 0x1526, 0x236e},
    {0x10b2, 0x2bf0, 0x29e4, 0x1299, 0x1549, 0x1d44, 0x006d, 0x018a,
     0x10ab, 0x1778, 0x3504, 0x060a, 0x1d04, 0x13db, 0x1d73, 0x2818},
    {0x276f, 0x15e3, 0x262e, 0x0054, 0x1831, 0x3965, 0x1ba2, 0x1acc,
   0x2d83, 0x12f5, 0x1221, 0x3b58, 0x09bd, 0x3d47, 0x386f, 0x1d25},
  {0x3f9c, 0x2525, 0x0dbc, 0x1768, 0x0430, 0x199d, 0x007d, 0x073f,
   0x3199, 0x06e1, 0x1b26, 0x1af5, 0x2551, 0x0228, 0x3b71, 0x2592},
  {0x12e4, 0x0ccb, 0x079f, 0x3f61, 0x16fe, 0x0c0f, 0x362e, 0x29f5,
   0x2f6c, 0x1fdd, 0x11ce, 0x2304, 0x191e, 0x3b0e, 0x3a94, 0x0f01},
  {0x3019, 0x1635, 0x2340, 0x0d23, 0x0ce2, 0x294f, 0x1a2c, 0x00cd,
   0x109a, 0x1272, 0x06de, 0x1ab3, 0x2605, 0x1ad7, 0x27cd, 0x229f},
  {0x34a7, 0x206c, 0x12d2, 0x033b, 0x055f, 0x2fbf, 0x39f4, 0x102e,
   0x0de3, 0x3fba, 0x0a3f, 0x1b14, 0x3402, 0x1b67, 0x254f, 0x15ed},
  {0x02bd, 0x3a38, 0x3247, 0x21e1, 0x3df4, 0x0142, 0x1b81, 0x2a50,
   0x21dd, 0x36de, 0x0318, 0x37b3, 0x018b, 0x1ce8, 0x1d87, 0x1afe},
  {0x2479, 0x3de8, 0x3fd5, 0x38b1, 0x15d9, 0x0bc2, 0x127c, 0x0a41,
   0x3da9, 0x0002, 0x0904, 0x0209, 0x26cc, 0x1cac, 0x11f7, 0x070e},
  {0x14c9, 0x2460, 0x1c0a, 0x2dd2, 0x1a3e, 0x2f3e, 0x14a5, 0x18c4,
   0x3068, 0x3b7d, 0x2d92, 0x3985, 0x2183, 0x16bc, 0x0622, 0x2ee1},
  {0x3dd1, 0x1f50, 0x1f78, 0x1be9, 0x104e, 0x1983, 0x2043, 0x2151,
   0x1651, 0x05b6, 0x1e2a, 0x009a, 0x0e64, 0x0f05, 0x30e4, 0x349e},
  {0x010f, 0x16d0, 0x37f1, 0x2d98, 0x20e6, 0x3084, 0x24d9, 0x3502,
   0x18f0, 0x0ba2, 0x11c6, 0x3bac, 0x1599, 0x3d42, 0x1feb, 0x2335},
  {0x2594, 0x17eb, 0x334c, 0x29a9, 0x379c, 0x3a21, 0x1dcc, 0x32d9,
   0x1b22, 0x0b85, 0x022e, 0x097b, 0x3f52, 0x27bd, 0x319f, 0x239c}
},
{ {0x1f00, 0x2cf3, 0x3290, 0x2dec, 0x07cb, 0x21c1, 0x2686, 0x00a5,
   0x089f, 0x25fe, 0x05f6, 0x2fe1, 0x0aba, 0x25dd, 0x1c1f, 0x0d18},
  {0x362c, 0x39fb, 0x20bc, 0x0ae1, 0x35fc, 0x03ef, 0x2468, 0x2fa7,
   0x0346, 0x1957, 0x37f6, 0x1414, 0x0394, 0x0911, 0x017a, 0x305e},
{0x38ef, 0x1835, 0x3102, 0x2d28, 0x0602, 0x0bf9, 0x17ed, 0x2760,
 0x3a80, 0x0e79, 0x02a2, 0x31bd, 0x32b0, 0x3f82, 0x01c9, 0x043c},
{0x258e, 0x0c91, 0x1876, 0x0383, 0x2d6c, 0x0519, 0x1921, 0x0c24,
 0x30b0, 0x34de, 0x2427, 0x0a58, 0x1eaf, 0x31e7, 0x3b55, 0x0dcd},
{0x0d8f, 0x2e2c, 0x0932, 0x0599, 0x0913, 0x280d, 0x0fe8, 0x3331,
 0x291e, 0x37cc, 0x15aa, 0x1354, 0x058b, 0x3b5b, 0x1aee, 0x31b5},
{0x046b, 0x0d3b, 0x3058, 0x1d07, 0x2447, 0x2ed2, 0x1bd4, 0x00e4,
 0x39b8, 0x3a8b, 0x0d27, 0x1461, 0x019c, 0x09c4, 0x0852, 0x1756},
{0x0f29, 0x0090, 0x3d09, 0x233f, 0x153e, 0x11f5, 0x32f2, 0x3757,
 0x1291, 0x064b, 0x23b3, 0x3d83, 0x01c1, 0x38c1, 0x375d, 0x3338},
{0x3b3e, 0x14a9, 0x256f, 0x1f83, 0x311c, 0x117d, 0x2e49, 0x38f1,
 0x198d, 0x036a, 0x222f, 0x2780, 0x0626, 0x2a04, 0x19aa, 0x110e},
{0x0ba3, 0x02df, 0x3466, 0x0259, 0x33e0, 0x02ab, 0x0a1f, 0x11db,
 0x01d2, 0x0014, 0x23fe, 0x248b, 0x3fe4, 0x3c51, 0x1d70, 0x20d7},
{0x279c, 0x0795, 0x140a, 0x3d2d, 0x317b, 0x1f3a, 0x2505, 0x3b07,
 0x144a, 0x09d0, 0x2607, 0x0a89, 0x0b9b, 0x3f1e, 0x2bed, 0x08ed},
{0x16e0, 0x267a, 0x0206, 0x163e, 0x1199, 0x1645, 0x10ed, 0x232b,
 0x3898, 0x308e, 0x0564, 0x1c03, 0x05e0, 0x2041, 0x3753, 0x2d64},
{0x3610, 0x1ad5, 0x2c6f, 0x3776, 0x302b, 0x089d, 0x058a, 0x26fd,
 0x14ee, 0x1d14, 0x018e, 0x2d1a, 0x135f, 0x0c1f, 0x209a, 0x2f75},
{0x2696, 0x2333, 0x05a3, 0x2ea4, 0x11d7, 0x1aff, 0x1095, 0x36e5,
 0x2754, 0x1ec1, 0x0e5b, 0x309d, 0x3074, 0x141c, 0x188a, 0x285d},
{0x0b52, 0x0263, 0x3d03, 0x0cfc, 0x02a5, 0x07a3, 0x05fb, 0x1000,
 0x3a35, 0x1afd, 0x1776, 0x3ac2, 0x2346, 0x2ebb, 0x2f01, 0x3361},
{0x3f3e, 0x2091, 0x2687, 0x054a, 0x0480, 0x195f, 0x17e3, 0x09e9,
 0x3eb6, 0x382a, 0x2282, 0x0701, 0x3dde, 0x2843, 0x16ea, 0x2e7e},
  {0x2480, 0x0a8e, 0x3a78, 0x1553, 0x307e, 0x10eb, 0x0b99, 0x17fb,
   0x3ae8, 0x0220, 0x2d7a, 0x0c41, 0x1b5f, 0x3d2a, 0x2d00, 0x327c}
},
{ {0x22d4, 0x28ed, 0x0924, 0x08d3, 0x1266, 0x3ce3, 0x3f0f, 0x33ca,
   0x00fc, 0x3e0e, 0x1306, 0x2b39, 0x0094, 0x13df, 0x24b1, 0x0a80},
  {0x04b9, 0x2b45, 0x1332, 0x1b09, 0x26bf, 0x00d0, 0x251c, 0x2511,
   0x20c8, 0x31ff, 0x3df1, 0x21ad, 0x245e, 0x3cad, 0x391b, 0x1ec7},
  {0x2a1d, 0x3219, 0x07ac, 0x2762, 0x3623, 0x3efa, 0x00bd, 0x1dd8,
   0x1b05, 0x254b, 0x0a8f, 0x314b, 0x2cc2, 0x29a8, 0x13aa, 0x1139},
  {0x1b8e, 0x14b3, 0x3765, 0x2371, 0x275c, 0x2bf2, 0x355c, 0x2735,
   0x283e, 0x32a8, 0x3382, 0x168a, 0x0d0a, 0x1e18, 0x365c, 0x15d5},
  {0x1ef5, 0x144d, 0x0bbc, 0x2307, 0x068c, 0x1f6a, 0x1487, 0x3ed0,
   0x10a2, 0x1b4a, 0x0dac, 0x2831, 0x0ece, 0x0417, 0x2501, 0x1d48},
  {0x29ac, 0x1e9b, 0x00c4, 0x28fb, 0x2c1b, 0x3c7c, 0x1a46, 0x0f95,
   0x2ec6, 0x122c, 0x241d, 0x32e0, 0x23c6, 0x3f86, 0x2b5b, 0x191f},
  {0x34b4, 0x2fac, 0x0ba6, 0x06cb, 0x2836, 0x3980, 0x0734, 0x0565,
   0x39f8, 0x3b2d, 0x121d, 0x25ea, 0x356d, 0x0277, 0x2238, 0x0f69},
  {0x3de4, 0x2452, 0x1cb7, 0x2dc2, 0x241c, 0x0702, 0x0df2, 0x2667,
   0x05ab, 0x2203, 0x37ce, 0x1ae6, 0x2022, 0x306f, 0x0725, 0x158f},
  {0x3727, 0x124d, 0x15a1, 0x3557, 0x084b, 0x03d4, 0x2899, 0x2d25,
   0x01b1, 0x1c6b, 0x29e6, 0x2256, 0x3a64, 0x00d7, 0x0eca, 0x1c22},
  {0x37f5, 0x1800, 0x0986, 0x15d6, 0x227c, 0x2ec9, 0x3030, 0x3c46,
   0x02b1, 0x1c6f, 0x04d9, 0x3bbc, 0x15ac, 0x1b93, 0x3ad7, 0x15ba},
  {0x187f, 0x17d3, 0x05c1, 0x0224, 0x01ac, 0x29be, 0x17e4, 0x12f9,
   0x252d, 0x0153, 0x38fc, 0x0fc1, 0x0f35, 0x133e, 0x0c8a, 0x3b60},
  {0x0823, 0x012c, 0x304b, 0x3d79, 0x20dc, 0x26b4, 0x375a, 0x2955,
     0x3c77, 0x01bb, 0x1615, 0x2380, 0x2257, 0x2763, 0x3032, 0x3a4d},
    {0x3469, 0x2791, 0x01d6, 0x02b3, 0x0457, 0x2003, 0x156b, 0x208a,
     0x37d8, 0x05cb, 0x1e1e, 0x02a3, 0x10c1, 0x2e96, 0x2aa1, 0x0e99},
    {0x348d, 0x05a4, 0x0356, 0x1a04, 0x1f3c, 0x3bb6, 0x2d6b, 0x0be2,
     0x0959, 0x25ac, 0x1959, 0x2cd4, 0x0616, 0x15ef, 0x3acf, 0x27a7},
    {0x1bf0, 0x14b0, 0x2ea2, 0x374e, 0x3501, 0x1a21, 0x1a12, 0x2769,
     0x2d3c, 0x2fa8, 0x1cf0, 0x30e3, 0x1999, 0x0335, 0x3c90, 0x208f},
    {0x1475, 0x33c3, 0x01b6, 0x06d0, 0x1c42, 0x3aeb, 0x230e, 0x0775,
     0x12f3, 0x1bd2, 0x2857, 0x27a9, 0x2766, 0x2104, 0x39a7, 0x2b72}
  },
  { {0x26b5, 0x0d43, 0x29ea, 0x0bed, 0x3ccc, 0x2ed6, 0x081a, 0x1c83,
     0x127d, 0x3ee2, 0x2c5c, 0x2b57, 0x1d4b, 0x1d43, 0x09a9, 0x3ec9},
    {0x0af1, 0x27fc, 0x1cd6, 0x16c1, 0x1bfc, 0x3090, 0x0c59, 0x06b7,
     0x0b82, 0x163b, 0x242d, 0x39eb, 0x04b5, 0x0722, 0x2088, 0x3c88},
    {0x2016, 0x0cd0, 0x0b5c, 0x272e, 0x3d0d, 0x0e59, 0x1b12, 0x35f1,
     0x2520, 0x01ec, 0x0f23, 0x1b6c, 0x3d4f, 0x143f, 0x036e, 0x3012},
    {0x2563, 0x226a, 0x11c9, 0x213e, 0x0c47, 0x38a2, 0x0639, 0x39c8,
     0x13d1, 0x2a49, 0x0c5e, 0x26be, 0x3daa, 0x36aa, 0x3073, 0x155f},
    {0x01a6, 0x1f06, 0x007a, 0x2507, 0x1f3f, 0x2c97, 0x039a, 0x2365,
     0x3158, 0x073b, 0x2709, 0x11fe, 0x092e, 0x38ca, 0x38be, 0x398c},
    {0x20f0, 0x3d65, 0x0af3, 0x297c, 0x088b, 0x26af, 0x3233, 0x245a,
     0x12d3, 0x39f2, 0x1fb6, 0x2163, 0x0175, 0x3ea4, 0x2b49, 0x125f},
    {0x1261, 0x3a3b, 0x3ec5, 0x30ee, 0x1358, 0x0b89, 0x16b4, 0x1353,
     0x392b, 0x372f, 0x3cda, 0x3b74, 0x277b, 0x3ab5, 0x3fc2, 0x32de},
    {0x28cc, 0x176d, 0x1dda, 0x3832, 0x3587, 0x3da2, 0x06fd, 0x32e1,
     0x1bab, 0x1cdb, 0x0fd2, 0x0748, 0x105c, 0x278e, 0x1576, 0x1333},
    {0x23ad, 0x369f, 0x23be, 0x22fa, 0x07ad, 0x043f, 0x07b2, 0x0179,
     0x1ca5, 0x288e, 0x3b3b, 0x0e3c, 0x3d1c, 0x22f0, 0x08ff, 0x0fbd},
    {0x2d60, 0x344d, 0x2744, 0x0367, 0x2ff1, 0x17c1, 0x0f4b, 0x1e7f,
     0x384c, 0x257a, 0x22ee, 0x2742, 0x34aa, 0x1457, 0x17be, 0x3f14},
    {0x3ce0, 0x3693, 0x13c9, 0x26b9, 0x26e2, 0x3621, 0x0fdd, 0x21fd,
     0x316f, 0x3a6b, 0x25e8, 0x133c, 0x1391, 0x25ad, 0x1038, 0x06ae},
    {0x1c10, 0x31b9, 0x1ad8, 0x0342, 0x2eb6, 0x1ab4, 0x36f7, 0x2b2d,
     0x3feb, 0x199b, 0x3404, 0x2d4f, 0x23fa, 0x27f4, 0x0e85, 0x2d4d},
    {0x29d3, 0x0933, 0x1620, 0x2519, 0x16d1, 0x3eb4, 0x0c7c, 0x1724,
     0x1f6f, 0x333d, 0x1f33, 0x2488, 0x22a4, 0x2bf3, 0x28a8, 0x1d8b},
    {0x1a97, 0x1e02, 0x2680, 0x3818, 0x1827, 0x36ce, 0x2d05, 0x0001,
     0x1160, 0x2971, 0x3114, 0x33e1, 0x155a, 0x2439, 0x3fa2, 0x1fa7},
    {0x2825, 0x198a, 0x377b, 0x3d48, 0x1779, 0x09b4, 0x27c6, 0x3801,
     0x1149, 0x2b31, 0x1b0a, 0x05f1, 0x1da2, 0x0eda, 0x22e2, 0x307d},
    {0x19eb, 0x18bd, 0x2377, 0x292e, 0x3b78, 0x1a1a, 0x376d, 0x16cd,
     0x3bd0, 0x31aa, 0x2172, 0x178c, 0x0cbf, 0x3bef, 0x28c5, 0x119b}
  },
  { {0x23f9, 0x1386, 0x28ff, 0x1124, 0x2fcc, 0x0424, 0x168e, 0x2994,
     0x2798, 0x095b, 0x31bc, 0x33f1, 0x1f64, 0x1888, 0x3209, 0x3cab},
    {0x1bf4, 0x09ef, 0x0946, 0x12d8, 0x34c6, 0x3571, 0x20ee, 0x04a4,
     0x3c5d, 0x06d1, 0x3922, 0x30a1, 0x3e94, 0x2631, 0x2906, 0x2487},
    {0x2b86, 0x0144, 0x1574, 0x065e, 0x2cf7, 0x36d5, 0x1cc1, 0x2807,
     0x0f9e, 0x2bd6, 0x1ec6, 0x2175, 0x03a2, 0x341d, 0x282e, 0x268a},
    {0x0f5c, 0x24dd, 0x14f2, 0x2159, 0x0d5c, 0x1b4f, 0x253a, 0x2d9c,
     0x37fc, 0x0dc5, 0x210e, 0x20e5, 0x3cf4, 0x18b9, 0x0dcf, 0x176e},
    {0x054f, 0x0bc5, 0x29bd, 0x0fec, 0x1498, 0x0983, 0x2426, 0x243e,
    0x1bbd, 0x324b, 0x2f5c, 0x0892, 0x20ec, 0x1602, 0x321f, 0x2810},
   {0x2461, 0x069d, 0x04e6, 0x1319, 0x1fc3, 0x2f9f, 0x205a, 0x249c,
    0x1b44, 0x117b, 0x292f, 0x0cbc, 0x36f1, 0x3d94, 0x022f, 0x2308},
   {0x06d3, 0x3167, 0x0dc4, 0x31e5, 0x0e33, 0x212f, 0x037d, 0x1e1f,
    0x1582, 0x1649, 0x2136, 0x301d, 0x2a97, 0x01f8, 0x09b3, 0x3d88},
   {0x1118, 0x203e, 0x2e36, 0x2fc5, 0x092f, 0x1587, 0x10e2, 0x17bb,
    0x0fc8, 0x282b, 0x1dce, 0x2bc1, 0x345c, 0x389d, 0x27ce, 0x157e},
   {0x015a, 0x07da, 0x122a, 0x1ba0, 0x0c1d, 0x3cb4, 0x3d0f, 0x2e1a,
    0x0713, 0x33f4, 0x12b8, 0x3c24, 0x1ae5, 0x0470, 0x38a3, 0x3511},
   {0x106e, 0x1f28, 0x1366, 0x3a89, 0x0b3b, 0x3295, 0x3dc7, 0x2af3,
    0x2e89, 0x3f81, 0x0a2e, 0x0010, 0x0865, 0x0956, 0x2269, 0x1f5c},
   {0x28aa, 0x17b2, 0x167b, 0x36fc, 0x0ed5, 0x34a2, 0x2f19, 0x174c,
    0x1c50, 0x244a, 0x30ce, 0x16a4, 0x31ce, 0x384f, 0x19d4, 0x368c},
   {0x3fee, 0x016f, 0x324d, 0x1a9b, 0x01c2, 0x2489, 0x30c8, 0x32fd,
    0x2315, 0x34a3, 0x35af, 0x05aa, 0x3fd9, 0x37dd, 0x186b, 0x1a0f},
   {0x29e5, 0x0061, 0x0059, 0x3a17, 0x1d2d, 0x33ff, 0x2a98, 0x3901,
    0x25a1, 0x01af, 0x16b8, 0x07b1, 0x1127, 0x3885, 0x299a, 0x058c},
   {0x2226, 0x289e, 0x207f, 0x11bf, 0x2661, 0x2c87, 0x2997, 0x06ea,
    0x2271, 0x363e, 0x0222, 0x26e1, 0x3457, 0x1ee7, 0x128d, 0x3c17},
   {0x3d85, 0x2727, 0x3851, 0x0add, 0x246c, 0x33af, 0x3794, 0x317d,
    0x0591, 0x3f2e, 0x297b, 0x3b32, 0x178f, 0x1c5b, 0x3da3, 0x39a5},
   {0x361a, 0x3532, 0x2a79, 0x1605, 0x3c57, 0x2395, 0x3b10, 0x263e,
    0x37ab, 0x1fd2, 0x0223, 0x2541, 0x2a16, 0x190f, 0x16fa, 0x0021}
 },
 { {0x1529, 0x3e20, 0x29c4, 0x3e79, 0x0f73, 0x314a, 0x2db1, 0x36b7,
    0x3dc3, 0x288b, 0x367e, 0x2a7d, 0x36ae, 0x21e4, 0x3596, 0x031b},
{0x24e0, 0x39a9, 0x1317, 0x25d4, 0x27a8, 0x2fdc, 0x25ef, 0x1618,
 0x2633, 0x12c9, 0x04c6, 0x0e6b, 0x2884, 0x0b67, 0x231c, 0x1b6d},
{0x11ed, 0x2c66, 0x257d, 0x162f, 0x0401, 0x1986, 0x3231, 0x00e6,
 0x36f3, 0x2be1, 0x0947, 0x1eee, 0x1c82, 0x1eb7, 0x09c1, 0x3c65},
{0x2e5f, 0x2cbc, 0x31ba, 0x2c9a, 0x0654, 0x1b77, 0x2710, 0x0a91,
 0x3828, 0x05db, 0x0c6e, 0x139a, 0x1194, 0x22f9, 0x2de2, 0x0651},
{0x31d9, 0x0a4e, 0x2fe3, 0x0eac, 0x3369, 0x2d7f, 0x0f59, 0x17b4,
 0x1e68, 0x2f96, 0x29b9, 0x0131, 0x3e46, 0x3d58, 0x009f, 0x1a8a},
{0x3527, 0x22da, 0x22d6, 0x30c9, 0x29ec, 0x2878, 0x3c6d, 0x2640,
 0x3aa6, 0x1cb1, 0x3e95, 0x2060, 0x3a22, 0x1a29, 0x3f22, 0x0c29},
{0x1328, 0x119c, 0x115d, 0x11a4, 0x1dec, 0x31a8, 0x3efb, 0x392f,
 0x3487, 0x23ab, 0x2fb9, 0x03b0, 0x202a, 0x3151, 0x2181, 0x3251},
{0x0dc8, 0x0acd, 0x0a3a, 0x3cd2, 0x06b8, 0x0d86, 0x02ff, 0x0b4c,
 0x3ed9, 0x042b, 0x3d3d, 0x0732, 0x3746, 0x2f1f, 0x236a, 0x0116},
{0x0edb, 0x315f, 0x28ba, 0x21ea, 0x3aa0, 0x1e65, 0x2188, 0x26e0,
 0x3a00, 0x0674, 0x07ed, 0x03fb, 0x1f9f, 0x0092, 0x31d4, 0x2b78},
{0x2abc, 0x0fa2, 0x3268, 0x3488, 0x094b, 0x3b26, 0x0075, 0x1e3e,
 0x3346, 0x393a, 0x309a, 0x31c9, 0x22cc, 0x3cfd, 0x01b0, 0x1c5c},
{0x181f, 0x2b7f, 0x14cb, 0x22a9, 0x14c1, 0x005c, 0x1f63, 0x3de5,
 0x1247, 0x1dcf, 0x2e5e, 0x2f9d, 0x0155, 0x1b2b, 0x0a63, 0x2e75},
{0x3f8f, 0x1352, 0x3070, 0x1b21, 0x3e12, 0x2c57, 0x374d, 0x293d,
 0x2382, 0x3e5d, 0x369e, 0x0cd7, 0x2707, 0x2b79, 0x17d6, 0x19ad},
{0x2834, 0x3d2c, 0x313a, 0x1f94, 0x32b9, 0x16c0, 0x04e7, 0x3e1d,
 0x1dc1, 0x2f26, 0x05f8, 0x120e, 0x1fdb, 0x37fe, 0x3510, 0x3c4e},
{0x39d6, 0x3970, 0x0d9d, 0x00ce, 0x2c8f, 0x30f1, 0x1422, 0x3658,
 0x1801, 0x03c1, 0x0528, 0x2442, 0x3998, 0x34cc, 0x3241, 0x17c2},
    {0x281b, 0x3263, 0x19e8, 0x219f, 0x2927, 0x255f, 0x1d72, 0x3175,
     0x2338, 0x063e, 0x1930, 0x16a5, 0x16e7, 0x14f7, 0x1c07, 0x35d3},
    {0x230f, 0x1192, 0x0f3a, 0x06c9, 0x29cd, 0x3a4a, 0x1243, 0x2a9f,
     0x2399, 0x0cd1, 0x26d4, 0x2b4a, 0x077d, 0x2ce9, 0x3791, 0x1c66}
  },
  { {0x347c, 0x25e5, 0x0de9, 0x294e, 0x2e55, 0x381c, 0x297a, 0x3fa5,
     0x1189, 0x3f05, 0x1793, 0x32bc, 0x3dce, 0x3092, 0x2a89, 0x151d},
    {0x0d56, 0x0439, 0x28d7, 0x2270, 0x2de0, 0x2851, 0x1739, 0x3682,
     0x2cc8, 0x05f0, 0x23f2, 0x0a11, 0x3164, 0x3287, 0x10e7, 0x3477},
    {0x2d5f, 0x2da7, 0x0b06, 0x316d, 0x0035, 0x0978, 0x1d7c, 0x370a,
     0x093e, 0x1c53, 0x298b, 0x0ed0, 0x19bd, 0x3769, 0x1faa, 0x0b44},
    {0x1e6f, 0x02f4, 0x0c43, 0x2a5d, 0x0119, 0x3b6f, 0x34e0, 0x0661,
     0x32c0, 0x17ee, 0x25f5, 0x3e75, 0x1f9b, 0x06a6, 0x2bad, 0x1846},
    {0x05be, 0x18a6, 0x2ddd, 0x39fe, 0x0968, 0x223d, 0x2f00, 0x2855,
     0x36c9, 0x184b, 0x1134, 0x0a8d, 0x3f2a, 0x203c, 0x0ea3, 0x22c7},
    {0x1251, 0x220d, 0x2f2f, 0x23c9, 0x2721, 0x3e96, 0x2069, 0x0f8f,
     0x0610, 0x3ba6, 0x3978, 0x040f, 0x31fc, 0x2f47, 0x3d3a, 0x03fe},
    {0x3392, 0x12ef, 0x30d9, 0x0dee, 0x2d14, 0x1407, 0x2cd2, 0x0bfa,
     0x057d, 0x0106, 0x1f45, 0x07bf, 0x24bf, 0x1be2, 0x2bfa, 0x0365},
    {0x0347, 0x3935, 0x0d1a, 0x22d8, 0x1b43, 0x3d1d, 0x19f3, 0x2fbe,
     0x29dc, 0x2351, 0x028f, 0x1203, 0x1c4a, 0x025c, 0x11e9, 0x0340},
    {0x1780, 0x2573, 0x1d35, 0x2f82, 0x2244, 0x2859, 0x0898, 0x1143,
     0x0265, 0x150b, 0x13cd, 0x00ff, 0x013d, 0x3d66, 0x3bed, 0x2d03},
    {0x3c08, 0x04a3, 0x108b, 0x21d2, 0x01d0, 0x0835, 0x065d, 0x0efa,
     0x0f2f, 0x06a4, 0x23a2, 0x3330, 0x1e14, 0x10dc, 0x01b3, 0x1e86},
    {0x3855, 0x0896, 0x27f1, 0x2e61, 0x3d2b, 0x34bb, 0x121e, 0x12e5,
   0x1ca9, 0x04ae, 0x1390, 0x2ac1, 0x3d44, 0x063b, 0x3d3b, 0x1660},
  {0x3315, 0x0527, 0x10f6, 0x012a, 0x002d, 0x08de, 0x1c67, 0x2f87,
   0x043a, 0x1a35, 0x0689, 0x0e81, 0x33a2, 0x2d4b, 0x3db2, 0x1638},
  {0x393d, 0x0db2, 0x3026, 0x3e19, 0x1b18, 0x1665, 0x3a24, 0x357c,
   0x039f, 0x079a, 0x1eab, 0x082c, 0x3f08, 0x3ce2, 0x235f, 0x087d},
  {0x2fb5, 0x0163, 0x2089, 0x0024, 0x0e93, 0x2d39, 0x35b7, 0x1e5b,
   0x0a9c, 0x0b43, 0x154e, 0x0f2b, 0x1ad4, 0x040b, 0x39e4, 0x3a2b},
  {0x0cf9, 0x1e0d, 0x2a92, 0x3419, 0x0950, 0x134a, 0x2d26, 0x295c,
   0x2d07, 0x0fa6, 0x0c6b, 0x2c11, 0x3a04, 0x3d67, 0x2aed, 0x27a1},
  {0x26d0, 0x13a2, 0x3d4a, 0x11b7, 0x3539, 0x1572, 0x2733, 0x22f5,
   0x15df, 0x0861, 0x367c, 0x0b41, 0x35d4, 0x34ab, 0x11cd, 0x0d91}
},
{ {0x2a15, 0x30d6, 0x27e9, 0x2e54, 0x2e9d, 0x1c2f, 0x24b2, 0x1d85,
   0x1330, 0x1b5c, 0x3272, 0x3e15, 0x2f78, 0x1f7c, 0x2a2b, 0x23d2},
  {0x0d0d, 0x2741, 0x0676, 0x17dc, 0x3e38, 0x0768, 0x293a, 0x11d5,
   0x0970, 0x2dbb, 0x3f9a, 0x35fb, 0x1794, 0x3c6a, 0x05fa, 0x146f},
  {0x3569, 0x1d53, 0x3ef4, 0x1f7e, 0x2b36, 0x31f0, 0x1ec2, 0x2d95,
   0x0e3b, 0x0ed2, 0x09e0, 0x18e9, 0x0432, 0x2d17, 0x101e, 0x09df},
  {0x2e90, 0x3a39, 0x1245, 0x36ca, 0x1b7f, 0x39f9, 0x0242, 0x2aa3,
   0x003a, 0x1a78, 0x3a0e, 0x0357, 0x300e, 0x1450, 0x3156, 0x1362},
  {0x09ce, 0x1b35, 0x0bb5, 0x1ae9, 0x24ad, 0x0b60, 0x178b, 0x36fa,
   0x1c78, 0x0b4b, 0x14a1, 0x3972, 0x0de6, 0x0500, 0x0b25, 0x245d},
  {0x2015, 0x31ab, 0x0c46, 0x16c9, 0x110c, 0x1726, 0x040c, 0x0817,
   0x07b9, 0x15f2, 0x3e51, 0x28f3, 0x3e8e, 0x307f, 0x0d6c, 0x3081},
  {0x0c20, 0x3954, 0x2473, 0x0c58, 0x01b9, 0x0d1c, 0x27eb, 0x2481,
   0x18d0, 0x0b0c, 0x179e, 0x22bc, 0x200d, 0x089c, 0x3a1b, 0x2217},
    {0x2b50, 0x24fc, 0x32a4, 0x15ea, 0x2b97, 0x3cdd, 0x2890, 0x33a3,
     0x2331, 0x3cc3, 0x1ca4, 0x18d7, 0x27fd, 0x2783, 0x157a, 0x02f6},
    {0x3ff2, 0x3006, 0x37f4, 0x3e86, 0x2a88, 0x06b1, 0x1e32, 0x1a14,
     0x2d6f, 0x2671, 0x24cb, 0x3702, 0x0b49, 0x286d, 0x07ab, 0x17a1},
    {0x1bf9, 0x1a51, 0x1aa0, 0x1a4d, 0x3aef, 0x061e, 0x38e0, 0x3688,
     0x13c4, 0x074b, 0x1fcb, 0x1594, 0x23aa, 0x0efb, 0x331b, 0x2455},
    {0x04e2, 0x30ad, 0x1f59, 0x0f87, 0x14e8, 0x183a, 0x317e, 0x1d37,
     0x3e72, 0x066b, 0x0152, 0x2862, 0x1297, 0x20c2, 0x2b92, 0x08fc},
    {0x140f, 0x3324, 0x2eea, 0x1215, 0x2c71, 0x120a, 0x2385, 0x3942,
     0x3281, 0x33f2, 0x2144, 0x28bf, 0x09f8, 0x118c, 0x0373, 0x394d},
    {0x11bd, 0x3dcf, 0x18ac, 0x07e8, 0x0f38, 0x0cff, 0x00b9, 0x295a,
     0x36d8, 0x3c94, 0x3dd8, 0x26fe, 0x2c23, 0x2f92, 0x170f, 0x3836},
    {0x31f8, 0x1041, 0x195c, 0x3ba9, 0x18e7, 0x2cd8, 0x1a09, 0x1d3b,
     0x25b2, 0x0882, 0x31ee, 0x10fb, 0x3cb6, 0x1b53, 0x213c, 0x0cf5},
    {0x3258, 0x186f, 0x20df, 0x1ccd, 0x1685, 0x0caa, 0x1952, 0x2745,
     0x193e, 0x032b, 0x3d02, 0x0698, 0x225b, 0x3af7, 0x08ea, 0x1909},
    {0x1cca, 0x32c3, 0x00b6, 0x04d0, 0x2235, 0x3681, 0x0b8c, 0x1fda,
     0x38f3, 0x09bf, 0x0fe7, 0x2a00, 0x0060, 0x0adf, 0x07fb, 0x12b0}
  },
  { {0x0eb3, 0x2ab6, 0x03b3, 0x1d91, 0x357b, 0x3cdf, 0x231d, 0x336e,
     0x2347, 0x27e3, 0x2390, 0x3e8a, 0x333c, 0x11fc, 0x09ec, 0x153d},
    {0x0291, 0x022b, 0x12b2, 0x0d6f, 0x3203, 0x28ef, 0x19d9, 0x2ad2,
     0x1ab5, 0x21d7, 0x2e3d, 0x270a, 0x2725, 0x0434, 0x3bad, 0x08dd},
    {0x11b2, 0x1053, 0x043b, 0x161a, 0x32ac, 0x0453, 0x014a, 0x2867,
     0x39ea, 0x1f27, 0x10ef, 0x271a, 0x38bc, 0x23c1, 0x27b7, 0x12ba},
    {0x0d26, 0x378a, 0x3f0a, 0x076c, 0x0e3a, 0x2cad, 0x3ec3, 0x2ada,
   0x2517, 0x2969, 0x0b1a, 0x047c, 0x3df2, 0x12fc, 0x24da, 0x301e},
  {0x0a68, 0x25e0, 0x0352, 0x18cb, 0x20e3, 0x3559, 0x29b5, 0x35f3,
   0x012e, 0x396b, 0x1976, 0x0321, 0x2824, 0x0782, 0x3a50, 0x1a8d},
  {0x05cf, 0x0ee5, 0x1b13, 0x0cac, 0x26e4, 0x3e49, 0x273f, 0x2a63,
   0x117f, 0x1d7b, 0x2b6e, 0x3961, 0x3a45, 0x2140, 0x198c, 0x1081},
  {0x3273, 0x35f2, 0x30f6, 0x1186, 0x1283, 0x3505, 0x1aa2, 0x1658,
   0x2b21, 0x1492, 0x1fd0, 0x19f4, 0x1078, 0x1ccc, 0x237d, 0x0f30},
  {0x1da6, 0x09eb, 0x2e97, 0x3daf, 0x274f, 0x19f6, 0x3202, 0x1601,
   0x182b, 0x1bdb, 0x29ce, 0x1e3b, 0x1f26, 0x13a7, 0x0657, 0x0d5d},
  {0x2b3b, 0x113c, 0x34ff, 0x31d1, 0x0146, 0x1c49, 0x1a24, 0x06c8,
   0x3b25, 0x24ae, 0x24a5, 0x1857, 0x146e, 0x268f, 0x13fe, 0x2700},
  {0x006c, 0x0254, 0x3896, 0x334d, 0x312d, 0x2e9b, 0x2a4d, 0x0362,
   0x21ff, 0x10d2, 0x2715, 0x1c23, 0x2c50, 0x3726, 0x041f, 0x2432},
  {0x3461, 0x2f5f, 0x1f9e, 0x07a9, 0x107c, 0x1209, 0x3fa7, 0x02e7,
   0x34da, 0x2616, 0x3bf2, 0x1551, 0x23f5, 0x229c, 0x3b6c, 0x3d9f},
  {0x1e7b, 0x3535, 0x0294, 0x1c77, 0x0fc4, 0x132f, 0x24cf, 0x096c,
   0x0aa8, 0x20d2, 0x1088, 0x0f12, 0x08f7, 0x24de, 0x0f72, 0x2c4d},
  {0x1539, 0x0b5b, 0x259f, 0x294c, 0x0e44, 0x05a2, 0x3091, 0x1be5,
   0x2ddf, 0x1cce, 0x2005, 0x3c28, 0x3c22, 0x0792, 0x0db6, 0x154a},
  {0x1a6a, 0x175f, 0x1edc, 0x0349, 0x0526, 0x1883, 0x39c9, 0x3762,
   0x26eb, 0x225e, 0x1e42, 0x38e2, 0x3bc6, 0x1163, 0x0e5f, 0x3e6f},
  {0x02e8, 0x0693, 0x0858, 0x3121, 0x306c, 0x0ae9, 0x2030, 0x35ba,
   0x0a51, 0x3bc9, 0x0093, 0x2e07, 0x0b30, 0x221d, 0x3f98, 0x0a7e},
  {0x3f6e, 0x25c3, 0x0740, 0x01a9, 0x2f94, 0x3e62, 0x3140, 0x3e3a,
   0x08f8, 0x3c0f, 0x2946, 0x3141, 0x28a5, 0x26f5, 0x3f75, 0x290d}
},
{ {0x2832, 0x31eb, 0x2149, 0x1db7, 0x137e, 0x358a, 0x2ea3, 0x2b95,
   0x162c, 0x36a1, 0x13b4, 0x3c69, 0x1e64, 0x3f48, 0x2006, 0x10ad},
  {0x04a0, 0x24b3, 0x2f89, 0x0d36, 0x0293, 0x0902, 0x29e2, 0x043d,
   0x0efc, 0x2297, 0x284f, 0x0ab9, 0x2757, 0x35ca, 0x08fd, 0x3f68},
  {0x21d5, 0x117e, 0x281d, 0x2cb5, 0x360b, 0x38bd, 0x33fd, 0x323a,
   0x2d88, 0x04c5, 0x21cf, 0x0ed1, 0x0b18, 0x2ab7, 0x1830, 0x3cb3},
  {0x2c0d, 0x0fd4, 0x048c, 0x2be6, 0x206e, 0x077f, 0x3963, 0x0141,
   0x0428, 0x1657, 0x2b71, 0x24cc, 0x206b, 0x10b4, 0x114f, 0x0a5b},
  {0x3460, 0x3a8a, 0x3fda, 0x2ef5, 0x0d0b, 0x1dc0, 0x33c5, 0x20be,
   0x1e95, 0x07d7, 0x2206, 0x274d, 0x35ab, 0x3602, 0x0a40, 0x0a31},
  {0x108e, 0x10a6, 0x1b1f, 0x1f75, 0x0d60, 0x0745, 0x3c33, 0x1a1f,
   0x0125, 0x2120, 0x0398, 0x04c2, 0x3a54, 0x185c, 0x2c8d, 0x000b},
  {0x2770, 0x1c11, 0x09b1, 0x31ec, 0x0ea9, 0x0bad, 0x1387, 0x3066,
   0x3375, 0x15f0, 0x3d9e, 0x0af6, 0x38b0, 0x144b, 0x0eb0, 0x076b},
  {0x1584, 0x2f24, 0x06d4, 0x3ea7, 0x0279, 0x19de, 0x254c, 0x390f,
   0x2453, 0x25bb, 0x1c98, 0x34fc, 0x0a54, 0x11de, 0x0d69, 0x296c},
  {0x2fce, 0x3057, 0x3caf, 0x3c67, 0x2c20, 0x2367, 0x3f80, 0x34ae,
   0x0238, 0x2883, 0x0c49, 0x39ae, 0x0b33, 0x04cf, 0x230d, 0x3211},
  {0x080c, 0x1da3, 0x2a01, 0x1953, 0x077c, 0x156f, 0x0ca3, 0x1f0d,
   0x112e, 0x2cb7, 0x1956, 0x2905, 0x3761, 0x0507, 0x0a47, 0x20ed},
  {0x13c3, 0x3c07, 0x0b7b, 0x2611, 0x26ac, 0x349f, 0x3380, 0x1555,
   0x1e78, 0x0f45, 0x0197, 0x0af9, 0x1644, 0x308d, 0x0107, 0x3879},
  {0x0213, 0x2987, 0x387b, 0x12bf, 0x20d9, 0x3ecf, 0x08b1, 0x17ef,
   0x3da8, 0x2694, 0x1cd5, 0x1684, 0x2d1d, 0x30f4, 0x3cc1, 0x3c8b},
  {0x31b1, 0x0438, 0x365a, 0x3732, 0x36bd, 0x1d8d, 0x2302, 0x0aef,
   0x3b1b, 0x2a4e, 0x308c, 0x100f, 0x04f9, 0x3fc6, 0x0c38, 0x27df},
  {0x22e1, 0x2a52, 0x2856, 0x18f9, 0x0a65, 0x018f, 0x2871, 0x1614,
   0x315d, 0x09fb, 0x0c61, 0x074e, 0x124e, 0x1650, 0x2743, 0x0012},
  {0x1d99, 0x0fa8, 0x3a9b, 0x194b, 0x2d5d, 0x35d5, 0x3f3c, 0x1e0c,
   0x1a86, 0x180a, 0x28bb, 0x23b1, 0x2dfa, 0x071d, 0x02bf, 0x26f6},
  {0x0186, 0x38e3, 0x0c45, 0x2327, 0x0114, 0x1325, 0x1e82, 0x0fd0,
   0x042a, 0x0f0f, 0x164e, 0x2044, 0x0a6c, 0x12cd, 0x391f, 0x03b8}
},
{ {0x031f, 0x1f48, 0x1280, 0x13d9, 0x1c4d, 0x371e, 0x3d5a, 0x05ed,
   0x1c40, 0x1fb1, 0x133a, 0x372d, 0x0ff9, 0x2bfc, 0x123b, 0x22eb},
  {0x08d5, 0x0bea, 0x1417, 0x2d5a, 0x1b15, 0x04ad, 0x2926, 0x0f8c,
   0x2759, 0x1b9b, 0x2bc7, 0x0c92, 0x331f, 0x087b, 0x034f, 0x2cb3},
  {0x2c9c, 0x0bae, 0x068d, 0x215a, 0x202c, 0x1b2d, 0x3c45, 0x2659,
   0x3fb4, 0x25eb, 0x3a2d, 0x29f9, 0x04b7, 0x04b4, 0x358d, 0x198e},
  {0x3c0e, 0x00f0, 0x1460, 0x19f8, 0x0ec1, 0x2c53, 0x3017, 0x1566,
   0x3060, 0x051b, 0x1462, 0x22b5, 0x151a, 0x32ed, 0x1523, 0x3eb8},
  {0x3350, 0x021e, 0x1865, 0x0d6b, 0x1e8c, 0x1737, 0x1d84, 0x164f,
   0x3fff, 0x0c22, 0x1a2e, 0x1231, 0x0e24, 0x15e5, 0x2a0a, 0x238a},
  {0x0a5e, 0x2dac, 0x21d6, 0x0fc7, 0x0d61, 0x3f12, 0x17d2, 0x2acc,
   0x1630, 0x334b, 0x2063, 0x2d42, 0x20c6, 0x2992, 0x2c00, 0x0e16},
  {0x22e7, 0x2823, 0x025f, 0x305f, 0x18ab, 0x2368, 0x3a2a, 0x2bdf,
   0x37a7, 0x36a7, 0x0a18, 0x2a67, 0x0ba9, 0x2b5f, 0x1ef3, 0x265f},
  {0x3305, 0x335d, 0x342c, 0x1d1c, 0x165e, 0x3b5f, 0x364a, 0x2e19,
   0x2ff8, 0x3546, 0x3152, 0x29da, 0x0807, 0x22e4, 0x01e2, 0x158d},
  {0x33a4, 0x2336, 0x048d, 0x2c2d, 0x3cc6, 0x06a3, 0x2bbe, 0x0bee,
   0x2d15, 0x20a9, 0x0b7c, 0x3414, 0x0e77, 0x1b3f, 0x3f0c, 0x0fe6},
  {0x02b0, 0x2143, 0x1a44, 0x1453, 0x063d, 0x2983, 0x103e, 0x1bac,
       0x3248, 0x2bb1, 0x3082, 0x1670, 0x08c0, 0x0b4d, 0x1aba, 0x367d},
      {0x271f, 0x1573, 0x0d68, 0x06ab, 0x065b, 0x34a6, 0x28ee, 0x0c9a,
       0x2dbe, 0x1f6b, 0x293f, 0x01e9, 0x2fad, 0x33fc, 0x3fc1, 0x1cc4},
      {0x045a, 0x1a59, 0x113f, 0x1e6c, 0x2df3, 0x02d2, 0x0c54, 0x142f,
       0x1872, 0x3857, 0x0772, 0x3e4b, 0x020b, 0x07af, 0x0819, 0x16a9},
      {0x15fe, 0x322e, 0x199e, 0x3dd4, 0x3413, 0x0371, 0x258f, 0x3f8a,
       0x3f51, 0x17bd, 0x2ff7, 0x3b8c, 0x06cc, 0x269d, 0x1632, 0x1e40},
      {0x0444, 0x3561, 0x2b46, 0x1f8b, 0x2f77, 0x1706, 0x1322, 0x2bbf,
       0x1b75, 0x3113, 0x03d7, 0x10b6, 0x30b2, 0x1e55, 0x0cec, 0x2ca6},
      {0x2090, 0x1955, 0x112b, 0x0380, 0x166c, 0x252e, 0x3252, 0x1922,
       0x28fa, 0x033a, 0x1f57, 0x104a, 0x0604, 0x389e, 0x2ec1, 0x01c3},
      {0x1a26, 0x0791, 0x02db, 0x0786, 0x2734, 0x1a10, 0x016a, 0x217f,
       0x15ee, 0x0b05, 0x2a2f, 0x1741, 0x2534, 0x2042, 0x208d, 0x2bcb}
    },
    { {0x2b10, 0x071a, 0x1a95, 0x05d4, 0x0316, 0x182d, 0x32ff, 0x0567,
       0x1641, 0x29ae, 0x1579, 0x19a8, 0x1a37, 0x1ac8, 0x1fbd, 0x2211},
      {0x3c30, 0x234d, 0x0299, 0x2bd7, 0x09a4, 0x2cdf, 0x29a4, 0x2062,
       0x2d7b, 0x1225, 0x211c, 0x0b10, 0x38d8, 0x3ad4, 0x3eee, 0x2086},
      {0x3a63, 0x2c4a, 0x3e31, 0x2af8, 0x05fe, 0x3bc3, 0x2f12, 0x1e26,
       0x0039, 0x0361, 0x131b, 0x03a0, 0x075b, 0x3a2f, 0x0bc4, 0x0e63},
      {0x0ac5, 0x058f, 0x1676, 0x1a6d, 0x3c9f, 0x3b9b, 0x0aab, 0x1c35,
       0x2f98, 0x03ad, 0x3a98, 0x3988, 0x3804, 0x2584, 0x2717, 0x2e01},
      {0x0ea2, 0x199c, 0x29a3, 0x2c68, 0x135b, 0x25de, 0x31cb, 0x3581,
       0x1aad, 0x174b, 0x2c0c, 0x1834, 0x1591, 0x22d2, 0x08c6, 0x3e91},
      {0x3d6e, 0x20d5, 0x3573, 0x3b1a, 0x380c, 0x0cd3, 0x144c, 0x38e6,
       0x2888, 0x30c2, 0x0464, 0x277c, 0x37c1, 0x332c, 0x3c3a, 0x1f15},
  {0x1820, 0x3993, 0x0ee9, 0x2ce3, 0x284a, 0x2321, 0x30b1, 0x10c8,
   0x02e9, 0x1082, 0x239e, 0x3191, 0x3f84, 0x3094, 0x02c1, 0x20b9},
  {0x2606, 0x0b0e, 0x2930, 0x2edd, 0x2f1c, 0x0e25, 0x111d, 0x1a74,
   0x0c68, 0x28f6, 0x3bf6, 0x1785, 0x1cae, 0x3106, 0x08cc, 0x088d},
  {0x1680, 0x023b, 0x2980, 0x0d34, 0x116e, 0x1f07, 0x18be, 0x2e4d,
   0x0618, 0x18f2, 0x01c4, 0x2f25, 0x2c6c, 0x379d, 0x2822, 0x3d21},
  {0x03b4, 0x12ea, 0x0f4d, 0x2808, 0x39cd, 0x3b5e, 0x2522, 0x2d55,
   0x03af, 0x1971, 0x0ede, 0x1f0c, 0x2498, 0x2b07, 0x23ec, 0x06ed},
  {0x18a2, 0x0890, 0x30c6, 0x2e93, 0x0247, 0x35d6, 0x24e3, 0x3685,
   0x2521, 0x263a, 0x32aa, 0x1061, 0x3c44, 0x37c6, 0x293e, 0x3f4c},
  {0x2ee0, 0x3dc5, 0x08a8, 0x05a1, 0x1aa5, 0x29c1, 0x3575, 0x2fd0,
   0x2d19, 0x2bbc, 0x03e1, 0x1d22, 0x175d, 0x2650, 0x2a29, 0x232d},
  {0x11f6, 0x1206, 0x16ef, 0x27dd, 0x3664, 0x373b, 0x2e9e, 0x1eea,
   0x3734, 0x10cb, 0x08d6, 0x24dc, 0x085d, 0x0642, 0x1e1a, 0x301c},
  {0x3f63, 0x0063, 0x2a87, 0x3c68, 0x349b, 0x0b9c, 0x3243, 0x1a03,
   0x27e6, 0x09ed, 0x28d2, 0x3b2b, 0x3ad2, 0x3f37, 0x10fc, 0x381b},
  {0x0154, 0x3470, 0x0b73, 0x0784, 0x385e, 0x2e67, 0x2fb7, 0x1a5f,
   0x3a68, 0x25ee, 0x0f0c, 0x1e83, 0x0134, 0x149f, 0x247c, 0x1633},
  {0x1011, 0x1998, 0x0cb6, 0x2fa0, 0x29bb, 0x0270, 0x2676, 0x37ea,
   0x1db1, 0x3b57, 0x1401, 0x0655, 0x2dc0, 0x2fba, 0x0c9c, 0x34a0}
},
{ {0x397d, 0x380d, 0x3690, 0x2037, 0x3538, 0x10db, 0x0b5f, 0x293b,
   0x1746, 0x13e8, 0x3b19, 0x316a, 0x30ed, 0x24be, 0x2b8b, 0x2c14},
  {0x2f30, 0x23c0, 0x32a0, 0x06a8, 0x3dd2, 0x21a4, 0x0ab1, 0x10f2,
   0x0b0d, 0x35e2, 0x149a, 0x3b20, 0x2a39, 0x1d6a, 0x327a, 0x29c8},
  {0x2515, 0x3ed6, 0x00d1, 0x138c, 0x36db, 0x17c9, 0x1bc4, 0x2917,
 0x2e22, 0x01d9, 0x0262, 0x201b, 0x187d, 0x0d83, 0x195a, 0x0bd7},
{0x30ab, 0x1113, 0x02b9, 0x002e, 0x08a2, 0x0422, 0x3ed8, 0x3309,
 0x2963, 0x3255, 0x2d02, 0x032c, 0x38b3, 0x19e1, 0x2be7, 0x10ea},
{0x17a7, 0x0fc9, 0x2ffb, 0x0191, 0x371d, 0x0783, 0x0309, 0x0c89,
 0x2b73, 0x2fdb, 0x21cc, 0x16eb, 0x18cd, 0x1383, 0x22aa, 0x0614},
{0x03b9, 0x04af, 0x05fd, 0x10da, 0x1fa5, 0x31af, 0x2ee5, 0x3afb,
 0x26ce, 0x0fc3, 0x14ba, 0x25dc, 0x101b, 0x043e, 0x2f43, 0x1c5d},
{0x0b20, 0x0135, 0x354a, 0x2320, 0x067b, 0x3b05, 0x363a, 0x1978,
 0x0ad3, 0x0e2a, 0x2726, 0x2491, 0x1c8b, 0x0351, 0x3e7f, 0x35a6},
{0x2e65, 0x0a8b, 0x0e15, 0x1429, 0x051f, 0x1a6e, 0x3555, 0x20aa,
 0x0448, 0x0a29, 0x3907, 0x2fcf, 0x1128, 0x39d4, 0x2419, 0x148f},
{0x3d92, 0x21b9, 0x10de, 0x0bc8, 0x39f6, 0x15d4, 0x0a78, 0x3cbb,
 0x3715, 0x335b, 0x3930, 0x2797, 0x1e53, 0x113d, 0x33dd, 0x2e82},
{0x109f, 0x374f, 0x09fa, 0x079d, 0x3e14, 0x25a3, 0x2330, 0x34e7,
 0x2f48, 0x0b92, 0x2024, 0x1480, 0x1014, 0x3d31, 0x1849, 0x1a76},
{0x11ff, 0x15d3, 0x3583, 0x23e6, 0x3ef8, 0x3679, 0x3968, 0x2985,
 0x1ede, 0x2050, 0x08a4, 0x398b, 0x1bc6, 0x3d37, 0x1089, 0x11c5},
{0x162d, 0x34d9, 0x2409, 0x3fb3, 0x364d, 0x0227, 0x1448, 0x1889,
 0x16a8, 0x00c2, 0x0b28, 0x2932, 0x0408, 0x2a83, 0x0bb3, 0x3bb9},
{0x2bf7, 0x138b, 0x3484, 0x04be, 0x165c, 0x07f2, 0x373d, 0x3a88,
 0x1beb, 0x1985, 0x0f22, 0x155e, 0x2c49, 0x227d, 0x1a02, 0x309e},
{0x273a, 0x3119, 0x0da4, 0x0f49, 0x2033, 0x0bb2, 0x1027, 0x1e73,
 0x289a, 0x372c, 0x155c, 0x3e74, 0x2d2f, 0x30d7, 0x0e26, 0x0f99},
{0x3045, 0x1097, 0x14a4, 0x2254, 0x074d, 0x305d, 0x1852, 0x34f9,
 0x10f1, 0x3c87, 0x1348, 0x1baa, 0x3228, 0x3e26, 0x07ea, 0x240a},
{0x1c3f, 0x0cba, 0x3fb6, 0x3a6f, 0x00af, 0x08d8, 0x3782, 0x29fa,
   0x0ba8, 0x38ff, 0x1ed5, 0x1327, 0x0243, 0x05d7, 0x3d80, 0x0c63}
},
{ {0x294d, 0x3ea5, 0x0d1d, 0x3619, 0x24bb, 0x1edf, 0x190a, 0x0afe,
   0x2b13, 0x1468, 0x2690, 0x33a5, 0x0a4c, 0x1c68, 0x0eb2, 0x3447},
  {0x176a, 0x0201, 0x2d12, 0x0c09, 0x3cd3, 0x1e94, 0x21a3, 0x05b7,
   0x2ce4, 0x2f60, 0x044a, 0x2422, 0x0f5d, 0x0866, 0x07a4, 0x142b},
  {0x3ce9, 0x036b, 0x1d46, 0x2fd2, 0x0485, 0x29c2, 0x2f0b, 0x2a7b,
   0x1501, 0x3376, 0x3750, 0x2d24, 0x3171, 0x2c59, 0x36c5, 0x073c},
  {0x12d4, 0x3670, 0x3c76, 0x1208, 0x1015, 0x2103, 0x1b3a, 0x0934,
   0x1773, 0x257c, 0x2c0e, 0x2eeb, 0x1f98, 0x15f5, 0x1810, 0x00d6},
  {0x3542, 0x36d1, 0x3dc0, 0x0f7d, 0x2b4d, 0x2c36, 0x3a23, 0x138a,
   0x1a48, 0x25b8, 0x1003, 0x05a7, 0x19fc, 0x2e5b, 0x072f, 0x3934},
  {0x19a0, 0x1ee9, 0x37e3, 0x263f, 0x18e1, 0x166e, 0x3179, 0x231f,
   0x29f2, 0x14dc, 0x2288, 0x3fa4, 0x1626, 0x316e, 0x39a3, 0x338f},
  {0x307b, 0x1e63, 0x1a06, 0x1caa, 0x2fff, 0x0e39, 0x3494, 0x1ca8,
   0x09e2, 0x39a0, 0x099e, 0x11c7, 0x152a, 0x30dd, 0x354f, 0x36cd},
  {0x000a, 0x208e, 0x1496, 0x0fee, 0x33c1, 0x00a4, 0x2c2c, 0x0143,
   0x09a0, 0x041c, 0x28ce, 0x15e8, 0x2e3f, 0x21ae, 0x20cd, 0x2e8d},
  {0x2f4f, 0x3fd4, 0x2aa9, 0x04b8, 0x246f, 0x05e7, 0x3c58, 0x0fae,
   0x3cd4, 0x224f, 0x2fa4, 0x31c4, 0x23e4, 0x008d, 0x2716, 0x0dbd},
  {0x166d, 0x36e3, 0x0989, 0x0908, 0x1c13, 0x216f, 0x26cf, 0x2877,
   0x2941, 0x1051, 0x07c5, 0x1df8, 0x0314, 0x1fc5, 0x1efa, 0x3aad},
  {0x098f, 0x3c27, 0x2e88, 0x11e1, 0x2b1c, 0x361b, 0x24e6, 0x01a5,
   0x2a78, 0x162e, 0x084e, 0x3fde, 0x2325, 0x10be, 0x3e2a, 0x1f49},
  {0x1024, 0x0a73, 0x3b7a, 0x03e9, 0x0df1, 0x21e9, 0x149b, 0x1e17,
   0x3f01, 0x27a2, 0x0f5b, 0x29ab, 0x0d44, 0x2066, 0x2efb, 0x148d},
  {0x3f26, 0x0d9e, 0x3e55, 0x327f, 0x34f2, 0x3eff, 0x1822, 0x3b48,
   0x2c5a, 0x0a1b, 0x169e, 0x088f, 0x0650, 0x0f47, 0x3087, 0x2200},
  {0x272f, 0x1908, 0x3456, 0x1f52, 0x13b1, 0x05cc, 0x2198, 0x091c,
   0x0aaa, 0x1652, 0x39a6, 0x0875, 0x095f, 0x212e, 0x072c, 0x2e7c},
  {0x3163, 0x1d1a, 0x1ded, 0x38ad, 0x3aba, 0x0695, 0x237c, 0x0ac1,
   0x029d, 0x2a73, 0x3b49, 0x1111, 0x0416, 0x0099, 0x06dc, 0x3067},
  {0x0df9, 0x15b6, 0x1028, 0x1690, 0x357d, 0x0ea6, 0x3dbe, 0x3207,
   0x0668, 0x145b, 0x1b89, 0x0659, 0x0509, 0x1784, 0x2e34, 0x3779}
},
{ {0x1dd2, 0x11bb, 0x1f18, 0x045d, 0x3d7b, 0x06c4, 0x304f, 0x30df,
   0x2598, 0x0ff5, 0x30a5, 0x39cb, 0x0388, 0x2920, 0x1536, 0x1321},
  {0x3474, 0x1c9f, 0x04e8, 0x3098, 0x23dc, 0x353e, 0x2c52, 0x1425,
   0x1823, 0x2c4f, 0x317f, 0x3899, 0x3845, 0x1489, 0x3882, 0x16e6},
  {0x0de5, 0x29b6, 0x3cef, 0x0390, 0x0774, 0x2009, 0x0da5, 0x03eb,
   0x2109, 0x15d2, 0x1395, 0x313f, 0x04a2, 0x29b7, 0x3c32, 0x12b4},
  {0x0eef, 0x04d6, 0x3f30, 0x3d69, 0x1f46, 0x233d, 0x1ce7, 0x00da,
   0x37b9, 0x1505, 0x0847, 0x2d01, 0x357a, 0x014b, 0x3450, 0x2475},
  {0x182a, 0x2f06, 0x3873, 0x1c99, 0x3349, 0x296e, 0x3313, 0x1465,
   0x07e4, 0x23f7, 0x0fd8, 0x0418, 0x0181, 0x0307, 0x3a29, 0x2839},
  {0x0db7, 0x19ce, 0x3a93, 0x27da, 0x09c7, 0x0489, 0x0a05, 0x194c,
   0x0188, 0x1d16, 0x370c, 0x0a6a, 0x2768, 0x2bda, 0x0443, 0x1533},
  {0x3b3d, 0x0840, 0x090c, 0x1af9, 0x2eb5, 0x0a15, 0x13a3, 0x0dc0,
   0x070f, 0x24e7, 0x2c18, 0x3ddc, 0x1884, 0x3dcb, 0x00ee, 0x2152},
  {0x03f7, 0x1d29, 0x1d30, 0x24a1, 0x3443, 0x366f, 0x2f97, 0x24b5,
   0x37b4, 0x0fbe, 0x0ccf, 0x3033, 0x08f1, 0x178e, 0x10fe, 0x3aa2},
  {0x2803, 0x3053, 0x3b3c, 0x34cb, 0x274c, 0x0537, 0x31e6, 0x1019,
     0x1efe, 0x31a2, 0x3afe, 0x1de0, 0x0025, 0x01d5, 0x24d0, 0x03de},
    {0x2e48, 0x0719, 0x259c, 0x29d0, 0x0936, 0x310d, 0x011b, 0x3acb,
     0x23f1, 0x19cd, 0x323e, 0x220a, 0x317a, 0x37ec, 0x07dc, 0x1389},
    {0x2902, 0x00fe, 0x123d, 0x1444, 0x3079, 0x0c14, 0x2fcb, 0x33ef,
     0x22f7, 0x104b, 0x17bc, 0x3a51, 0x013f, 0x2875, 0x057f, 0x27c5},
    {0x22a6, 0x3730, 0x20d1, 0x2ba6, 0x3de1, 0x35ef, 0x2456, 0x3649,
     0x2a6c, 0x30cd, 0x11c4, 0x1cc3, 0x1ab6, 0x024c, 0x1d67, 0x30a3},
    {0x25c5, 0x1d69, 0x16db, 0x28ad, 0x3d26, 0x3a3d, 0x342b, 0x0cd9,
     0x1bd8, 0x24f4, 0x2251, 0x0ded, 0x253d, 0x21f5, 0x1610, 0x0def},
    {0x2403, 0x37dc, 0x0926, 0x1e93, 0x033c, 0x103b, 0x1f22, 0x2232,
     0x19c6, 0x1359, 0x1be3, 0x06f3, 0x19e2, 0x1861, 0x2417, 0x20d3},
    {0x0644, 0x0598, 0x3aae, 0x0889, 0x1655, 0x2776, 0x3e52, 0x3680,
     0x1a38, 0x03b5, 0x35e5, 0x0555, 0x0d88, 0x359b, 0x1195, 0x2c0f},
    {0x26bd, 0x33b6, 0x1b2a, 0x1e67, 0x0b29, 0x3d60, 0x39a1, 0x399d,
     0x3118, 0x3887, 0x3235, 0x05d9, 0x0b07, 0x1d63, 0x2eca, 0x03b2}
  },
  { {0x25e1, 0x281a, 0x377e, 0x16cf, 0x044b, 0x0af4, 0x14aa, 0x35f8,
     0x3c92, 0x3e4a, 0x22bb, 0x0271, 0x1e11, 0x2d54, 0x3b9a, 0x2225},
    {0x0915, 0x1eba, 0x3c50, 0x2a0f, 0x226d, 0x1b1d, 0x0705, 0x03ba,
     0x1d66, 0x2201, 0x30c0, 0x25a6, 0x0e3d, 0x1c88, 0x0b7a, 0x15c5},
    {0x10dd, 0x3553, 0x14c4, 0x01dc, 0x004a, 0x1812, 0x2b52, 0x1a79,
     0x0a32, 0x32b5, 0x1300, 0x2eec, 0x3ddd, 0x1e8b, 0x04e0, 0x0c3d},
    {0x318b, 0x1483, 0x211e, 0x2bdc, 0x2b20, 0x0c4f, 0x12e9, 0x37d3,
     0x3628, 0x11e5, 0x1a31, 0x2001, 0x2df4, 0x2228, 0x351f, 0x3611},
    {0x31c2, 0x12d5, 0x1b94, 0x0395, 0x11c1, 0x1324, 0x2118, 0x0765,
     0x086e, 0x234b, 0x38c8, 0x05c2, 0x0210, 0x2d3d, 0x0fe5, 0x1b58},
  {0x04f0, 0x2cb6, 0x105d, 0x2f63, 0x039e, 0x0dd8, 0x2df7, 0x0393,
   0x1355, 0x2fd4, 0x2cbb, 0x3014, 0x2c54, 0x39db, 0x083a, 0x071f},
  {0x0cde, 0x36d4, 0x0058, 0x3d91, 0x1bec, 0x01bf, 0x1098, 0x37ad,
   0x13b8, 0x21ab, 0x1a4b, 0x076a, 0x38df, 0x0261, 0x030e, 0x2ef2},
  {0x0350, 0x25a7, 0x3854, 0x300f, 0x0c39, 0x2274, 0x14de, 0x24f5,
   0x1809, 0x3130, 0x0b48, 0x3645, 0x1e31, 0x3b24, 0x2973, 0x1402},
  {0x3567, 0x3a10, 0x2aa5, 0x3b7f, 0x0d07, 0x051c, 0x2786, 0x3784,
   0x0d63, 0x200b, 0x0a20, 0x21b0, 0x32ef, 0x3161, 0x3eb3, 0x01e8},
  {0x00a9, 0x001f, 0x1d96, 0x3db3, 0x112c, 0x04f6, 0x36ec, 0x2355,
   0x19f2, 0x1357, 0x0d2e, 0x19b4, 0x27f0, 0x1f4d, 0x1979, 0x053e},
  {0x1af2, 0x182c, 0x1970, 0x38b4, 0x2a93, 0x173c, 0x0cfa, 0x218a,
   0x0ac6, 0x17da, 0x39cf, 0x0b95, 0x331e, 0x3ae7, 0x1e2b, 0x344c},
  {0x1994, 0x1d49, 0x2a64, 0x0601, 0x179d, 0x0ae5, 0x03cd, 0x0fd6,
   0x3650, 0x1f43, 0x0b8e, 0x1981, 0x3a52, 0x1f29, 0x2df6, 0x114d},
  {0x0b65, 0x33d3, 0x2f4d, 0x0919, 0x3eb0, 0x24ff, 0x395c, 0x180b,
   0x0ed8, 0x039d, 0x088c, 0x006a, 0x2393, 0x1169, 0x3e2b, 0x07b0},
  {0x13ff, 0x0e1c, 0x37d5, 0x1351, 0x2253, 0x3981, 0x2c76, 0x2d63,
   0x36da, 0x3bfb, 0x2465, 0x02fd, 0x2ce7, 0x2190, 0x0d78, 0x149e},
  {0x07e3, 0x2fdd, 0x3c0b, 0x2f1b, 0x12a2, 0x2485, 0x3830, 0x1fab,
   0x38e5, 0x1d24, 0x091d, 0x29e0, 0x29b4, 0x17cf, 0x0245, 0x21a0},
  {0x219a, 0x2051, 0x18b6, 0x37fa, 0x27fa, 0x1c3d, 0x02ae, 0x1e6b,
   0x1790, 0x1511, 0x2a42, 0x3280, 0x2fc9, 0x013b, 0x3cf2, 0x3c74}
},
{ {0x0ed3, 0x3459, 0x382d, 0x170b, 0x13cc, 0x21bf, 0x329c, 0x125d,
   0x0b98, 0x2575, 0x381d, 0x2e63, 0x0b71, 0x2965, 0x2fc0, 0x328e},
  {0x1664, 0x14fb, 0x1716, 0x2384, 0x3e5c, 0x3a57, 0x0185, 0x2248,
 0x2db0, 0x0ad7, 0x3a40, 0x060b, 0x30aa, 0x3d59, 0x1d76, 0x2111},
{0x1816, 0x2c16, 0x234e, 0x14e5, 0x31e0, 0x2c85, 0x18d6, 0x1a73,
 0x1cdc, 0x0028, 0x3c1d, 0x3b93, 0x2dda, 0x1bf7, 0x2cf9, 0x0d7f},
{0x3671, 0x092d, 0x1e1c, 0x265c, 0x1fd4, 0x3c9e, 0x3104, 0x3d90,
 0x0da1, 0x287b, 0x099b, 0x2469, 0x15e2, 0x2fd5, 0x10c4, 0x3af4},
{0x19ee, 0x04bd, 0x3a48, 0x3b90, 0x2b6f, 0x13ce, 0x3ee8, 0x3a65,
 0x0799, 0x044d, 0x0cc9, 0x3188, 0x0ecb, 0x268c, 0x36c4, 0x29fd},
{0x3149, 0x33c4, 0x24df, 0x2c70, 0x3426, 0x3a58, 0x0fb6, 0x0b76,
 0x1cee, 0x19d7, 0x3452, 0x0b1e, 0x2db7, 0x329d, 0x38b5, 0x1abf},
{0x0e90, 0x1ae7, 0x3306, 0x00f9, 0x336c, 0x3712, 0x2a19, 0x20ad,
 0x1ef9, 0x189a, 0x29ff, 0x3d05, 0x108d, 0x0995, 0x2eba, 0x34d2},
{0x290f, 0x34d3, 0x07b7, 0x36f9, 0x037a, 0x18a4, 0x10ae, 0x0441,
 0x22df, 0x2792, 0x326f, 0x2b23, 0x1255, 0x3e68, 0x1569, 0x2fec},
{0x3dca, 0x1c31, 0x2a6b, 0x2a3b, 0x0d4c, 0x30e7, 0x0190, 0x06c7,
 0x2dcc, 0x011a, 0x1f6d, 0x0ab0, 0x129a, 0x1c62, 0x1692, 0x0e86},
{0x2e80, 0x0a7a, 0x1096, 0x3bd7, 0x2c7c, 0x0082, 0x0062, 0x327e,
 0x20a6, 0x323b, 0x20bd, 0x1320, 0x1c14, 0x0bdf, 0x1875, 0x1114},
{0x3f5c, 0x3bd2, 0x12d1, 0x258a, 0x0eab, 0x0e50, 0x1e52, 0x157f,
 0x37da, 0x1c3b, 0x26fc, 0x2fd8, 0x2767, 0x3076, 0x0ad8, 0x0f21},
{0x2a0d, 0x12b3, 0x08ae, 0x096b, 0x3d3e, 0x20b1, 0x0149, 0x232f,
 0x2107, 0x0d38, 0x22a2, 0x1ea8, 0x1546, 0x35a4, 0x00c3, 0x1050},
{0x08af, 0x232a, 0x376b, 0x3a42, 0x363f, 0x08d1, 0x1747, 0x3aac,
 0x01d8, 0x3f7f, 0x1dbe, 0x0162, 0x0cc5, 0x05dd, 0x158a, 0x11d9},
{0x2651, 0x2349, 0x2c8e, 0x2895, 0x189d, 0x1738, 0x0522, 0x1932,
 0x063f, 0x1d02, 0x2e4e, 0x397e, 0x2b37, 0x0339, 0x37bb, 0x19dc},
{0x3428, 0x3999, 0x2c86, 0x3a79, 0x137d, 0x2b66, 0x34ec, 0x1179,
   0x0e54, 0x3d8f, 0x3e9d, 0x16d2, 0x364e, 0x05f7, 0x3678, 0x36bf},
  {0x335c, 0x37ac, 0x0bb6, 0x19a9, 0x2b87, 0x2aab, 0x0382, 0x169b,
   0x3d96, 0x1963, 0x017b, 0x1d4a, 0x1fb9, 0x0e31, 0x0f36, 0x26d6}
},
{ {0x228c, 0x2348, 0x0bfd, 0x0f48, 0x10cd, 0x2a55, 0x30b5, 0x3f2c,
   0x10c7, 0x0894, 0x16c5, 0x3159, 0x1d5d, 0x1007, 0x124a, 0x3b2f},
  {0x2e30, 0x397a, 0x0a16, 0x0b13, 0x2496, 0x3485, 0x26b8, 0x0459,
   0x358f, 0x228e, 0x34c7, 0x1054, 0x3624, 0x0315, 0x0ec6, 0x0244},
  {0x0e75, 0x227b, 0x0f82, 0x2ff2, 0x3a92, 0x2704, 0x05b9, 0x2139,
   0x08a7, 0x13d6, 0x1b66, 0x0ec7, 0x18b5, 0x2e66, 0x1c56, 0x32d4},
  {0x1e59, 0x0814, 0x1a40, 0x2c7d, 0x06f8, 0x228d, 0x3b4b, 0x337e,
   0x0563, 0x375c, 0x2d04, 0x1984, 0x2013, 0x21e7, 0x3889, 0x3dcd},
  {0x3f96, 0x3345, 0x27a6, 0x20b7, 0x24aa, 0x03bf, 0x1f72, 0x0113,
   0x276e, 0x008c, 0x0174, 0x2531, 0x3ce8, 0x2119, 0x1f37, 0x1a05},
  {0x2af0, 0x202f, 0x3bce, 0x27f3, 0x340a, 0x0f5a, 0x262d, 0x0543,
   0x3078, 0x2f61, 0x2864, 0x0643, 0x1254, 0x002c, 0x126a, 0x3e58},
  {0x32d0, 0x27bb, 0x2ef6, 0x0ac2, 0x1be1, 0x0450, 0x01df, 0x1713,
   0x154c, 0x19a1, 0x20c9, 0x2fd6, 0x0f7b, 0x1937, 0x0d6a, 0x2956},
  {0x1116, 0x0283, 0x124c, 0x3e3d, 0x2c15, 0x0fde, 0x3ab1, 0x3db6,
   0x21d8, 0x34c2, 0x052f, 0x1774, 0x1c76, 0x3663, 0x01fd, 0x15a3},
  {0x36e9, 0x2dc6, 0x2599, 0x3577, 0x28f5, 0x3f5d, 0x1ef8, 0x3ab2,
   0x236d, 0x2ac3, 0x21f1, 0x3e04, 0x028e, 0x2fb1, 0x0bca, 0x03ee},
  {0x2b47, 0x35a3, 0x35e1, 0x0d98, 0x0240, 0x3520, 0x24c7, 0x15bd,
   0x1748, 0x253b, 0x1b45, 0x1d82, 0x3c9d, 0x13f3, 0x1193, 0x30f7},
  {0x2134, 0x14d3, 0x376e, 0x30b9, 0x2127, 0x31d8, 0x27d1, 0x28f8,
   0x2c2f, 0x3af0, 0x125e, 0x39cc, 0x2255, 0x3063, 0x39ba, 0x369a},
  {0x1ada, 0x2750, 0x3afc, 0x373f, 0x2e4c, 0x2e7b, 0x1c0e, 0x0850,
   0x1cec, 0x239b, 0x2421, 0x1f66, 0x2b82, 0x0057, 0x0f64, 0x0d8c},
  {0x3368, 0x203d, 0x0233, 0x0be6, 0x38a6, 0x1677, 0x23f6, 0x3764,
   0x3876, 0x0229, 0x22ac, 0x216b, 0x1d0b, 0x3a97, 0x33e6, 0x167d},
  {0x351d, 0x26ca, 0x24b7, 0x3a13, 0x0518, 0x0576, 0x3190, 0x334e,
   0x062e, 0x1123, 0x2d7e, 0x09be, 0x0bd4, 0x1686, 0x12a1, 0x189b},
  {0x2ab0, 0x3a1a, 0x0992, 0x1623, 0x1509, 0x23d3, 0x1188, 0x15ae,
   0x2b5d, 0x0580, 0x08dc, 0x3c8c, 0x13da, 0x0c67, 0x27d5, 0x32f0},
  {0x27d3, 0x1f16, 0x10a0, 0x2753, 0x3a62, 0x02ca, 0x1f7d, 0x278f,
   0x2e03, 0x13ed, 0x1643, 0x02dd, 0x0a34, 0x103a, 0x258c, 0x1c5a}
},
{ {0x1318, 0x205c, 0x00b4, 0x2bd9, 0x271e, 0x2d96, 0x0a95, 0x25ff,
   0x29ca, 0x34bc, 0x1121, 0x1b71, 0x166f, 0x2764, 0x2ffa, 0x01ca},
  {0x0cb4, 0x18b4, 0x1b50, 0x3cd1, 0x18fc, 0x1277, 0x3548, 0x26a1,
   0x2c90, 0x3d10, 0x23bc, 0x33df, 0x2323, 0x2057, 0x032a, 0x16a0},
  {0x04d4, 0x0e27, 0x2f51, 0x130c, 0x2ac9, 0x15a8, 0x3293, 0x2fe8,
   0x1aac, 0x1dad, 0x03f2, 0x24f7, 0x2b99, 0x35ec, 0x3b98, 0x2049},
  {0x30b3, 0x1bb2, 0x3b76, 0x0c50, 0x1fd3, 0x2bfb, 0x3abc, 0x2425,
   0x2689, 0x1d1b, 0x0198, 0x0dce, 0x021c, 0x0466, 0x0ba5, 0x0aee},
  {0x3fe8, 0x1f11, 0x3763, 0x343d, 0x23ff, 0x1cf1, 0x3109, 0x2c4c,
   0x1707, 0x07e2, 0x2d23, 0x1a63, 0x28b5, 0x235a, 0x0268, 0x342d},
  {0x1df0, 0x3f25, 0x274a, 0x07c9, 0x15bc, 0x0269, 0x321a, 0x17bf,
   0x197e, 0x11a3, 0x2040, 0x2076, 0x3b4a, 0x02d8, 0x05ce, 0x05ad},
  {0x0864, 0x33a9, 0x0f79, 0x0c16, 0x3799, 0x0aa5, 0x0f6a, 0x169f,
   0x2cb8, 0x3275, 0x3029, 0x3ca7, 0x15cf, 0x0747, 0x31ac, 0x0e5d},
  {0x274b, 0x30b8, 0x109d, 0x354c, 0x0dbe, 0x3103, 0x0f44, 0x102a,
   0x077a, 0x1b63, 0x03a5, 0x1a2a, 0x13d4, 0x3f83, 0x2c32, 0x2d8d},
  {0x38eb, 0x3a4c, 0x0bf7, 0x0617, 0x34cd, 0x373e, 0x25c8, 0x2f65,
   0x0b22, 0x0627, 0x31c0, 0x14f1, 0x1e07, 0x0fb2, 0x1084, 0x042d},
  {0x2c6a, 0x1da0, 0x1100, 0x25e4, 0x37af, 0x1f91, 0x1b7b, 0x0586,
   0x0918, 0x0a0c, 0x2879, 0x34db, 0x2b17, 0x159f, 0x2617, 0x1f68},
  {0x00d4, 0x14dd, 0x08ef, 0x1982, 0x32c7, 0x0306, 0x09b0, 0x1b85,
   0x3a28, 0x2740, 0x284c, 0x2849, 0x3d8e, 0x14ed, 0x0803, 0x3216},
  {0x3c72, 0x3401, 0x24c8, 0x017f, 0x0b16, 0x3314, 0x0d51, 0x3a49,
   0x2ad0, 0x0252, 0x248e, 0x287e, 0x0239, 0x015e, 0x31f3, 0x3cce},
  {0x3840, 0x261d, 0x07fc, 0x117c, 0x3e0a, 0x1ba4, 0x1b1b, 0x2466,
   0x2f7c, 0x3fe0, 0x0042, 0x0b24, 0x04bc, 0x09e8, 0x2eb2, 0x19af},
  {0x0d1f, 0x2bea, 0x1138, 0x15a6, 0x1d3d, 0x02ce, 0x18fd, 0x153c,
   0x14cd, 0x0c94, 0x3ef2, 0x020d, 0x0b38, 0x0cc6, 0x3b28, 0x0e06},
  {0x284b, 0x368a, 0x1336, 0x0a57, 0x1273, 0x3c5f, 0x2f4b, 0x08f9,
   0x176b, 0x1935, 0x14c0, 0x2137, 0x18bc, 0x0640, 0x0360, 0x1d0d},
  {0x2600, 0x0d00, 0x0bdc, 0x0e56, 0x0d9c, 0x31bb, 0x0cfe, 0x0578,
   0x0568, 0x052d, 0x1cfe, 0x09cf, 0x1030, 0x0b27, 0x1e85, 0x0c69}
},
{ {0x2f7f, 0x02f3, 0x2585, 0x3e83, 0x3521, 0x39a8, 0x1418, 0x3c2c,
   0x3bfc, 0x2d13, 0x34e6, 0x3b1e, 0x2bd1, 0x12dd, 0x0cf2, 0x2db5},
  {0x116c, 0x2ae4, 0x3271, 0x0c04, 0x10bb, 0x1213, 0x27e2, 0x2609,
   0x2a9b, 0x0dc3, 0x3e7b, 0x2d1b, 0x3495, 0x1344, 0x2828, 0x3d19},
  {0x0e5a, 0x0468, 0x26ba, 0x09c5, 0x1e99, 0x2219, 0x20a3, 0x358e,
   0x22f8, 0x2e45, 0x3ede, 0x2026, 0x3ea0, 0x06a7, 0x179a, 0x3ab6},
  {0x35bd, 0x0171, 0x1ba9, 0x2a1f, 0x3230, 0x2bc4, 0x3ee6, 0x3435,
   0x27ac, 0x1205, 0x33aa, 0x03b1, 0x06ef, 0x24a3, 0x15da, 0x15a7},
    {0x013c, 0x202d, 0x066f, 0x3593, 0x05bb, 0x17ff, 0x0572, 0x10f7,
     0x2681, 0x02dc, 0x2718, 0x0313, 0x2714, 0x064e, 0x2e14, 0x0c3b},
    {0x09e6, 0x0a14, 0x3bae, 0x0493, 0x03bb, 0x11b1, 0x24a2, 0x14b8,
     0x3770, 0x3ca6, 0x22e5, 0x0b54, 0x3c36, 0x0036, 0x143a, 0x0c84},
    {0x1f42, 0x386a, 0x3177, 0x2101, 0x3a7b, 0x1c15, 0x177a, 0x2925,
     0x0cb1, 0x02cb, 0x1924, 0x15b1, 0x267f, 0x232c, 0x159b, 0x156c},
    {0x0f50, 0x3c41, 0x3941, 0x3056, 0x071c, 0x3af6, 0x39be, 0x2b01,
     0x3083, 0x33d4, 0x0891, 0x26b0, 0x36b4, 0x2064, 0x1dee, 0x1235},
    {0x27c8, 0x2d87, 0x00d9, 0x30a8, 0x2790, 0x22e0, 0x3320, 0x2029,
     0x16e2, 0x223a, 0x267d, 0x2d06, 0x022a, 0x252f, 0x360c, 0x2f0f},
    {0x1d6c, 0x1806, 0x29b0, 0x3886, 0x2dd1, 0x2e52, 0x2162, 0x05b4,
     0x2d0f, 0x3bd1, 0x2aba, 0x060e, 0x08bb, 0x2588, 0x26e8, 0x1eed},
    {0x0b4e, 0x16e3, 0x3666, 0x013e, 0x1862, 0x27d2, 0x1408, 0x3a14,
     0x09cc, 0x3a32, 0x2984, 0x355b, 0x2dd7, 0x0180, 0x339a, 0x31a6},
    {0x1b00, 0x18c3, 0x27f6, 0x36be, 0x3a0c, 0x305b, 0x30d8, 0x01e5,
     0x2919, 0x3fdc, 0x0a69, 0x0979, 0x1c1e, 0x1590, 0x0d93, 0x1108},
    {0x0778, 0x2a61, 0x0726, 0x1519, 0x1fb7, 0x1285, 0x14c2, 0x1493,
     0x3c59, 0x2a69, 0x1b88, 0x25cd, 0x3299, 0x166b, 0x2295, 0x1fee},
    {0x153f, 0x1479, 0x1ff7, 0x3344, 0x1312, 0x27c2, 0x0b75, 0x2885,
     0x2f3a, 0x3859, 0x2555, 0x3e03, 0x0a0e, 0x15dc, 0x1ac4, 0x1bad},
    {0x1edd, 0x3c56, 0x352a, 0x189f, 0x25c0, 0x2ea0, 0x243a, 0x0b45,
     0x3065, 0x0d32, 0x210a, 0x38b6, 0x1104, 0x217a, 0x36b2, 0x2a0c},
    {0x0c53, 0x0db0, 0x253c, 0x1d15, 0x009c, 0x07cc, 0x27ed, 0x2a75,
     0x3868, 0x0cef, 0x378c, 0x2af7, 0x09ae, 0x3e27, 0x3ded, 0x1a0d}
  },
  { {0x01f6, 0x37d9, 0x2f1a, 0x0592, 0x1fcc, 0x1b54, 0x07cd, 0x145d,
 0x3989, 0x3e48, 0x0922, 0x31c1, 0x31d3, 0x3d28, 0x347e, 0x3531},
{0x31e9, 0x1796, 0x0295, 0x39a2, 0x04a5, 0x360a, 0x1cad, 0x17aa,
 0x0bf0, 0x209d, 0x2cc5, 0x2d38, 0x00ed, 0x3701, 0x04a8, 0x23d0},
{0x36af, 0x09c8, 0x2918, 0x32cb, 0x3f09, 0x321e, 0x30ea, 0x251a,
 0x27e1, 0x251e, 0x2bb0, 0x31fe, 0x16ab, 0x1507, 0x048a, 0x37bc},
{0x395d, 0x0ddd, 0x2fea, 0x18a1, 0x3508, 0x31cc, 0x132b, 0x15e7,
 0x36bc, 0x1f62, 0x14ca, 0x22e8, 0x08e7, 0x0d25, 0x34a1, 0x0452},
{0x0943, 0x0dc1, 0x23bf, 0x2dd4, 0x1c8c, 0x33b7, 0x0985, 0x328f,
 0x1458, 0x1ab0, 0x303e, 0x1bfd, 0x19b1, 0x09fc, 0x21b4, 0x0a0d},
{0x08e4, 0x0649, 0x2967, 0x29f0, 0x25e9, 0x0edc, 0x0fb9, 0x286b,
 0x3388, 0x29a6, 0x238d, 0x39ee, 0x2b41, 0x0e98, 0x3837, 0x288c},
{0x0b3e, 0x3cfb, 0x296b, 0x1416, 0x171a, 0x27e0, 0x0845, 0x18a3,
 0x1267, 0x20ab, 0x26f2, 0x3f03, 0x218c, 0x3ac1, 0x2558, 0x1903},
{0x3f00, 0x0455, 0x2d5c, 0x238b, 0x3d30, 0x05fc, 0x10a3, 0x31b0,
 0x2ba5, 0x02f0, 0x01a2, 0x2d90, 0x1637, 0x1989, 0x20a2, 0x1183},
{0x0bc9, 0x2080, 0x2328, 0x3b44, 0x1377, 0x1278, 0x250b, 0x32a3,
 0x34cf, 0x10a4, 0x1880, 0x0f1a, 0x3936, 0x31f6, 0x24ca, 0x32bd},
{0x0760, 0x079e, 0x01c5, 0x0db8, 0x387c, 0x3296, 0x113b, 0x2f0d,
 0x22dd, 0x2a3f, 0x230c, 0x2c5e, 0x2814, 0x05b1, 0x35f4, 0x00f5},
{0x0683, 0x3e3f, 0x0e7a, 0x20ea, 0x24ea, 0x0138, 0x1065, 0x1f2c,
 0x07d0, 0x1048, 0x3223, 0x3742, 0x19c3, 0x13be, 0x26cd, 0x1938},
{0x35f7, 0x0d20, 0x1af6, 0x000c, 0x0230, 0x2fa6, 0x37b0, 0x2795,
 0x242f, 0x2332, 0x3807, 0x0d24, 0x3f5e, 0x3467, 0x025a, 0x22d3},
{0x260d, 0x2c33, 0x3b3a, 0x35f5, 0x0e4b, 0x26ed, 0x2bc2, 0x0a19,
 0x20a1, 0x1bcb, 0x1625, 0x2755, 0x2ca9, 0x00e3, 0x2dfc, 0x33b4},
{0x0e96, 0x0f6c, 0x3700, 0x2c78, 0x353c, 0x032f, 0x1122, 0x0305,
   0x3b2c, 0x3cb5, 0x3642, 0x0818, 0x1af7, 0x118f, 0x2221, 0x331c},
  {0x0dba, 0x20da, 0x053c, 0x0679, 0x394e, 0x3444, 0x286a, 0x14d5,
   0x1968, 0x39f3, 0x158c, 0x25d6, 0x1bb3, 0x0c76, 0x029e, 0x0b03},
  {0x1583, 0x3cd7, 0x1530, 0x0cf4, 0x3e36, 0x0c52, 0x120c, 0x0d4b,
   0x3931, 0x2d2d, 0x235b, 0x1541, 0x2334, 0x13b9, 0x0c1c, 0x215c}
},
{ {0x1adb, 0x2f0e, 0x01f4, 0x173a, 0x1157, 0x1456, 0x3728, 0x3391,
   0x1a16, 0x3924, 0x267e, 0x0697, 0x0264, 0x0e48, 0x061c, 0x3f07},
  {0x2fd1, 0x3aa5, 0x1cc0, 0x0d55, 0x2e70, 0x24c2, 0x29e3, 0x30cc,
   0x055a, 0x131d, 0x1656, 0x30a4, 0x2194, 0x105f, 0x375e, 0x2bd2},
  {0x34d7, 0x2898, 0x08c2, 0x389a, 0x2f40, 0x2fbc, 0x1e44, 0x1527,
   0x0b36, 0x311e, 0x0703, 0x32d1, 0x2ca2, 0x0eb7, 0x224a, 0x2a9d},
  {0x19ef, 0x3436, 0x1887, 0x1769, 0x014f, 0x34a9, 0x15b5, 0x329b,
   0x153a, 0x306e, 0x1bf8, 0x0158, 0x1ff4, 0x378d, 0x28d5, 0x2911},
  {0x17c0, 0x17b6, 0x2e95, 0x2579, 0x37f7, 0x02fb, 0x2e20, 0x29a0,
   0x2283, 0x2761, 0x03c6, 0x2018, 0x3e2c, 0x13fa, 0x0246, 0x0b34},
  {0x36f0, 0x0eaa, 0x1bbc, 0x3e5b, 0x1e9c, 0x3a6d, 0x3708, 0x36a5,
   0x1ff8, 0x0652, 0x3f57, 0x12c1, 0x3336, 0x2c47, 0x14c6, 0x0fd5},
  {0x3403, 0x19e7, 0x0aed, 0x1164, 0x1a3a, 0x397f, 0x00f3, 0x3506,
   0x1596, 0x0d01, 0x219e, 0x06cd, 0x389f, 0x3d9c, 0x37b5, 0x275d},
  {0x3550, 0x0920, 0x2a59, 0x2772, 0x3e53, 0x3bbb, 0x3c62, 0x0121,
   0x2fb2, 0x32db, 0x3f3d, 0x3d64, 0x2847, 0x02ea, 0x116b, 0x13bd},
  {0x11e3, 0x119d, 0x32ae, 0x3e08, 0x3dfd, 0x1a3f, 0x20af, 0x3625,
   0x2602, 0x2125, 0x1168, 0x2df1, 0x3be4, 0x177e, 0x10ca, 0x0577},
  {0x3773, 0x2d8a, 0x01c8, 0x1d89, 0x1ddf, 0x1ce2, 0x0762, 0x00dd,
   0x17fc, 0x0122, 0x2730, 0x3fab, 0x1b49, 0x1d92, 0x1734, 0x0d72},
  {0x0909, 0x37e2, 0x3b96, 0x3528, 0x2157, 0x20f3, 0x205f, 0x1855,
   0x3b82, 0x2305, 0x1e5e, 0x1be4, 0x02ad, 0x38b7, 0x110a, 0x18fe},
  {0x2d41, 0x07b3, 0x0df6, 0x1b74, 0x22b8, 0x31ca, 0x060f, 0x2d9e,
   0x2dd9, 0x2f9b, 0x3016, 0x0b81, 0x24d3, 0x2092, 0x085b, 0x1b8c},
  {0x2e72, 0x124b, 0x2c26, 0x2b29, 0x1ac0, 0x1305, 0x24ee, 0x2f35,
   0x051e, 0x1433, 0x2785, 0x16e1, 0x2da1, 0x20de, 0x31c5, 0x1578},
  {0x35e0, 0x3289, 0x3f89, 0x07d3, 0x3260, 0x362f, 0x1018, 0x3196,
   0x094d, 0x3fb0, 0x3099, 0x1131, 0x1bd7, 0x34b1, 0x2164, 0x1b6a},
  {0x0ab2, 0x1d93, 0x291f, 0x3979, 0x2637, 0x325d, 0x3df5, 0x0715,
   0x081e, 0x36a6, 0x171f, 0x350a, 0x0607, 0x0429, 0x19e5, 0x336a},
  {0x12fd, 0x3f39, 0x08c1, 0x1a11, 0x3e13, 0x348f, 0x1aec, 0x06f5,
   0x06c3, 0x00a2, 0x398f, 0x0841, 0x1634, 0x080e, 0x3b81, 0x1795}
},
{ {0x2093, 0x1bf3, 0x238e, 0x0454, 0x26a6, 0x3a12, 0x13c5, 0x3097,
   0x0718, 0x39d0, 0x0513, 0x33ee, 0x28b3, 0x0236, 0x3aff, 0x2124},
  {0x2b94, 0x1cb8, 0x1cd4, 0x1e0f, 0x1069, 0x209f, 0x00e2, 0x18eb,
   0x2e7a, 0x1bff, 0x0fb5, 0x2749, 0x1ce5, 0x2581, 0x1799, 0x1e75},
  {0x0c60, 0x3579, 0x325a, 0x3639, 0x28b0, 0x1145, 0x379f, 0x2e05,
   0x1d3e, 0x302f, 0x33d7, 0x25d1, 0x11dc, 0x01b7, 0x15db, 0x2fa5},
  {0x28c7, 0x3983, 0x3126, 0x16f0, 0x31ad, 0x23e1, 0x3f42, 0x1df2,
   0x1a70, 0x2636, 0x268e, 0x3d99, 0x12f4, 0x15d1, 0x2eda, 0x22ba},
  {0x315a, 0x3137, 0x1b3c, 0x0687, 0x0d85, 0x1ae8, 0x27d7, 0x134f,
   0x1405, 0x3825, 0x3932, 0x1c3e, 0x3224, 0x1cff, 0x09c6, 0x2c3f},
  {0x15ad, 0x323d, 0x0111, 0x00c5, 0x0fad, 0x1c87, 0x3d41, 0x2699,
   0x2bcd, 0x29cb, 0x12d7, 0x30ff, 0x1154, 0x10c0, 0x0c2a, 0x2289},
  {0x0343, 0x0520, 0x3817, 0x2d80, 0x00ec, 0x2400, 0x1cd2, 0x3913,
   0x0824, 0x06db, 0x2dfb, 0x0d3d, 0x0549, 0x34d6, 0x0c6a, 0x3f19},
  {0x29af, 0x3908, 0x37df, 0x2844, 0x0e62, 0x12a4, 0x0136, 0x09f5,
   0x19c1, 0x078d, 0x186d, 0x22d1, 0x25b3, 0x284d, 0x1540, 0x093a},
  {0x0780, 0x1103, 0x27db, 0x1c86, 0x367f, 0x11fb, 0x2528, 0x2628,
   0x04c0, 0x3ee9, 0x0785, 0x3bde, 0x1473, 0x3d32, 0x3536, 0x0ecc},
  {0x2975, 0x3ff5, 0x3ead, 0x3085, 0x1b0d, 0x3861, 0x2f1d, 0x26df,
   0x2b75, 0x325e, 0x38dd, 0x2816, 0x0e14, 0x03c5, 0x0006, 0x37ed},
  {0x0303, 0x2747, 0x1fa8, 0x107a, 0x1ced, 0x0b9f, 0x0ee7, 0x098d,
   0x00db, 0x17cb, 0x1eb0, 0x3c84, 0x34ee, 0x210d, 0x03d5, 0x171d},
  {0x04e5, 0x1b3b, 0x1512, 0x1bf5, 0x3d63, 0x2557, 0x3db9, 0x12e2,
   0x2ba2, 0x229b, 0x3592, 0x3e81, 0x1f8d, 0x0a33, 0x02da, 0x1eb4},
  {0x3c55, 0x0b8a, 0x0c99, 0x291b, 0x2959, 0x1238, 0x27c0, 0x2beb,
   0x25cb, 0x01cb, 0x0ddb, 0x3f9d, 0x2adc, 0x0dda, 0x058d, 0x2e68},
  {0x2370, 0x1ff5, 0x3916, 0x2a3e, 0x2f3f, 0x1621, 0x2d74, 0x24b9,
   0x184d, 0x3347, 0x125a, 0x14cc, 0x3d33, 0x107d, 0x14a8, 0x00a0},
  {0x0fdb, 0x1093, 0x3a76, 0x0137, 0x3a6c, 0x2912, 0x2ab3, 0x3ba5,
   0x2a96, 0x3410, 0x230b, 0x3236, 0x2ea7, 0x0929, 0x3e7c, 0x13dd},
  {0x1554, 0x2edc, 0x2d9f, 0x07ba, 0x3877, 0x3b7e, 0x3df0, 0x0e60,
   0x133d, 0x3c81, 0x1544, 0x355a, 0x27ba, 0x1727, 0x1d32, 0x06cf}
},
{ {0x11d8, 0x0ef3, 0x3f17, 0x2ad9, 0x3729, 0x09b6, 0x266a, 0x2a5f,
   0x3ab7, 0x15f9, 0x1ec4, 0x320e, 0x25c7, 0x30bf, 0x0cbb, 0x359d},
  {0x2756, 0x0d4f, 0x3c16, 0x2000, 0x0ba0, 0x2ac6, 0x1df5, 0x16e8,
   0x04c8, 0x07e1, 0x0673, 0x0a30, 0x0794, 0x3871, 0x2357, 0x26f3},
  {0x07c7, 0x345f, 0x03da, 0x1a90, 0x23d4, 0x09c9, 0x149d, 0x364c,
   0x0ac4, 0x23af, 0x3aee, 0x3774, 0x1452, 0x154f, 0x39e8, 0x2d58},
{0x046d, 0x1697, 0x0aec, 0x35df, 0x2732, 0x2300, 0x31be, 0x3dc9,
 0x27e7, 0x11cc, 0x2a0e, 0x0474, 0x2007, 0x2298, 0x3ac5, 0x3d87},
{0x0165, 0x076d, 0x0dd6, 0x3cbc, 0x1683, 0x02bc, 0x00c8, 0x2f9e,
 0x35c4, 0x2072, 0x2781, 0x1731, 0x0f24, 0x2294, 0x1002, 0x0efd},
{0x0de7, 0x12fb, 0x1361, 0x1c51, 0x31e3, 0x1eff, 0x0c6c, 0x3bb2,
 0x1609, 0x0cfd, 0x224d, 0x2ae6, 0x14ab, 0x32c4, 0x21f2, 0x0b83},
{0x0ab3, 0x210b, 0x012d, 0x25f9, 0x3464, 0x10bd, 0x2dae, 0x1c84,
 0x2ba9, 0x3b45, 0x1e0a, 0x3a3e, 0x2702, 0x142c, 0x10a5, 0x1e25},
{0x1e24, 0x0c75, 0x1cc8, 0x3e28, 0x0590, 0x0d10, 0x3c9b, 0x064c,
 0x1db3, 0x24c5, 0x19d2, 0x13fd, 0x02b4, 0x1043, 0x3285, 0x3694},
{0x33ec, 0x3c86, 0x21fb, 0x016e, 0x22ca, 0x003f, 0x0e32, 0x2b61,
 0x14bb, 0x31b3, 0x36b8, 0x13f1, 0x0767, 0x30c5, 0x0b7f, 0x32d7},
{0x34f1, 0x0183, 0x2e4a, 0x17d0, 0x3652, 0x29df, 0x1705, 0x2dd0,
 0x0948, 0x0a10, 0x2737, 0x078f, 0x10cc, 0x0f07, 0x0034, 0x21da},
{0x0460, 0x0999, 0x285c, 0x2b55, 0x08bd, 0x03bd, 0x32cf, 0x2c93,
 0x1b9f, 0x3165, 0x045e, 0x2b2c, 0x158e, 0x3751, 0x26a9, 0x3479},
{0x25c6, 0x33f8, 0x1668, 0x2176, 0x081f, 0x0abf, 0x3974, 0x0bbd,
 0x07a8, 0x248d, 0x2c35, 0x16ec, 0x1d0c, 0x225c, 0x396d, 0x0973},
{0x3740, 0x01f0, 0x0431, 0x0630, 0x1d7f, 0x388a, 0x0af2, 0x171c,
 0x29cc, 0x2644, 0x3135, 0x0756, 0x1ea4, 0x074c, 0x27fb, 0x021f},
{0x174d, 0x0c7d, 0x3256, 0x1056, 0x0ca9, 0x2848, 0x0216, 0x3b77,
 0x24d5, 0x0dc9, 0x347f, 0x38c6, 0x066a, 0x1bdd, 0x2e39, 0x2670},
{0x35a0, 0x37a3, 0x0993, 0x0abe, 0x15a0, 0x052a, 0x1b02, 0x1745,
 0x399a, 0x2fc6, 0x011e, 0x155d, 0x33e8, 0x2241, 0x2ec0, 0x06dd},
{0x0385, 0x2171, 0x2177, 0x1fe5, 0x1608, 0x2de8, 0x2882, 0x19ca,
 0x3abb, 0x1899, 0x1562, 0x0529, 0x3f79, 0x3227, 0x1d95, 0x0fa7}
```

```
},
{ {0x3400, 0x01f3, 0x2b04, 0x2719, 0x1d38, 0x0851, 0x3a82, 0x0fd9,
   0x190d, 0x2604, 0x118b, 0x1c5e, 0x35be, 0x39ff, 0x09d9, 0x3a27},
  {0x25a0, 0x09a6, 0x3d56, 0x3606, 0x2978, 0x001b, 0x2642, 0x172e,
   0x37b2, 0x04aa, 0x0a9e, 0x187b, 0x1f54, 0x1338, 0x0cdd, 0x3008},
  {0x0831, 0x0053, 0x3ec8, 0x1bb6, 0x344e, 0x1de6, 0x3809, 0x319e,
   0x15de, 0x1042, 0x2197, 0x38ab, 0x19c4, 0x3f5b, 0x35aa, 0x2161},
  {0x057c, 0x097e, 0x23ee, 0x2682, 0x06ad, 0x1cc9, 0x11cb, 0x2fe0,
   0x3767, 0x2693, 0x187a, 0x17f4, 0x3600, 0x06e7, 0x120f, 0x01de},
  {0x1de3, 0x2991, 0x215f, 0x1a99, 0x1ce1, 0x3322, 0x2dc5, 0x3992,
   0x2a95, 0x36fb, 0x1f92, 0x1503, 0x0a03, 0x28cd, 0x07d2, 0x25b7},
  {0x3c70, 0x078e, 0x2499, 0x0952, 0x03d8, 0x2025, 0x25c9, 0x275a,
   0x1cf5, 0x0d7b, 0x0a70, 0x0c10, 0x2459, 0x22e6, 0x1f97, 0x32a5},
  {0x1cbc, 0x3d13, 0x3d7a, 0x1067, 0x2a14, 0x1ab7, 0x0779, 0x10bc,
   0x0379, 0x2a76, 0x3ace, 0x1ace, 0x2ff0, 0x1688, 0x3335, 0x3d73},
  {0x2850, 0x031c, 0x2a36, 0x014d, 0x1f8e, 0x210f, 0x107e, 0x3da7,
   0x0a94, 0x2881, 0x2cd3, 0x1603, 0x010d, 0x2eb0, 0x3edb, 0x3004},
  {0x1b1a, 0x3b73, 0x09e5, 0x267c, 0x10df, 0x1257, 0x3ee0, 0x3052,
   0x3d5d, 0x05ac, 0x2ae1, 0x0dff, 0x3646, 0x0f6f, 0x3c13, 0x3018},
  {0x37db, 0x2bee, 0x18b7, 0x30fc, 0x2bae, 0x203f, 0x2806, 0x15cc,
   0x1bb4, 0x14d1, 0x06ba, 0x38a5, 0x055e, 0x0207, 0x16d5, 0x173e},
  {0x0f3e, 0x2448, 0x1b5d, 0x15ab, 0x35c6, 0x0853, 0x2116, 0x1b8a,
   0x1fe2, 0x3c23, 0x2e09, 0x3496, 0x1a67, 0x1b62, 0x159e, 0x2bb7},
  {0x2f13, 0x1d5b, 0x2fee, 0x3b35, 0x349d, 0x023f, 0x145c, 0x32b3,
   0x1ed7, 0x23d7, 0x3a1d, 0x0a37, 0x2de7, 0x3785, 0x368e, 0x3f45},
  {0x3f0d, 0x31de, 0x07c1, 0x1419, 0x0dd3, 0x0a79, 0x150a, 0x23bb,
   0x19d8, 0x2f5a, 0x0cb9, 0x08c7, 0x3214, 0x0e7d, 0x082f, 0x2a4f},
  {0x1413, 0x1763, 0x11f1, 0x089b, 0x0407, 0x1d57, 0x3578, 0x168b,
   0x3643, 0x20fb, 0x1f23, 0x139c, 0x17f7, 0x3ec7, 0x0b59, 0x07e7},
  {0x029b, 0x04ea, 0x13a1, 0x1b82, 0x35fe, 0x0744, 0x1ebf, 0x36b1,
   0x33bb, 0x0766, 0x0d89, 0x0211, 0x36f5, 0x0e6d, 0x184a, 0x0d09},
  {0x0b04, 0x3a77, 0x2c17, 0x3739, 0x0aaf, 0x114c, 0x00e9, 0x25b4,
   0x2c05, 0x2bb8, 0x286e, 0x2141, 0x2f34, 0x29d2, 0x083b, 0x34b3}
},
{ {0x243c, 0x31df, 0x126b, 0x0412, 0x1f58, 0x12ee, 0x30cf, 0x2182,
   0x0965, 0x2ae7, 0x3387, 0x3310, 0x2b64, 0x2801, 0x20b3, 0x383d},
  {0x33e2, 0x32f1, 0x0a21, 0x13e0, 0x3ed3, 0x126c, 0x0516, 0x39b5,
   0x1c71, 0x38bf, 0x3905, 0x330c, 0x2d94, 0x3139, 0x2995, 0x327b},
  {0x01be, 0x2db9, 0x05c8, 0x0f39, 0x3792, 0x2a8f, 0x25bd, 0x1ee2,
   0x3d62, 0x329a, 0x26dc, 0x0533, 0x0cc4, 0x18a5, 0x3a9a, 0x3c93},
  {0x0a27, 0x2058, 0x3437, 0x207c, 0x0556, 0x31d0, 0x154b, 0x33d1,
   0x0160, 0x1e46, 0x100e, 0x2e0b, 0x29a1, 0x1e7c, 0x3a8c, 0x15cd},
  {0x3d23, 0x0cc7, 0x3fa6, 0x0d99, 0x23d8, 0x04c4, 0x2512, 0x0069,
   0x2059, 0x05d1, 0x2942, 0x36ea, 0x3fcf, 0x24f9, 0x24e1, 0x1cf9},
  {0x17a3, 0x281c, 0x380b, 0x15fc, 0x38f0, 0x0acb, 0x16a2, 0x3bc5,
   0x2a81, 0x2430, 0x3f35, 0x275f, 0x1f79, 0x26ae, 0x3c63, 0x142d},
  {0x2f28, 0x25d3, 0x1f96, 0x3595, 0x0781, 0x2c41, 0x2301, 0x179f,
   0x11b8, 0x21fc, 0x0c4b, 0x3994, 0x39ac, 0x2cfd, 0x1f5a, 0x1eeb},
  {0x3db5, 0x0f43, 0x1701, 0x393f, 0x3266, 0x08e1, 0x0c26, 0x3a02,
   0x07a1, 0x38af, 0x0cce, 0x2db2, 0x15a5, 0x0944, 0x35a5, 0x3049},
  {0x1368, 0x2928, 0x3123, 0x115f, 0x1782, 0x29eb, 0x0b57, 0x39e9,
   0x14a6, 0x049c, 0x2bfe, 0x3fca, 0x24b6, 0x0302, 0x2312, 0x2bb9},
  {0x1bca, 0x252b, 0x3558, 0x2989, 0x288d, 0x3dd5, 0x02b7, 0x3037,
   0x3da4, 0x161d, 0x2c24, 0x0159, 0x2889, 0x3e17, 0x02a1, 0x051d},
  {0x34c9, 0x1c9e, 0x1012, 0x1b04, 0x16ac, 0x2d2e, 0x31ea, 0x0085,
   0x3f40, 0x2337, 0x0f62, 0x1200, 0x192b, 0x1765, 0x1528, 0x299b},
  {0x3cee, 0x27ff, 0x0b2d, 0x3ebd, 0x35dc, 0x3af3, 0x11c2, 0x030c,
   0x155b, 0x0f6d, 0x23fd, 0x380e, 0x2486, 0x0052, 0x137b, 0x1196},
  {0x3631, 0x3bbf, 0x2b7d, 0x07d5, 0x0a0b, 0x2d3b, 0x0fcd, 0x37c4,
   0x396f, 0x1f2a, 0x16b3, 0x111c, 0x3ab4, 0x3c5c, 0x0751, 0x1e80},
  {0x04b6, 0x1dab, 0x3fbb, 0x020e, 0x341b, 0x1c93, 0x399b, 0x1e13,
   0x2c08, 0x3d40, 0x07ee, 0x0708, 0x286c, 0x2c43, 0x06bf, 0x14ad},
  {0x1b0e, 0x2993, 0x00a7, 0x093c, 0x03dc, 0x18e5, 0x2f52, 0x3d45,
   0x1248, 0x1e23, 0x22b1, 0x3b27, 0x2929, 0x2454, 0x2ca5, 0x078c},
  {0x32b8, 0x1dc9, 0x06c1, 0x06e2, 0x3b08, 0x008f, 0x392c, 0x2ef7,
   0x026c, 0x09cb, 0x1d4d, 0x07f5, 0x3ced, 0x3027, 0x15ec, 0x0c0d}
},
{ {0x1558, 0x1cf2, 0x126e, 0x28c3, 0x3b56, 0x3bf1, 0x22f2, 0x1e84,
   0x0998, 0x35ac, 0x3c5b, 0x0612, 0x3aa9, 0x222e, 0x19a6, 0x0733},
  {0x3ba0, 0x0723, 0x0496, 0x1917, 0x221f, 0x2caa, 0x2d2a, 0x293c,
   0x24a0, 0x1e89, 0x2b77, 0x1b97, 0x2094, 0x2317, 0x3909, 0x1c92},
  {0x073d, 0x2ed1, 0x1694, 0x237f, 0x25d2, 0x38ec, 0x2c12, 0x29fe,
   0x0fcc, 0x03a4, 0x14d0, 0x1207, 0x3d22, 0x3987, 0x0079, 0x0686},
  {0x1b9c, 0x2b3c, 0x214e, 0x2169, 0x2dd5, 0x2ac4, 0x2abb, 0x1e41,
   0x123f, 0x3d82, 0x390b, 0x004d, 0x1dc8, 0x1fea, 0x1bc5, 0x1ffd},
  {0x37c8, 0x2ced, 0x2414, 0x19dd, 0x16d7, 0x219b, 0x1c8f, 0x0cf3,
   0x0597, 0x18a0, 0x14e7, 0x0ad4, 0x2b02, 0x0bc7, 0x3b68, 0x3041},
  {0x044e, 0x17ca, 0x18e0, 0x0128, 0x2a1c, 0x3baf, 0x1c8d, 0x2b9c,
   0x2cb4, 0x337b, 0x0e0f, 0x3d06, 0x1d54, 0x068f, 0x3a3c, 0x2056},
  {0x0ab7, 0x2970, 0x1dc3, 0x16c6, 0x1133, 0x04e3, 0x3194, 0x2449,
   0x3a7f, 0x0435, 0x047d, 0x303a, 0x31f7, 0x35b0, 0x23b5, 0x0a72},
  {0x0628, 0x3e7e, 0x3f46, 0x0789, 0x1a8e, 0x338a, 0x1d90, 0x24d1,
   0x1e03, 0x1c44, 0x3b95, 0x3486, 0x1f2e, 0x2538, 0x379a, 0x078a},
  {0x2250, 0x28dd, 0x0f68, 0x03f5, 0x2484, 0x285b, 0x27f2, 0x3525,
   0x1ffc, 0x2aa2, 0x182f, 0x23b2, 0x0c3f, 0x1873, 0x16ca, 0x35da},
  {0x143e, 0x0c95, 0x0b61, 0x3b29, 0x2a7c, 0x3fa9, 0x191d, 0x3946,
   0x32f5, 0x19f5, 0x21e2, 0x1678, 0x1db5, 0x16df, 0x323c, 0x03ed},
  {0x03cf, 0x06a9, 0x1662, 0x0b23, 0x285f, 0x04ab, 0x3f87, 0x0f31,
   0x1fb4, 0x0f6b, 0x379e, 0x2b51, 0x3944, 0x06b3, 0x3b47, 0x3e66},
  {0x3df9, 0x38ed, 0x0cf6, 0x29ba, 0x0736, 0x0027, 0x1a7a, 0x3bda,
   0x06fa, 0x35ed, 0x1877, 0x0c7e, 0x3f13, 0x32d3, 0x0fa9, 0x2b03},
  {0x2e86, 0x1e33, 0x273d, 0x2aca, 0x20a0, 0x3d14, 0x0589, 0x2ec5,
   0x3132, 0x3dfa, 0x28e3, 0x1f30, 0x2339, 0x3660, 0x2c1c, 0x2ba1},
  {0x0c64, 0x3f34, 0x2f8c, 0x1191, 0x1d8e, 0x390d, 0x0a0a, 0x36df,
   0x24bd, 0x05c3, 0x270d, 0x35eb, 0x353a, 0x2b0a, 0x2c07, 0x387d},
  {0x0739, 0x0298, 0x10ee, 0x2067, 0x0477, 0x145f, 0x047f, 0x35ce,
   0x34c0, 0x2316, 0x0880, 0x09d1, 0x0872, 0x0fd7, 0x0daa, 0x3d95},
  {0x2ee3, 0x1762, 0x1d5a, 0x17a9, 0x2073, 0x3dc6, 0x2a80, 0x1521,
   0x2c82, 0x2497, 0x2abe, 0x1871, 0x35e4, 0x3399, 0x07de, 0x2230}
},
{ {0x1349, 0x36ee, 0x28e6, 0x05af, 0x0a00, 0x1f2b, 0x13a4, 0x1fff,
   0x02ec, 0x3d2e, 0x2191, 0x2aee, 0x16b1, 0x00cc, 0x13b2, 0x1893},
  {0x3590, 0x2868, 0x1240, 0x3465, 0x1f89, 0x3c0a, 0x3dbf, 0x0dd5,
   0x2156, 0x09dc, 0x3a05, 0x045f, 0x2db8, 0x0806, 0x19b0, 0x3d6d},
{0x086c, 0x363d, 0x18bb, 0x1296, 0x2bba, 0x2c55, 0x192d, 0x1b5e,
 0x301b, 0x0c08, 0x1375, 0x322d, 0x1712, 0x00b7, 0x3574, 0x1c9d},
{0x3d9b, 0x3883, 0x1912, 0x1469, 0x1dc6, 0x3b0c, 0x1aaa, 0x3022,
 0x0200, 0x0c8f, 0x360e, 0x2939, 0x30d2, 0x2c92, 0x1c63, 0x3ef1},
{0x3e23, 0x24e5, 0x3bd9, 0x287f, 0x256a, 0x0816, 0x1315, 0x0b21,
 0x1337, 0x39b0, 0x1f65, 0x3d5b, 0x13d2, 0x27ee, 0x111a, 0x2509},
{0x1a43, 0x2cba, 0x3283, 0x0118, 0x0c0c, 0x3424, 0x2e37, 0x38d1,
 0x3dad, 0x0bcb, 0x0f02, 0x341f, 0x31f9, 0x090f, 0x14ea, 0x0fa5},
{0x0820, 0x3fc4, 0x2406, 0x1404, 0x2c61, 0x2be4, 0x339b, 0x0629,
 0x3ffa, 0x2cb9, 0x1d12, 0x08a0, 0x2947, 0x36eb, 0x35c8, 0x1653},
{0x0297, 0x2ccc, 0x1e00, 0x1e98, 0x3917, 0x0991, 0x09f2, 0x3ad8,
 0x33ed, 0x24db, 0x3fbd, 0x3986, 0x1136, 0x32ca, 0x1604, 0x06d6},
{0x3e80, 0x31da, 0x00ea, 0x0c3a, 0x3e4c, 0x0bc3, 0x29a2, 0x0b55,
 0x3618, 0x0368, 0x193d, 0x264b, 0x024d, 0x0c00, 0x09ca, 0x0adb},
{0x3393, 0x12ae, 0x2ed9, 0x25ca, 0x3d74, 0x0b77, 0x1025, 0x3c8e,
 0x3ca8, 0x172a, 0x129f, 0x0996, 0x27ea, 0x1b07, 0x13e3, 0x150c},
{0x03ec, 0x3724, 0x18b3, 0x115a, 0x14c8, 0x192e, 0x1a8f, 0x13ac,
 0x27a5, 0x1718, 0x2ba0, 0x3bdd, 0x2c44, 0x2a6a, 0x3f1c, 0x287c},
{0x3475, 0x33a8, 0x3a1c, 0x15c1, 0x2d56, 0x0d03, 0x02ba, 0x2f33,
 0x13a8, 0x36ad, 0x38c2, 0x384a, 0x2d6e, 0x265d, 0x3195, 0x1759},
{0x14d2, 0x033f, 0x00c7, 0x0608, 0x2c9e, 0x2597, 0x005a, 0x1565,
 0x11ab, 0x04ca, 0x22ce, 0x3478, 0x2aa4, 0x3897, 0x0dab, 0x14b2},
{0x08cb, 0x30e8, 0x0bf3, 0x3e1b, 0x3e35, 0x282f, 0x2516, 0x24a4,
 0x3e98, 0x1ffb, 0x24ab, 0x0954, 0x3a08, 0x17c7, 0x3fdf, 0x3cfa},
{0x2287, 0x0f93, 0x38cd, 0x1e19, 0x2306, 0x2ebf, 0x110d, 0x1e8d,
 0x28c4, 0x0f7a, 0x2b05, 0x325f, 0x05b5, 0x21dc, 0x188c, 0x387e},
    {0x23b6, 0x2ec3, 0x01ae, 0x11e4, 0x2f37, 0x3c98, 0x355f, 0x0275,
     0x34ce, 0x1488, 0x3237, 0x306d, 0x343e, 0x3b09, 0x2214, 0x320d}
  }
};
```

# VITA

Richard Lloyd Churchill

Candidate for the Degree of

Master of Science

Thesis: RICHARD LLOYD CHURCHILL

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2011.

Completed the requirements for the Bachelor of Science in Chemistry and Philosophy at Oklahoma State University, Stillwater, Oklahoma in 1980.

Experience:

Systems Engineer, Telex Corp., Tulsa, OK 1985-1988

Systems Engineer / Systems Architect, Compaq Computer Corp., Houston, TX 1988-2001

Programmer / Software Architect / Consultant, Houston, TX 2001-2004

Programmer / Software Architect / Consultant, Stillwater, OK 2004-2011

Name: Richard Lloyd Churchill Date of Degree: December, 2011

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: MODIFIED MCLAREN-MARSAGLIA PSEUDO-RANDOM NUMBER GENERATOR AND STOCHASTIC KEY AGREEMENT

Pages in Study: 416 Candidate for the Degree of Master of Science

Major Field: Computer Science

Findings and Conclusions: A discussion of problems in cryptographic applications, with a brief survey of pseudo-random number generators (PRNG) used as synchronous stream ciphers, leads to a discussion of the McClaren-Marsaglia shuffling PRNG, and some means of altering its structure to both provide a more secure PRNG and to provide effective means by which to inject aperiodicity into a modified form of McClaren-Marsaglia. A discussion of two closely related protocols using this modified form of McClaren-Marsaglia as means by which correspondents may agree upon a set of random bits in a manner suitable for use in cryptographic applications is then presented, with implementation in the C programming language of the second protocol. Analysis of the protocols concludes that a reasonable expectation of confidentiality and cryptographic strength in the agreed bit-sequence is obtained.

ADVISER'S APPROVAL: Dr. H. K. Dai