UNIVERSITY OF TARTU
Faculty of Mathematics and Computer Science
Institute of Computer Science
Chair of Cryptography

Emilia Käsper

Complexity Analysis of Hardware-Assisted Attacks on A5/1

Master's Thesis

Supervisor: Prof. Helger Lipmaa

TARTU 2006

Acknowledgements

This Master's thesis was written as part of the GO-SEC project at Helsinki University of Technology. The project was financed by the National Technology Agency of Finland and several Finnish telecommunication companies. I thank the partners of GO-SEC, the Laboratory for Theoretical Computer Science and the Signal Processing Laboratory at Helsinki University of Technology for supporting this research and providing me with an excellent working environment for the duration of the project.

I am grateful to my supervisor Helger Lipmaa for his numerous suggestions and remarks, and especially for his contributions to optimising the software implementation. I would also like to thank Kaisa Nyberg, Jan Villemson and Sven Laur for their helpful comments, and Kimmo Järvinen for giving me insight into the hardware world.

Contents

Chapter 1

Introduction

The A5/1 stream cipher is an encryption algorithm used in the GSM standard of mobile communications. The standard originally incorporated two ciphers: A5/1 was used mainly in Europe, whereas A5/2, an intentionally weaker version of the A5 algorithm was created for export. Recent cryptanalytic attacks against A5/2 have caused 3GPP (3rd Generation Partnership Project, [3GP]), who is currently responsible for maintaining and developing GSM technical specifications, to withdraw A5/2 from the standard. According to [GSM04], there were over 1.7 billion GSM subscribers as of March 2006. With 3G networks still under development and A5/2 in removal, a vast majority of those users will rely on A5/1 to protect the confidentiality of their voice communications during the coming years.

The history of GSM dates back to the 1980s and the A5/1 encryption algorithm was developed already at the end of the same decade. The first initiative to create a pan-European mobile communications network was met in 1982, when the Groupe Spéciale Mobile was established and development of the new digital cellular standard begun. This was also the birth of the GSM acronym: only later was the original French acronym changed to Global System for Mobile Communications. In 1989, the responsibility for specification development passed to the newly created European Telecommunications Standards Institute (ETSI); during the next year, most of the GSM specifications were published. However, the encryption algorithms were initially kept secret in hope to increase security. The "security through obscurity" principle did not prove a very good idea: the general design of GSM ciphering algorithms leaked in 1994, and by 1999, both A5/2 and A5/1 algorithms had been reverse-engineered.

The subject of this work is the cryptanalysis of A5/1, with special focus on attacks that could benefit from hardware assistance. Special-purpose hardware has proved useful for example in cracking the DES block cipher [Fou98].

The goal of this Master's thesis is to analyse whether it is also possible to speed up the cryptanalysis of the A5/1 stream cipher with the aid of hardware, in particular, field programmable gate arrays (FPGAs). The nature of the thesis is theoretical: descriptions of attack algorithms and estimates on their time and memory complexity are meant to help a hardware specialist in selecting the method most suitable for hardware implementation. Also, we are aware of one previous hardware implementation of a cryptanalytic attack against A5/1, which we cover in Section 4.1.4.

The thesis is organised as follows: Chapter 2 introduces the reader to the general architecture of GSM security and gives necessary background about other aspects of GSM voice transmission. The first part of Chapter 3 describes the A5/1 encryption algorithm. The second part summarises some general concepts in the cryptanalysis of stream ciphers. This concludes the introductory part.

The major part of our original work is presented in Chapter 4. In this chapter, we analyse three different attacks against A5/1, belonging to a generic class of attacks known as guess-and-determine attacks. Apart from reviewing work done by other authors, our own contribution is the following. First, in Section 4.1, we give a detailed complexity analysis for the Anderson-Keller-Seitz attack presented in this section. Keller and Seitz have attempted such analysis in [KS01], however, we have found that their analysis is imprecise, and present our exact results. Second, we have also implemented the attack in software. The implementation enables us to verify our theoretical results as well as to give an estimate on the time complexity of the attack on a PC. Implementation results are presented in the end of the section.

Next, we also use our results from Section 4.1 to give a precise estimate on the complexity of the Biham-Dunkelman attack, presented in Section 4.2. Also, as a smaller original contribution, we generalise the Biham-Dunkelman attack by proposing a trade-off curve between the amount of required plaintext and computational complexity.

Chapter 5 proceeds with reviewing other known attacks against A5/1. In each section of this chapter, we describe the key ideas of one attack and provide implementation details, where available. We conclude each section by discussing possibilities for hardware implementation. Finally, in Chapter 6, we give a comparison table of the parameters for all of the presented attacks.

Chapter 2

Introduction to GSM Security

This chapter gives a general introduction to the GSM system. We start by giving information about speech encoding, error correction coding and speech transmission in GSM. In the next section, we move on to the more specific field of security in the GSM system. This chapter is meant to give necessary background information for understanding the security of the A5/1 encryption algorithm. We do not analyse the security of the GSM system as a whole and all security claims that we make later in this thesis will only concern the cryptographic strength of the A5/1 cipher.

2.1 GSM Speech Transmission

Speech in GSM is digitally coded at a rate of 13 kbps, so-called full-rate speech coding. The speech signal is divided into 20 millisecond samples, so one block of the GSM speech codec output is 260 bits long. Each block is then protected from errors, using cyclic and convolutional encoding, and the output size of the encoder is 456 bits. Before transmission, this data is encrypted, using a bitwise exclusive-or (XOR) operation with a pseudo-random bit sequence generated by the encryption algorithm. The fact that the encrypted data is redundant—260 bits of actual data are expanded into 456 bits by the encoder—has been exploited to mount a ciphertext-only attack on the encryption algorithm. For more details on GSM error-correction coding, we refer the reader to the relevant standard [ETSa].

The GSM system uses Time Division Multiple Access (TDMA) to allow several users to share the same channel. The time unit of the TDMA system is a *burst*, which lasts 3/5200 seconds (approximately 0.577 ms). One burst is capable of carrying 114 data bits, and so the size of a burst is 25% of the size of an error correction coded block, or equivalently 5 ms of conversation.

0		8		16		24		...		448		burst N
1		9		17		25		...		449		burst N+1
2		10		18		26		...		450		burst N+2
3		11		19		27		...		451		burst N+3
	4		12		20		28		...		452	burst N+4
	5		13		21		29		...		453	burst N+5
	6		14		22		30		...		454	burst N+6
	7		15		23		31		...		455	burst N+7

Figure 2.1: The block diagonal interleaver of the GSM system

However, the division of a block into bursts is slightly more complicated.

While error correction codes employed in the GSM system make it possible to detect and correct bit errors, they cannot guard against disturbances causing a large number of consecutive erroneous bits—a situation common to the radio interface. To further protect against such errors, each 456-bit speech sample is interleaved before transmission. First, the 456 bits are divided into 8 blocks of 57 bits each: bits $0, 8, \ldots, 448$ form block I, bits $1, 9, \ldots, 449$ form block II etc, until the last block of 57 bits (block VIII) will then contain the bits numbered $7, 15, \ldots, 455$. The first four blocks of 57 bits are then placed in the even-numbered bits of four consecutive bursts. The other four blocks of 57 bits are placed in the odd-numbered bits of the next four bursts. This type of interleaver is called the block diagonal interleaver (see Figure 2.1).

Since a burst can carry 114 data bits, each burst carries blocks from two consecutive samples, and one sample is distributed over eight bursts. This means that even if an entire burst is lost during transmission, the loss for a sample block is only 12.5% of the total number of bits.

A burst also contains some fixed bits to control modulation, so the total length of one burst is 156.25 bits. However, only the 114 data bits are encrypted before transmission. Encryption takes place as a final step after interleaving and before modulation; decryption is carried out symmetrically after demodulation. The steps of speech processing before modulation are depicted in Figure 2.2.

In addition to digitally encoded speech, the mobile and the network also exchange signalling data during the set-up phase of a call, and also during the call itself, e.g., to report signal strength. While speech data is exchanged on traffic channels, signalling data is sent on dedicated control channels. Both types of channels are bidirectional, i.e., data is sent and received simultaneously. Signalling data is protected from errors with similar error correction

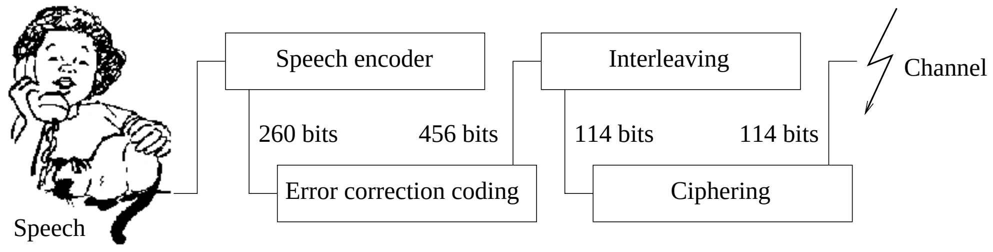

Figure 2.2: Speech coding in the GSM system

methods, but the input to the encoder is now a message of fixed length 184 bits and the output block size is again 456 bits. Interleaving is done in a slightly different manner. Namely, the first four blocks of 57 bits are again placed in the even-numbered bits of four consecutive bursts, but the other four blocks are placed in the odd-numbered bits of the *same* four bursts. Hence, a burst of signalling data carries data from only one signalling message. This type of interleaver is called the block rectangular interleaver. Finally, the 114 significant bits of a signalling data burst are encrypted in exactly the same way as bits of speech data.

Bursts in GSM are sent in frames. One TDMA frame contains eight bursts from eight different users and consecutively lasts approximately 4.62 milliseconds. The difference between the frame length and the corresponding length of conversation (5 ms) is compensated by the fact that not all frames carry speech data. Of each 26 consecutive TDMA frames, 24 are reserved for traffic channels, one frame is used for signalling data on the Slow Associated Control Channel (SACCH) and the last frame is an idle frame, i.e., nothing is transmitted at all.

This structure is called the TDMA 26-Multiframe. In this multiframe structure, there are separate channels for uplink and downlink information, so the total number of information bits transmitted on a channel within each frame is $2 \cdot 114 = 228$ bits per user. Now, for each user, 26 frames carry 24 blocks of (uplink and downlink) speech data of 114 bits, or equivalently, $24 \cdot 5 = 120$ milliseconds of conversation. We see that the total duration of the 26 frames, together with the data frame and the idle frame, is also 120 milliseconds, as expected.

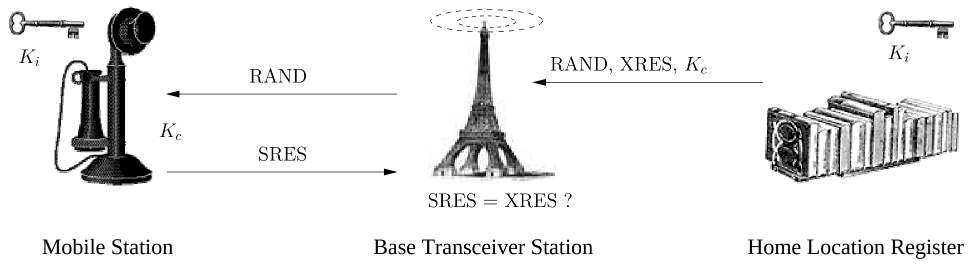

Figure 2.3: User authentication in the GSM system

2.2 Security in the GSM System

For each user i in the GSM network, there exists a permanent 128-bit secret key K_i that is stored in two locations: the user's Subscriber Identity Module (SIM) card and the Authentication Centre (AuC) of the network. The key is never supposed to leave either of these locations. Checking whether the user has access to his key K_i (and is hence, hopefully, a legitimate user) is done by challenging the SIM card to do a computation that can only be done with the correct key K_i. The base transceiver station (BTS) can do this without obtaining knowledge of the secret key K_i in the following way.

When a mobile station (MS)—a user—logs on to the network, the BTS requests an *authentication triplet* from the AuC of the user's Home Location Register (HLR). The HLR/AuC first selects a 128-bit random value RAND. It then computes an authentication value XRES (eXpected RESponse) and the session key K_c from RAND and the user's permanent secret key K_i (recall that the AuC is the only other party in possession of the key K_i).

The three values—the 128-bit RAND, the response value XRES and the 64-bit session key K_c—form the authentication triplet that the HLR/AuC sends to the BTS. The BTS forwards RAND to the MS. The MS performs the same computations as the HLR/AuC did before. It obtains two values: the session key K_c and the authentication value SRES (Signed RESponse), and sends the latter to the BTS. If the MS is in possession of the correct secret key K_i, the value of SRES will match the expected value XRES. Vice versa, it should be the case that if the value of SRES matches the expected value XRES, then the MS is indeed in possession of the correct key. If the values match, authentication succeeds and the key K_c, now known by both the user and the BTS, is used for encryption until the next authentication occurs (see Figure 2.3).

The response value and the session key are generated by the HLR and the MS with two one-way functions A3 and A8, respectively. Neither of these algorithms is specified in the GSM specifications; rather, they are left for the service provider to implement. In practice, they are usually implemented together as COMP128, a "reference implementation" from the GSM specifications, or one of its later, improved versions. An overview of security issues related to COMP128 can be found at [COM].

Authentication in GSM systems is one-way: the user is authenticated to the base station, but not vice versa. This opens up possibilities for active attacks with a fake base station.

After the session key K_c has been generated and the encryption process has been initialised, ciphering takes place at both the mobile station and the base station. The encryption algorithm of GSM is called A5. Each frame, the A5 algorithm produces the same pseudorandom sequence of 2×114 bits at both ends. Encryption is achieved by simple bitwise XOR with the plaintext block; the decryption operation is obviously identical to encryption. The first 114-bit block output by the algorithm is now used to encrypt data on the network side and, once received, to decrypt it on the mobile station side. The second block is used analogously to encrypt data transmitted from the mobile station to the base station.

As explained above, encryption of calls in the GSM system is not end-to-end. Radio traffic between the end user and the nearest base station is encrypted; however, base stations have access to plaintext, and on (landline) links between network stations, data is sent in the clear. Moreover, control data, including the session key used for ciphering, is sent to the base station unencrypted. This is one of the most criticised properties of GSM security. Also, session key setup, which is triggered by the authentication procedure, may occur as often—or as rarely—as the network operator wishes. In practice, the same key may be in use for days. As we shall see from some of the attacks, a large amount of data encrypted with the same key increases the vulnerability of the cipher.

The ciphering procedure described above applies to GSM security. GPRS connections are secured using a different encryption algorithm (GEA), which is not publicly known. Readers interested in other aspects of GPRS security may read [eK01].

Chapter 3

The A5/1 Encryption Algorithm

In this chapter, we describe the A5/1 encryption algorithm. The description was first obtained by methods of reverse engineering and can be found at [And94], together with a reference software implementation in C programming language [BGW99]. According to [BSW01], the GSM organisation has confirmed the correctness of the algorithm. We end this chapter by introducing the general concept of the cryptanalysis of stream ciphers.

3.1 The A5 Family of Stream Ciphers

There are multiple versions of the encryption algorithm referred to as A5:

- A5/0 is a dummy cipher that provides no encryption;
- A5/1 (the subject of this Master's thesis) is the original A5 algorithm used in Europe but also in North America and increasingly all over the world;
- A5/2 is an intentionally weaker encryption algorithm created for export;
- A5/3 is a strong encryption algorithm created as part of the 3rd Generation Partnership Project (3GPP).

Following the principle of "security through obscurity", the designs of A5/2 and A5/1 were initially kept secret. Sketchy designs leaked in 1994 and the exact designs were reverse engineered in 1999. A recent ciphertext-only attack on A5/2 by Barkan, Biham and Keller [BBK03] requires less than 5.5 hours of one-time precomputation and recovers the session key in less than a second on a regular PC. Attacking A5/1 still requires notably more time and memory resources. The development of A5/3 has been public from the start and there are no known security problems with the algorithm, which is based on the KASUMI block cipher.

A5/2 was designed intentionally weak due to export limitations. Following the publication of the Barkan-Biham-Keller attack, the 3GPP organisation recently withdrew A5/2 from its standard. Export limitations have been removed and 3GPP is encouraging networks to remove A5/2 support at the earliest opportunity [3GP04]. Since support for A5/3 version is present only in Third Generation networks, and Second Generation networks are gradually dismissing the weak A5/2 version, A5/1 will be the prevailing algorithm of GSM encryption almost all over the world during the coming years. Moreover, 3G networks will have to ensure backwards compatibility with A5/1 until old handsets are no longer used, so the cryptanalysis of A5/1 is still an important topic of research.

3.2 The A5/1 Stream Cipher

A5/1 is a generic (as opposed to block cipher based) synchronous stream cipher. Three linear feedback shift registers (LFSRs) R_1, R_2 and R_3 of lengths 19, 22 and 23 bits, respectively, are used as its main building blocks. Table 3.1 shows the primitive (i.e., maximal length) feedback polynomials of the registers.

Register	Feedback polynomial	tap positions	period
R_1	$x^{19} \oplus x^5 \oplus x^2 \oplus x \oplus 1$	18, 17, 16, 13	$2^{19} - 1$
R_2	$x^{22} \oplus x \oplus 1$	21, 20	$2^{22} - 1$
R_3	$x^{23} \oplus x^{15} \oplus x^2 \oplus x \oplus 1$	22, 21, 20, 7	$2^{23} - 1$

Table 3.1: Feedback polynomials of the LFSRs in the A5/1 construction

A schematic description of the A5/1 cipher is given in Figure 3.1. Throughout this thesis, we refer to bits 18, 21 and 22—leftmost bits in the figure—of registers R_1, R_2 and R_3, respectively, as the most significant bits of the respective registers. Conversely, the bits at position 0—rightmost in the figure—are referred to as the least significant bits.

When a register is clocked, the bits at its tap positions are XORed together and stored in the least significant bit of the register, after its bits have been shifted one position to the left. The most significant bit of each register

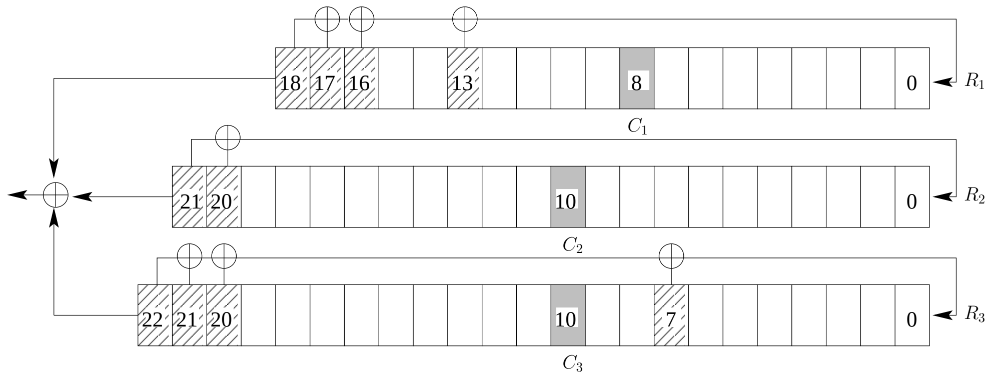

Figure 3.1: The three LFSRs of the A5/1 stream cipher

is its output bit. The output of the A5/1 generator is at each clocking cycle obtained by XORing the output bits of the three registers.

The algorithm takes as input two parameters: a 64-bit secret session key K_c and a 22-bit counter F_n, derived from the publicly known TDMA frame number. During the operation of the cipher, the contents of the three registers are re-initialised with a new counter before the encryption/decryption of each frame.

In the initialisation phase, all registers are clocked regularly at each cycle. During the first 64 clockings, the 64 bits of the session key K_c—from least significant bit (lsb) to most significant bit (msb)—are XORed into the lsb of each of the registers in parallel. During the next 22 cycles, the frame number F_n is mixed into the registers analogously. In the key setup phase, the cipher produces no output. The state of the three registers after key setup is called the *initial state* of frame n. Due to the way the contents of the registers are initialised, the initial state is a linear combination of the publicly known frame counter F_n and the secret session key K_c. Hence, when the initial state of any given frame is known, the secret key is revealed.

After initialisation, an irregular clocking rule is introduced. The middle bits of the three registers—bits 8, 10 and 10, denoted by C_1, C_2 and C_3, respectively—are the clocking bits used to determine the stop/go clocking. At each step, the majority of the three bits is calculated and only those registers for which the clocking bit agrees with the majority are clocked. Hence, if $C_1 = C_2 = C_3$, all registers are clocked; if $C_1 = C_2 = C_3 \oplus 1$, registers R_1 and R_2 are clocked, and so on. The probability for an individual LFSR to be clocked is 3/4 and the majority clocking rule guarantees that at least two of the three registers are clocked at each cycle.

With the stop/go clocking started, the registers are first clocked for 100 clock cycles without producing any output. Finally, the registers are stop/go clocked for 228 cycles and an output bit is produced after each clock cycle. Hence, the first output bit is produced after the registers have been clocked irregularly for 101 clock cycles. Half of the produced 228 bits are then used to encrypt uplink traffic, while the remaining bits are used to decrypt downlink traffic in the current frame. The amount of discarded bits is crucial in correlation attacks against the cipher (c.f. Chapter 5, Section 5.2).

One more observation should be made about encryption. Namely, the GSM protocol is implemented in such a way that if one of the conversation parties is silent, transmission from that side is cut. In this case, only half of the 228 bits are actually used in encryption; the other half is just discarded. For the cryptanalyst, this brings bad news, since it reduces the amount of available ciphertext and plaintext. In all complexity estimates throughout this thesis, we assume that from each time frame during the conversation all 228 bits of keystream can be extracted.

At any moment during the operation of the cipher, the contents of the three registers form the *internal state* of the cipher. Note that the internal state size of A5/1 is equal to the secret key length. In modern stream ciphers (e.g., SNOW 2.0), the state size is usually at least twice the key size. This requirement is also pointed out in the ECRYPT Call for Stream Cipher Primitives, published in November 2004, as a prevention measure for time-memory trade-off attacks [ECR]. All known attacks against A5/1 are internal state recovery attacks, followed by initial state reconstruction and session key recovery.

3.3 Cryptanalysis of the A5/1 Stream Cipher: Generic Ideas

Cryptanalysis of a stream cipher is usually based on the **known plaintext** assumption. This means that the attacker has access to a certain amount of corresponding pairs of plaintext and ciphertext. Assume that the attacker holds a chunk of plaintext $X = (x_0, x_1, \ldots, x_n)$ and a corresponding chunk of ciphertext $Y = (y_0, y_1, \ldots, y_n)$ (where x_i and y_i are bits, $0 \leq i \leq n$). The encryption operation of a stream cipher is typically—and also in the case of A5/1—simple bitwise exclusive-or of the plaintext $X = (x_0, x_1, \ldots, x_n)$ and keystream $Z = (z_0, z_1, \ldots, z_n)$:

$$\begin{aligned}(y_0, y_1, \ldots, y_n) &= (x_0, x_1, \ldots, x_n) \oplus (z_0, z_1, \ldots, z_n) \\ &= (x_0 \oplus z_0, x_1 \oplus z_1, \ldots, x_n \oplus z_n) \ ,\end{aligned}$$

or briefly,

$$Y = X \oplus Z \ . \tag{3.1}$$

This means that the attacker can immediately deduce the keystream produced by the cipher by rewriting (3.1) as

$$Z = X \oplus Y \ ,$$

i.e.,

$$(z_0, z_1, \ldots, z_n) = (x_0 \oplus y_0, x_1 \oplus y_1, \ldots, x_n \oplus y_n) \ .$$

The task of the attacker is now to find the key that was used to produce this output. In the following sections, we use the terms known plaintext and known keystream as synonyms, since knowledge of one reveals the other.

In the case of the A5/1 cipher, access to known plaintext then means either direct access to the keystream generator or access to the output of the speech codec (or signalling data). Some ideas for obtaining plaintext are presented in Chapter 5, Section 5.3.

If an attack is mounted without any knowledge of the plaintext, only using the encrypted data, it is called a ciphertext-only attack. In the case of the A5/1 cipher, such an attack would require only passive eavesdropping on radio channels to obtain encrypted data and would hence be very powerful. On the other hand, a ciphertext-only attack still requires some (statistical) knowledge of the plaintext. If nothing about the plaintext is known, then in (3.1), Z can be any keystream produced with a valid key and X can be anything. Hence, any keystream will yield a valid plaintext and there is no way to decide which of the keys was actually used for encryption.

In real-life settings, plaintext is never “anything”. In the case of A5/1 in the GSM setting, plaintext is digitally encoded speech or signalling data. Moreover, as was described in Section 2.1 of Chapter 2, all data is expanded by employing error-correction coding. This means that although output from the speech codec is compressed, input to the encryption algorithm is redundant. In Section 5.3, we describe how the redundancy can be exploited to mount a ciphertext-only attack on the cipher.

In the following chapters, several types of attacks against the A5/1 cipher are described. When comparing these attacks from the viewpoint of efficiency and practical threat to security, the following four parameters should be considered:

- **Data complexity**. In the common known plaintext model, data complexity means **known plaintext requirement**—the amount of plaintext-ciphertext pairs needed to complete the attack with a given success probability. In the case of a ciphertext-only attack, data complexity simply means the amount of ciphertext needed. In practical settings, there is usually an upper limit to the amount of plaintext encrypted using the same key. In the case of A5/1, we know that the amount of plaintext encrypted, using *the same session key and frame number combination*, is 228 bits. The amount of conversation encrypted, using *the same session key but different frame numbers* depends on the frequency of authentication. In any case, obtaining seconds of known plaintext or minutes of known ciphertext might be feasible, whereas re-authentication makes obtaining hours of conversation from one session an impossible task.

- **Time complexity**. In the theory of cryptanalysis, time complexity is usually given as a number of operations needed to complete the attack. The worst-case time complexity of an exhaustive search is equivalent to the size of the keyspace—2^{64} in the case of A5/1—and the efficiency of any other attack can be compared to this measure. If an attack succeeds in finding the key in less than 2^{64} time, the cipher is said to be broken.

  In practical settings, we are of course interested in the actual wall clock time needed to carry out the attack. The attack time then depends on the type of operations referred to in the theoretical calculations, and an attack requiring 2^{40} cipher clockings is in practice somewhat faster than an attack requiring 2^{40} steps of "solving a system of linear equations".

  The time needed to complete an attack can be divided into **precomputational complexity** and **attack-time complexity**. Precomputations are usually done only once, before the attack is launched, so time requirements are slightly more relaxed. An attack requiring a month of one-time precomputations, but thereafter finding the key in seconds during actual attack time is more feasible than an attack requiring a month of computations each time a new key is attacked.

  Finally, we distinguish **worst-case complexity** and **average-case complexity**. Worst-case complexity is the time after which the attack is bound to finish, whereas average-case complexity refers to the time after which the attack is expected to finish. For example, in the case of an exhaustive search over 2^{64} possible keys, the right key will definitely be found after 2^{64} tests, but it is expected to be found after half of the tests, i.e., in time 2^{63}.

- **Memory complexity**. The required amount of memory needed to perform the attack is most often crucial if the attack has a precomputation stage. Memory complexity is directly related to precomputation time complexity: large memory requirement infers long precomputation time.

- **Success probability**. We consider two types of attacks. *Deterministic* attacks are guaranteed to succeed within time T, given an amount D of plaintext, using M memory. *Probabilistic* attacks, on the contrary, have success probability $p < 1$ for fixed parameters T, M and D.

Chapter 4

Guess-And-Determine Attacks

This chapter, which forms the core of the thesis, deals with a certain class of attacks against stream ciphers in general and the A5/1 cipher in particular, namely, guess-and-determine attacks. Guess-and-determine attacks were the first attacks proposed against A5/1; they are also probably the easiest to understand. Later, several other attacks with significantly lower computational complexity have been found. However, guess-and-determine attacks still prevail in one area—the amount of known plaintext required. We believe that in the real world, obtaining known plaintext may be more difficult than obtaining a sufficient amount of computation power, and so guess-and-determine attacks are still worth investigating, despite their relatively high computational complexity.

4.1 Anderson-Keller-Seitz Attack

In this section, we discuss the first proposed guess-and-determine attack against the A5/1 stream cipher. The core of a guess-and-determine attack lies in guessing part of the cipher's internal state and deriving the remaining unknown state bits from known keystream. Thus, this attack is a known plaintext attack. The proposed attack requires only one or two frames of known plaintext, no precomputation and, correspondingly, no notable amount of memory. On the other hand, the computational complexity of such an attack is quite high—over 2^{50} A5/1 clock cycles, which amounts to over two years of computation time on our test platform PC.

The next three sections focus on theoretical attack models and their complexity. Section 4.1.1 introduces the main concept of the attack, which was first mentioned by Anderson [And94]. In Section 4.1.1, we analyse a concrete attack model that was used in a hardware implementation by Keller and Seitz [KS01]. From now on, we refer to this attack as the Anderson-Keller-Seitz (AKS) attack. Section 4.1.1 contains original theoretical results on the computational complexity of the attack. A previous attempt at complexity analysis by Keller and Seitz is discussed in Section 4.1.3—we show that their case study is incomplete and, as a consequence, they underestimate the complexity of the attack over 4 times.

In the final section, we compare two implementations of the AKS attack. First, we describe our own implementation of the attack in software. We include some notes on implementation and a time complexity estimate. Then, we discuss the hardware implementation by Keller and Seitz and compare the efficiency of the two implementations.

4.1.1 Key Ideas of the Attack

Even before the exact design of the A5/1 algorithm was reverse engineered, first attacks against the cipher were published. When the sketchy design leaked, Anderson immediately pointed out that a trivial guess-and-determine attack, where the contents of two linear feedback shift registers are guessed and the content of the last register is then derived from known keystream, has computational complexity of about 2^{40} tests.

(Un)fortunately, the complexity estimate given by Anderson is too optimistic. Namely, the contents of two registers are not sufficient to derive the contents of the third register in a straightforward manner, since it is not possible to determine, which registers are being clocked each cycle, without knowing the clocking bit of the third register. A rough approach would be to additionally guess the 11 less significant bits of the third register. This would reveal the clocking sequence and would allow to determine the remaining 12 bits of the internal state. Since the combined length of the two shortest LFSRs is 41 bits, the worst-case complexity of the attack would now increase to 2^{52} tests, where each test consists of deriving the remaining 12 bits of the third register and further checking the solution against known keystream to eliminate false possibilities. The average-case complexity of the same attack is then 2^{51} tests, since we can expect to examine half of the possibilities before we find the right state.

This attack can be slightly improved by noting that some of the 2^{11} possibilities for the less significant bits of the third register lead to an early contradiction with the known keystream even before all the remaining unknown bits have been recovered. Keller and Seitz have taken advantage of this improvement and implemented the attack in hardware. In the next section, we proceed to describe the attack in more detail and present a detailed complexity analysis for both hardware and software implementations.

We stress that the internal state of the cipher recovered by this attack does not directly reveal the secret session key. The attack recovers the state of the registers at the moment when the cipher starts producing output, i.e., after 100 bits of discarded output. Techniques for reversing the algorithm for 101 cycles to recover the initial state and derive the session key were first proposed by Golić and are discussed in the end of Section 4.1.2.

An Improvement of the Attack

We shall now present a concrete attack model from [KS01] and give a theoretical complexity analysis of the model. First, we present the stages of the attack step-by-step. Then, we compute the average-case complexity of the attack in high-level operations. Finally, we give two implementation-specific complexity estimates in A5/1 clock cycles.

Our computations throughout this thesis will be based on the assumption that the LFSR sequences are mutually independent and uniformly random. Although this is not the case, the randomness assumption is reasonable, since the individual LFSR sequences of A5/1 are maximum-length sequences with good statistical pseudo-random properties. In particular, it is known that the output sequence of a maximum-length LFSR satisfies a set of properties known as Golomb's randomness postulates. Also, it should be noted that the output function of A5/1—a ternary exclusive-or with inputs from the outputs of its three registers—is maximum-order correlation immune. Recall that an n-ary Boolean function is maximum order correlation immune if its output is statistically independent of any $n-1$ inputs to the function, and exclusive-or clearly satisfies this property. A nice explanation of both Golomb's postulates and correlation immunity can be found in e.g. [MvOV96]. Furthermore, the results of our practical simulations, discussed in Section 4.1.4 also support the reasonableness of such assumptions.

The Precise Attack Model

We start by giving some notations. Let our time unit be an A5/1 clock cycle and let the time when the first output bit is produced be $t = 0$. Denote the (unknown) most significant bits of registers R_1, R_2 and R_3 at time t by $O_1(t)$, $O_2(t)$ and $O_3(t)$, respectively. Denote the clocking bits of the registers at time t by $C_1(t)$, $C_2(t)$ and $C_3(t)$, respectively. Finally, suppose that we are given the known keystream sequence $\mathcal{Z}(t) = Z(0), Z(1), \ldots, Z(227)$ from one frame. Our task is then to determine the internal state of the cipher that generates this sequence. Note that at each cycle, the following relation holds:

$$Z(t) = O_1(t) \oplus O_2(t) \oplus O_3(t) \ . \tag{4.1}$$

We proceed as follows. First, we guess the contents of registers R_1 and R_2 at time $t = 0$. Equation (4.1) can then be rewritten as

$$O_3(t) = Z(t) \oplus O_1(t) \oplus O_2(t) \ , \tag{4.2}$$

where the right-hand-side is now known, provided that the clocking sequence is known. In particular, we can immediately determine the unknown bit $O_3(0)$ from the first known output bit $Z(0)$ and the guessed bits $O_1(0)$ and $O_2(0)$. In general, every time the third register is clocked, the next output bit of R_3 is determined by known keystream. Finally, when the entire state of register R_3 has been derived, we can continue to clock the registers and compare the output against known keystream to eliminate false possibilities for the internal states of registers R_1 and R_2.

The difficulty here lies in the fact that the clocking sequence of the registers depends not only on the guessed states of R_1 and R_2, but also on the 11 less significant bits (bits 0 to 10) of R_3. Hence, we would need to guess 11 more bits, yielding the aforementioned worst-case complexity of 2^{52} tests.

Fortunately, it turns out that we can do a little better. Namely, some of the 2^{52} partial internal states cannot be completed to a full 64-bit state that produces the required keystream. Therefore, we proceed to derive the contents of the third register bit-by-bit. As we shall soon see, this enables us to eliminate inconsistent guesses early on in the search.

The search, depicted in Figure 4.1 goes as follows. First, since $Z(0)$, $O_1(0)$ and $O_2(0)$ are known, bit $O_3(0)$ can immediately be derived from (4.2). Next, we guess the clocking bit $C_3(0)$ and clock the registers according to the majority rule.

At any time t, when guessing bit $C_3(t)$, there are two possible cases. First, assume that the clocking bits $C_1(t)$ and $C_2(t)$ of registers R_1 and R_2 are different. In this case, $C_3(t)$ agrees with either $C_1(t)$ or $C_2(t)$ and, due to the majority rule, register R_3 is definitely clocked. There are two possibilities for the clocking bit $C_3(t)$ and we need to check both of them. Either case leads us back to the beginning of the loop depicted in Figure 4.1 and we proceed to derive the next output bit of R_3.

Second, suppose that registers R_1 and R_2 have identical clocking bits at some time t, i.e., $C_1(t) = C_2(t)$. Then these registers are both clocked. Now, consider the guess $C_3(t) = C_1(t) \oplus 1$, i.e., assume that the clocking bit $C_3(t)$ disagrees with the majority and, consequently, the third register is not clocked. Then the most significant bit of R_3 remains the same during the next clock cycle, i.e., $O_3(t + 1) = O_3(t)$. The next keystream bit $Z(t + 1)$

can then be compared to the XOR of the most significant bits of R_1, R_2 and R_3. If these bits differ, the guessed internal state does not produce correct keystream, so the option $C_3(t) = C_1(t) \oplus 1$ can be eliminated. In other words, it has to be that $C_3(t) = C_1(t) = C_2(t)$ and register R_3 is clocked. Now, the clocking bit $C_3(t)$ is uniquely determined. The probability that a random guess for $C_3(t)$ leads to such a contradiction in one clock cycle is thus

$$\begin{aligned}\Pr[\mathsf{fail}(1)] &= \Pr[C_1(t) = C_2(t) \neq C_3(t)] \cdot \\ &\quad \Pr[Z(t+1) \neq O_1(t+1) \oplus O_2(t+1) \oplus O_3(t)] \\ &= \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8} \ .\end{aligned}$$

Here, the probability is taken over all possible values for bits $C_1(t)$, $C_2(t)$, $C_3(t)$, $O_1(t+1)$, $O_2(t+1)$ and $O_3(t) = Z(t) \oplus O_1(t) \oplus O_2(t)$, assuming that the LFSR sequences of R_1 and R_2 are mutually independent and uniformly random.

Observe that at most one of the two guesses for $C_3(t)$ leads to a contradiction. Thus, by mutual exclusivity, the probability that either of the two guesses for $C_3(t)$ leads to a contradiction (so that $C_3(t)$ is uniquely determined) is 1/4. Therefore, the average number of possibilities that we need to consider is at most $(1 \cdot 1/4 + 2 \cdot 3/4)^{11} = (7/4)^{11} \approx 471$, so the heuristic reduces the number of tests approximately 4 times.

However, we can refine this search a bit further. Namely, what if R_3 is not clocked at cycle t but the keystream bit and the XOR bit are identical, so that we do not get a contradiction? Then, the clocking bit of R_3 remains the same during the next cycle, so we can look one step further. If R_3 is clocked at cycle $t+1$, we cannot determine whether our initial guess $C_3(t) = C_1(t) \oplus 1$ was right or not. If R_3 is not clocked, we can again perform the output bit check. Continuing analogously, one of the two things is bound [1] to happen: either R_3 is clocked or we get a contradiction in the output bit check. In the latter case, we know that R_3 had to be clocked at cycle t, so we have reduced the possibilities for the clocking bit $C_3(t)$ from two to one. In the former case, we cannot outrule either guess for the clocking bit of R_3 at cycle t, so we must keep both possibilities.

Figure 4.1 gives a schematic description for the search. The search has two end states: FAIL means that our last guess for $C_3(t)$ is inconsistent with

[1]Unless registers R_1 and R_2 are both in the constant all-zero state, register R_3 will definitely be clocked after a finite number of cycles. If it indeed happens that registers R_1 and R_2 are both in the all-zero state, then for a certain (constant) keystream the test will "hang" until we run out of available keystream. However, our test will finish with very high probability and avoiding the contrary is just a simple matter of implementation.

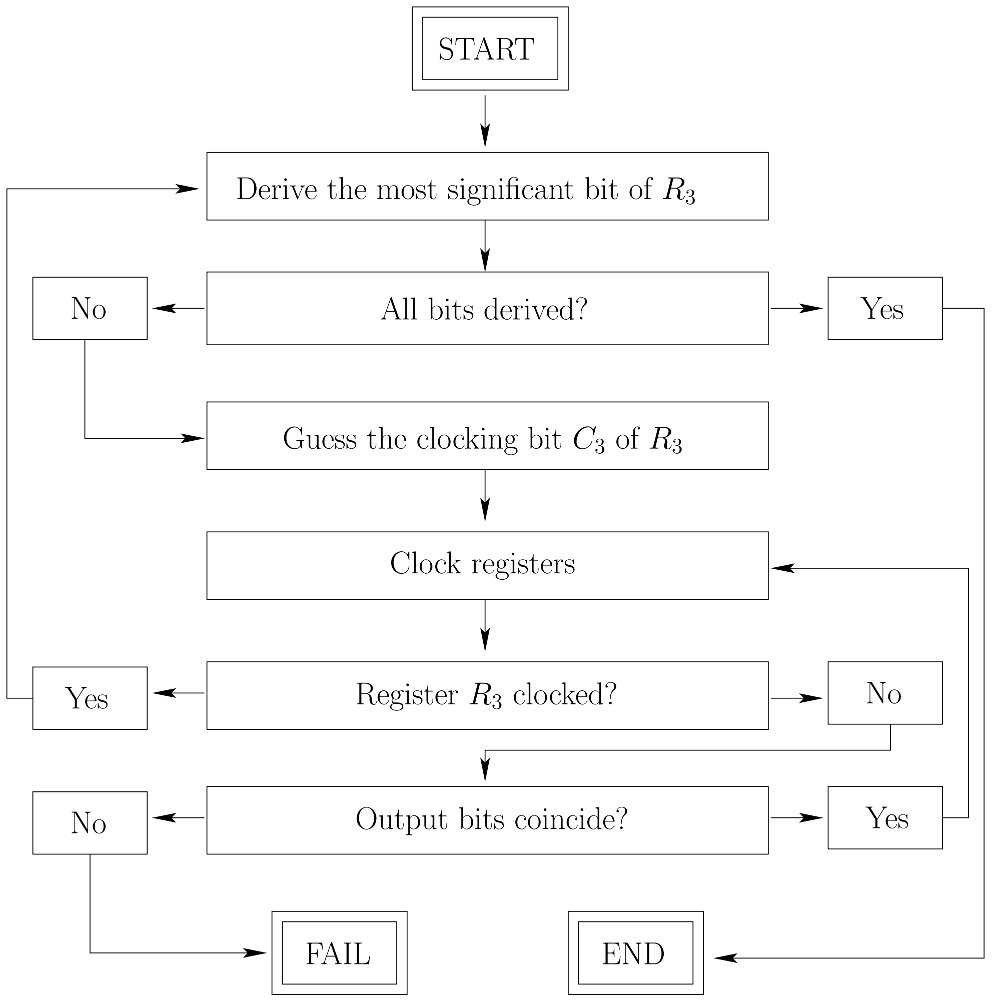

Figure 4.1: A schematic description of the improved guess-and-determine search

the output keystream, hence, the algorithm must return to the point where the last assumption about the clocking bit $C_3(t)$ was made. END means that we have found a complete candidate internal state for the three registers and continue to check the solution. We shall call these internal states *candidate solutions*.

In Figure 4.2, an example of a contradictory guess is given. If the output sequence is $\mathcal{Z} = 0, 0, 1, \ldots$, the most significant bit of R_3 must be 1. Now, the clocking bit $C_3(t)$ is uniquely determined: $C_3(t) = 1$. Indeed, if we set $C_3(t) = 0$, then register R_3 is not clocked for two consecutive cycles and the output produced is $\mathcal{Z}' = 0, 0, 0, \ldots$, so the second clock cycle yields a contradiction.

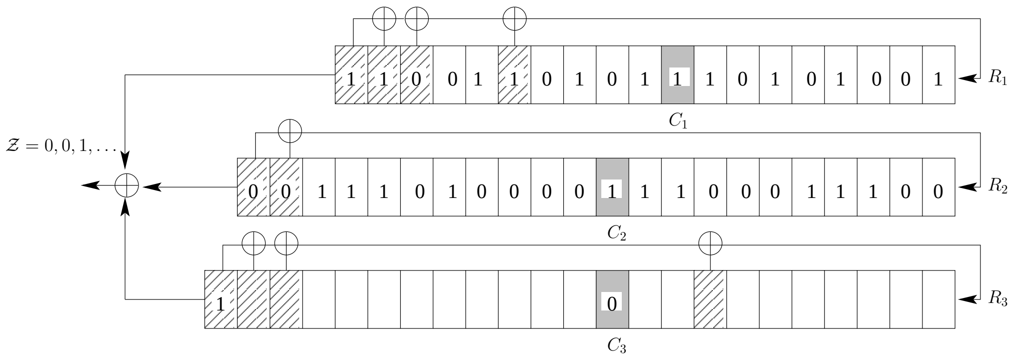

Figure 4.2: A contradictory state for the output sequence $\mathcal{Z} = 0, 0, 1, \ldots$

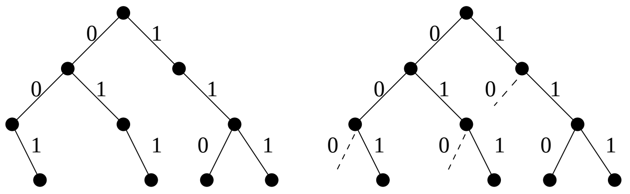

Figure 4.3: A binary solution tree for $n = 3$. Here, (001), (011), (110) and (111) are valid solutions, whereas (000), (010) and (10) lead to a contradiction.

4.1.2 Average-Case Complexity of the Attack

In the following, we shall use a binary solution tree to model our search algorithm. Obviously, for each guess of registers R_1 and R_2, we can use a full binary tree of depth 11 to represent all possible choices for the 11 clocking bits of register R_3. In our search, we then start from the root and step down one level in the tree every time register R_3 is clocked. If we did not eliminate contradictions on the fly, our search algorithm would need to go through the entire binary tree 2^{41} times, once for every guess of registers R_1 and R_2.

Now, in order to model contradictions, we need to build a tree of *consistent* solutions. Such a tree can be obtained by pruning the full binary tree at the nodes where a contradiction is found. A toy example of such a pruned tree is given in Figure 4.3: the tree on the left is a pruned solution tree; the tree on the right also contains deleted edges. It should be noted that we always prune at most one of the two subtrees of a node, since at most one guess for the clocking bit of register R_3 can lead to a contradiction. Hence, in our binary solution tree, all leaves are located at level 11. Each path to a leaf (or shortly, each leaf) corresponds to one consistent choice for the 11 bits of register R_3, and vice versa.

In order to calculate the complexity of the AKS attack, we need to find the average number of leaves in a solution tree. For this, we first find the average number of children of a non-leaf node. As stated above, all computations in this chapter will be based on the (essentially false but practically reasonable) assumption that the output sequences produced by the three LFSRs are mutually independent and uniformly random. For example, when we make a random guess for the 19 bits of R_1 at $t = 0$, we assume that R_1 produces random bits even for $t \geq 19$ (while, in reality, each bit is a linear combination of previous bits). We are now ready to formulate and prove the first result.

Proposition 4.1. *Assume that the bits entering the clock control of registers R_1 and R_2 and the output bits produced by registers R_1 and R_2 are mutually independent and uniformly random for all $t \geq 0$. A non-leaf node in a binary solution tree then has $12/7$ children on average, where the average is taken over all solution trees defined by the known keystream $\mathcal{Z}(t)$ and all possible bit combinations for R_1 and R_2.*

Proof. We need to count the average number of edges leaving a non-leaf node. Thus, we need to determine the probability of a contradiction, i.e., the probability that an edge is deleted. The probability that we run into a contradiction during the first cycle is

$$\begin{aligned}\Pr[\mathsf{fail}(1)] &= \Pr[C_1(t) = C_2(t) \neq C_3(t)] \cdot \\ &\quad \Pr[Z(t+1) \neq O_1(t+1) \oplus O_2(t+1) \oplus O_3(t)] = \frac{1}{8} \ .\end{aligned}$$

The probability that we run into a contradiction during the second cycle is

$$\begin{aligned}\Pr[\mathsf{fail}(2)] &= \Pr[C_1(t) = C_2(t) \neq C_3(t)] \cdot \\ &\quad \Pr[Z(t+1) = O_1(t+1) \oplus O_2(t+1) \oplus O_3(t)] \cdot \\ &\quad \Pr[C_1(t+1) = C_2(t+1) \neq C_3(t)] \cdot \\ &\quad \Pr[Z(t+2) \neq O_1(t+2) \oplus O_2(t+2) \oplus O_3(t)] \\ &= \frac{1}{4} \cdot \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8^2} \ .\end{aligned}$$

Analogously, a contradiction takes place during the ith cycle if (a) register R_3 is not clocked for i cycles; (b) no contradiction is found for $i-1$ cycles; and (c) a contradiction is found during the ith cycle, so

$$\Pr[\mathsf{fail}(i)] = \frac{1}{4^i} \cdot \frac{1}{2^{i-1}} \cdot \frac{1}{2} = \frac{1}{8^i} \ . \tag{4.3}$$

The overall probability of a contradiction is then

$$\Pr[\mathsf{fail}] = \sum_{i=1}^{\infty} \Pr[\mathsf{fail}(i)] = \sum_{i=1}^{\infty} \frac{1}{8^i} = \frac{1}{7} \ . \tag{4.4}$$

Here, working under the randomness assumption means, amongst other things, that our computations do not take into account the upper bound for the number of cycles that register R_3 can remain unclocked, but the difference in probabilities is clearly very small.

Now, since the probability of deleting an edge is 1/7, we delete one of the two edges of a non-leaf node with probability 2/7. Indeed, we always prune at most one of the two subtrees, so the two events corresponding to deleting the two edges are mutually exclusive.

We conclude that a non-leaf node of a solution tree has $1 \cdot 2/7 + 2 \cdot 5/7 = 12/7$ children on average. □

As an easy consequence, we can now state the main result of the complexity analysis.

Corollary 4.2. *Assume that the conditions of Proposition 4.1 hold. Assume also that all* 2^{64} *internal states occur with equal probability* 2^{-64}. *The average-case complexity of the AKS attack is then approximately* $2^{48.6}$ *tests.*

Proof. From Proposition 4.1, it follows that a binary solution tree has on average $(12/7)^n$ nodes at depth n. Since the number of candidate solutions is equal to the number of leaves in the tree, we conclude that we get on average

$$\mathbf{E}[\mathsf{solns}] = \left(\frac{12}{7}\right)^{11} \approx 376 \tag{4.5}$$

candidate solutions for each of the 2^{41} tests. We can expect to perform half of the tests before we find the right internal state, which gives us a complexity estimate

$$\mathbf{E}[\mathsf{T}] = 2^{41} \cdot \mathbf{E}[\mathsf{solns}] \cdot \frac{1}{2} \approx 2^{48.6} \ .$$

□

In other words, we have shown that the effective key length of the A5/1 cipher is reduced from 64 bits to less than 50 bits.

This result is different from the complexity analysis of the same attack model by Keller and Seitz [KS01]. In Section 4.1.3, we discuss their analysis and point out the corrections we have made. We have also verified our results in a practical implementation; our theoretical and simulation results agree completely.

A Complexity Estimate in A5/1 Clock Cycles

When we turn our interest to implementations, we are more interested in finding the time complexity of the attack in low-level operations. In the current case, a natural choice for the operation is one A5/1 cipher clock cycle. The term "clock cycle" should be handled with care here—during the attack, each cycle involves other operations than just shifting the three registers. Nevertheless, for a hardware implementation, an estimate in clock cycles gives a direct relation between the implementation clock frequency and the total expected complexity in wall clock time. In a software implementation, an A5/1 clock cycle is more vague, since one A5/1 clock cycle involves several CPU clock cycles. Still, these estimates are helpful when planning an implementation strategy. We shall give two different complexity estimates that correspond to a typical software and hardware implementation, respectively.

We shall again use the solution tree model to explain the results. Recall that a binary solution tree represents the search corresponding to one choice for the 41 bits of registers R_1 and R_2: non-leaf nodes correspond to intermediate internal states of the cipher; edges mark consecutive guesses for the bits of R_3; and leaves correspond to candidate solutions. The tree has been pruned in such a way that if a guess is contradictory, the corresponding edge (together with the whole subtree) has been deleted. However, since we are counting clock cycles, we must also take into account the time spent on finding contradictions. Therefore, we use the tree model with deleted edges (see Figure 4.3, right).

We divide the attack into three phases. Phase 1 consists of finding candidate solutions. Phase 2 consists of eliminating false solutions by comparing the output generated by a candidate internal state against further known keystream. When a candidate solution passes Phase 2, its correctness should be verified by rewinding the algorithm to the initial state and comparing against another frame—this is Phase 3. The time complexity of Phase 1 is estimated in the following section. Phase 2 is inspected in Section 4.1.2. Techniques for Phase 3 are also briefly discussed in the same section, although the time complexity is in this case notably smaller.

Software-Oriented Strategy

Suppose first that the implementation is a simple depth-first traversal of the tree, and the intermediate states of the registers are kept in memory, as it is feasible in a software implementation. Then, each edge in the tree is traversed once and the time required to complete one of the 2^{41} tests is proportional to the number of edges in the tree. Let us start with some simple facts.

Proposition 4.3. *Assume that the bits entering the clock control of registers R_1 and R_2 and the output bits produced by registers R_1 and R_2 are mutually independent and uniformly random for all $t \geq 0$. A binary solution tree then has approximately 525 non-leaf nodes and approximately 1050 edges on average, where deleted edges are included in the count and the average is taken over all solution trees defined by the known keystream $\mathcal{Z}(t)$ and all possible bit combinations for R_1 and R_2.*

Proof. Since a non-leaf node was shown to have 12/7 children on average, and all leaf nodes are located at level 11, the average number of non-leaf nodes is

$$\mathbf{E}[\mathsf{nl-nodes}] = \sum_{i=0}^{10} \left(\frac{12}{7}\right)^i \approx 525,$$

as desired. Furthermore, each non-leaf node has exactly two outgoing edges, provided that we count deleted edges. Thus, the average number of edges in a tree is $\mathbf{E}[\mathsf{edges}] \approx 1050$. $\square$

Consider now a non-leaf node in a tree and the corresponding internal state of the registers. We will find the average number of cycles needed to process a guess for the next clocking bit of R_3, i.e., the average number of cycles spent before register R_3 is clocked. In solution tree terms, we are evaluating the average number of cycles required to traverse an edge, counting also for the deleted edges leading to a contradiction. Multiplying the result with the number of edges will give us the desired complexity estimate.

Proposition 4.4. *Assume that the bits entering the clock control of registers R_1 and R_2 and the output bits produced by registers R_1 and R_2 are mutually independent and uniformly random for all $t \geq 0$. Also, assume that each edge in a solution tree is traversed exactly once. Traversing one edge in a solution tree with deleted edges then takes 8/7 A5/1 clock cycles on average, where the average is taken over all solution trees defined by the known keystream $\mathcal{Z}(t)$ and all possible bit combinations for R_1 and R_2.*

Proof. Let cycles be the number of cycles required to traverse an edge. Processing a guess for the next clocking bit of R_3 always takes at least one clock cycle, so $\Pr[\mathsf{cycles} \geq 1] = 1$. Next, processing a guess takes at least two clock cycles if and only if the following events take place: (a) register R_3 stops for the first clock cycle, i.e., $C_1(t) = C_2(t) \neq C_3(t)$; and (b) no contradiction is found after the first clock cycle, i.e., $Z_3(t+1) = O_1(t+1) + O_2(t+1) + O_3(t)$.

Since R_3 stops with probability 1/4 and the output bit check fails with probability 1/2, we conclude that

$$\begin{aligned}\Pr[\mathsf{cycles} \geq 2] &= \Pr[C_1(t) = C_2(t) \neq C_3(t)] \cdot \\ &\quad \Pr[Z_3(t+1) = O_1(t+1) + O_2(t+1) + O_3(t)] \\ &= \frac{1}{4} \cdot \frac{1}{2} = \frac{1}{8} \ .\end{aligned}$$

In general, traversing an edge takes at least i cycles if (a) register R_3 is not clocked for $i-1$ cycles; and (b) no contradiction is found for $i-1$ consecutive checks. Therefore, the probability that traversing an edge takes at least i cycles is

$$\Pr[\mathsf{cycles} \geq i] = \frac{1}{4^{i-1}} \cdot \frac{1}{2^{i-1}} = \frac{1}{8^{i-1}} \ , \tag{4.6}$$

where the first term in the product corresponds to event (a) and the second to event (b). Conclusively, traversing an edge takes

$$\mathbf{E}[\mathsf{cycles}] = \sum_{i=1}^{\infty} \Pr[\mathsf{cycles} \geq i] = \sum_{i=1}^{\infty} \frac{1}{8^{i-1}} = \frac{8}{7}$$

cycles on average. □

Therefore, the average time required to traverse a solution tree is

$$\mathbf{E}[\mathsf{tree}] = \mathbf{E}[\mathsf{cycles}] \cdot \mathbf{E}[\mathsf{edges}] \approx 1199$$

A5/1 clock cycles. It is now straightforward to give a complexity estimate in A5/1 clock cycles for a typical software implementation.

Corollary 4.5. *Assume that the conditions of Propositions 4.3 and 4.4 hold. Assume also that all 2^{64} internal states occur with equal probability 2^{-64}. The average-case time complexity of Phase 1 in a software-oriented attack is then $2^{50.2}$ A5/1 clock cycles.*

Proof. We can expect to perform half of the tests before we find the right internal state. Multiplying the number of tests, the average number of edges in a tree, the average number of clock cycles required to traverse an edge and dividing the result by 2 yields

$$\mathbf{E}[\mathsf{T}_1^{\mathsf{soft}}] = 2^{41} \cdot \mathbf{E}[\mathsf{tree}] \cdot \frac{1}{2} \approx 2^{50.2} \ .$$

□

Finally, we remark that on average 3.2 clock cycles are spent on finding each candidate solution.

Hardware-Oriented Strategy

In the depth-first search used in the software-oriented strategy, we need to store in memory the internal states of registers R_1 and R_2 for each level in the tree, in order to be able to backtrack to the last branching point. Since the efficiency of a hardware implementation depends on the area used up by the test, a different strategy might prove useful for a hardware implementation. Namely, instead of keeping up to 11 backtracking points, it is possible to keep a simple 11-bit counter to record which 11-bit combinations for the bits of R_3 have already been checked. The state of the counter then represents 11 consecutive guesses for the bits of R_3: every time we reach a leaf node in the tree (i.e., we find a candidate solution), registers R_1 and R_2 are re-initialised and the counter is increased by 1. That is, each check is started over from the root node. For example, assume that the 11 consecutive guesses (00101010111) yield a candidate solution. Then we would proceed by setting the clocking bits of R_3 to (00101011000) for the next check. Contradictions can be processed analogously: suppose that the counter is set to (00101010111), but at level 5 we encounter a contradiction, so that (00101) cannot be completed to a full solution. Then, the remaining guesses beginning with the contradictory prefix (00101) should be skipped and the counter should be set to (00110000000) for the next check.

We proceed to compute the time complexity of the attack in A5/1 clock cycles. Note that while in case of a software-oriented strategy, each edge is traversed once, in case of the current strategy, each *path* is traversed once. This means that edges closer to the root are traversed more often. Thus, we need to recompute the expected number of cycles required to traverse an edge, taking into account the weights of the edges.

Proposition 4.6. *Assume that the bits entering the clock control of registers R_1 and R_2 and the output bits produced by registers R_1 and R_2 are mutually independent and uniformly random for all $t \geq 0$. Also, assume that each path in a solution tree is traversed exactly once. Traversing a solution tree with deleted edges then takes approximately* 6374 *A5/1 clock cycles on average, where the average is taken over all solution trees defined by the known keystream $\mathcal{Z}(t)$ and all possible bit combinations for R_1 and R_2.*

Proof. First, we show that the average number of cycles required to traverse an edge is independent of the fact whether the edge is a deleted edge (i.e., one leading to a contradiction) or not. According to (4.4), the probability of an edge being deleted is $\Pr[\mathsf{fail}] = 1/7$. Analogously, according to (4.3),

$$\begin{aligned}\Pr[\mathsf{cycles} \geq i \,\&\, \mathsf{fail}] &= \sum_{j=i}^{\infty} \Pr[\mathsf{fail}(j)] = \sum_{j=i}^{\infty} \frac{1}{8^i} = \frac{1}{8^{i-1}} \cdot \frac{1}{7} \\ &= \Pr[\mathsf{cycles} \geq i] \cdot \Pr[\mathsf{fail}] \ ,\end{aligned}$$

where the last equality is obtained from (4.6). Thus,

$$\mathbf{E}[\mathsf{cycles}|\mathsf{fail}] = \mathbf{E}[\mathsf{cycles}] = \frac{8}{7} \ .$$

Next, note that each deleted edge is traversed exactly once. For all other edges, the number of times an edge is traversed equals the number of leaves in the subtree hanging from that edge. The average number of such leaves, in turn, depends only on the level the edge is located at. Thus, the weighted average for the number of clock cycles required to traverse an edge is still $\mathbf{E}[\mathsf{cycles}] = 8/7$.

Finally, we count the total number of edge traversals $\mathbf{E}[\mathsf{trav}]$. There are two types of paths: paths of length 11 ending with a leaf and paths of various lengths ending with a deleted edge. The average number of paths of length 11 equals the average number of leaves in a tree, that is, $(12/7)^{11} \approx 376$. The average number of paths of length i ending with a deleted edge equals the average number of one-child nodes at depth $i-1$, that is, $(12/7)^{i-1} \cdot 2/7$. Summing up the traversals, we get

$$\mathbf{E}[\mathsf{trav}] = 11 \cdot \left(\frac{12}{7}\right)^{11} + \sum_{i=1}^{11} i \cdot \frac{2}{7} \cdot \left(\frac{12}{7}\right)^{i-1} \approx 5577 \ .$$

We conclude that the average time required to traverse a solution tree is

$$\mathbf{E}[\mathsf{tree}] = \mathbf{E}[\mathsf{cycles}] \cdot \mathbf{E}[\mathsf{trav}] \approx 6374 \ .$$

□

We have also verified this result by doing an implementation of the hardware-oriented strategy in software. Over 100 000 000 simulations, the average time required to traverse a tree was 6373.83 cycles (theoretical estimate 6373.60). We also remark that on average, $\mathbf{E}[\mathsf{tree}]/\mathbf{E}[\mathsf{solns}] \approx 17$ clock cycles are spent on each solution. Again, it is now straightforward to compute the total time complexity.

Corollary 4.7. *Assume that the conditions of Proposition 4.6 hold. Assume also that all 2^{64} internal states occur with equal probability 2^{-64}. The average-case complexity of Phase 1 in a hardware-oriented attack is then approximately $2^{52.6}$ A5/1 clock cycles.*

Proof. As usual, we can expect to perform half of the tests before we find the right internal state. Multiplying the number of tests, the average number of cycles per test and dividing the result by 2 yields

$$\mathbf{E}[\mathsf{T}_1^{\mathsf{hard}}] = 2^{41} \cdot \mathbf{E}[\mathsf{tree}] \cdot \frac{1}{2} \approx 2^{52.6} \quad .$$

□

Eliminating False Solutions

After a search branch has been completed and a candidate solution has been found, it also needs to be compared against further known keystream bits to check whether the candidate internal state indeed generates the expected sequence. Our final result shows that this can be done in two clock cycles on average.

Proposition 4.8. *Assume that the output sequence produced by an incorrect candidate internal state is uniformly random. Discarding an incorrect solution then takes 2 clock cycles on average, where the average is taken over all possible output sequences.*

Proof. Under the assumption that the output sequence generated by an internal state is uniformly random, we can expect this sequence to differ from the remaining known keystream already in the first bit with probability 1/2. The probability that the sequences coincide in the first bit, but differ in the second bit, is then 1/4, and, generally, the probability that the sequences first differ in the nth bit is 2^{-n}. Thus, the number of steps required to eliminate a false candidate solution is a geometric random variable with parameter $p = 1/2$ and expectation $1/p = 2$. □

In practice, one would want to set an upper limit for the total number of compared bits. Since there are 2^{64} possible internal states, 64 bits might be a suitable threshold.

We conclude the complexity analysis by comparing the total average time required to complete Phases 1 and 2 of the AKS attack using software-oriented and hardware-oriented strategies.

Corollary 4.9. *Assume that the conditions of Proposition 4.8 and Corollaries 4.5 and 4.7 hold. The overall average-case complexity of the AKS attack is then $2^{50.9}$ A5/1 clock cycles in the software-oriented case and $2^{52.8}$ A5/1 clock cycles in the hardware-oriented case.*

Proof. The average time spent on Phase 2 is in both cases

$$\mathbf{E}[\mathsf{T}_2] = 2^{41} \cdot 2 \cdot \mathbf{E}[\mathsf{solns}] \cdot \frac{1}{2} = 2^{41} \cdot \left(\frac{12}{7}\right)^{11} \approx 2^{49.6}$$

clock cycles. The overall average-case complexity of the attack is now

$$\mathbf{E}[\mathsf{T}^{\mathsf{soft}}] = \mathbf{E}[\mathsf{T}_1^{\mathsf{soft}}] + \mathbf{E}[\mathsf{T}_2] \approx 2^{50.9}$$

A5/1 clock cycles in the software-oriented case and

$$\mathbf{E}[\mathsf{T}^{\mathsf{hard}}] = \mathbf{E}[\mathsf{T}_1^{\mathsf{hard}}] + \mathbf{E}[\mathsf{T}_2] \approx 2^{52.8} \tag{4.7}$$

A5/1 clock cycles in the hardware-oriented case. □

It remains to invert the algorithm to extract the secret session key (Phase 3), however, Phases 1 and 2 will dominate in the overall complexity of the attack. Strategies for completing Phase 3 will be discussed next.

Initial State Reversion

Previously, we have described attack models for reconstructing the internal state of the cipher at the time when the cipher produces its first output bit, i.e., $100+1$ clock cycles after initialisation. In order to successfully complete the attack, it is also necessary to reverse the internal state and find the initial state of the cipher after the initialisation phase.

Suppose that the attacker succeeds in finding the initial state. Since the frame counter is publicly known, the algorithm can then be reversed to find the state after the mixing of the session key but *before the mixing of the frame counter*, i.e., 64 cycles after the start of initialisation. This reversal is straightforward, since during initialisation, registers are clocked in a regular fashion, without the stop/go clocking rule. After the effect of the frame counter has been eliminated, there is actually no need to derive the session key. The state after the input of the session key is common to all frames and can be used directly in decryption.

Denote the initial state, i.e., the state after the initialisation has been completed by $S(0)$. We are interested in recovering this unknown state from the now known internal state $S(101)$. This can be done by guessing individually for each of the three registers the number of times it has been clocked. For each such guess, the registers can first be reverted to their corresponding states in $S(0)$. Next, the A5/1 algorithm can be run forward for 101 cycles to test whether the state after 101 stop/go-clocked steps is indeed $S(101)$.

Even if we do not take into account any of the mutual clocking constraints, we know that each of the registers was clocked between 0 and 101 times. This gives us $102^3 \approx 2^{20}$ states to test. Since backwards clocking is an independent process for each register, it can be done by e.g. table look-up. Running the cipher forward with irregular clocking to obtain $S(101)$ from $S(0)$ will take 101 clock cycles. Altogether, even in the worst case and with the most naive implementation, it will take less than 2^{30} A5/1 clock cycles to find the initial state.

The search can further be made much more efficient by taking into account that at each step, each register is clocked with probability 3/4, and therefore, the expected number of clocks is $(3/4) \cdot 101 \approx 76$. Starting the search from the most probable values for each LFSR will clearly make finding a solution much more efficient.

However, it can happen that several initial states produce the same internal state $S(101)$—whereas only one of them is the correct $S(0)$ that reveals the session key after eliminating the effect of the frame counter. If one wants to take into account this possibility, then one has to perform an exhaustive search over all possible clockings. Still, due to the majority rule in clocking, there is no need to go over all possible 102^3 values. One can simply take into account that since at each cycle at least two registers are clocked, during 101 cycles the three LFSRs have to be clocked at least 202 times.

In any case, the complexity of this part of the attack is insignificant compared to that of deriving the internal state $S(101)$.

4.1.3 Discussion on a Previous Complexity Analysis

In this section, we discuss the complexity analysis of the Anderson-Keller-Seitz attack given by Keller and Seitz. First, we review their estimate on the average number of candidate solutions. Citing the analysis of [KS01]:

> If the clock bits of R1 and R2 are identical, then both registers are clocked. If we assume the clock bit of R3 to be different, then R3 will not be clocked, and its most significant bit in the next cycle will remain the same. The output bit generated by an exclusive-or operation of the most significant bits can then be compared to the bit of the output sequence $\mathcal{O}$. If they differ, then this possibility was a false one and the clock bits must be equal. If the output bit produced and the bit from the output sequence are identical, we pursue this possibility. Thus, in one half of the situations (R1 and R2 having identical clock bits) on the average,

we reduce the number of possibilities from two to one and have to check $(3/2)^{11} \approx 85$ cases.

These results are different from the results presented above, where the average number of cases is approximately 376. According to our analysis, this argumentation is not correct. Namely, if the exclusive-or of the most significant bits and the bit of the output sequence differ, then we can indeed decide that all clock bits must be equal. However, if the bits are identical, we cannot decide anything—we still need to consider both cases. This analysis therefore excludes some candidate solutions and an implementation according to this strategy may fail to find the key. According to (4.5) (c.f. Section 4.1.2), the correct value for the number of candidate solutions is 376.

Keller and Seitz state that each check inspects 14 output bits on average. Although not explained in the paper, this implies that they follow the hardware-oriented strategy, i.e., each check is started from the beginning. Since the number of inspected output bits is equivalent to the number of A5/1 clock cycles spent, they claim to spend 14 clock cycles per solution. This value is different from our theoretical estimate, which is approximately 17 clock cycles. One reason to this difference could be that we have taken into account also those cycles that lead to a contradiction and thus never yield a full solution. Simulating the attack in software, we have also found that our practical results agree with our theoretical estimate, however, we admit that there may be differences in implementations that we are not aware of.

As to the extraction of the right solution from the set of candidate solutions, Keller and Seitz state that "false possibilities can be eliminated fast". We have shown above that it takes on average 2 clock cycles to eliminate false candidates. Taking this into account, the number of clock cycles spent on each candidate solution in the hardware-oriented strategy is now 19.5 according to our analysis, or 16 according to the claims of Keller and Seitz.

4.1.4 Implementations

In this section, we compare two implementations of the Anderson-Keller-Seitz guess-and-determine attack. The first is our own implementation in software, whereas the second is an FPGA implementation by Keller and Seitz. The first attack is implemented by using software-oriented strategy; for the FPGA attack, we do not know the implementation details, but the computations of the authors imply that hardware-oriented strategy was followed (c.f. Section 4.1.2). We compare our theoretical results on the attack complexity with statistics from simulations and estimate the wall clock time complexity of our implementation on our test platform. We also give an estimate for the time requirement of the hardware implementation, however, these results should be considered less precise, since the implementation is not our own and its full details are not known to us.

A Software Implementation of the Attack

We have implemented the Anderson-Keller-Seitz guess-and-determine attack in software, using the first strategy, i.e., doing a depth-first search, while keeping the intermediate states in memory. All the following statistics are averaged over 100 million test runs with pseudorandom selections of the states of registers R_1 and R_2, and of the output keystream. The simulations were run on a PC with the following parameters:

CPU:	Intel Celeron M, 1300MHz, Banias-512 core
Memory:	256 MB
Operating system:	Windows XP Home Edition SP 1
Compiler:	GCC v.3.3.3 (cygwin special), optimisation level O4

Table 4.1: Test platform for the AKS attack in software

The first part of the attack constitutes of deriving all candidate solutions of register R_3 for given register states R_1 and R_2, and known keystream. In our implementation, on average 375.759 candidate solutions were found for each test, with 375.764 being the theoretical estimate. Finding one candidate solution took 3.1916 A5/1 clock cycles on average, whereas our theoretical estimate was 3.1915 cycles. Thus, both results agree almost fully with the theoretical results presented above.

In the second part of the attack, each candidate solution was checked against 64 bits of known keystream. Each such check took 1.999993 cycles on average, which is again fully consistent with the theoretical result (exactly 2 cycles) under the randomness assumption.

The time consumed by the first part of the attack was 6 minutes for 100 million simulations. Since on average, 2^{40} tests need to be performed before the right state is found, our simulation covers a fraction of $2^{-13.4}$ of the whole attack. Therefore, the first part of the attack is expected to take slightly over one year on a single PC with these parameters.

The second part of the attack—eliminating false possibilities—took 5 minutes for the same number of simulations. Hence, this part of the attack would roughly double the attack time.

The total time consumption of the attack was 11 minutes for 100 million simulations. This means that one test machine would find the correct state in

$$T = \frac{2^{40} \cdot 11}{10^7} \text{ minutes} \approx 840 \text{ days} \ ,$$

i.e., in less than 2.5 years. Therefore, we can expect 1000 such PC-s to find the key in less than one day, using only one frame of known plaintext.

Some notes on optimisation

To speed up our implementation, we clocked the registers one byte at a time, not one bit at a time. Since the contents of the LFSRs were stored in 32-bit registers but the longest shift register is only 23 bits long, we used 8 of the remaining bits to "buffer" the next bits of the registers. Consider, for example, the second shift register. If the contents of register R_2 at time t are

$$(b_{21}, b_{20}, \ldots, b_0)$$

from the most significant bit (output bit) to the least significant bit, then the next 8 bits that we store to the least significant bits of a 32-bit register are

$$\begin{aligned} &(b_{21} \oplus b_{20}, b_{20} \oplus b_{19}, \ldots, b_{14} \oplus b_{13}) \\ = &(b_{21}, b_{20}, \ldots, b_{14}) \oplus (b_{20}, b_{19}, \ldots, b_{13}). \end{aligned}$$

Similar equations can be written for the other two registers. Now, we do not need to update the least significant bit of the registers at each clocking. Instead, we can update the 8 extra bits of the registers every 8 cycles, and at each clocking, simply shift the registers one bit to the left. The cost of updating 8 bits is implementation-wise equivalent to updating just one bit.

An FPGA Implementation of the Attack

We will next give a time complexity estimate for the hardware implementation. Since we use measurement data from the paper by Keller and Seitz combined with our theoretical estimates, these results should not be considered as exact implementation measurements. Rather, they give a rough estimate on the efficiency of an FPGA-based implementation compared to a software implementation.

We start with the following notations. Let the real clock frequency of the FPGA (in A5/1 clock cycles) be F cycles per second. Note that this value can be significantly lower than the theoretical maximum frequence of the device. Furthermore, assume that P instances of the implementation can be fitted on one FPGA chip, each doing a share of the 2^{41} tests, so that P is the degree of parallelisation. According to (4.7), the average-case complexity of the attack is $\mathbf{E}[\mathsf{T}^{\mathsf{hard}}] = 2^{52.8}$ A5/1 clock cycles. Thus, given these parameters, the expected wall clock time to complete the attack is

$$T = \frac{\mathbf{E}[\mathsf{T}^{\mathsf{hard}}]}{F \cdot P} \ ,$$

In other words, in order to complete the attack in less than T seconds, the parameters of the FPGA need to satisfy

$$F \cdot P \geq \frac{2^{52.8}}{T} \ .$$

In particular, the attack time of our implementation is $2^{26.1}$ seconds. Thus, in order for the implementation to be faster than our software implementation on our test platform, we need

$$F \cdot P \geq 2^{26.7} \ .$$

The FPGA used for implementation by Keller and Seitz had the following basic parameters:

FPGA type:	Xilinx XC4062
Configurable Logic Blocks (CLBs):	2304
Theoretical system performance:	80 MHz [Xil99]

Table 4.2: Test platform for the AKS attack in hardware

The implementation by Keller and Seitz occupied 313 of the FPGA's 2304 CLBs, so $P = \lfloor \frac{2304}{313} \rfloor = 7$ instances were fitted on one chip. The achieved clock frequency was $F = 18.65$ MHz, which is less than 25% of the theoretical maximum performance. We see that

$$F \cdot P = 7 \cdot 18.65 \cdot 10^6 = 2^{27.0} \ ,$$

so the hardware implementation has only marginal advantage over our implementation on our test platform. More precisely, the hardware implementation is faster by a factor of 1.2, so an attack would be completed in 700 days, slightly less than 2 years.

We see that the FPGA used in 2001 is not significantly faster than our PC (and even in 2001, a PC with parameters similar to our test machine would already have been available). On the other hand, an FPGA attack implemented now on a modern device would benefit not only from higher clock frequency but also from a larger number of available CLBs, which tends to grow faster. A Xilinx Virtex-II Pro XC2VP125 device, as suggested by Kairus, has 55616 CLBs and would therefore be $55616/2304 \approx 24$ times faster [Kai03]. The attack time would be guaranteed to drop to a month, even without taking into account any other benefits of a newer device. In addition, the device has better system performance—unfortunately, the gap between theoretical performance and implementation-specific clock frequency makes the benefit rather impossible to predict. As the XCVP125 device is said to achieve frequencies of 300 MHz, we may speculate that it should be at least an additional $300/80 \approx 4$ times faster. Hence, one such device would complete the attack in a week.

Finally, we would like to stress that these comparisons should be taken with a heavy grain of salt. Namely, we have only compared the *time* and ignored the second component important for the attacker—the cost of implementation. In order to estimate the precise advantage of an FPGA over a PC, we would need to compare the "value for price". That is, the question we should ask is, "Given the same amount of money for implementation costs, which one is faster, an FPGA or a PC?" Such analysis is beyond the scope of this thesis, but we hope that the comparison given above has given the reader a rough idea of the effort needed to break the A5/1 cipher.

4.2 Biham-Dunkelman Attack

In the previous section, we considered perhaps the most basic attack against a stream cipher, namely, a guess-and-determine attack. As the name implies, such attacks constitute of guessing some part of the secret key (or, equivalently, cipher internal state) and determining the rest of the unknown bits by using known keystream.

Next, we look more deeply into the structure of the A5/1 cipher, and point out certain very specific weaknesses that can be exploited to attack the cipher. Tricks and techniques presented in this chapter are due to Biham and Dunkelman [BD00]. We describe their attack model, but we also explain how it is related to the previous ideas and show how the different approaches can be combined. Furthermore, we generalise their attack by proposing a trade-off curve between computational complexity and plaintext requirement. This original result provides the attacker with more flexibility in scenarios where the rather large plaintext requirement of the Biham-Dunkelman attack (also, BD attack) might previously have been the main bottleneck.

4.2.1 Key Ideas of the Attack

The Biham-Dunkelman attack, as all previously presented attacks, attempts to recover the internal state of the cipher during its operation. It operates in the known plaintext model. The attack is expected to be a thousand times faster than the Anderson-Keller-Seitz attack, so the expected time complexity is less than a day on a PC and a few minutes on an FPGA; also, it does not require any precomputation. On a downside, the attack requires up to half a minute of known plaintext.

The key idea of this attack is to wait for an event that leaks a large amount of information about the internal state. It turns out that a situation where one of the registers is not clocked for a long time is such an event. Assume that we know a time instance where the third register does not move for 10 clock cycles. Denote the contents of the three registers at this time point by $R_1[18], \ldots, R_1[0]$, $R_2[21], \ldots, R_2[0]$ and $R_3[22], \ldots, R_3[0]$, with $R_1[18]$, $R_2[21]$ and $R_3[22]$ being the output bits and $R_1[8]$, $R_2[10]$ and $R_3[10]$ the clocking bits.

At such a time point, we are able to recover the state of the first two registers plus two bits from the third register by guessing only 12 bits. Namely, for those 10 clock cycles that the third register does not move, the 10 consecutive clocking bits of R_1 and R_2 are the complement of the (one and the same) clocking bit of R_3. Hence, by guessing bit $R_3[10]$, we can determine the following bits: $R_1[8], \ldots, R_1[0]$ and $R_2[10], \ldots, R_2[1]$. Additionally, we will know the next bit to enter the least significant position of R_1, i.e., the bit $R_1[-1] = R_1[18] \oplus R_1[17] \oplus R_1[16] \oplus R_1[13]$ (see Figure 4.4, top).

Next—and this is where the information leakage takes place—the same output bit of R_3 will contribute to 11 consecutive rounds of output. So, we next guess this output bit, $R_3[22]$. We already have 10 bits of information about register R_1, so by guessing 9 more bits ($R_1[17], \ldots, R_1[9]$), we can recover all of R_1. (Bit $R_1[18]$ can be recovered from the known bit $R_1[18] \oplus R_1[17] \oplus R_1[16] \oplus R_1[13]$.) Since we now know the output bits of both R_1 and R_3, this will immediately reveal the output bit of R_2, $R_2[21]$. Similarly, during the next 10 clock cycles at which register R_3 is not clocked (and, conversely, both R_1 and R_2 are clocked) the only unknown bit in the output exclusive-or will be the bit from the second register. With the aid of known keystream, we can recover 10 more bits of register R_2 ($R_2[20], \ldots, R_2[11]$) during those 10 cycles. Now, we have recovered 21 bits of R_2, so it will take one more guess (bit $R_2[0]$) to determine the full contents of the second register (see Figure 4.4, bottom).

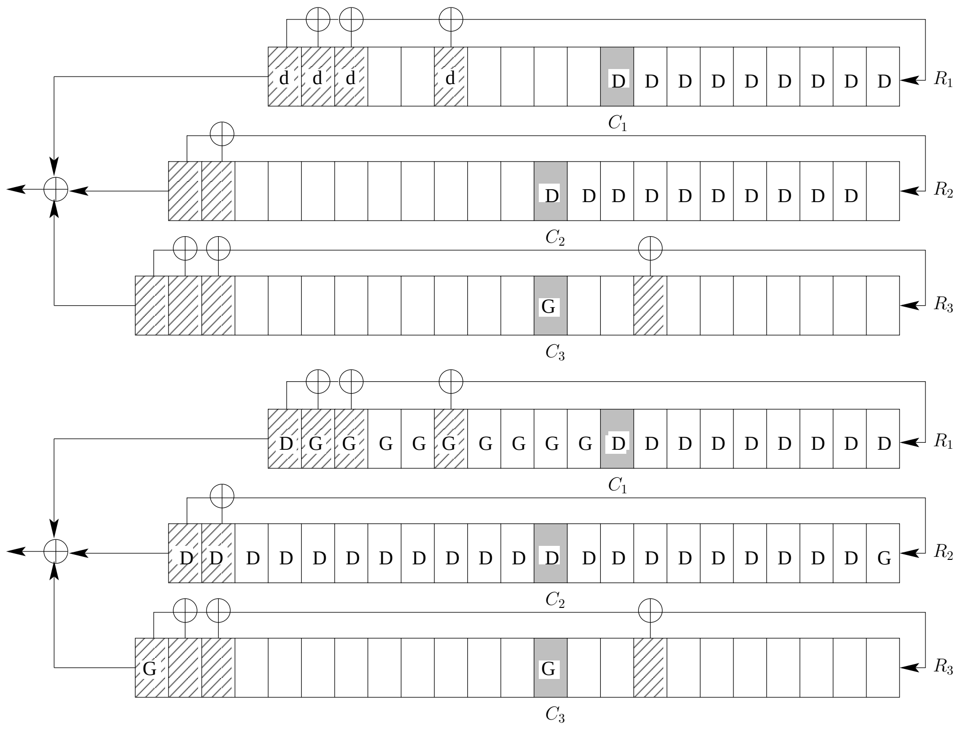

Figure 4.4: Top: Guessing the clocking bit of R_3 (marked with G) determines the 19 clock bits marked with D and the sum of the four bits marked with d. Bottom: Guessing altogether 12 bits of the internal state (marked with G) determines 31 bits of the state (marked with D).

We have now recovered all the bits of R_1 and R_2, and two bits from R_3 by guessing 12 bits. In other words, this event has gained us $19+22+2-12 = 31$ bits of information about the internal state. Of course, things are not quite as simple as they appear, since we were working under the assumption that we know the location of the information-leaking event. Such an event will happen in one out of 2^{20} possible cipher states. Thus, we will need to probe about 2^{20} different locations (from many different frames) by trial-and-error before the event actually occurs.

We are now actually in a situation familiar from earlier attacks. In the Anderson-Keller-Seitz (AKS) guess-and-determine attack model (c.f. Chapter 4.1), we guessed the contents of two registers and proceeded to determine the contents of the third register. Here, we have also determined the contents of registers R_1 and R_2, albeit in a different manner, so we could proceed exactly as in the AKS attack model to complete the attack (with the minor difference that we already know 2 bits from R_3, bits $R_3[22]$ and $R_3[10]$).

4.2.2 A Trade-Off Between Computational and Plaintext Complexity

As we saw in the previous section, certain events cause the A5/1 cipher to leak information about its internal state. In other words, if the cipher reaches a certain internal state, this state can be recovered with less effort than in an ordinary guess-and-determine attack. Thus, in this section, we address the following three questions:

- How much do we gain in computational cost by waiting for the event when the third register is not clocked for $n = 10$ cycles?
- What is the price that we pay for this gain in known plaintext?
- Is it possible to extend this attack for other values of n?

Consider first the question of computational cost. Recall that in order to locate an event where R_3 stays put for 10 cycles, 2^{20} different starting locations need to be probed on average. For each such location, guessing 12 bits immediately reveals 31 more bits. Hence, the 41 bits of registers R_1 and R_2, plus 2 bits from register R_3, can be determined with effort 2^{32}. As a comparison, in the Anderson-Keller-Seitz attack model, we exhaustively searched through all 2^{41} possible values of registers R_1 and R_2, and obtained the additional 2 bits of R_3 with one guess (guessing the clocking bit and determining the output bit). Thus, for the same 43 bits of internal state, we need to consider 2^{42} possibilities in the AKS model, compared to only 2^{32}

possibilities in the in the BD model. Conclusively, we are able to speed up our attack by a factor of more than 1000.

Naturally, this advantage does not come for free. We pay for the speed-up in required plaintext. In the AKS attack, one or two frames of keystream were sufficient. In this attack, we need 2^{20} different starting locations. One frame contains 228 bits and as we need about 64 bits to uniquely identify the right solution, $228 - 63 = 165$ bits in one frame can serve as a possible starting location. Hence, the plaintext requirement is approximately

$$D = \frac{2^{20}}{165} \approx 2^{12.6} \text{ frames} \ ,$$

or approximately 30 seconds of conversation. [2]

Finally, we address the third question. Namely, in real-life scenarios, obtaining 30 seconds of conversation plaintext is rather infeasible. Thus, we would like a trade-off between computational cost and plaintext requirement. For example, we could search for a situation where the third register stays put for, say, only $n = 5$ clock cycles. This event is much more frequent but still leaks some information. The following original result gives a trade-off curve for $0 \leq n \leq 10$.

Proposition 4.10. *Assume that at each time during the operation of the cipher, each 64-bit internal state occurs with equal probability* 2^{-64}. *Fix* n *to be an integer,* $0 \leq n \leq 10$. *Then, 43 bits of the A5/1 internal state can be determined with only* 2^{42-n} *bit guesses, using an expected amount of* $2^{2n}/165$ *frames of known plaintext.*

Proof. The attack will be based on the information leak from the event when the third register does not move for n clock cycles. Denote the contents of registers R_1, R_2 and R_3 by $R_1[18], \ldots, R_1[0]$, $R_2[21], \ldots, R_2[0]$ and $R_3[22], \ldots, R_3[0]$, respectively, where $R_1[18]$, $R_2[21]$ and $R_3[22]$ are output bits and $R_1[8]$, $R_2[10]$ and $R_3[10]$ are clocking bits. Then, the desired event happens if and only if

$$\begin{aligned} R_1[8] &= R_1[7] = \cdots = R_1[8-(n-1)] \\ &= R_2[10] = R_2[9] = \cdots = R_2[10-(n-1)] \neq R_3[10] \ , \end{aligned}$$

where $R_1[-1] = R_1[18] \oplus R_1[17] \oplus R_1[16] \oplus R_1[13]$. Note that if any one of the $2n+1$ bits present in this condition is fixed, the remaining $2n$ bits are immediately determined. Thus, one out of 2^{2n} internal states satisfies this

[2]The authors of [BD00] state that the plaintext requirement is over 2 minutes, which is incorrect according to our computations.

property, so in order to locate our event, on average 2^{2n} locations need to be probed. If we require 64 bits of output to uniquely determine the right solution, we are left with 165 possible starting locations in each frame. The claim on plaintext requirement follows.

Now, we attempt to recover all bits of R_1 and R_2, plus the first clocking bit and first output bit of R_3, 43 bits in total. First, guessing the clocking bit of the third register (bit $R_3[8]$) reveals n consecutive clocking bits of the first two registers (bits $R_1[8], R_1[7], \dots, R_1[8-n]; R_2[10], R_2[9], \dots, R_2[10-n]$), since these have to be the complements of the guessed bit. This gives us $2n$ bits of information about the two registers. Next, guess the output bit $R_3[22]$ of R_3 and all the remaining $19-n$ bits of R_1. Then, $R_2[21]$ can be determined from $R_3[22]$, $R_1[18]$ and the known keystream bit. Also, during the next n clock cycles, the output bit of R_3 remains the same, so the output bit of R_2 can again be determined from known keystream and the outputs of R_1 and R_3. Thus, we can recover $n+1$ bits of R_2. It remains to guess the remaining $22-n-(n+1) = 21-2n$ bits of R_2.

We see that of the 43 bits, only $1+1+(19-n)+(21-2n) = 42-3n$ bits need to be guessed. The procedur needs to be repeated for 2^{2n} different locations, giving a total effort of 2^{42-n}, as desired. □

Compared to the AKS guess-and-determine attack, this attack is faster by a factor of 2^n for the selected n. Of course, the trade-off curve cannot be extended to infinity: the third register cannot stay put infinitely (unless registers R_1 and R_2 are both in the all-zero state). Even increasing the value of n above 10 is not reasonable, as the plaintext requirement grows prohibitively high. On the other hand, decreasing the value to, say, $n = 8$, still gives a speed-up of about $2^8 = 256$ times, whereas the plaintext requirement is reduced to less than two seconds. In general, increasing the computational complexity by a factor of two reduces the plaintext requirement by a factor of four.

Figure 4.5 depicts various points on the trade-off curve. Data requirement in frames has been converted to conversation seconds. The endpoints of the curve correspond to the AKS attack and the original BD attack. Finally, we remark that it is possible to speed up the AKS attack 8 times without increasing the average plaintext requirement. This is due to the fact that for generalised BD attack with $n = 3$, we need on average $2^6 = 64$ different starting locations, which can be obtained from one frame. However, we must keep in mind that this is the *average* plaintext requirement of the BD attack, whereas the AKS attack *always* succeeds, using only one frame of data.

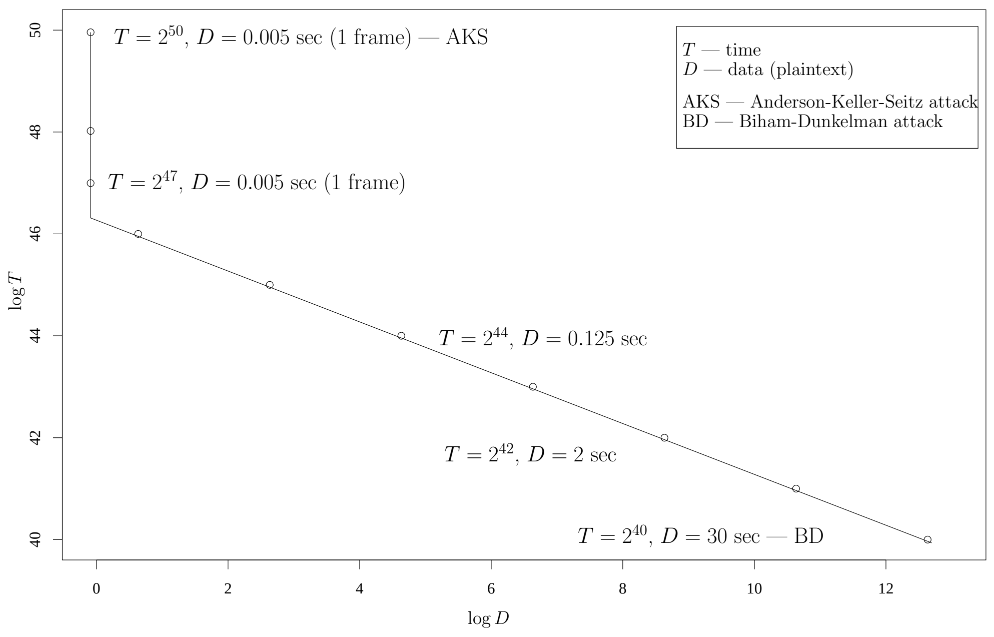

Figure 4.5: A trade-off between computational complexity and plaintext requirement

4.2.3 Possibilities for Hardware Implementation

The Biham-Dunkelman attack is very similar to the Anderson-Keller-Seitz attack presented in [KS01]: the key idea of the attack is to guess the contents of registers R_1 and R_2 and determine the contents of register R_3 from known keystream. Since the search space for R_1 and R_2 can be split into smaller subspaces, the attack is easily parallelisable and is thus suitable for hardware implementation. Actually, the AKS attack and the BD attack are essentially identical implementation-wise. The only difference lies in the way in which the contents of the first two registers are initialised: in the AKS case, this is a simple exhaustive search; for the BD attack, an additional assumption limits the possibilites, and the search is performed over multiple locations of keystream. After initialisation of R_1 and R_2, the state of R_3 can be determined, using the familiar algorithm depicted in Figure 4.1 of Section 4.1.

In Section 4.1.4, we estimated that a modern FPGA should be able to complete the AKS guess-and-determine attack in a week. The BD attack is roughly 1000 times faster. Thus, in the Biham-Dunkelman attack model, finding the secret session key should be a question of minutes on a single hardware device. Naturally, the attack is valid only if the attacker is in possession of a sufficient amount—approximately 30 seconds—of plaintext, which is a serious drawback in real-life scenarios.

4.3 Golić Attack

The Anderson-Keller-Seitz guess-and-determine attack (from now on, AKS attack) described in Section 4.1 was informally proposed by Anderson in 1994. The first cryptanalytic paper on A5/1 by Golić was published in 1997. In this paper, Golić states that the AKS attack based on guessing the contents of two registers cannot work, because the clocking depends on the third unknown register as well. (We have seen that it actually can work, only the complexity is higher than initially suggested.) Golić proposes a new attack which is said to achieve average-case computational complexity of around 2^{40} tests. However, this complexity cannot be directly compared to the complexity in the AKS model—$2^{48.6}$ tests—since deriving a solution in the Golić model additionally requires solving a system of 64 linear equations in 64 variables. An implementation by Pornin and Stern leads us to believe that a straightforward software implementation of the Golić attack is not remarkably faster than the AKS attack; however, they also propose a clever mixed software-hardware implementation that significantly outperforms a software-only attack [PS00]. We proceed to describe the theoretical attack model and the implementations.

4.3.1 Key Ideas of the Golić Attack

The main idea of Golić's guess-and-determine attack is again to guess certain bits of the internal state and determine the remaining bits from known keystream. The attack makes use of the fact that the only source of nonlinearity in the cipher is the irregular clocking rule. Once we know the clocking sequence, each bit of known keystream can be related to three bits of the internal state via a linear equation of the form $x + y + z = c$, where x, y, z are variables and c is constant. Thus, the key idea of the attack is to treat the internal state as a set of 64 variables and collect enough linearly independent linear equations to uniquely solve the system. In order to recover the full 64-bit internal state, at least 64 linearly independent equations are needed.

Of course, things are not quite as simple, since the clocking sequence is not known. Thus, the attack begins by guessing a number of bits entering the clock control. Each guessed bit also adds an equation of the form $x = 0$ or $x = 1$ to the system. The number of guessed bits required to construct a sufficient amount of linear equations then determines the complexity of the attack.

Let us first guess the "lower half" of each of the three registers (9, 11 and 11 bits, respectively). Fixing the values of these bits gives the first 31 equations. Also, since the first output bit can be expressed as an exclusive-or of the three most significant bits of the registers, we obtain another equation for those three bits.

Next, since the probability of a register to be clocked is 4/3, it will be possible to clock the registers for another $(4/3) \cdot 9 \approx 12$ cycles on average, before we "run out" of guessed bits in register R_1. The output bits from these clock cycles will give rise to another 12 equations. Now, we have obtained a system of approximately $1 + 31 + 12 = 44$ linear equations.

Let us also verify that these equations are mutually linearly independent. After guessing 31 bits, the three registers still contain 10, 11 and 12 unknown bits in the most significant bit positions, respectively. One bit from each register will be used in the first equation obtained from the very first output bit, which leaves us 9, 10 and 11 unused bits. During the next 12 clock cycles, we clock each register an expected number of 9 times. Since all registers have at least as many unknown bits, a new unused bit will appear in each of the equations with high probability, and hence the equations will all be linearly independent (mutually, and also from all previous equations).

To sum it up, we now have 44 bits of information about the internal state. Therefore, guessing 20 of the unknown bits would suffice to derive the whole state. However, this would give a total computational complexity of $2^{31+20} = 2^{51}$ tests, which is even higher than in the AKS model.

Instead of going through all 2^{20} possible values, Golić proposes to continue guessing clocking bits, which on the other hand can be expressed as linear combinations of our variable bits by the property of an LFSR. The advantage here is that we can reduce the number of possible clocking triples by taking into account constraints from previous equations. Namely, once the initially guessed bits move to the most significant positions, we can start comparing known keystream bits against known output to eliminate false possibilities.

To better understand the attack, let us look at an illustrated example. Figure 4.6 displays two A5/1 internal states. The upper figure depicts the beginning of the attack, where the 31 clocking bits of the registers have been guessed. Each clock cycle gives a new linear equation in the unknown bits marked with '?'. The lower figure illustrates the situation 9 clock cycles later. We have now run out of clocking bits for register R_1. The next guessed bit (circled in the figure) can be expressed as a linear combination of previous bits. We still need to wait for R_1 to be clocked once, R_2 to be clocked 4 times and R_3 to be clocked 6 times, before we can start comparing known keystream against known output from the LFSRs.

Golić suggests that the whole complexity of the attack is approximately

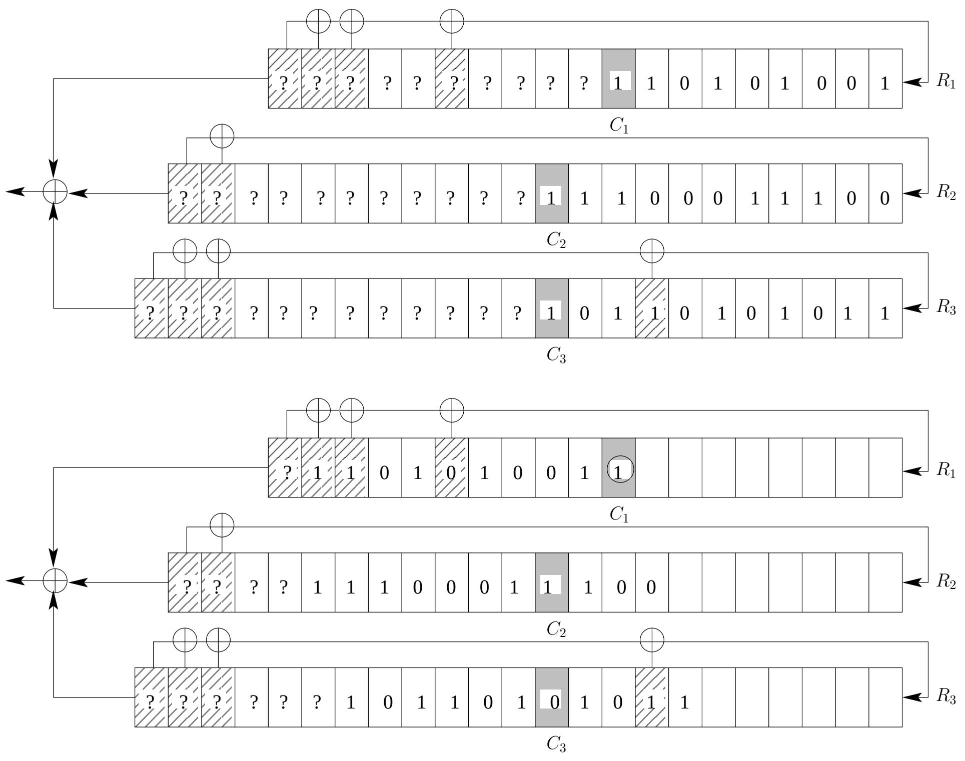

Figure 4.6: The partial internal state of the registers in the beginning of the Golić attack and 9 clock cycles later

2^{41}. Zenner has given a more detailed analysis and concludes that the average-case complexity is between 2^{42} and 2^{43} [Zen99]. This is somewhat lower than the average-case AKS attack complexity $2^{48.6}$. However, recall that each test in the Golić model gives a system of 64 linear equations in 64 variables, so we also need to take into account the time required to solve the system.

Both Golić's attack and the Anderson-Keller-Seitz attack based have the following ideas in common:

- Guess partial contents of the LFSRs, so that the clocking sequence is determined;
- At each clock cycle, impose restrictions on the remaining unknown bits from known keystream;
- At each cycle, also check that the previous guesses are consistent with known keystream;
- Check each candidate solution against further bits to eliminate false candidates.

The main difference of the two approaches lies (a) in the selection of initially guessed bits; and consequently (b) in the way the information from known keystream is used. In the AKS attack, bits are derived directly, whereas in the Golić attack, keystream bits are used to derive linear equations in bits of the internal state.

4.3.2 Implementations

In this section, we describe implementations of the Golić attack by Pornin and Stern. The authors propose two approaches: a software-only attack and a mixed software-hardware implementation. The average-case wall clock time complexity of a software-only attack is 200 days on their test platform. This is already somewhat better than our implementation of the AKS attack (840 days), but of course, since we use a different test platform, these results cannot be benchmarked.

The second implementation proposed by Pornin and Stern is a software-hardware tradeoff. The idea here is to derive partial solutions on a software platform and complete the attack by exhaustive search on a dedicated hardware device. For the Golić attack, this means that instead of constructing a full system of 64 linearly independent linear equations, a smaller system is constructed. Such a system is underdetermined, which means that there are multiple solutions satisfying the system. The dedicated hardware device then searches through these solutions exhaustively.

When constructing the partial systems of equations, Pornin and Stern take a slightly different approach from Golić. Instead of guessing clocking bits, they guess which registers are clocked and express this information also through linear equations. For example, if registers R_1 and R_2 move and register R_3 stays, this can be expressed as

$$\begin{aligned} c_1 \oplus c_2 &= 0 \\ c_1 \oplus c_3 &= 1 \ , \end{aligned}$$

where c_1, c_2 and c_3 are the clocking bits of R_1, R_2 and R_3, respectively. Using this approach, each clock cycle gives 2 bits of information about the clocking bits, since there are 4 possibilities for the clocking sequence. The known bit of output gives another equation, so each clock cycle produces three equations.

As an example, suppose that the cipher is run in software for 18 clock cycles. Each cycle, there are 4 possibilities for selecting the clocking sequence, so the total software load is $4^{18} = 2^{36}$, i.e., we obtain 2^{36} different systems of equations. From 17 cycles, $3 \cdot 18 = 54$ equations are generated, so there are 2^{10} solutions to each system. The hardware then probes all of them, giving a total hardware load 2^{46}. For this particular configuration, Pornin and Stern's software platform completes its job in 0.85 days and the FPGA platform requires 2.5 days.

In order to see the improvement, we can give a nice estimate. Namely, for an attacker, the deciding factor is not just the attack time, but the ratio of time and *cost*. Pornin and Stern state that the cost of their FPGA platform is about 2/3 the cost of the software platform (at the time of writing the article). As we saw, a single test PC would complete the attack in 200 days on average, so three such PCs would require 67 days. For the same amount of money, the attacker could instead purchase one PC and three FPGAs, which would finish the attack in 0.85 days, so there is approximately a factor 80 improvement in performance.

Chapter 5

Other Attacks on A5/1

In this chapter, we give an overview of other, more recent attacks against the A5/1 cipher. One section is dedicated to both time-memory trade-off attacks and correlation attacks. While these attacks are generally faster than guess-and-determine attacks, their plaintext requirement tends to be prohibitively high for real-life scenarios. Thus, in the final section of this chapter, we investigate possibilities for attacking the A5/1 cipher in a ciphertext-only scenario, where the attacker only has access to encrypted communication.

5.1 Time-Memory Trade-Off Attacks

This section focuses on time-memory trade-off (TMTO) attacks against the A5/1 cipher. TMTO attacks against cryptographic primitives date back to 1980, when Hellman proposed a method for attacking DES and block ciphers in general [Hel80]. At the time, it was not seen how this method could be employed to attack stream ciphers. In 1995, Babbage proposed a TMTO attack model for stream ciphers [Bab95]. The same method was discovered independently by Golić, applying the attack in particular to A5/1 [Gol97].

In this section, we describe a more recent work by Biryukov et. al. [BSW01]. They propose two attacks against the A5/1 cipher: the first is an enhancement of the Babbage-Golić attack, whereas the second is actually an application of the original Hellman method.

5.1.1 Key Ideas of A Time-Memory Trade-Off Attack

Time-memory trade-off attacks are, once again, internal state recovery attacks in the known plaintext model. The task of the attacker is thus to find the internal state that generates a given keystream sequence. A rough approach would be to exhaustively search through all possible internal states until the right keystream sequence is encountered. TMTO attacks make use of this idea but apply it in a more clever way. In a nutshell, a TMTO attack against A5/1 goes as follows. First, perform part of the exhaustive search, that is, select a subset of the 2^{64} internal states, compute for each state the output sequence it generates, and store the results. If we want each output sequence to correspond to one internal state only, 64 bits is a suitable output sequence length. Note that this part of the attack is independent of the actual observed keystream, so it must be done only once. We call this part the precomputation stage of the attack. In the real-time phase of the attack, the attacker then simply observes generated output until a previously stored sequence appears. Then, the corresponding internal state can be immediately determined from the stored results. Clearly, the more state-output pairs we store during precomputation, the shorter the waiting time until a stored keystream sequence actually appears in the cipher's output. This is where TMTO attacks get their name: there is a trade-off between precomputation memory requirement and real-time phase attack time, or, even more importantly, known plaintext requirement.

So far, we have not used any specific properties of the A5/1 cipher. Indeed, TMTO attacks can in principle be applied to any stream cipher. Since the search is performed over internal states rather than secret keys, TMTO attacks are more efficient than exhaustive search for ciphers with a relatively short internal state size. In particular, the internal state of the A5/1 cipher is only 64 bits long, or as long as the secret key. For modern stream ciphers, the recommended state size is at least twice the size of the key [ECR]. Let us now see how the size of the state affects the efficiency of a TMTO attack.

The Birthday Paradox

Suppose that we decide to store on disk the output sequences for M internal states. The question we would now like to ask is, "How long do we have to wait before we observe a sequence that is also stored on our disk?". The following variation of the well-known Birthday Paradox gives an answer to this question.

Lemma 5.1 (Extended Birthday Paradox). *Let $M \subset \{1, 2, \ldots, n\}$ be a set of $|M| = m$ integers. Draw t integers uniformly at random from the range $[1, n]$. If the parameters m, t and n satisfy*

$$m \cdot t \geq n \ ,$$

then it is likely that amongst the t randomly selected integers there is an integer that collides with an element of M.

Proof. First, we find the probability that there is no collision. The probability that one integer drawn uniformly at random from $[1, n]$ does not collide with the elements of M is $(n - m)/n$. The probability that none of the t integers collide with the elements of A is thus $((n - m)/n)^t$. We conclude that the probability of a collision is

$$\Pr[\mathsf{collision}] = 1 - \left(\frac{n-m}{n}\right)^t \ .$$

For m small compared to n, we can roughly estimate

$$\Pr[\mathsf{collision}] \approx 1 - e^{-\frac{mt}{n}} \geq 1 - e^{-1} > 0.5 \ .$$

□

Mathematically, this is not a rigorous proof—we cannot conclude that the value of the probability "likely" is always greater than 0.5 or any other desired constant. For now, we just say that the collision probability is greater than 0.5 for a sufficiently large n and for a reasonable selection of m and t.

In the case of A5/1, the number of possible 64-bit sequences is $n = 2^{64}$. The set of stored 64-bit output sequences corresponds to the set M, and the observed outputs correspond to the t random selections. The task is to find a point (m, t) on the trade-off curve $m \cdot t = 2^{64}$, such that m is a feasible memory requirement and t is a feasible plaintext requirement. For fixed m and t, the TMTO attack is probabilistic, i.e., the attack will succeed with a given probability $p = \Pr[\mathsf{collision}] < 1$. In comparison, guess-and-determine attacks (with the exception of Biham-Dunkelman's attack) have success probability $p = 1$.

Next, we describe the key ideas of two different TMTO attacks on A5/1. The biased birthday attack has suggested parameters m = 200 GB and t = 2 minutes; the random subgraph attack is more powerful, requiring only 2 seconds of conversation at the expense of only slightly larger memory requirements (400 GB).

5.1.2 Biased Birthday Attack

Sampling of State-Output Pairs

First, we describe the process of selecting the state-output pairs that will be stored during precomputation.

In order to save up on disk space, it would be nice to select states producing output with a common prefix, as it is then not necessary to store the common part for each pair. For example, consider a 16-bit pattern α

(we will see the motivation for choosing this particular length in a moment). We would like to find all of the approximately 2^{48} states that generate a sequence starting with α. On the other hand, we would like the process to be faster than the 2^{64} complexity of going through all possible internal states and discarding those that do not produce a sequence starting with α.

Flaws in the design of A5/1 make it possible to sample those 2^{48} states in 2^{48} time. Namely, the placement of the clocking bit makes the bits that affect the clock control and those that affect the output unrelated for about 16 clock cycles. The algorithm for generating these states bares resemblance with the algorithm used in Golić's guess-and-determine attack described in Section 4.3 of the previous chapter. Exactly as in Golić's attack, one can guess the least significant bits of the three registers, starting with the bits that determine the clock control. All possibilities for the three most significant bits that result in the right output value can then be identified at each clock cycle. This process can be continued without more complicated trial-and-error as long as at least one of the output bits is undetermined. Since the three registers move on average 3/4 of the time, and have 10, 11, and 12 undetermined bits in the beginning, respectively, this process can indeed be continued for about 16 clock cycles in the same simple manner. As a conclusion, it is possible to generate all internal states producing a sequence starting with α without exhaustively searching over the set of all internal states. While trying all the 2^{64} states is computationally infeasible, a 2^{48} preprocessing stage is already barely doable.

Disk Storage

During the preprocessing stage, we are interested in storing state-output pairs, preferably in such a way that the stored output is long enough to identify a unique internal state. The authors of [BSW01] propose to limit the sequence length to 51 bits instead of 64 bits, claiming that this is usually sufficient for unique identification. When considering disk storage, the first 16 bits of output are now constant and need not be stored. The next 35 bits can be stored in sorted order—which means that it is sufficient to store only the increments. Also, the 64-bit internal states can be encoded with shorter, 48-bit names (since there are only about 2^{48} of them). Further, if a few bits are left unspecified (at the expense of trying a small number of candidates later), each state-output pair can be stored in 6 bytes. Storing all 2^{48} pairs would still require unrealistic amounts of space, but a subset of 2^{35} pairs requires less than 200 GB of storage and can easily be fitted on one commercially available hard disk. We shall later see that 2^{35} pairs are sufficient to succeed in the attack.

Biased Sampling

The key to the success of the attack lies in the clever selection of the subset of pairs. Namely, it turns out that not all internal states have an equal probability of occurrence. Selecting states with a higher probability significantly increases the success probability.

More precisely, if an internal state occurs while the cipher is producing output, then at least 101 clock cycles must have passed from the end of the initialisation (the cipher simply does not produce any output earlier). On the other hand, since our identifying output is $16 + 35 = 51$ bits long, the cipher must run further for at least 51 cycles and we cannot use any of the last 51 states. Consequently, if the state at the end of the initialisation is $S(0)$, then the states that we can consider in the attack are between $S(101)$ and $S(278)$.

Conversely, if an internal state occurs while the cipher is producing output, then it has to be reachable from some initial state with at least 101 and at most 278 cycles. Through extensive sampling, Biryukov et. al. made a crucial observation that about 85% of the states can *never* be reached with more than 100 steps. The reason behind this is the non-bijective state update function. This means that a vast majority of all possible internal states actually never occur while the cipher is producing output, and storing those states on the disk is not helpful at all. Moreover, there was a huge variance amongst the remaining 15% of states—some could be reached from only one initial state, whilst others had over 26,000 descendants. Figure 5.1 illustrates this situation. The upper dots represent the internal states and the black area represents the corresponding candidate initial states. The tree of the leftmost state dies out without reaching depth 101; the middle state can be reached from many initial states; and the rightmost state can be reached from relatively few initial states.

Each initial state, together with the frame number, corresponds to one particular secret key. In order for the attack to succeed, an internal state derived from the correct initial state has to be stored on the disk. Then, output from the cipher eventually collides with some output sequence stored on the disk and the corresponding internal state is revealed. If we choose for storing the internal states with the highest number of descendants at levels 101 and below, then our chances for success increase. In other words, looking again at Figure 5.1, it is in our best interest to have the internal states cover as large a black area as possible.

In the following analysis, we assume that amongst all internal states that generate a sequence beginning with the pattern α, we have chosen for storage a subset of 2^{35} states with the highest number of candidate initial states. The

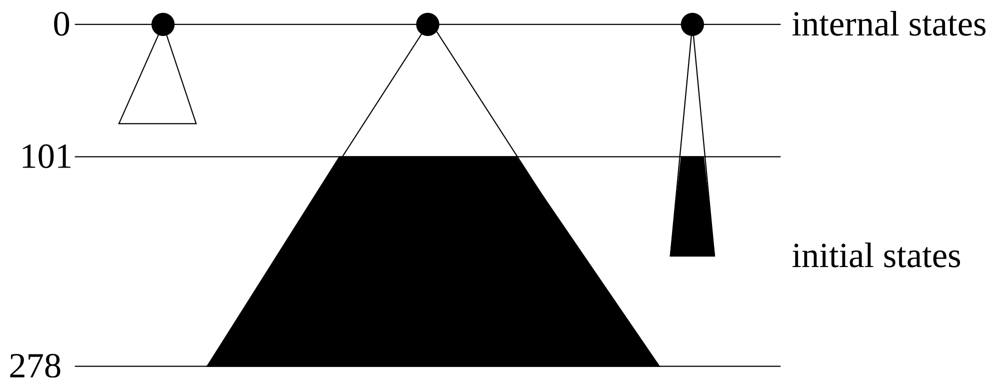

Figure 5.1: A variance in the distribution of reachable internal states and their corresponding initial states

extensive sampling by Biryukov et. al. showed that the average number of initial states for these stored states is expected to be approximately 12,500.

Real-Time phase of the Attack

Now, once the precomputation phase is complete, we have 2^{35} state-output pairs stored on the disk and sorted according to the output sequences. During the real-time phase of the attack, we compare the output of the cipher to the outputs stored on the disk. If some 51 consecutive output bits happen to collide with a stored output, then the corresponding internal state is revealed. Since the clock cycle at which the collision happens is known, it remains to invert the algorithm—for example, by using methods described in Section 4.1.2—to recover the corresponding candidate initial state(s). Should there be more than one such candidate, then each of them needs to be tested against further keystream. Since an internal state has on average 12,500 possible initial states in the "black belt", we can expect to have about $12,500/178 \approx 70$ candidate initial states at a fixed clock cycle, i.e., at a fixed level of the belt.

It remains to compute the probability of the collision. Of course, this probability depends on the length of the plaintext that we have access to. Assume that we have access to two minutes' (i.e., 120 seconds) worth of plaintext.

The probability that an internal state occurs during cipher operation is proportional to the number of initial states it can be derived from (the black area on Figure 5.1). If the number of possible initial states for a given internal state s is $W(s)$, then the probability of encountering this state during 120 seconds of conversation is

$$\Pr_T(s) = \frac{W(s)}{2^{64}} \cdot \frac{120}{5 \cdot 10^{-3}} \ ,$$

assuming that no internal state occurs more than once. On the other hand, we need to calculate the probability $\Pr_M(s)$ that a given state is stored on the disk. If we want to achieve the average weight 12,500, we can simply set a threshold k and define $\Pr_M(s) = 0$ for $W(s) < k$ and $\Pr_M(s) = 1$ for $W(s) \geq k$. As we saw above, this threshold will give us about 2^{35} stored states.

The probability of a collision can now be easily calculated by the formula

$$\begin{aligned} \Pr[\mathsf{collision}] &= \sum_s \Pr_M(s) \cdot \Pr_T(s) = \sum_{s|W(s)\geq k} \Pr_T(s) \\ &= \sum_{s|W(s)\geq k} \frac{W(s)}{2^{64}} \cdot \frac{120}{5 \cdot 10^{-3}} = 2^{35} \cdot \frac{12,500}{2^{64}} \cdot \frac{120}{5 \cdot 10^{-3}} \approx 0.56 \ . \end{aligned}$$

Hence, it is likely that a collision will actually occur.

Finally, we can compare how the selection of special states improves the efficiency of the attack when compared to storing just random states. During two minutes of conversation, approximately $178 \cdot 120/5 \cdot 10^{-3} \approx 2^{22}$ internal states can be inspected. If 2^{35} *random* internal states are stored on the disk, then the product $2^{22} \cdot 2^{35} = 2^{57}$ is about 100 times smaller than 2^{64}. To raise the success probability above 50%, it would be necessary to increase both the available storage and the available plaintext, together by a factor of 100. As we saw above, careful selection of stored states fills the gap in a much more feasible way.

5.1.3 Random Subgraph Attack

In this section, we discuss a different attack model, based on the original Hellman time-memory trade-off for block ciphers. When compared to the biased birthday attack, the random subgraph attack needs less available data (seconds instead of minutes). The real-time computation time is slightly higher but still feasible (a few minutes according to the authors).

Key Idea of the Hellman Trade-Off

We start by describing the trade-off model for block ciphers. Let E be any block cipher. For a given plaintext P, the encryption function f from keys K to ciphertexts is defined as $f(K) = E_K(P)$. Assuming that all the plaintexts, ciphertexts and keys have the same binary size, f can be considered as a random function over a common space U.

The idea proposed by Hellman is to choose a large number m of starting points and iterate f on them t times, i.e., compute $f(K)$, $f(f(K))$,...,$f^t(K)$ for each starting point K. On disk, only the startpoint-endpoint pairs $(K, f^t(K))$ are stored in increasing endpoint order. If we are given a ciphertext $f(K)$ that appears along one of these paths, we can continue applying f to the ciphertext, until we hit the endpoint of a path. Then, we can jump to the startpoint of the path and continue applying f. The last point before we hit $f(K)$ again is then likely to be the correct key K.

Since it is difficult to cover a random graph with random paths, Hellman proposed to use several variants of f (e.g., by permuting output bits). Suppose that we use t variants f_i and iterate each one of them t times on m start points. Then, each ciphertext is likely to be covered by one of the paths if the parameters satisfy

$$mt^2 \geq |U| \ .$$

For these parameters, the total memory requirement is $M = mt$ and the running time is $T = t^2$, so the corresponding time-memory trade-off curve is

$$M\sqrt{T} = |U| \ .$$

Applying the Attack to A5/1

In the case of the A5/1 cipher, the desired function f can be defined as the function from 64-bit internal states to their corresponding 64-bit output sequences. If we assume that the maximal memory at our hands is $M = 2^{36}$, then the corresponding time required is $T = 2^{56}$, which is much worse than in the biased birthday attack.

The basic idea of the new random subgraph attack is again to consider only the 2^{48} states producing an output sequence starting with the 16-bit pattern α. These states can be described by their 48-bit names, also, 48 bits suffice to uniquely define the output, since the first 16 bits are constant. Hence, if x is a 48-bit sequence, the computation goes as follows: expand the 48 bits to the full 64-bit internal state; compute the next 64 bits of output; delete the prefix α and take the remaining 48 bits as $f(x)$.

The recommended implementation gives the parameters m and t values $m = 2^{24}$ and $t = 2^{12}$. This means that 2^{12} variants of the function f are considered, each of which is iterated 2^{12} times, starting from 2^{24} randomly chosen 48-bit strings. The total memory requirement of the attack is still $M = mt = 2^{36}$ but the time required has dropped from 2^{56} to $T = t^2 = 2^{24}$.

The real-time phase of the attack requires only about 2 seconds of plaintext, since the pattern α is likely to appear in the output during that time.

The authors have proposed several small improvements to further enhance the attack. These can be found in the original paper [BSW01]. As a conclusion, they state that the actual attack takes several minutes on a single PC (as of 2000). Taking into account the reduced plaintext requirement—now only a few seconds—,this attack is very powerful, once the precomputation stage is completed.

5.1.4 Possibilities for Hardware Implementation

In time-memory trade-off attacks, most of the computational effort goes into the preprocessing stage. For both the biased birthday attack and the random subgraph attack, this stage consists of computing internal states producing output starting with a particular sequence. These computations are essentially the same as computations in guess-and-determine attacks: part of the internal state is guessed and the remaining part is derived from known keystream. The computations can be easily parallelised and would thus be rather suitable for hardware implementations.

However, the computations need to be carried out only once, which makes them unattractive for building special purpose hardware. Another drawback is the need to constantly record computation outputs, so it is the hard disk access that might become a bottleneck.

Finally, one might want to ask if guess-and-determine attacks become rudimentary in the light of time-memory trade-off attacks, which require approximately the same amount of computational effort, but only once. It might seem to be the case, since also the memory requirement of the attacks presented above is quite small in terms of today's hardware (and thus, also implementation cost is low). Still, guess-and-determine attacks prevail in at least one parameter, that is, the amount of required plaintext. While trade-off attacks require plaintext equal to seconds up to minutes of conversation, guess-and-determine attacks can make do with only one or two frames. Also, time-memory trade-offs are always probabilistic. This means that for the given time and memory parameters, there is a good (over 50 per cent) chance of recovering the session key but the success is not guaranteed, as opposed to guess-and-determine attacks.

5.2 Ekdahl-Johansson Correlation Attack

In this chapter, we discuss a correlation attack against the A5/1 stream cipher proposed by Ekdahl and Johansson [EJ03]. This attack is especially interesting, since it points out a new weakness in the cipher. While all previously presented attacks relied in their success on the relatively small internal state of the cipher, the crucial parameter in the Ekdahl-Johansson attack is the number of discarded bits prior to keystream generation.

In 2004, Maximov, Johansson and Babbage proposed another correlation attack against A5/1 [MJB04]. Their attack is based on the Ekdahl-Johansson approach, but uses some new techniques to improve the attack parameters. Since the attack is optimised for software implementation, and much of the enhancement comes from heuristics and tricks of implementation, we leave the details of this attack out of the scope of this thesis. In the summarising chapter, the reader can find the parameters of the complexity of the attack.

5.2.1 Key Ideas of the Attack

Similarly to other previously presented attacks, correlation attacks work in the known plaintext model. The key idea of a correlation attack on a stream cipher is the following: find known correlations between (some of) the bits of the internal state and the keystream output. Next, if the attacker has access to a sufficiently large amount of keystream (where the "sufficiency" of the amount depends on the strength of the correlation), he can determine information about the internal state (and, hence, about the secret key) with high confidence.

The first problem that we need to tackle when mounting a correlation attack on A5/1 is the varying frame counter. Namely, in order to make use of the correlations, one has to collect data over several frames. On the other hand, the cipher is re-initialised with a new counter every frame, so how can we find a correlation that holds for all frames, regardless of the counter?

Here, the linear fashion in which the cipher is (re)-initialised provides the answer. Denote the 64-bit secret session key by $K_c = (k_0, \ldots, k_{63})$ and the known 22-bit frame counter for frame n by $F_n = (f_0, \ldots, f_{21})$. Denote by $R_1(0), R_1(1), \ldots$ the output of the first linear feedback shift register *when clocked regularly* after the initialisation has been completed. Similarly, denote the consecutive output bits of registers R_2 and R_3 from *regular* clocking by $R_2(0), R_2(1), \ldots$ and $R_3(0), R_3(1), \ldots$, respectively. Finally, denote the known keystream from a given frame by $Z(0), Z(1), \ldots, Z(228)$. For now, we assume that the cryptanalyst knows either the whole frame or none of it, although we later see that the attacker needs only the 40 first bits from each frame.

When we look at the regular clocking, each output bit of each of the three LFSRs is a *known* linear combination of the secret key bits and the frame counter bits. For each output bit $R_1(t)$ of the first register, we can write out the following linear equation

$$R_1(t) = \sum_{i=0}^{63} c_{1,i}(t)k_i \oplus \sum_{i=0}^{21} d_{1,i}(t)f_i \ , \tag{5.1}$$

where $c_{1,i}(t)$ and $d_{1,i}(t)$ are known binary coefficients. In other words, the output bit $R_1(t)$ can be split into the *key part* and the *frame counter part*. Denote the key part by $\hat{k}_1(t)$ and the frame counter part for $\hat{f}_1(t)$, then Equation 5.1 translates to

$$R_1(t) = \hat{k}_1(t) \oplus \hat{f}_1(t) \ .$$

Similar equations can be written for registers R_2 and R_3:

$$R_2(t) = \hat{k}_2(t) \oplus \hat{f}_2(t) \ ,$$

$$R_3(t) = \hat{k}_3(t) \oplus \hat{f}_3(t) \ . \tag{5.2}$$

Here, the important thing to note is that the key part is constant over all frames and the frame counter part is known for all frames.

Of course, this alone does not help us much. We would like to relate the bits $R_j(t)$ with the output bits $Z(t)$ but the irregular clocking rule prohibits us from knowing which bits of the three registers were used to combine the output bit $Z(t)$. However, we do have some useful information about the clocking probabilities. Consider the first output bit $Z(0)$. Before this bit is produced, 101 bits of output are discarded. We know that each register is clocked with probability 3/4. Hence, we can expect that after 101 irregular clockings, each of the registers has been clocked about 76 times.

Assume for a moment that all three registers were indeed clocked exactly 76 times. Then we can write out the following equation:

$$Z(0) = R_1(75) \oplus R_2(75) \oplus R_3(75) \ .$$

The equations from (5.2) now give us

$$Z(0) = \hat{k}_1(75) \oplus \hat{f}_1(75) \oplus \hat{k}_2(75) \oplus \hat{f}_2(75) \oplus \hat{k}_3(75) \oplus \hat{f}_3(75) \ .$$

Rewriting this equation once more gives

$$\hat{k}_1(75) \oplus \hat{k}_2(75) \oplus \hat{k}_3(75) = \hat{f}_1(75) \oplus \hat{f}_2(75) \oplus \hat{f}_3(75) \oplus Z(0) \ , \tag{5.3}$$

where the left hand side is constant for all frames and the right hand side is known for all frames. From now on, denote the right hand side for frame n by

$$O_n(75, 75, 75, 0) = \hat{f}_1(75) \oplus \hat{f}_2(75) \oplus \hat{f}_3(75) \oplus Z(0) \ . \qquad (5.4)$$

In the above, we made the assumption that all three LFSRs were clocked exactly 76 times. This assumption will not hold for most frames but we can determine the probability that it holds. In the case above, the probability is found to be about 10^{-3}. (We will return later to explain how this probability was calculated.) Now, if our assumption holds, then Equation 5.3 is true with probability 1. If our assumption does not hold, then the equation is still true with probability 1/2. We have now come to our correlation:

$$\begin{aligned} & \Pr[\hat{k}_1(75) \oplus \hat{k}_2(75) \oplus \hat{k}_3(75) = O_n(75, 75, 75, 0)] \\ = \ & \Pr[\text{assumption correct}] \cdot 1 + \Pr[\text{assumption wrong}] \cdot \frac{1}{2} \\ = \ & \frac{1}{2} + \frac{1}{2} \cdot 10^{-3}. \end{aligned}$$

The left hand side of Equation 5.3 is now equal to the known right hand side with probability biased from 1/2. If we have access to a sufficient amount of frames, the common value $\hat{k}_1(75) \oplus \hat{k}_2(75) \oplus \hat{k}_3(75)$ can be determined with high confidence. The value of this sum gives us one bit of information about the secret key. By considering other such triplets, the entire key can be derived.

In order to determine the number of frames required to make the right decision, we need to introduce more weapons of math instruction from probability theory. We present the following theorem without proof.

Theorem 5.1 (A Chernoff Bound)**.** *Let $X_1, X_2, \ldots, X_n$ be independent binary random variables. For $1 \leq i \leq n$, let $\Pr[X_i = 1] = p > 1/2$. Define $X = \sum_{i=1}^{n} X_i$. Then,*

$$\Pr\left[X \leq \frac{n}{2}\right] \leq e^{-2n(p-1/2)^2} \ .$$

The theorem gives a lower bound for the success of a majority decision for n independent, equally likely events. In the case of our attack, each event indicates the correctness of our decision for one frame—each decision is correct with probability $p = 1/2 + 1/2 \cdot 10^{-3}$. Suppose that we take the majority vote over 6 million frames. Then the probability that less than half of our decisions are right is bounded from above by

$$\Pr[\mathsf{error}] \leq e^{-2 \cdot 6 \cdot 10^6 \cdot (10^{-3}/2)^2} = e^{-3} \leq 0.05 \ ,$$

so we can have 95% confidence in our decision. However, 6 million frames of keystream amount to over 8 hours of unencrypted conversation, clearly an impossible goal.

5.2.2 An Improvement of the Attack

The attack described above recovers the key bit-by-bit but requires millions of frames of keystream to achieve an acceptable confidence level. Even one million frames correspond to over one hour of conversation, so in practice, this attack is still unrealistic.

In the basic attack above, we considered the probability of the clocking triple $(75, 75, 75)$ occurring at the position where the first keystream bit $Z(0)$ is output, i.e., after 101 bits of discarded output. Let the keystream bit $Z(t)$ be produced at position $v = t + 101$. To improve the attack, note that the same triple might also occur in other positions, and we can use all those positions to calculate the correlation probability. Namely, denote the probability of a triple (cl_1, cl_2, cl_3) occurring at position v by $\Pr[(cl_1, cl_2, cl_3), v]$. Given a triple (cl_1, cl_2, cl_3), it is now possible to calculate an interval $\mathcal{I}$ where this triple has a significant probability of occurrence. In frame n, denote the probability of the corresponding sum of $\hat{k}$-bits being zero by

$$p_n(cl_1, cl_2, cl_3) = \Pr[\hat{k}_1(cl_1) \oplus \hat{k}_2(cl_2) \oplus \hat{k}_3(cl_3) = 0] \ .$$

Then, the correlation probability can be calculated as a weighted voting over all positions in interval $\mathcal{I}$:

$$\begin{aligned} p_n(cl_1, cl_2, cl_3) \quad &= \quad \sum_{v \in \mathcal{I}} \Pr[(cl_1, cl_2, cl_3), v] \cdot [O_n(cl_1, cl_2, cl_3, v - 101) = 0] + \\ & \quad\quad \frac{1}{2} \cdot \left(1 - \sum_{v \in \mathcal{I}} \Pr[(cl_1, cl_2, cl_3), v] \right) \ , \end{aligned} \tag{5.5}$$

where $O_n(cl_1, cl_2, cl_3, v-101) = \hat{f}_1(cl_1) \oplus \hat{f}_2(cl_2) \oplus \hat{f}_3(cl_3) \oplus Z(v-101)$, exactly as in Equation 5.4.

Calculating the Probabilities

Next, we need a method for calculating the probability $\Pr[(cl_1, cl_2, cl_3), v]$ of a clocking triple occurring at a certain position. Under the usual assumption of independent uniformly distributed clocking bits, we can give a simple closed formula for the probability:

$$\Pr[(cl_1, cl_2, cl_3), v] = \frac{\binom{v}{v-cl_1}\binom{cl_1}{v-cl_2}\binom{cl_1+cl_2-v}{v-cl_3}}{4^v} \ .$$

The formula is derived as follows: the first LFSR is *not* clocked $v - cl_1$ times. There are $\binom{v}{v-cl_1}$ possible position combinations for this to happen. Next, we know that if one of the LFSRs is not clocked, then the other two are definitely clocked. Hence, the second LFSR will definitely move in the $v - cl_1$ positions where the first LFSR did not move. From the remaining $v - (v - cl_1) = cl_1$ positions, we have to choose $v - cl_2$ positions where this register is not clocked; there are $\binom{cl_1}{v-cl_2}$ ways to do this. Finally, the third register can stop only if both of the other registers move. There are $v-(v-cl_1)-(v-cl_2) = cl_1+cl_2-v$ such positions, out of which we can select $v - cl_3$ in $\binom{cl_1+cl_2-v}{v-cl_3}$ ways. As a final step, the number of clocking sequences satisfying our constraints is divided by the number of all possible clocking sequences, of which there are 4^v.

After these probabilites are calculated, we can estimate the probability given in Equation 5.5. In order to make use of the information over all available frames, the log-likelihood ratio is used. Based on the log-likelihood result, a hard decision can then be taken to estimate the sum $\hat{k}_1(cl_1) \oplus \hat{k}_2(cl_2) \oplus \hat{k}_3(cl_3)$ to be either 0 or 1.

Mounting the Attack

For each clocking triple (cl_1, cl_2, cl_3), the estimate on the value $\hat{k}_1(cl_1) \oplus \hat{k}_2(cl_2) \oplus \hat{k}_3(cl_3)$ gives us one bit of information about the key. Hence, in order to reconstruct the secret key, we need to consider at least 64 such triples. This is done as follows.

First, pick an interval C_1 and let each of the clocking values cl_1, cl_2, cl_3 run through this interval. If the length of the interval is n, the hard estimates on the sums $\hat{k}_1(cl_1) \oplus \hat{k}_2(cl_2) \oplus \hat{k}_3(cl_3)$ yield a system of linear equations in $3n$ variables (the $\hat{k} - bits$) and n^3 equations (the estimates). For example, if $C_1 = [79, \ldots, 86]$, then the system has 24 variables and 512 equations. The problem of finding the correct values of the variables then becomes a problem of linear decoding; if the estimates are reasonably accurate, 24 bits of information about the key can be obtained. In the actual attack, this is implemented as an exhaustive search over 24 binary variables.

However, 24 bits are not sufficient to recover the secret key. In order to obtain at least 64 bits, the length of the interval should be increased to at least 22. Now, this would make the search more complex than a brute-force attack, with 2^{66} possible combinations to consider. Instead, a divide-and-conquer strategy can be used. Namely, we can select several intervals and combine their solutions. For $n = 8$, we could set $C_1 = [79, \ldots, 86]$, $C_2 = [87, \ldots, 94]$ and $C_3 = [95, \ldots, 102]$. This gives in total 72 bits of information about the key, which is more than enough. The total cost of the exhaustive search is only $3 \cdot 2^{24}$.

When the solutions for these three intervals have been found, they can be combined and loaded in the shift registers. To verify the solutions, the registers then need to be clocked backwards regularly for 79 clocks, plus another 22 clocks used for loading the frame number. Next, the frame number should be loaded and the 100 premix clocks should be run. Then, it is possible to check the produced keystream against the expected keystream and validate or reject the solution.

In order to increase the chances for success, it is possible to save for each interval a list of closest solutions and try all possible combinations over the three lists. Also, longer intervals with overlapping bits can be used to discard mismatching solutions.

5.2.3 Possibilities for Hardware Implementation

Ekdahl and Johansson have implemented the attack in software and it works reasonably well. For 70,000 frames of available plaintext (about 5 minutes and 20 seconds of conversation), the success rate has been brought to over 75%, while the attack time remains within a few minutes. There are two parameters which seem to affect the efficiency of the attack: the amount of available plaintext and the configuration of search intervals. Two tendencies strike out from the simulations. Firstly, and rather unsurprisingly, the success rate grows as the amount of plaintext grows. Secondly, the success rate grows—albeit more slowly—as the search intervals get larger (and, hence, the computational complexity increases). There is also a third parameter that was not experimented with in the implementations, namely, the number of closest solutions stored for each interval. Clearly, increasing this number would also improve the attack at the expense of more computational work.

Most of the computational effort in the attack goes into linear decoding and solution checking. These are both simple operations—the latter being almost equivalent to running A5/1 itself—that could well benefit from hardware implementation. Both stages can be parallelised to an arbitrary extent, and some pipelining might be possible between the two stages. It is an interesting open question, whether and how much it is possible to increase the success rate of the attack and/or to decrease the plaintext requirement of the attack by putting more computational effort into the implementation than a regular PC allows.

5.3 Ciphertext-Only Attacks

In this chapter, we briefly discuss options for attacking the A5/1 cipher within a ciphertext-only scenario. Since obtaining conversation plaintext is perhaps the key obstacle in implementing all the previously described attacks, finding a ciphertext-only attack would be a huge benefit for the attacker.

Suppose that the attacker only has access to some ciphertext, which is obtained by an exclusive-or operation between plaintext and keystream. In order to determine which is the correct plaintext-keystream pair to form this ciphertext, the plaintext needs to exhibit some redundancy. Fortunately, in the settings in which the cipher is actually used, redundancy is present in the plaintext, and there are at least two different ways to exploit this.

Redundancy in Control Data

The first observation that should be made is that, apart from speech data, also signalling data is encrypted before transmission. The bits in the signalling data frames often have a very specific structure with many fixed bits, so such information can be exploited.

In particular, we point out a proposal made in [MJB04]. In a conversation, normally only one party speaks at a time, so there is a lot of silence. In GSM conversations, transmission is cut during silence. However, the fact that also background noise is cut can be very annoying for a listener. To solve this problem, background noise, or so-called “comfort noise”, is generated at the listener’s side. The GSM standard specifies the Silence Descriptor frame (SID frame) that contains information about this comfort noise [ETSb]. In order for the receiving side to identify this frame, the frame contains a 95-bit fixed all-zero codeword.

The SID frame is sent in the beginning of a silence period and then twice each second. Hence, attacks that require only little plaintext could easily benefit from this information. In particular, success is very likely for the guess-and-determine attacks of both Anderson-Keller-Seitz and Golić, which require less than 95 bits of plaintext from one frame—so one SID frame should be sufficient to complete the attack.

Redundancy from Error Correction Coding

The second observation which can be made is that data (both speech and signalling data) is encrypted *after* error-correction coding is applied. This means that all encrypted data contains redundancy with known structure. Barkan, Biham and Keller describe a time-memory trade-off similar to the random subgraph attack presented in Section 5.1.3 exploiting this redundancy [BBK03].

In brief, the attack goes as follows. Consider for example signalling data. For this type of data, error correction coding expands 184 bits into 456 bits, which are then interleaved and divided into four frames. The coding operation and the interleaving operation can be modelled together as a 456×184 matrix G. If the 184-bit message to be coded is M, then the resulting 456-bit codeword is $P = G \cdot M$.

From basic coding theory, we know that there are $456 - 184 = 272$ linearly independent equations that describe the set of all valid codewords. Let the corresponding 272×456 parity-check matrix be H, i.e., $H \cdot P = 0$ for all valid codewords P.

Now, assume that we have obtained four frames of ciphertext and combined it to a 456-bit word C. Let P and K denote the corresponding plaintext and keystream, respectively. Then, $C = P \oplus K$ and

$$H \cdot C = H \cdot (P \oplus K) = (H \cdot P) \oplus (H \cdot K) = \mathbf{0} \oplus (H \cdot K) = H \cdot K \ .$$

Now, since matrix H is fixed and ciphertext C is known, $H \cdot K$ is a known 272-bit word that depends only on the output of the A5/1 generator during four consecutive frames. Hence, we can model the operation of the cipher as a function $h : \{0,1\}^{64} \mapsto \{0,1\}^{272}$ that takes the 64-bit internal state of the cipher after the initialisation of the first frame and outputs the 272-bit word from the four frames. Inverting h then reveals the internal state and thus breaks the cipher.

If we consider only the first 64 bits of this 272-bit word, then h is a function from 64-bit words to 64-bit words, and we can apply the Hellman trade-off to h, in the same fashion as it was done in the random subgraph attack (c.f. Section 5.1.3 of Chapter 5).

The authors of the attack have evaluated various points on the trade-off curve. For example, if we assume access to 20 minutes of encrypted conversation—note that we only need *ciphertext*, which can be obtained by passive eavesdropping—then 35 PCs would complete the preprocessing in one year, the result would require 600 GB of storage, and the actual attack could be completed real-time by one PC. As another example, if the available data is limited to 8 seconds, then the memory requirement grows to 35-70 TB. Also, it would then take 5000 PCs to complete the preprocessing in one year, and several hundred PCs to complete the attack in real-time.

It is interesting to note that a similar attack against the weak encryption algorithm A5/2 requires only a few hours of one-time precomputation, less than 1 GB of storage, and recovers the secret key in less than a second.

Moreover, even if the network uses A5/1 or even A5/3, a vast majority of mobile devices still support the weak A5/2, which opens up several possibilities for man-in-the-middle attacks. Scenarios for such attacks are described in [BBK03].

Chapter 6

A Comparison of the Attacks

In the final chapter of this Master's thesis, we give a compact comparison of known attacks against the A5/1 stream cipher. We compare the following parameters:

- **Plaintext requirement**, or, equivalently, keystream requirement. We give the required number of frames, and the corresponding amount of actual conversation. For correlation attacks, we give an interval; the success probability of the attack grows as plaintext availability increases.

- **Ciphertext requirement**. For those attacks that require plaintext, ciphertext requirement is equal to plaintext requirement. For ciphertext-only attacks, known ciphertext is the only type of ciphering data required.

- **Memory requirement**. This is only given for attacks that require precomputation. We do not estimate the memory requirement of the real-time phase of the attack, since this is usually very small. For time-memory trade-off attacks, we list some points on the trade-off curve; other configurations are possible.

- **Precomputation complexity**. Estimates are given in time units and reflect the time it would take one "regular PC" to complete the attack, based on the best implementations known to us. For precomputation, the time is usually fixed beforehand, so we do not distinguish between worst-case and average-case complexity. It should be noted that these figures are taken from articles written over several years and are thus "of illustrative value". On the other hand, we feel that comparing, say, bit exclusive-ors to matrix multiplications in Big-Oh notation is even more of a comparison between apples and oranges. We leave it to the readers to apply Moore's law to level the differences.

- **Real-time computation complexity**. Again, estimates are given in time units for a single PC, except for the Anderson-Keller-Seitz hardware attack implemented on an FPGA, and the Golić attack implemented on a mixed platform. Estimates for guess-and-determine attacks are based on average-case complexity. For other, probabilistic attacks, the given time is the worst-case complexity for a given success probability. For both precomputation and real-time computation, we make an exception in the case where no estimate is given in the corresponding article, and provide some approximate figures to give a very rough idea of the required time.

Table 6.1 summarises the results. The attacks are presented in the order in which they were described in this thesis. The years refer to the year the corresponding article was published, except for the Anderson-Keller-Seitz software attack, in which case the year refers to our own software implementation.

Attack type	Year	Plaintext (frames)	Ciphertext (frames)	Memory	Pre-computation	Attack time
Anderson-Keller-Seitz (software)	2005	1 (5 ms)	1	—	—	2.5 years
Anderson-Keller-Seitz (hardware)	2001	1 (5 ms)	1	—	—	2 years (FPGA)
Biham-Dunkelman	2000	6000 (30 s)	6000	—	—	estimated $<$ 1 day (2005)
Biham-Dunkelman (our proposal)	2000	400 (2 s)	400	—	—	estimated 3-4 days (2005)
Golić	1997/2000	1 (5 ms)	1	—	—	2.5 days (PC/FPGA)
Biased birthday	2000	24,000 (2 min)	24,000	200 GB	unknown (2^{48})	1 second
Random subgraph	2000	400 (2 s)	400	400 GB	unknown (2^{48})	minutes
Ekdahl-Johansson	2003	30,000–70,000 (3–6 min)	30,000–70,000	—	—	minutes
Maximov-Johansson-Babbage	2004	2000–10,000 (10-50 s)	2000–10,000	—	—	minutes
Barkan-Biham-Keller	2003	—	240,000 (20 min)	600 GB	35 years	minutes
Barkan-Biham-Keller	2003	—	1600 (8 s)	35 TB	5000 years	minutes–hours

Table 6.1: A comparison of the attacks on A5/1

Chapter 7

Conclusion

The A5/1 stream cipher used in GSM communications is possibly the most widespread cipher in the world, with several hundred million users relying on its security. After its reverse-engineering in 1994, it has also become one of the most studied ciphers in the cryptographic community. Several attacks against the cipher have been published, some of which work in real-time on a regular PC. However, attacks with feasible computational load usually require a large amount of available plaintext—several seconds or even minutes of unencrypted communication. Thus, it is an interesting question whether attacks that are computationally more demanding but require only a very small amount of plaintext can be enhanced by a hardware implementation. While studying the hardware implementability of previously published attacks against A5/1, we have come to the following conclusions.

Firstly, guess-and-determine attacks are particularly suited for hardware implementation. In 2001, when one such implementation was published, it did not offer much benefit compared to a software implementation on a standard PC. However, over time, PCs and FPGAs have improved roughly equally in speed but FPGAs have also notably improved in capacity. Since guess-and-determine attacks allow for parallelisation to an almost arbitrarily large degree, an FPGA in 2005 could do in one week the work that took an FPGA in 2001 over two years. On a negative side—negative, of course, from the attacker's viewpoint—, we show that the complexity of a simple guess-and-determine attack is higher than previously speculated: we have calculated the complexity to be over 2^{50} A5/1 clock cycles.

Secondly, we show that there is a nice trade-off between the computational complexity and plaintext requirement of a guess-and-determine attack. Namely, the attacker can achieve a factor two reduction in computational complexity by increasing plaintext requirement by a factor of four.

Finally, we investigate correlation attacks, which rely on a large amount of plaintext to make use of small correlations present in this data. We conclude that such attacks can also benefit from hardware assistance. We leave it an open question whether increased computational effort can bring the plaintext requirement from the current several minutes down to a more realistic level.

Jadašifri A5/1 riistvarapõhiste rünnete keerukusanalüüs

Magistritöö

Resümee

GSM-sides kasutatav jadašiffer A5/1 on mitmesaja miljoni kasutajaga maailma enamlevinumaid šifreid. Pärast esialgu salajas hoitud algoritmi avalikukstulemist 1994. aastal on A5/1 ka krüptograafide poolt üks enamuurituid šifreid. Osad avaldatud rünnetest töötavad tavalisel PC-l reaalajas. Samas on kiiremate rünnete eelduseks, et ründajal on juba eelnevalt juurdepääs sekunditele või isegi minutitele dekrüpteeritud vestlusele. Käesolevas töös uuritakse, kas ründeid, mis vajavad vaid väikest kogust avateksti ja on seetõttu praktikas kasutatavad, on võimalik muuta efektiivsemaks, kasutades spetsiaalset riistvara.

Töö esimeses pooles uuritakse mõista-mõista rünnete (ingl. k. *guess-and-determine attacks*) keerukust ning antakse ühe ründemudeli jaoks täpne keerukushinnang. Osutub, et sellised ründed on üldiselt aeglasemad, kui varasemad hinnangud lubasid arvata—“harilikul” lauaarvutil kulub ühe võtme lahtimurdmiseks üle kahe aasta. Samas on sellised ründed peaaegu kuitahes suures ulatuses paralleliseeritavad. See võimaldab hästi ära kasutada spetsiaalset riistvara. Kui ühe protsessoriga PC puhul saame ära kasutada ainult tema kiirust, siis FPGA korral tuleb arvesse ka kiibi suurus—suuremale kiibile saab implementeerida mitu rünnet paralleelselt.

Lisaks keerukushinnangule pakutakse töö esimeses pooles välja ka ühe mõista-mõista ründe üldistus. Nimelt on ründajal võimalik vähendada vajamineva avateksti hulka 4 korda, suurendades vastavalt arvutusvõimsust 2 korda.

Töö teises pooles antakse ülevaade ülejaanud teadaolevatest rünnetest A5/1 šifri vastu ja hinnatakse iga ründe korral riistvaraimplementatsiooni otstarbekust.

Bibliography

[3GP] 3rd Generation Partnership Project. http://www.3gpp.org (last visited 14.03.2006).

[3GP04] 3GPP TSG SA WG3 Security: Evaluations of mechanisms to protect against Barkan-Biham-Keller attack, May 2004.

[And94] Ross Anderson. Newsgroup Communication. http://yarchive.net/phone/gsmcipher.html (last visited 14.03.2006), 1994.

[Bab95] Steve Babbage. A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers. In *European Convention on Security and Detection, IEE Conference Publication No. 408*, 1995.

[BBK03] Elad Barkan, Eli Biham, and Nathan Keller. Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication. In *Advances in Cryptology - CRYPTO 2003. Proceedings*, volume 2729 of *Lecture Notes In Computer Science*, pages 600–616. Springer Verlag, 2003.

[BD00] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher. In *Progress in Cryptology—INDOCRYPT 2000. Proceedings*, volume 1977 of *Lecture Notes In Computer Science*, pages 43–51. Springer Verlag, 2000.

[BGW99] Marc Briceno, Ian Goldberg, and David Wagner. A Pedagogical Implementation of A5/1. http://jya.com/a51-pi.htm (last visited 14.03.2006), 1999.

[BSW01] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Fast Software Encryption: 7th International Workshop, FSE 2000. Proceedings*, volume 1978 of *Lecture Notes In Computer Science*, pages 1–18. Springer Verlag, 2001.

[COM] GSM Cloning. http://www.isaac.cs.berkeley.edu/isaac/gsm.html (last visited 14.03.2006).

[ECR] ECRYPT—European Network of Excellence for Cryptology. eSTREAM—the ECRYPT Stream Cipher Project. http://www.ecrypt.eu.org/stream (last visited 14.03.2006).

[EJ03] Patrik Ekdahl and Thomas Johansson. Another Attack on A5/1. In *IEEE Transactions on Information Theory*, volume 49, pages 284–289, 2003.

[eK01] Geir Bjåen and Erling Kaasin. Security in GPRS. Master's thesis, Agder University College, 2001.

[ETSa] ETSI EN 300 909: Digital cellular telecommunications system (Phase 2+); Channel coding. GSM 05.03 version 8.5.1 Release 1999.

[ETSb] ETSI TS 100 963: Digital cellular telecommunications system (Phase 2+); Comfort Noise Aspects for Full Rate Speech Traffic Channels. (3GPP TS 06.12 version 8.1.0 Release 1999).

[Fou98] Electronic Frontier Foundation. *Cracking DES*. O'Reilly & Associates Incorporated, 1998.

[Gol97] Jovan Dj. Golić. Cryptanalysis of Alleged A5 Stream Cipher. In *Advances in Cryptology—EUROCRYPT '97. Proceedings*, volume 1233 of *Lecture Notes In Computer Science*, pages 239–255. Springer Verlag, 1997.

[GSM04] GSM World. http://www.gsmworld.com (last visited 14.03.2006), 2004.

[Hel80] Martin E. Hellman. A Cryptanalytic Time-Memory Trade-Off. In *IEEE Transactions on Information Theory*, volume 26, pages 401–406, 1980.

[Kai03] Jaakko Kairus. Potential of a Hardware Assisted Attack on A5/1 Security. Technical report, Helsinki University of Technology, 2003.

[KS01] Jörg Keller and Birgit Seitz. A Hardware-Based Attack on the A5/1 Stream Cipher. In *Proceedings of the 2001 Arbeitsplatzcomputer (APC)*, München, Germany, 2001.

[MJB04] Alexander Maximov, Thomas Johansson, and Steve Babbage. An Improved Correlation Attack on A5/1. In *Selected Areas in Cryptography: 11th International Workshop, SAC 2004. Revised Selected Papers*, volume 3357 of *Lecture Notes In Computer Science*, pages 1–18. Springer Verlag, 2004.

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[PS00] Thomas Pornin and Jacques Stern. Software-hardware trade-offs: Application to a5/1 cryptanalysis. In *Cryptographic Hardware and Embedded Systems - CHES 2000. Proceedings*, volume 1965 of *Lecture Notes In Computer Science*, pages 318–327. Springer Verlag, 2000.

[Xil99] Xilinx. *XC4000E and XC4000X Series Field Programmable Gate Arrays*, May 1999.

[Zen99] Erik Zenner. Kryptographische Protokolle im GSM-Standard: Beschreibung und Kryptanalyse. Master's thesis, University of Mannheim, 1999.