SANDIA REPORT

SAND2004-5114
Unlimited Release
Printed October 2004

Securing Mobile Code

Erik Anderson, Cheryl Beaver, William Neumann and Richard Schroeppel
Cryptography and Information Systems Surety Department

Phil Campbell
Networked System Survivability and Assurance Department

Hamilton Link
Advanced Information and Control Systems Department

Lyndon Pierson
Advanced Networking Integration Department

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

SAND2004-5114
Unlimited Release
Printed October 2004

Securing Mobile Code

Erik Anderson, Cheryl Beaver, William Neumann, and Richard Schroeppel
Cryptography and Information Systems Surety Department

Phil Campbell
Networked System Survivability and Assurance Department

Hamilton Link
Advanced Information and Control Systems Department

Lyndon Pierson
Advanced Networking Integration Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0785

Abstract

If software is designed so that the software can issue functions that will move that software from one computing platform to another, then the software is said to be "mobile." There are two general areas of security problems associated with mobile code. The "secure host" problem involves protecting the host from malicious mobile code. The "secure mobile code" problem, on the other hand, involves protecting the code from malicious hosts. This report focuses on the latter problem.

We have found three distinct camps of opinions regarding how to secure mobile code. There are those who believe special distributed *hardware* is necessary, those who believe special distributed *software* is necessary, and those who believe *neither* is necessary. We examine all three camps, with a focus on the third.

In the distributed software camp we examine some commonly proposed techniques including Java, D'Agents and Flask. For the specialized hardware camp, we propose a cryptographic technique for "tamper-proofing" code over a large portion of the software/hardware life cycle by careful modification of current architectures. This method culminates by decrypting/authenticating each instruction within a physically protected CPU, thereby protecting against subversion by malicious code.

Our main focus is on the camp that believes that neither specialized software nor hardware is necessary. We concentrate on methods of code obfuscation to render an entire program or a data segment on which a program depends incomprehensible. The hope is to prevent or at least slow down reverse engineering efforts and to prevent goal-oriented attacks on the software and execution. The field of obfuscation is still in a state of development with the central problem being the lack of a basis for evaluating the protection schemes. We give a brief introduction to some of the main ideas in the field, followed by an in depth analysis of a technique called "white-boxing". We put forth some new attacks and improvements on this method as well as demonstrating its implementation for various algorithms. We also examine cryptographic techniques to achieve obfuscation including encrypted functions and offer a new application to digital signature algorithms. To better understand the lack of security proofs for obfuscation techniques, we examine in detail general theoretical models of obfuscation. We explain the need for formal models in order to obtain provable security and the progress made in this direction thus far.

Finally we tackle the problem of verifying remote execution. We introduce some methods of verifying remote exponentiation computations and some insight into generic computation checking.

Contents

List of Figures

List of Tables

Chapter 1

Introduction: What is "secure mobile code?"

If software is designed so that the software can issue functions that will move that software from one computing platform to another, then the software is said to be "mobile." (Such software presumes a distributed computing environment that can carry out such move functions.) There are two general areas of security problems associated with mobile code. The "secure host" problem involves protecting the host from malicious mobile code. The "secure mobile code" problem, on the other hand, involves protecting the code from malicious hosts. This report focuses on the latter problem. We note that this issue applies not only to "mobile" software that can move itself, but also to any code running on a host computer that is not completely trusted such as, say, for a remote computation; nevertheless, we will usually refer to the problem as the "secure mobile code" problem.

The secure mobile code problem assumes that the code is on an adversary's computer and that (a) the adversary can at any time examine any of the bits comprising the code and its data, (b) can change any of those bits, and (c) has similar access to all of the registers during execution should the adversary decide to execute the code, allowing for replay attacks. In addition, (d) the adversary can change bits after execution. That is, the adversary has complete control of the code before, during, and after execution. Based on the above assumed capabilities we present a simple, corresponding model of defense, in three levels and shown in Table 1.1, where the adversary would have to achieve level i in order to achieve level $i + 1$.

We have found three distinct camps of opinions regarding how to secure mobile code. There are those who believe special distributed *hardware* is necessary, those who believe special distributed *software* is necessary, and those who believe *neither* is necessary.[1] In this report, we will examine all three camps, with a focus on the third.

When neither specialized software nor hardware is present, secure mobile code must be executable as is. It is unlike, say, encrypted code that requires a pre-processing step (decryption) before it is executable. Each instruction is in plaintext. However, the word "secure" in secure mobile code implies that the code is nevertheless protected in some way.

[1] Perhaps there is a fourth group that thinks it is impossible to secure mobile code with any technique.

Level	Prevent the adversary from...
1	...understanding the intention of the code.
2	...being able to change the code in a goal-directed way.
3	...being able to create his own unobfuscated (and thus maintainable) copy of the functionality of the code.

Table 1.1. Defense Model

Unfortunately, the term says nothing directly about data: is that not to be secure also? To remove any confusion with encrypted code and at the same time to avoid problems with the word "secure" and to distinguish code from data we think that the terms "executable protected code" (EPC) and "readable protected data" (RPD) are superior and less likely to confuse. Such code is generally referred to as "obfuscated" code, suggesting that the code is protected by its being obscured or confused in such a way that is it is difficult to make sense of it but it is at the same time executable as is: it is not encrypted. As a consequence, the portion of this report dealing with solutions requiring neither specialized software or hardware concentrates on obfuscation.

A homely analogy may be of help in grasping the concept of obfuscated code. Suppose we visit a friend and find that he is in the kitchen with various jars and bags and bowls on the table. We ask him what he is doing. He could make one of two replies.

Obfuscated Description:

> He replies that he is putting two cups of this – he points – into this bowl – again he points – and four tablespoons of this – again he points – in as well, whereupon he says he will stir it up and leave it to one side for a few minutes. Meanwhile he says that he will put four of these and two cups of that and four tablespoons of this and four cups of that and a cup of this and a half a cup of that, along with a cup and three quarters of this, all in the blender and stir it up. After a few minutes he will combine these ingredients with the ones in the other bowl, then he will add many cups of this while he stirs. That will complete the first step.

Plaintext Description:

> He replies that he is making bread.

For completeness, there is a third reply. Encrypted Description:

> Jx $ruiq9 pwvj k' dm 9ejfsc k%scw.

Note that the encrypted reply has exactly the same number of characters as the unobfuscated one (for ease of comparison we show the spaces unencrypted), unlike the obfuscated one which has a distinctly different number of characters. Note also that the encrypted reply is not "executable" as is: it makes no sense. One would have to decrypt it before proceeding.

When we look at obfuscated code, then, presuming we cannot break the obfuscation we cannot see the "big picture".

Obfuscated Description:

> Here is a load of address 10 to register 2. And here is a move of address 17 to address 12. And here is a zero of register 4. And so on.

Plaintext Description:

> Here is a sort routine.

The central problem currently facing obfuscation is the lack of a basis for evaluating protection schemes. [2] When presented with code that we are told is obfuscated we have no gauge by which to measure the obfuscation. Granted, the code may be unintelligible to us-and may still be unintelligible after much examination-but this is qualitative, subjective, and unrepeatable. To make matters worse obfuscation, like steganography, is best when it is not even noticed, when the code does not even have the appearance of being obfuscated, when we sail right through it and are confident that we know precisely what it is doing, completely fooled. A step forward would be a black box that could tell us which of two obfuscated versions of a code were stronger, even if we had no theory of how the black box worked. As it is, we do not know of any such black box, let alone any theory.

So the floodgate, so to speak, that is holding back progress on obfuscation is a gauge by which to measure strength.

Oddly enough the need for a gauge itself is curious. Someone once remarked that software obfuscation should be easy since obfuscation seems to be software's natural state. We seem to spend all of our time doing our best just to keep the cursed stuff from slipping through our fingers and on into obfuscated oblivion. Yet, by one of those puzzles of nature, now that we see a use for obfuscated code we seem unable to produce it.

This report is organized as follows. In Chapter 2 we briefly discuss some proposed software only approaches to securing mobile code including Java, D'Agents and Flask.

In Chapter 3 we discuss specialized hardware solutions including a Sandia developed method for *shrink wrapping* software that is then run on a special trusted hardware volume.

The remaining chapters are devoted to software solutions requiring neither specialized software or hardware. In particular, we concentrate there on the approach of code obfuscation. In Chapter 4 we examine some of the important approaches to obfuscation put forth in the past few years.

In Chapter 5 we look in detail at the obfuscation method known as "white-boxing". In particular we describe a white-box implementation applied to the cipher DES as well as some attacks against it. We show how the method can be improved to withstand the attacks and also apply the white-boxing method to the AES cipher.

In Chapter 6 we describe some cryptographic methods to achieve obfuscation including

[2] As Loureiro notes, "[Obfuscation's] major drawback is the lack of theoretical foundations in order to establish precise definitions of security, and accordingly to be able to quantify the security of the underlying transformations" ([22], page 21).

encrypted functions and applications to digital signature algorithms. Although the cryptographic approaches do allow for some provable results against breaking the obfuscation, they only apply to a small class of functions.

In Chapter 7 we examine in detail general theoretical models of obfuscation. We explain the need for formal models in order to obtain provable security and the progress made in this direction thus far.

Chapter 8 describes some possible future research directions in the field of obfuscation.

Finally in Chapter 9, we tackle the complex issue of verifying execution of code on remote machines. In particular, we put forth some new methods for verifying exponentiation computations and examine the idea of generic execution checking.

Chapter 2

Specialized Software Solutions

One approach to the implementation of mobile code in general is to execute the code on distributed software. The hardware and its location may be foreign, so to speak, but the system software is just like home. In this chapter we will consider three examples of this approach – the Java, D'Agents, and Flask – each closer to the hardware than the last. Our conclusion is that the problem of trust persists, even when we control the software that has exclusive control of the hardware.

2.1 Java

Java is probably the most famous example of an implementation of mobile code via distributed software. The "distributed software" in this case is the Java Virtual Machine (JVM). In order to execute a code written in Java, the user first "compiles" the source code, translating it from the high-level language of Java into the intermediate language of Java "bytecodes." The bytecodes are the input language of the JVM. A JVM on a given platform executes Java bytecodes on that given platform. Once in bytecodes, a Java code can be executed on any platform that has a JVM, fulfilling the dictum, "Write once, run anywhere," meaning that a single version of a code written in Java can execute on any platform that has a JVM, as suggested in Figure 2.1. Of course we have to trust whatever lies below the JVM if we are to run our Java bytecodes "anywhere."

2.2 D'Agents

Another example of a mobile code system implemented via distributed software, this one at a level closer to the hardware than the JVM, is D'Agents [20]. D'Agents requires server code to be available on each machine that runs the agents. D'Agents agents can be written in more than one language: the D'Agents server code sits logically "beneath" the interpreters for those languages, such as the JVM. In turn, beneath the server code, sit "resource managers" for the CPU, the file system, libraries, the network, the screen, and other resources. Migration can be performed in a single instruction: "agent_jump." When agents migrate,

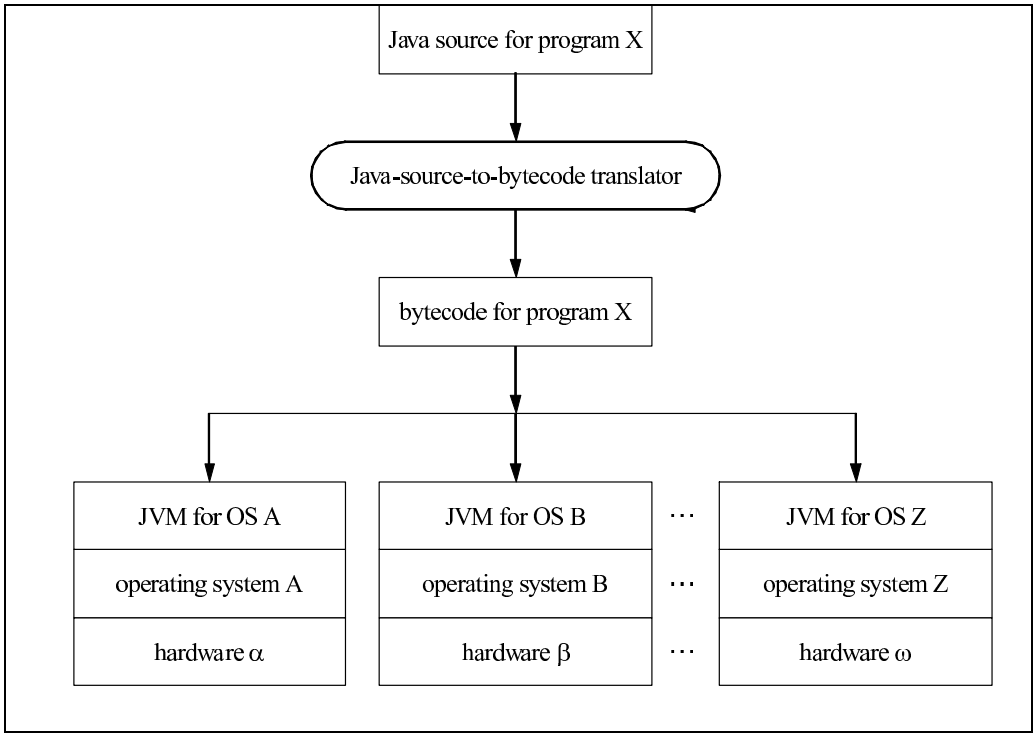

Figure 2.1. "Write once, run anywhere"

they can use encryption and/or signatures to provide confidentiality and/or authentication. However, if the target server of a jump does not trust the source machine, then the agent is downgraded to the untrusted "anonymous" status. Amongst machines in a single domain (i.e., controlled by the same organization) there is usually sufficient trust; trust between domains is problematic but outside the scope of D'Agents.

2.3 Flask

Below a JVM and below D'Agents code is an operating system. Within these software artifacts exist the same problems of trust that D'Agents faces explicitly and that the JVM faces implicitly. The problem, less delicately described, is spoofing or "bypass." An abstract architecture intended to address this issue is Flask [18], implemented, for example, in Security-Enhanced Linux (SE Linux) [43]. Minear presents a similar application of the Flask architecture to a Mach based system [33]. (See also Watson's description of TrustedBSD, a "trusted" form of FreeBSD [50].) The Flask architecture is shown in Figure 2.2. (Not shown in the figure is the existence of an object manager[1] for each type of object (e.g., process, file).)

[1]The object managers are part of the kernel but not part of Flask per se.

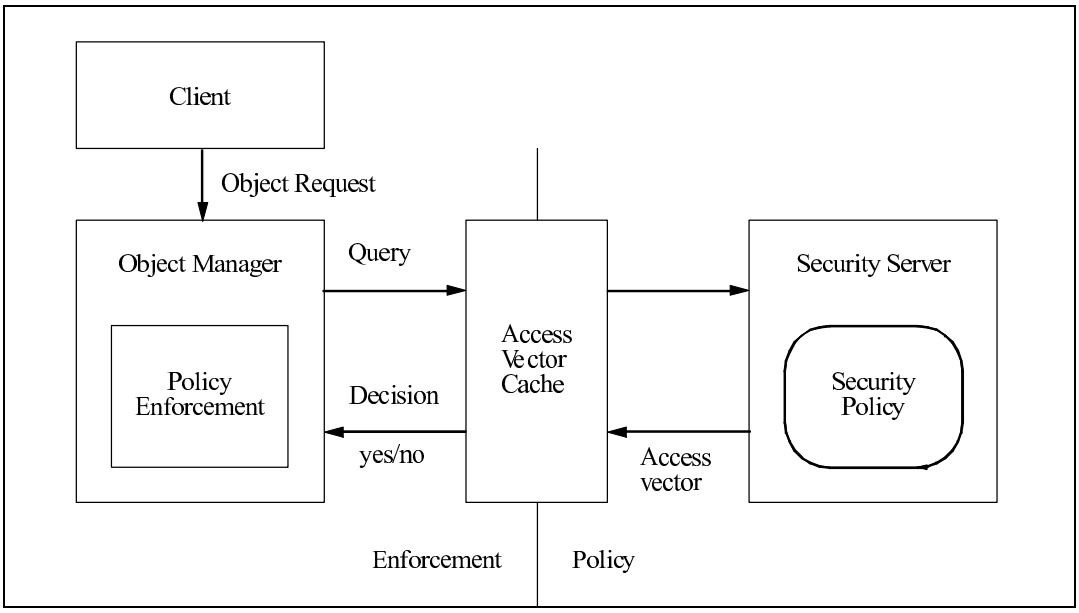

Figure 2.2. Flask Security Architecture (from [28] and [45]).

Flask provides flexible, fine-grained, mandatory access control. The figure shows some of the machinery that provides that capability. A Client (i.e., an application) requests approval to perform an action on an object. The request is directed to the manager of that object. This point in the kernel is referred to as a "control point." The object manager in turn queries the Security Server, effectively asking the question, "Can this subject take this action on this object?" The Security Manager bases its decision on (a) a security policy described in a file and (b) the label passed to the Security Manager from the Object Manager for the object in question. The Security Server returns a decision-yes or no. When the Object Manager has received the decision, it enforces it. The Access Vector Cache, placed between the Object Manager and the Security Server, reduces the performance penalty for the overhead of the architecture to an acceptable level, from about 500% down to approximately 5%. The description of the Security Policy resides in a file. Changing the file contents changes the policy. This ability is referred to as "flexibility."

Not shown in the diagram is the fine-grained nature of the access control in Flask. Flask presumes that every security-relevant access request, each control point, results in a query to the Security Server, the overhead for which is minimized by the Access Vector Cache.

The access control provided by Flask is intended to be mandatory. That is, once the Security Policy file is in place, there is a set of access control decisions that are made by the Security Server and are not at the discretion of the user (via the application) to make.

There are three general ways to mount a bypass attack on Flask:

1. spoof the Security Server;

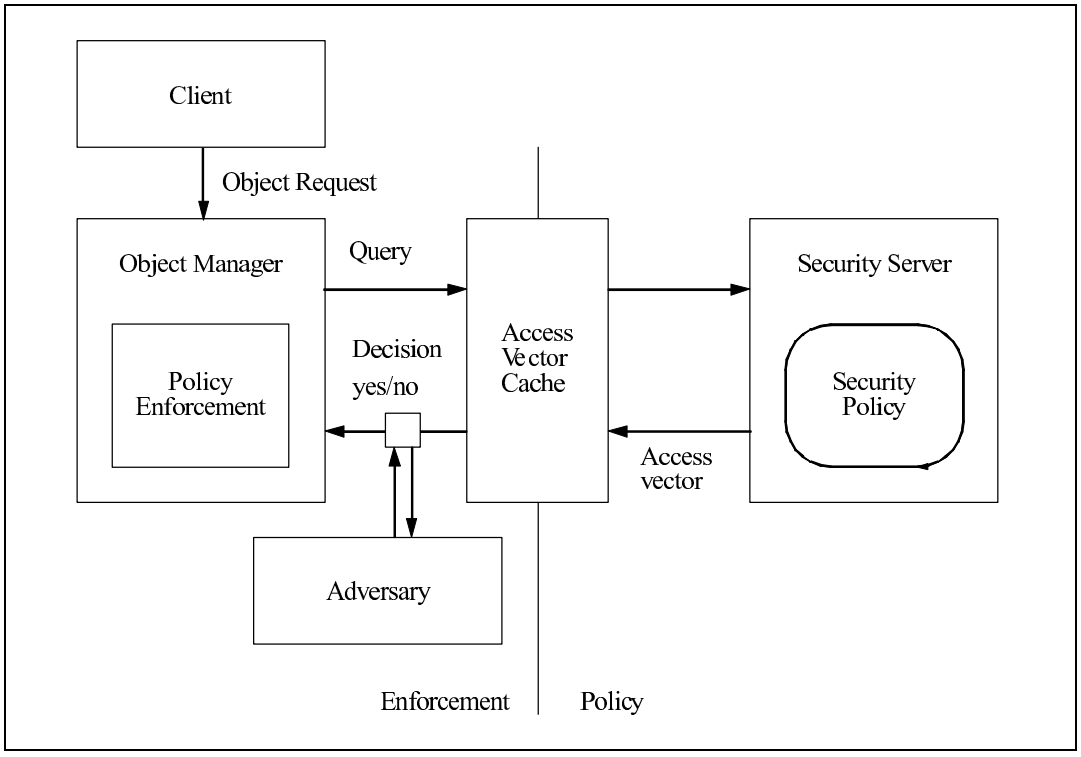

Figure 2.3. OS Bypass.

2. spoof the Security Policy; and

3. spoof the security labels.

The first attack is shown in Figure 2.3. After successfully mounting this attack the adversary implements a policy of the adversary's own choosing, including the policy that says that anyone can access anything.

The second attack involves changing the Security Policy, stored in a file. And the third attack involves changing the security labels of an object, making the object readable by anyone, for example.

A fourth attack, at least on SE Linux, focuses on the "security module," as explained here. When SE Linux was presented to Linus Torvalds at the 2.5 Linux Kernel Summit, Torvalds reportedly balked at the limitation that a single kind of Security Server represented. He suggested instead a "security framework." That is, he suggested that the Security Server also be flexible – defined in a file, in the same way that a Security Policy is defined in a file. As a result the Linux Security Module (LSM) architecture project began [27]. This architecture called for replacing the calls to the Security Server, at the control points in the kernel, with "hook functions" that would be defined by an LSM. Currently there are six LSMs available, including one for "SELinux." The LSM capability provides more flexibility, as well as providing another avenue of attack: spoof a legitimate security module.

There appear to be two general techniques to prevent bypass in Flask. The first is

referred to as "trusted path," meaning that between the Client and the Security Policy implemented in the Security Server there is only trusted code which is software in the "trusted code base" (TCB). The second consists of requiring code in the TCB to be used to change a Security Policy file, a security label, or a security module.

The TCB is necessary, since, as Loscocco et al. argue, the notion that "adequate security can be provided in applications [such as editors, compilers, and loader/link-editors] with the existing security mechanisms of mainstream operating systems" is a "flawed assumption" [29]. To attempt to do so results in a "fortress built upon sand," to use Baker's metaphor [5]. The problem is not so much the rain and the floods and the winds of Biblical metaphor, washing away the sand and leaving the fortress to topple over on its own. The problem is that the adversary can easily dig underneath the walls, effecting bypass.

In this Chapter we have reviewed three levels of distributed software for mobile code, exemplified by the JVM, the D'Agents code, and trusted operating systems. We have noted the persistence of the issue of trust, noteworthy by the extent to which Flask goes to attempt to circumvent it.

Chapter 3

Specialized Hardware Solutions

This chapter investigates methods for securing mobile code using specialized hardware techniques. The camp that believes special hardware is necessary points out that current computing architectures are "inherently insecure" because they are designed to execute ANY arbitrary sequence of instructions and are therefore subject to subversion by malicious code. By careful modification of current architectures, we can develop a cryptographic method of "tamper-proofing" code over a large portion of the software/hardware life cycle. This method culminates by decrypting/authenticating each instruction within a physically protected CPU, thereby protecting against subversion by malicious code.

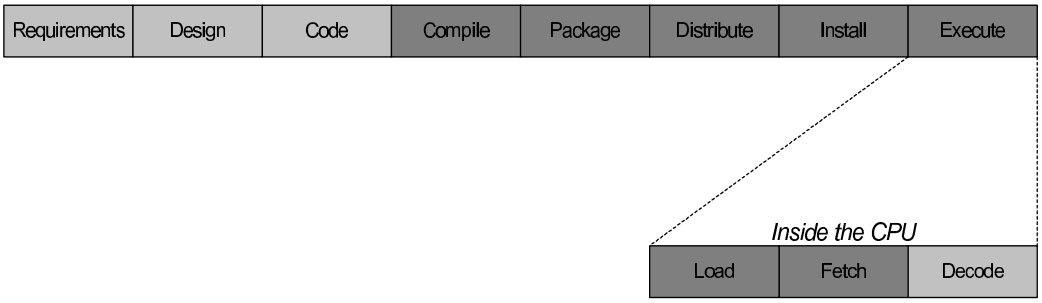

Figure 3.1. The goal is protection over a major portion of the software life cycle.

By moving trust around, cryptographically, the "trusted volume" that is required to be physically secure from subversion by an adversary can be made small, yet the ability to detect subversion and prevent execution of malicious code can be maintained. If the code intended to be mobile is encrypted and signed at its origin (within a trusted facility), and is decrypted and authenticated within a single integrated circuit containing the CPU, then the physical anti-tamper techniques can be applied to the single integrated circuit, rather than to a larger volume. It may be easier and less expensive to secure the smaller hardware volume than a larger volume.

Both encryption and authentication techniques must be applied to the code in order to protect it while outside the physically protected volume of the secured processor. En-

cryption is required to prevent reverse engineering of the code and to prevent bypass of the execution/access control mechanism. Authentication is required to assure the integrity of the code to be executed.

We define Faithful Execution (FE) as a software protection that begins when the software leaves the control of the developer and ends within the trusted volume of a target processor. That is, FE provides program integrity, even while the program is in execution. One method for providing FE involves the concept of a trusted kernel. However, trusted kernels require a lengthy accreditation process and do not protect the integrity of execution for processes that are being subjected to certain attacks (e.g., stack overflow attacks).

What is needed is a way to provide run-time code and data integrity checking that allows detection of tampering and provides confidentiality of the instruction stream. The method described here is to cryptographically "shrink-wrap" the program in a trusted verification facility, and to decrypt and authenticate the "shrink-wrapped" program within the protected volume of the integrated circuit containing the CPU itself. Properly designed and implemented, this method provides protection over a large portion of the software life cycle. This approach attempts to protect the confidentiality, authenticity, and integrity of code and data from the moment it is "shrink wrapped" at a trusted facility, through the distribution, storage, and loading phases, and up to the point where instructions or data are fetched from memory by the CPU. This method attempts to assure execution correctness or "faithful execution", presuming the correctness of the initial code and data has been validated.

The cryptographic "shrink wrapping" of the mobile code consists of encrypting and signing the instruction stream in a manner that authenticates the sequence of instructions (as well as the instructions themselves). This protected mobile code should be "unshrinkwrapped", as each instruction is fetched for execution, as close as possible to the execution process within the CPU. This could conceivably be done in protected hardware or in specially protected software. If this were performed in software, the program that decrypts, authenticates, and executes (perhaps in a virtual machine) the mobile code must be protected against reverse engineering by software obfuscation techniques which substitute for the "physically protected volume" that can be achieved in hardware. In addition, the decryption (and decryption key variables) must be protected against being used separately to decrypt the mobile code so that the code can not be executed by a non-decrypting (virtual or real) machine.

Key management for this technique can be further developed using public key or other techniques. The minimum requirement for securing mobile code is that the cryptovariables and means used to encrypt and sign the mobile code and to decrypt and authenticate the instruction stream at the point of execution must be protected from the adversary.

The following section describes a method of "unshrinkwrapping" and execution of mobile code in hardware. To expedite this research, the decryption, authentication, and execution is designed into reconfigurable hardware (a Field Programmable Gate Array (FPGA) or Programmable Logic Device (PLD)). The hardware designed into this FPGA can also be implemented in a "non-reconfigurable" Application Specific Integrated Circuit (ASIC). Physical anti-tamper techniques (that could be applied to such an FPGA or to an ASIC containing these functions) are beyond the scope of this research project.

3.1 FPGA Implementation

[1] The FE hardware architecture consists of three major components; the cryptographic Pre-processor, the interconnecting glue logic, and the Target processor as shown in Figure 3.2. The Target CPU, executes the desired instruction stream while the other, the Pre-processor, controls the Target's access to memory and performs the FE function.

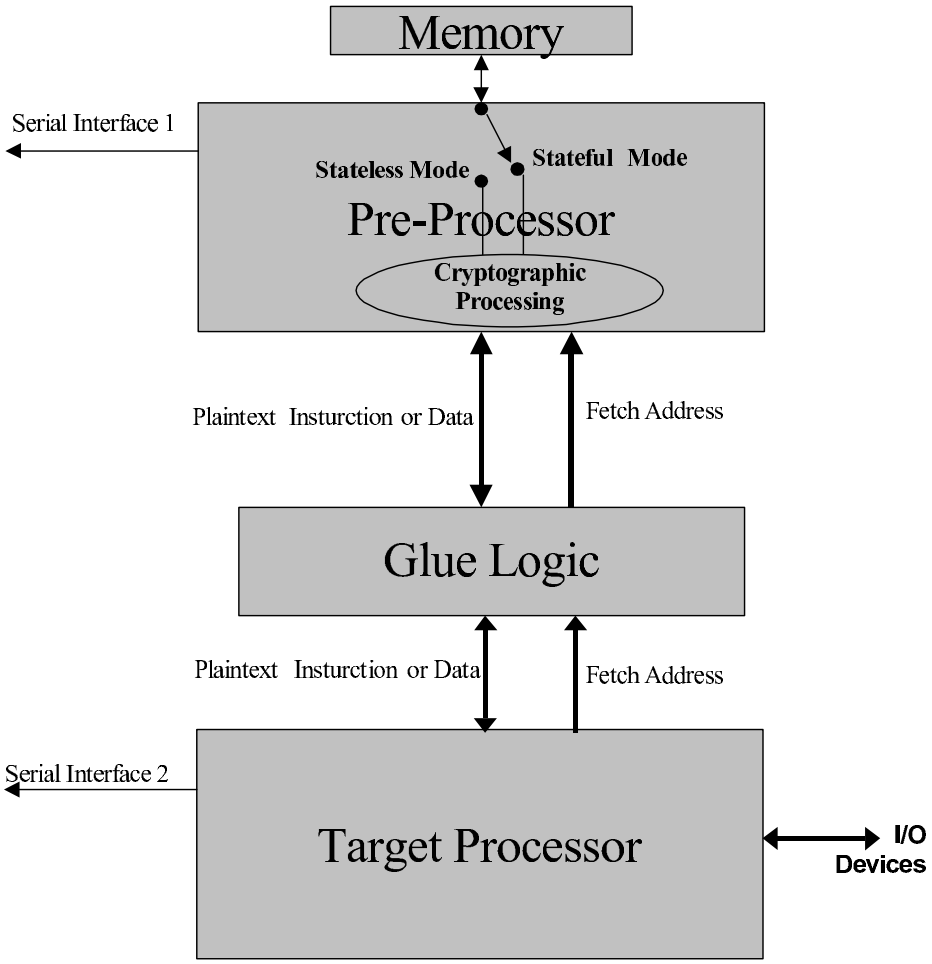

Figure 3.2. Major components of FE hardware architecture.

We have built a prototype FE engine in hardware using the latest generation of FPGAs. This specific implementation was developed using Altera's NIOS configurable soft-core embedded processor [35]. The NIOS processor offered many advantages to this project, in-

[1] Some of this work leverages work done in another LDRD called "Cryptographic Assurance of Execution Correctness"

cluding the ability to compile two Altera NIOS CPUs into a single Stratix 1S10 FPGA device.

Altera provides a very flexible bus architecture for NIOS called Avalon [3], with which we were able to implement numerous fully configurable ports. These ports were used to trigger interrupts instrumental in responding to instruction read and data write requests from the Target processor. Both the Pre-processor and the Target processor have separate Avalon buses. The Avalon bus is a parameterized interface to external logic blocks implemented as a switching fabric by Altera's SOPC Builder system development tool. The two Avalon buses are interconnected by glue logic that handles timing and signal handshaking.

The flexibility provided by the NIOS core was invaluable in the rapid prototyping of the FE processor. For example, we were able to implement the Pre-processor in software using one of the NIOS CPUs. As a prototype this arrangement provides significant flexibility, however for a more robust implementation that would execute faster, this function could later be replaced with special purpose logic written in a hardware description language.

The current implementation of the Pre-processor takes advantage of features in the NIOS CPU to improve performance by the use of custom hardware compiled instructions. We have compiled the processing intensive cryptographic functions that are required to perform the more complex bit and nibble manipulations into custom instructions that use the specialized hardware. The result was much faster executing cryptographic functions when compared to software implementations.

The software to decrypt and authenticate instructions and data, written in C, runs on the Pre-processor as a small kernel consisting of a "while loop" and Interrupt Service Routines (ISRs) to respond to read and write requests from the Target processor. The Target processor is a NIOS soft-core processor having 2K bytes of ROM, which contain a boot program called GERMS and a memory mapped, designer defined interface to the Pre-processor. The GERMS monitor is a small code segment provided by Altera that permits loading and execution of operating software. The designer defined interface includes address, data, and control lines. There are separate read and write data ports. The designed interface connects directly to the glue logic. All instructions and data are passed between the two NIOS processors via the designed interface and interconnecting glue logic.

The prototype operates as follows. Only the Pre-processor has access to application memory. The Pre-processor software and the Target processor distribution file are loaded into their respective memory locations using the Pre-processor's serial port. The Target processor boots its GERMS monitor (located in a local memory), and begins to accept commands via its serial UART. The Pre-processor software is started, enabling interrupts. An external command to the Target processor starts execution at the address of the distribution file. A request for the first instruction is sent to the Pre-processor. The Target processor waits for the Pre-processor to fetch and decode the requested instruction.

Each time a Target CPU application runs, several events occur. The application starts when the Target processor requests its first instruction from memory via the Pre-processor. This results in the address being latched and a read interrupt request being submitted to the Pre-processor. The Pre-processor places the Target CPU in a wait state and interrogates the application's header segment to determine the cryptographic context and retrieves the

correct keys from a secure memory location within the Pre-processor. After the first instruction request, the cryptographic context for that occurrence of the application is established and is not repeated for following instruction requests from that application occurrence. A new cryptographic context would necessitate a fresh context lookup.

The Pre-processor, upon receiving the read request, executes the ISR that fetches the instruction from memory. The ISR then performs any required decryption and authentication. Providing that the instruction is authentic, the Pre-processor passes the decrypted instruction back to the Target, again through the intervening glue logic, and removes the wait state. The Target then executes the instruction and increments its program counter to request the next instruction through the Pre-processor. Should the authentication check fail, the Pre-processor withholds passing the instruction on to the Target, and performs an exception handling process.

In this way, the Pre-processor/glue logic combination appears to the Target processor as a memory device with a variable cycle wait state. The application memory though, is not directly accessible by the Target processor, rather is only available through interrupt service requests to the Pre-processor. Data that the Target processor writes to memory is handled in a similar manner, but the data is encrypted before being written by the Pre-processor to memory, along with an authentication word.

3.2 Software Shrink Wrap Process

Faithful Execution protects software through a cryptographic process. The application of this protection, or the "shrink wrap" process protects both code confidentiality and code integrity by encoding and packaging the software at the instruction level. Individual instructions or sequential chains of instructions are encrypted to maintain confidentiality. Additionally, the encoding process includes redundant instruction information so as to allow authentication of each machine instruction and the execution order of the instructions.

After a trusted facility develops and certifies a piece of software as correct, it encodes the software for execution on the FE hardware. The encoding process repackages the software into five segments, that is, a software header, the encrypted software (instructions and data), the authentication data, initialization vector data, and Pre-processor instructions, as shown in Figure 3.3 and described below.

Implementation of FE permits several variations in its design. Both stateless and stateful instruction encryption as well as several encryption algorithms are possible. Additionally, it may be desired that different parts of a piece of software be protected in a different manner. In particular, the handling of a piece of software's code segment and data segment may be different. The shrink wrap header defines these variables to the Pre-processor as well as the memory size and the relative memory locations of the encoded software segments.

Our initial implementation of the FE hardware design uses the same 4-bit to 4-bit ECB encryption/decryption function as used with the Java VM implementation (extended through 8 slices for encryption of 32 bit words), again as a placeholder for a more robust

		Encoded Code:
		Header
Certified Code (Instructions & Data)	⇒	Encrypted Instructions & Data
		Authentication Data
		Initialization Vector
		Pre-processor Instructions

Figure 3.3. Shrink Wrapped Code Layout

cryptographic transform. This 4-bit ECB transform is sufficient for proof of concept. For the stateless FE implementation, application of this transform to various combinations of the instruction value and the instruction address provide the basis for deriving the encrypted instruction and the instruction authentication. We have developed several stateless and stateful schemes, using the application of the ECB transform, that employ between one and six 64-bit keys to produce the results for the encrypted data field and authenticated data field of the shrink wrap process. These schemes are being evaluated in terms of security robustness, performance, and complexity.

All of the information in memory consists of "values." We partition the universe of values into instructions and data. We further partition data into constants (values that can be stored in read-only memory) and what we call "variables" (values that must be stored in writable memory and which includes the stack and the heap).

The access pattern for instructions is always sequential. That is, we presume that if the current instruction is at address i, then the next instruction will be at address i+1 (except for instruction branches, of course). The access pattern for data - both constants and variables - is random.

Stateless protection can be used for either access pattern. Stateful protection is difficult and/or inefficient when applied to non-sequentially accessed data.

Writable data memory is subject to replay attacks since the value stored in any given location can change over the lifetime of the program. Read-only memory, on the other hand, is not subject to such attacks.

The initialization vector data segment and the Pre-processor instruction segment are necessary to handle the encryption and authentication of code sequences in a stateful FE implementation. The basis for stateful FE is "chaining" which is the cryptographic association of one instruction P_i with some other instruction P_j which in turn is associated with yet another instruction P_k to form a chain of instructions that are normally executed sequentially. The cryptographic mode we use as the basis for the association between instructions is Plaintext Block Chaining (PBC) [24], which requires an initialization value (IV) for the first instruction of a new sequence. At each code branch or jump instruction, the instruction chain sequence is broken, and a new IV is required. This data is stored in

the IV data segment.

Stateless protection implies that the cryptographic computations to decrypt and authenticate a given memory value are the same, regardless of the memory values previously processed. Therefore, in order to provide protection for the integrity of the sequence of instructions, the address of the memory value is incorporated into our stateless cryptographic processing. This complicates the authentication of relocatable code (in which the logical address as seen by the program is different from the physical address as seen by the underlying hardware). Since the processing of relocatable base addresses, etc. would be required to resolve such "address anchoring", our prototype only applies stateless protection to non-relocatable code.

Stateful protection implies that the cryptographic computations to decrypt and authenticate a given memory value depend on the successful decryption and authentication of all previously sequentially fetched values (since the most recent branch synchronization). In order to prevent the interchange of blocks of instructions (between branch targets, for example), the IV required to decrypt and authenticate the next block is only obtained by successful authentication of one of the appropriate previous instruction blocks. While this mode requires additional memory (the initialization values for each branch target), and requires identification of all branch targets for proper placement of the IV information, it is more easily applied to relocatable code and provides some additional independent protection for the instruction sequence.

Our prototype has been designed to implement both stateless (address anchored) and stateful decryption and authentication of instructions, and the stateless decryption and authentication of data, in order to evaluate the tradeoffs of robustness against subversion, processing performance, and implementation complexity. Now that the implementation details of these modes are established, we are carefully analyzing these tradeoffs, and evaluating the resulting implementations to gain insight into the relative robustness of these modes against subversion.

We have developed PC software that performs the encoding (shrink wrapping) upon the certified binary code. In our case, the binary code is an embeddable application we developed for the NIOS Target CPU. We develop the Target CPU application in a NIOS development environment, then move the binary file to a PC. There we execute the shrink wrapping program to cryptographically seal the binary application for transport to the FE environment.

3.3 Future Specialized Hardware Work

We have shown how mobile code can be computed securely in specialized hardware by using cryptographic assurance of execution correctness, in which decryption and authentication of the instruction sequence occurs only within a small, protected hardware volume. We have explained a method which cryptographically seals a piece of software at a code testing and certification facility, keeps it secure by distributing it in the encrypted form, and decrypts and authenticates it as it is being executed. We also described the related shrink wrap process to seal the certified mobile code for use in a specialize hardware environment.

Other work (not documented here but in reference [8]) describes a software prototype (of Cryptographically Assured Computation) in Java, where a Protection Engine was inserted between the Java Virtual Machine and the data store. While demonstrating some of the principles required, this software implementation lacks the obfuscation necessary to assure against reverse engineering of the decryption, authentication, and execution functions, nor to assure that the decryption function can not be used to separately decrypt and use the protected mobile code.

Ongoing work (if funded) will involve "black hatting" the hardware prototype to verify the security of the system and demonstrate proper operation. Additional work should be done to measure and optimize the instruction processing performance overhead incurred by several variations of the decryption and authentication processes. This technique should be experimentally applied to explore applications that could benefit from Faithful Execution and that can tolerate and justify the instruction processing overhead of the cryptographic operations. Future improvements could include inserting stronger cryptographic algorithms, more flexible modes of authentication, and faster hardware implementations to improve performance. Of great interest is the application of software obfuscation techniques to remove the requirement for the hardware based "protected volume".

Chapter 4

Software Solutions Requiring neither Specialized Software nor Hardware: Obfuscation Techniques

In this and the remaining chapters we examine methods and issues related to the notion of securing mobile code using neither specialized software nor hardware. The software must run on an adversary's computer, putting them in complete control of execution. The provider must assume that the adversary is able to run, stop, and restart the software at any point, reverse engineer components of the system, and see and manipulate all data. For software that must run on traditional hardware, an adversary can reverse engineer the entire program.

We will focus on using obfuscation to render an entire program or a data segment on which a program depends incomprehensible. The hope is to prevent or at least slow down reverse engineering efforts and to prevent goal-oriented attacks on the software and execution. The field of obfuscation is still in a state of development. We begin by giving a brief survey of important ideas in the field. We first present the top level of a taxonomy by Collberg[1][12] that categories sample obfuscation techniques. We use this to present several such techniques, as well as to provide structure for several obfuscation approaches that follow our discussion of Collberg's taxonomy. More examples of obfuscation can be found elsewhere [1].

4.1 Collberg's Taxonomy

Collberg presents a taxonomy of obfuscation techniques, each of which is referred to as a "transformation." The top level of the taxonomy consists of four categories of transformations: layout, control, data, and preventive. The first three categories can be intuitively defined based on the examples presented below. The last category is intended to "make

[1]For ease of reference we refer to Collberg, Thomborson, and Low's paper as though Collberg were the sole author.

Notation	Meaning	Examples

P,Q,R	A predicate. In Java, a boolean expression. In C this is an expression that evaluates to 0 (meaning false) or non-zero (meaning true).	If (P) ...
		If (Q) ...
		If (R) ...

P^T, Q^T, R^T	A predicate that always evaluates to true, independent of the values of any variables and functions used in the predicate.	If (1) ...
P^F, Q^F, R^F	A predicate that always evaluates to false, independent of the values of any variables and functions used in the predicate.	if (0) ...
$P^?, Q^?, R^?$	A predicate that sometimes evaluates to true and sometimes evaluates to false.	int i, j; read (i, j); if ($i > j$) ...
$< name >^{<=value>}$	A variable whose runtime value is known at compile time to be <value>.	int $i = 0$, $j = 1$; $i^{=0}$ $j^{=1}$
S<integer>	A series of statements, the precise nature of which is not important	S1; S2;

Table 4.1. Notation

known automatic [e.g., by a deobfuscator] techniques difficult" ([12], page 24).

We present one example of each category in Table 4.2 below. In Table 4.1 we explain the notation used in Table 4.2. The examples in Table 4.2 are presented in a high-level pseudo-code-sometimes to look like C and sometimes to look like Javathereby sidestepping advanced issues involving compiler optimizations and decompilers. All of the examples are from Collberg [12] except for the first.

Collberg considers that the "quality" of a given obfuscation transformation is a function of "potency," "resilience," and "cost." Potency is a measure of the strength of a given transformation against a human deobfuscator. Resilience is a measure of the strength of a given transformation against an automated deobfuscator. Cost is a measure of both the anticipated increased execution time and increased code size of a given transformation (and not the time or space required to perform the obfuscation).

4.2 Sander & Tschudin

Another technique for obfuscation is what is known as "secure function evaluation" or "computing with encrypted functions."[2] This is a data transformation in Collberg's scheme.

[2]There are generally two types of problems involved with the protection of data as opposed to function. One type, known as "secure multi-party computation" or "hiding data from an oracle," is typified by Yao's "Millionaires Problem" [52]: two millionaires want to find out who is wealthier but do not want to reveal to the other their wealth. The other type of problem is known as "computing with encrypted data" [39] (see also [1]) and uses homomorphic encryption schemes (see Chapter 6.

Transformation		Example	
		Before	After
Layout	Scramble Identifiers	int i, j, key;	int m1, function5, m2;

Control	[Convert] reducible to non-reducible flow graphs	do
		{
		S1;
		} while (P);
			1: if ($Q^?$)
			{
			2: S1;
			if (P) goto 3;
			goto 4;
			}
			else
			{
			3: S1;
			if (P)
			{
			if ($R^?$) goto 2;
			goto 1;
			}
			}
			4:

Data	Split variables	bool a, b;
		a = true;
		b = false;
		if (a) ... ;
		if (b) ... ;
			short a1, a2, b1, b2;
			a1 = 0; a2 = 1; // a = true;
			b1 = 0; b2 = 0 // b = false;
			// or: a1 = 1; a2 = 0; // a = true;
			// or: b1=1; b2=1; // b=false;
			int x = 2*a1+a2;
			if ((x==1) —— (x==2))...; //
			if(a)
			if (val (b1, b2))...; // if (B);
			int val (int i, int j)
			{
			if (i ==0)
			return (j);
			else
			return ((+1) % 2);
			}

<table>
<tr><th colspan="2" rowspan="2">Transformation</th><th colspan="2">Example</th></tr>
<tr><th>Before</th><th>After</th></tr>
<tr><td rowspan="2">Preventative</td><td>Use opaque predicates with side effects</td><td>{... S1; ... S2; ... }</td><td>int k = 0;
f();
{
k += 214748367;
// 2147483647 = $2^{31} - 1$
return (P^T)
}
g()
{
k -=2147483647;
return (P^T)
}
...
{ ...
if ($f()^T$)
S1;
if ($g()^T$)
S2;
...
}</td></tr>
<tr><td colspan="4">In the above example, if the adversary removes one but not both of the predicates, k will overflow and crash the executable, assuming that an int is stored in 32 bits using 2s complement</td></tr>
</table>

Table 4.2. Example Transformations (from Collberg's taxonomy)

For example the scheme that Sander & Tschudin [40] present uses additive and mixed multiplicative homomorphic properties of the Goldwasser-Micali probabilistic encryption scheme. That is, if we let "E(x)" denote the encryption of data item x, then there are efficient algorithms to compute E(x + y) given E(x) and E(y) and to compute E(xy) given E(x) and y. These properties enable Sander & Tschudin to protect computations that use polynomials. The scheme is restricted to polynomial/rational functions. Loureiro notes that whereas obfuscation in general lacks a theoretical foundation this approach applies to "more limited models such as circuits" and they have a "large complexity associated with each bit of output" ([22], page 27). We discuss this approach in more detail in Chapter 6.

4.3 Wroblewski

The approach presented by Wroblewski [51] operates on assembly language code and uses four obfuscation techniques:

1. reordering of instructions and blocks,
2. replacement,
3. simple insertion, and
4. complex insertion.[3]

The first and third techniques fit in Collberg's control category, with a little stretch. The other two techniques are not explicitly in Collberg's taxonomy.

Instructions and blocks that share no dependencies can be reordered. If dependencies are shared, then additional control structure will need to be added in the form of jumps to preserve the original ordering. Code that has no relevance to the current context can always be inserted. This is simple insertion. If for a sequence of statements there is an equivalent sequence, then that equivalent sequence can replace the original one. For example, the following two code fragments (from [2])

```
temp = a;
a = b;
b = temp;
```

and

```
a = a ⊗ b;
b = a ⊗ b;
a = a ⊗ b;
```

where a, b, and temp are all of the same scalar type and ⊗ denotes xor, are functionally equivalent but not equivalent in terms of the ease with which the general programmer will understand that they both swap values.

[3] Wroblewski does not name his two insertion types.

If the effect of the code on the current context can be later undone, then code that changes the current context can also be inserted, along with, at some other point in the program, the code that will undo that addition. This is complex insertion.

Note that Wroblewski presumes more analysis on a larger scale than does Collberg.

4.4 Wang

The approach presented by Wang [16] uses two techniques:

1. "flattening"[4] and
2. aliasing.

Collberg does not include flattening in the taxonomy. Collberg includes one mention of aliasing, but it is a severe subset of the technique Wang uses.

Flattening, as Wang describes it, is the process of converting the control structure of a procedure to a "universal" structure, as shown in Figure 4.1 (taken from Wang, Figure 4.4, page 66).

```
while ()
{
        switch ()
        {
            < procedure body >
        }
}
```

Figure 4.1. Universal Control Structure

The blocks[5] of the procedure become the statements of the switch. The statements that are at the same time both inside of the while statement and outside of the switch statement control the variable that determines which statement in the switch is executed next. For example, consider the procedure shown in Figure 4.2.

An equivalent universal structure for the code in Figure 4.2 is shown in Figure 4.3.[6]

[4]Not to be confused with Collbergs array flattening.

[5]A "block" is a sequence of instructions for which both of the following two conditions hold: (1) no instruction in the sequence is the target of a jump except possibly the first instruction, and (2) there is no jump instruction in the sequence except possibly the last instruction. That is, blocks are defined such that (a) jump targets, if any, are always at the beginning of a block and (b) jump instructions, if any, are always at the end of a block.

[6]Wang uses goto statements instead of a switch in this example.

```
int a, b;
a = 1;
b = 2;
while (a < 10)
 {
       b = a + b;
       if (b > 10 )
             b - -
        a++;
 }
 use b;
```

Figure 4.2. Sample Procedure

Note that the control-flow in Figure 4.2 has been converted into the data-flow in Figure 4.3. The code in Figure 4.3 is not difficult to decipher because the values assigned to swVar are hard coded. However, consider replacing statements such as "swVar = 2;" with "swVar = g[g[5] +g[g[23]]];" where g is a global integer array whose values are changed every so often during execution. (And now imagine adding several levels of indirection, as in g[g[***i]], where *i happens at the moment to be g[2] and **i happens to be &i.)

The other technique that Wang uses is aliasing: using more than one name for the same memory location. In general resolving aliases is undecidable. Several examples are shown in Figure 4.4.

4.5 Hohl

In this early paper Hohl [22] suggests the use of the following techniques:

1. "variable recomposition,"
2. "conversion of control flow elements into value-dependent jumps," and
3. "deposited keys."

Variable recomposition is what Collberg would call "split variables," a data transformation. The second approach approximates what Wang would call "flattening." The third approach gets values at runtime from some external source, thereby hindering static analysis. For example, the code could fetch the values that determine control flow. Riordan & Schneier use a similar approach in their "clueless" agents [37].

```
int swVar = 1;
while ( swVar < 7 )
{
     switch (swVar)
     {
          case (1):
               a = 1; b = 2;
               swVar = 2; break:
          case (2):
               if (!(a < 10)) swVar = 6;
               else swVar = 3;
                break;
           case (3):
                b = b + a;
                if (!(b > 10)) swVar = 5;
                else swVar = 4;
                break;
           case (4):
                b - -; swVar = 5;
                break;
           case (5):
                a++; swVar = 2;
                break;
           case (6):
                use (b); swVar = 7;
                break;
   }
 }
1:
```

Figure 4.3. Universal Structure Equivalent of Figure 4.2

```
Global and local reference aliasing:
    int *i = ...;
    main() { f(&i); ... }
    f (int **j) {
        int *k = *j;
        <k and i now point to the same location>
        ...
    }

Parameter aliasing:
    f (&i, &i);

Aliasing through return values:
    f () {
        int *i = ..., *j = ...;
        j = g ( &i );
        <i and j now point to the same location>
        ...
    }
    int *g (int **a ) { return ( *a ); }

Aliasing through side effects:
    f () {
        int *i = ..., *j = ...;
        g ( &i, &j );
        <i and j now point to the same location>
        ...
    }
    g ( int **a, int **b ) { a = b; }
```

Figure 4.4. Alias Examples

4.6 Ng

Ng [34] uses one technique: "intention obfuscation." This technique does not appear in Collbergs taxonomy.

Ng operates within the context of agents and he is interested in the "intention" of the code: what is the information that the owner wants to know? The easiest way to understand what Ng is talking about is to give an example. Consider the shopping agent shown in very high-level pseudo-code in Figure 4.5.

```
What is the price of your apples?
```

Figure 4.5. Shopping Agent

We presume that the "intention" of the agents owner (i.e., what the owner of the agents wants to know) is self-evident from the code in Figure 4.5, that the price of apples is really the information that is wanted. This intention could be a little obscured – or, as Ng would say, the entropy would be slightly increased - if the agent owner sent the agent shown in Figure 4.6 instead.

```
What is the price of your apples and
      the price of your oranges?
```

Figure 4.6. A Second Shopping Agent

The intention would be further obscured if the owner sent agents as shown in Figure 4.7 to additional stores, stores from which the agent owner would not consider buying (perhaps their quality is low).

```
What is the price of your pears and
      the price of your grapefruits?
```

Figure 4.7. A Third Shopping Agent

Even if an adversary were to receive the results of all of the agents, that adversary would still not be able to determine the intention of the agent owner.

Note that Ng is operating at a level above the code, at what he might call the "intention" level.

Chapter 5

White-Box Obfuscation

In this chapter we focus on an obfuscation technique different from the ones described in the previous chapter called "white-boxing". A system that is intended to be run on a malicious host is, by definition, not a black box because the adversary is able to view the program's execution as well as any intermediate results that are generated during computation. The *white-box attack context* was introduced by Chow et al. as a setting where the adversary is allowed to not only make such observations about the software, but is also able to examine and alter the software at will [11]. The system they implement in this context as a white-box program is a DES encryption algorithm. Ideally, a white-box encryption function would be so difficult to analyze that an adversary would be inclined to resort to a plaintext/ciphertext pairs attack, much as if the white-box implementation were a black box. Their implementation of DES does lend itself to some analysis, however, and a key can be extracted easily. We have built upon their ideas to create a similar white-box version of DES that is much less vulnerable to direct analysis.

One possible use for white-box cryptography would be the replacement of content encoding schemes such as the Content Scramble System (CSS) used to protect DVDs [13]. A chip could be made to perform decryption of DVD content using a white-box DES decryptor without revealing the encryption key, which an adversary would like to use to generate additional playable DVDs. As with CSS, such a system would still be vulnerable to duplication of raw data images, but it would be far more difficult for a criminal to generate properly encoded alterations or to extract the quantity of information needed to create a content player without the original content controls.

A second application for white-box cryptography of this sort would be in authenticating communications in low power processors in wireless broadcast situations. In such a setup, it would be desirable if, for example, directives broadcast from some central authority could be authenticated prior to being carried out. The obvious approach would be to sign the directives with the central authority's private key, and require that the nodes verify the signature before processing the command. However, in many situations, the nodes may not be powerful enough to verify the signature efficiently. With such low-power nodes, we would prefer to be able to use a symmetric-key message authentication code to verify the authenticity of the sender. Unfortunately, if a MAC is used, an adversary need only compromise a single node and recover the shared symmetric key in order to pose as the

central authority and issue counterfeit directives. Using white-box cryptography, we can hide the key from the adversary in a "verify-only" version of the code, which would prevent the attacker from issuing any counterfeit directives.

Chow et al. developed a white-box encoding for DES[17], however, their approach can be readily applied to other block ciphers such as Rijndael [14, 10]. The process of encoding reduces DES to small table lookups and then systematically reindexes and delinearizes those tables. In their publication of a white-box DES they admit vulnerability to a statistical bucketing attack on their encoding [11]. They address this vulnerability by augmenting DES with a nonstandard input and output permutation, but this makes their implementation a non-DES cipher. This may be a reasonable approach for many DRM applications but it is not appropriate for those requiring the use of standard encryption schemes. Jacob et al. [23] also performed some analysis of this DES encoding and described a fault-injection attack using differential cryptanalysis that reveals the DES key. This attack requires matching encodings of DES encryption and decryption, while most DRM applications would only include one or the other.

We have improved upon the work of Chow et al. and have implemented a white-box DES that is resilient to both the statistical bucketing attack described by Chow et al. and the differential cryptanalysis attack of Jacob et al. We have also implemented a modified statistical bucketing attack against Chow et al.'s DES that requires fewer encryptions by the target function than either of these prior attacks and requires only access to a white-box DES encryption function. Our alterations protect against this new attack as well. In addition, our implementation does not rely upon the input/output permutations described by Chow to protect against the statistical bucketing attack. Because of this, our implementation actually computes DES, and can interface with a standard implementation of DES without requiring knowledge of the input/output permutations. Our white-box DES is comparable in time and space requirements to its predecessor. We have profiled our implementation and believe that while it is substantially slower than a typical DES, the cost is appropriate to the threat model, the performance is acceptable for some uses, and a number of optimizations may still be made.

This chapter is organized as follows: Section 5.1 contains a summary of the creation of an instance of white box DES, which is provided for the reader who does not need the detail of Sections 5.2 and 5.3, but may need some background before moving on to the later sections of the chapter. A detailed description of white-boxing techniques is presented in Section 5.2. This is followed in Section 5.3 by a more precise look at how these techniques were applied to create a white-box implementation of DES. Section 5.4 reviews two known attacks against Chow et al.'s version of white-box DES, with an analysis of how our implementation prevents these attacks contained in Section 5.5. Section 5.6 discusses the use of these techniques in creating white-box versions of triple-DES and AES. Finally, Section 5.7 discusses optimizations that could be made to our implementation to yield smaller, faster versions.

$$y = Mx + v$$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{k-1} \end{bmatrix} = \begin{bmatrix} M_{0,0} & M_{0,1} & \cdots & M_{0,j-1} \\ M_{1,0} & M_{1,1} & \cdots & M_{1,j-1} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k-1,0} & M_{k-1,1} & \cdots & M_{k-1,j-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{j-2} \\ x_{j-1} \end{bmatrix} + \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{k-1} \end{bmatrix}$$

$$y_i = \left(\sum_{q=0}^{j-1} M_{iq} x_q \right) + v_i$$

$$m = ja, n = kb$$

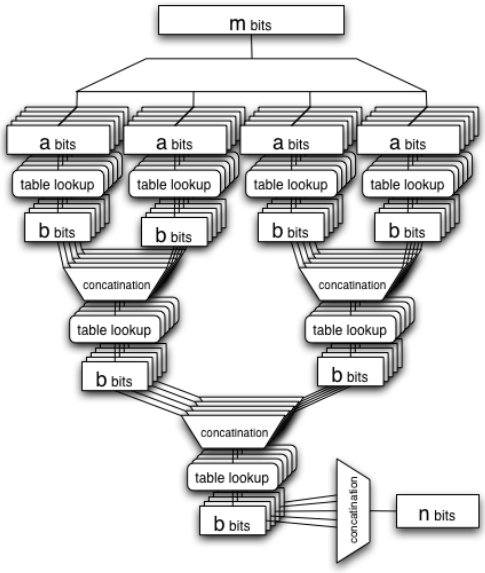

Figure 5.1. Decomposition of an affine transform

5.1 Overview of White-Box DES

Chow et al. and Jacob et al. described a white-box encoding of DES that is the basis for the implementation presented in Section 5.3. To provide motivation and context for the more in-depth descriptions that follow (or as review for those familiar with prior work), we briefly present the encoding of DES here. Definitions and details for specific elements of the white-box encoding process can be found in Section 5.2.

DES is usually described as a sequence of permutations, s-boxes, and xors on bit vectors. The permutations and xors are linear and can be represented by affine transformations (ATs), but because the s-boxes in DES are nonlinear, one can not represent DES as a single AT. We can, however, represent DES as alternating ATs and s-boxes. Chow outlines a process for re-representing an affine transformation with functionally equivalent code and data that cannot easily be used to recover the original AT. Once we represent an instance of DES (i.e., a key-specific DES) with ATs and s-boxes, we can prevent recovery of the ATs and simultaneously conceal the structure of the s-boxes to prevent recovery of the key that was used to generate them.

To represent DES permutations and xors for each round as a single AT, the s-box output must be accompanied by the original left and right input halves, L_r and R_r, for round r. This information is brought forward in parallel using 8-bit to 8-bit (8×8) T-boxes (obfuscated and key-impregnated s-boxes) that produce L_r, R_r, and the s-box results. R_r is included in the T-box output as the 16 duplicate bits passed into the s-boxes by the expansion permutation (EP), and 16 additional replicated bits of s-box input. The T-boxes allow the remaining DES components to be represented as matrix operations that are combined into a single AT in each round. In this manner, all of DES can be represented by alternating

ATs and T-boxes (Figure 5.3).

Chow et al. used a technique of matrix decomposition (Figure 5.1) to implement I/O-block encoding of affine transformations. The technique divides each AT, $Mx + v$, into independent equations that compute subvectors of the original result, y. Each subvector in the output is computed using a copy of the input vector, x, which is also divided into subvectors. Any subdivision of the input and output vectors will work. For clarity, Figure 5.1 depicts a uniform division of bit vector input into four a-bit vectors, and the creation of the output from five b-bit vectors. The first tables are 2^a-element multiplication tables with b-bit entries. The remaining tables are addition tables and have 2^{2b} b-bit entries, so tables that add two 4-bit values have 256 4-bit entries, while a table that adds two 8-bit values has 65,536 8-bit entries. We chose to divide input into 8-bit subvectors to match the 8-bit output of the T-boxes, and we initially divided the output into 4-bit subvectors to keep the addition tables from growing out of hand.

The delinearization step referred to by Chow et al. and Jacob et al. prevents an adversary from viewing the original contents of each table. Tables are delinearized by creating random permutations to rename their contents; for example, the elements of a table of 8-bit values would be renamed with a permutation of $[0 \ldots 2^8 - 1]$. The inverse of this permutation would be used to reindex the following table, which would subsequently have its contents renamed. For a system like DES, which can be represented entirely with ATs and table lookups, this process can be carried out on the entire implementation once the ATs have been tabularized.

Because the output of a table is delinearized, it is not possible to split the elements of the table into separate pieces without delinearizing the pieces separately. For this reason the block sizes available to us had to be divisors of eight. The smallest representation of the 96×96 matrices used to represent the rounds of DES would come from a 6×2 I/O-block encoding of the matrices, but such a division would have reduced the effectiveness of delinearization. For security, obfuscated blocks associated with the T-box input and output need to be as large as can be represented efficiently.

5.2 An Introduction To White-Boxing Techniques

How white-boxing is done can seem like a "black art" to those who are unfamiliar with the techniques. The existing literature on the subject is difficult to understand, and in some cases the available versions of the seminal papers contain errors in the details of the techniques and how they are applied. Furthermore there is no "compiler" or algorithm for mechanically converting an arbitrary function to its white-box equivalent. Instead, the individual techniques must be understood, and then one must determine when and how they can be applied to protect a given function.

The following sections are an attempt at providing a more accessible introduction to white-boxing techniques and terminology. In particular, Section 5.2.2 will describe the terminology and techniques of white-boxing, and Section 5.3 will show how those techniques can be used to protect a round of DES.

The terminology used here will follow closely that of [11].

5.2.1 White-Box Encoding Terminology

The primary goal when white-boxing is to encode a transformation so that some characteristic(s) of the transformation are hidden, despite an adversary's ability to see, and even manipulate, the internal computations of the transformation. In some cases, one may wish to hide the behavior of the transformation, or in others (such as DES), the behavior of the transformation may be publicly known, and one needs just to hide the key or some other portion of the data segment.

In the case of DES, the individual elements such as the s-box lookups and the permutations can be considered transformations, as can each round of DES, and the entire instance of DES itself. Having said that, we define encoding as follows:

Definition 1. Encoding: Let X be a transformation from m-bit inputs to n-bit outputs. Choose an m-bit bijection F and an n-bit bijection G. Call $X' = G \circ X \circ F^{-1}$ an *encoded* version of X. F is called an *input encoding* and G is called an *output encoding*. P' will be used to denote the encoded version of a transformation P. The transformation P' will then be represented as an s-box table, or a set of s-box tables.

Generally speaking, the bijections G and F will be non-linear transformations selected uniformly from the space of appropriately sized bijections. As an example of how this works, consider the affine transform A, implemented as an $m \times n$ matrix and an m-bit displacement vector. While a table describing A's input/output behavior would have 2^m entries, an adversary could determine the functionality of A by making just $m + 1$ queries to its black-box representation: One query with an all zero input reveals the value of the displacement vector, and an additional m queries, one for each of the canonical m-element basis vectors will yield the m rows of the matrix.

However, if one chooses random bijections G and F (n and m bit bijections, respectively) and creates the encoded $A' = G \circ A \circ F^{-1}$, and presents that to the adversary, then the above attack for reconstructing the transformation is nullified (see Section 5.2.3 for a brief discussion of the security of this encoding technique).

Typically, the inputs or outputs of a transformation will be too large to efficiently represent in a single table[1]. In some cases, it is possible to encode these transformations by using the concatenation of smaller bijections to construct a larger encoding.

Definition 2. Concatenated encoding: Consider bijections F_i of size n_i, where $n_1 + n_2 + \cdots + n_k = n$. The *function concatenation* $F_1 \wr F_2 \wr \cdots \wr F_k$ is the bijection F such that, for any n-bit vector $b = (b_1, b_2, \ldots, b_n)$, $F(b) = F_1(b_1, \ldots, b_{n_1}) \| F_2(b_{n_1+1}, \ldots, b_{n_1+n_2}) \| \cdots \| F_k(b_{n_1+\cdots+n_{k-1}+1}, \ldots, b_n)$, where $\|$ denotes vector concatenation. For such a bijection F, we have $F^{-1} = F_1^{-1} \wr F_2^{-1} \wr \cdots \wr F_k^{-1}$.

Often, the output of one encoded transformation will be used as the input to another

[1] For example, some of the transformations in white-box DES have 96-bit inputs and outputs which would require a table with 2^{96} 96-bit entries to represent.

transformation. When this occurs, it is vital that the output encoding of the first transformation correspond to the input transformation of the second.

Definition 3. Networked encoding: A networked encoding for computing $Y \circ X$, where the output of transformation X is used as the input of transformation Y, is an encoding of the form:

$$Y' \circ X' = \left(H \circ Y \circ G^{-1}\right) \circ \left(G \circ X \circ F^{-1}\right) = H \circ (Y \circ X) \circ F^{-1}.$$

Throughout this report, we will use the following notation to describe the sizes of the inputs and output of transformations, as well as the sizes of the values being transformed:

${}^{n}_{m}P$ will be used to emphasize that the transformation P takes m-bit input vectors and maps them to n-bit output vectors. For a matrix M, ${}^{n}_{m}M$ will denote a matrix with n rows and m columns, and for vectors ${}_{k}v$ represents a vector of k elements. ${}^{k}_{k}\mathbf{I}$ denotes the identity function on k-bit inputs. ${}^{n}_{m}\mathbf{e}$ denotes some *entropy transference function*. That is, some transformation from m-bit inputs to n-bit outputs such that if $m \leq n$ the mapping loses zero bits of information, and if $m > n$ then the mapping loses at most $n - m$ bits.

$\langle e_1, e_2, \ldots, e_k \rangle$ is a k-bit vector with elements e_i. Concatenation of vectors x and y is denoted $x \| y$. The ith element of a vector v is denoted v_i, and $v_{i..j}$ denotes the subvector of V containing elements of i through j inclusive, and ${}_{k}v$ denotes that v contains k elements. ${}_{k}\mathbf{e}$ denotes any vector with k elements, sometimes referred to as an *entropy vector*.

AT (affine transformation) denotes a vector-to-vector transformation, P, which can be defined for all ${}_{m}\mathbf{e}$ by ${}^{n}_{m}P({}_{m}\mathbf{e}) = {}^{n}_{m}M\,{}_{m}\mathbf{e} + {}_{n}d$, or by $P(\mathbf{e}) = M\mathbf{e} + d$, where M is a constant matrix, and d is a constant displacement vector. For this report, we only consider ATs over GF(2). Note that if A and B are ATs, then so are $A \wr B$ and $A \circ B$ (where defined).

5.2.2 White-Box Encoding Techniques

Ideally, one would like to represent the transformation in question as a single lookup table, as this format is equivalent to providing black-box access to the transformation, as all that is known about the transformation are its input/output pairings. Unfortunately, if the transformation is something like DES, this approach becomes infeasible, as the 64 bit input space would require tables that are simply too large to store. To avoid this problem, one can instead use a greater number of look-ups into a greater number of tables – similar to DES's use of eight 6-bit to 4-bit s-boxes instead of a single prohibitively large 48-bit to 32-bit s-box[2].

Of course, this approach brings with it a dramatic decrease in the security of the system. With a single s-box, a single bit difference in the key can affect all 32 bits of the output in a given round (for a fixed input). With DES's eight s-boxes, however, a single bit change in the key can affect at most four bits of the output in any given round. Furthermore, a single bit change in the input goes from affecting up to all 32 bits of the output to affecting at most 8 bits of the output, so an attackers work is cut down significantly. DES, of course,

[2]In truth this would be a 32-bit to 32-bit s-box as the expansion permutation of DES is a monomorphism over its 32-bit input space.

addresses this problem through the use of the expansion and P permutations, and by using multiple rounds to cascade these local effects over the entire block. With white-boxing, one needs to do more, as an adversary can observe the computation at any intermediate step. As a result, all atomic computations need to be secured individually – one cannot rely on a cascade of protection mechanisms.

The techniques described in the following section describe a way to build these secure atomic computations that can be composed to provide security for a cipher like DES, even in the white-box attack context.

5.2.2.1 Partial Evaluation

When part of the input to the transformation P is known at the time of the white-box encoding, one can often insert the input into the transformation prior to the encoding process. For example, given the key, one can partially evaluate the DES transformation by replacing the standard s-boxes in DES with s-boxes that have been re-indexed according to the appropriate round key. This set of key specific s-boxes will then be encoded using other white-boxing techniques.

5.2.2.2 Mixing Bijections

Mixing bijections are non-sparse bijective ATs that are used to increase the dependence of the output bits of some AT on all of its input bits. In DES, for example, when the permutations and xor operations are represented as ATs, they have a very sparse form, and each output bit depends only on one or two input bits. We can use mixing bijections to hide the sparse nature of the transformation. Consider such a sparse transformation P. We can generate a mixing bijection K, and compute $J = PK^{-1}$, so that $P = J \circ K$. In essence, this factors P into two new, non-sparse ATs whose output bits are all dependent on a large number of the input bits. Note that in isolation this too can be defeated by an adversary, by passing $m + 1$ values through both ATs and reconstructing the original AT as before.

5.2.2.3 I/O-Block Encoding

When encoding a transformation ${}^{n}_{m}P$ where m and n are large[3], we cannot simply wrap P with m and n bit bijections and present the encoding as an s-box table. This is because the size of the resulting table will be $O\left(n2^m\right)$.

In order to make encoding such transformations practical, we use I/O-block encoding, which divides the input and output into smaller pieces. That is, we divide the input to P into j blocks of a bits each, and divide the output into k blocks of b bits each, so $m = ja$ and $n = kb$. Now, rather than selecting bijections ${}^{m}_{m}F$ and ${}^{n}_{n}G$ to wrap around P, we will select ${}^{a}_{a}F_1, {}^{a}_{a}F_2, \ldots, {}^{a}_{a}F_j$ and ${}^{b}_{b}G_1, {}^{b}_{b}G_2, \ldots, {}^{b}_{b}G_k$ as encoding bijections for the blocks of the input

[3]Actually, the primary concern is when m is large, as the size of the resulting s-box varies linearly with n. Of course, this n bit output is likely to be passed as input to another transformation, where the size of n will be important.

and output. We then define $F_P = F_1 \wr F_2 \wr \cdots \wr F_j$ and $G_P = G_1 \wr G_2 \wr \cdots \wr G_k$[4], and define $P' = G_P \circ P \circ F_P^{-1}$ as before.

To use this technique, we will still need to split the computation of the original transformation P into smaller blocks. A method for doing this is presented in Section 5.2.2.7

5.2.2.4 Combined Function Encoding

Let P and Q be two transformations that can be performed in parallel. Instead of encoding them separately, it may be better to create an encoding of $P \| Q$, such as $G \circ (P \| Q) \circ F^{-1}$, using encodings G and F that span the concatenated input and output of $P \| Q$. Encoding the two transformations together this way mixes P's input and output entropy with Q's, making it harder for an adversary to determine the original functions P and Q. Note that this is not the same as concatenated function encoding as described in Section 5.2.1, as the encoding is applied to P and Q together, and not separately.

5.2.2.5 Bypass Encoding

It is sometimes desirable to add extra entropy to the inputs and outputs of a transformation; to make it harder to deduce the unencoded transformation for example. We can use a special case of combined function encoding, called bypass encoding to do this. For example, given a transformation ${}^{n}_{m}P$, we can add an extra a bits of input entropy and b bits of output entropy by selecting input and output encodings F and G, along with an appropriate entropy-transference function ${}^{b}_{a}\mathbf{E}$,[5] and creating ${}^{n+b}_{m+a}P' = G \circ \left(P \| {}^{b}_{a}\mathbf{E}\right) \circ F^{-1}$.

We call ${}^{b}_{a}\mathbf{E}$ the bypass component of P'. In the case where ${}^{b}_{a}\mathbf{E} = {}^{a}_{a}\mathbf{I}$, we call it the identity bypass.

5.2.2.6 Split Path Encoding

A special case of concatenated function encoding is split-path encoding. With split-path encoding, one creates a single encoding that is actually a concatenation of two separate encodings applied to a common input. For example, given the encoding ${}^{n}_{m}P$, one can define the new encoding ${}^{n+k}_{m}Q\left({}_{k}\mathbf{e}\right) = {}^{n}_{m}P\left({}_{k}\mathbf{e}\right) \wr {}^{k}_{m}R\left({}_{k}\mathbf{e}\right)$ for all ${}_{k}\mathbf{e}$ and for some fixed function ${}^{k}_{m}R$. The effect of this is that if P is a lossy transformation, the resulting Q may lose fewer (or no) bits of entropy from ${}_{k}\mathbf{e}$.

[4] In practice, we recommend also selecting the mixing bijections ${}^{m}_{m}J$ and ${}^{n}_{n}K$, and defining F_P and G_P as $(F_1 \wr F_2 \wr \cdots \wr F_j) \circ J$ and $(G_1 \wr G_2 \wr \cdots \wr G_k) \circ K$ respectively, in order to hide the likely sparse nature of the smaller transformation blocks.

[5] Recall, by definition the entropy-transference function must be lossless, so we must have that $a \geq b$.

5.2.2.7 Wide-Input Encoded ATs

Recall from Section 5.2.2.3 that we cannot represent encoded wide-input transformations as single s-boxes, as the size of the required tables varies exponentially with the size of the input to the transformation.

In order to use the I/O-blocked encoding technique, we need a way to split the computation of the transformation being encoded into reasonably sized blocks. For a wide-input AT A, we can do this by partitioning the matrix and vector component of A, as well as the input vector into smaller blocks, then using these blocks to subdivide the computation of A via standard linear algebra techniques. For example, rather than performing the following computation:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} & M_{0,4} & M_{0,5} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} & M_{1,4} & M_{1,5} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} & M_{2,4} & M_{2,5} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} & M_{3,4} & M_{3,5} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} + \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

We instead perform the following series of computations:

$$\begin{bmatrix} y_{00} \\ y_{10} \end{bmatrix} = \begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ M_{1,0} & M_{1,1} & M_{1,2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}, \quad \begin{bmatrix} y_{01} \\ y_{11} \end{bmatrix} = \begin{bmatrix} M_{0,3} & M_{0,4} & M_{0,5} \\ M_{1,3} & M_{1,4} & M_{1,5} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$\begin{bmatrix} y_{20} \\ y_{30} \end{bmatrix} = \begin{bmatrix} M_{2,0} & M_{2,1} & M_{2,2} \\ M_{3,0} & M_{3,1} & M_{3,2} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix}, \quad \begin{bmatrix} y_{21} \\ y_{31} \end{bmatrix} = \begin{bmatrix} M_{2,3} & M_{2,4} & M_{2,5} \\ M_{3,3} & M_{3,4} & M_{3,5} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix}$$

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} y_{00} \\ y_{10} \end{bmatrix} + \begin{bmatrix} y_{01} \\ y_{11} \end{bmatrix} + \begin{bmatrix} v_0 \\ v_1 \end{bmatrix}, \quad \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} y_{20} \\ y_{30} \end{bmatrix} + \begin{bmatrix} y_{21} \\ y_{31} \end{bmatrix} + \begin{bmatrix} v_2 \\ v_3 \end{bmatrix}$$

And our final result will be the concatenation of the two smaller vectors we obtained above.

Once subdivided in this manner, the transformation can be represented as a series of s-boxes which are now practical to store in memory. A great deal of space can be saved this way: recall that an m-bit to n-bit transformation would require $n2^m$ bits of storage to represent as a single table. After partitioning the input and output into j blocks of a bits in length and k blocks of b bits in length, respectively, this transformation can be represented as k sets of tables, where each table contains j a-bit to b-bit multiplication tables and $j-1$ $2b$-bit to b-bit addition tables[6].

One problem with this approach is that some blocks of the AT one wishes to encode may be singular, potentially zero, sub-matrices. The output of these singular blocks can contain

[6]Note that addition tables for a particular size are identical, but they must be duplicated and each encoded separately. The constant vector component of the AT may be folded into a tree of additions at any point and so does not add a table.

little to no entropy from the input, reducing the amount of work an adversary would need to do to reconstruct the AT. This situation can be avoided by using carefully selected mixing bijections (see Section 5.2.2.2). Rather than simply encoding the AT ${}^{n}_{m}A$, select ATs ${}^{n}_{m}A_1$ and ${}^{n}_{m}A_2$, such that $A_1 = A \circ A_2^{-1}$, then encode A_1 and A_2 separately into sets of s-boxes, and use the outputs of the A'_2 as the inputs to the A'_1 transformation. While this technique does not guarantee that there will not be singular blocks in either A_1 or A_2, it should lessen the amount of information leaked about A to an adversary.

That is, rather than simply encoding the AT ${}^{n}_{m}A$, select ATs ${}^{n}_{m}A_1$ and ${}^{n}_{m}A_2$, such that $A_1 = A \circ A_2^{-1}$, then encode A_1 and A_2 separately into sets of s-boxes, and use the outputs of the A'_2 as the inputs to the A'_1 transformation. While this technique does not guarantee that there will not be singular blocks in either A_1 or A_2, it should lessen the amount of information leaked about A to an adversary.

5.2.3 Bijective Encoding and Local Security

Given an encoded transformation $P' = G \circ P \circ F^{-1}$, we note that if the original transformation P is a bijection, then we can consider P' to be *locally secure*, that is, no information can be gained about P given just the s-box for P'. This is because given P', every possible bijection is a candidate for P, given appropriate choices for F and G. This is analogous to a one-time pad cipher where, given a ciphertext, every plaintext is possible, given the appropriate key.

When P is lossy, the encoded transformation P' is no longer locally secure, as it leaks some information to an adversary. An extreme example of this is a P whose output is 0 for all input. Given the corresponding P' an adversary can simply restrict his search space to constant functions. If one is not careful, even slightly lossy transformations can yield enough information to an adversary to enable an attack – such is the case with the statistical bucketing attacks discussed in Section 5.4.1.

5.3 White-Boxing Example: DES

The descriptions alone of the techniques in Section 5.2.2 may not help one create an encoded version of a transformation. To further clarify the ways these techniques can be used, we provide here a detailed look at the white-boxing of DES.

5.3.1 Unobfuscated DES

The structure of DES is shown in Figure 5.2. The incoming data block is split into two halves, where the right half is copied, with one copy becoming the left half of the input for the next round, and the other copy being mutated via a series of operations: An expansion permutation, an xoring of a round key, a pass through a set of substitution boxes, and a pass through another permutation before it is xored with the left half of the input. After 16 rounds of this transformation, the two halves are output, with the mutated right half

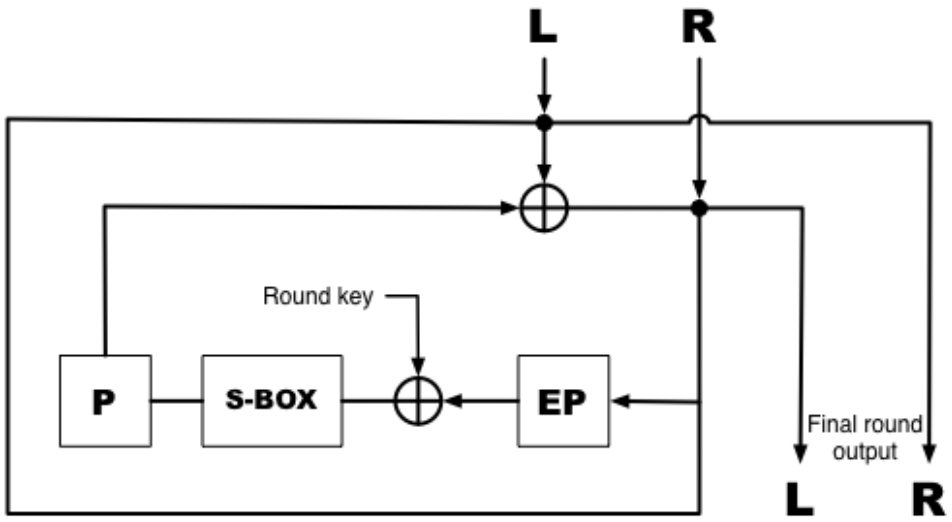

Figure 5.2. The basic structure of DES

forming the left half of the output, and the input right half remaining unchanged as the right half of the output.

There are four main portions of DES that need to be obfuscated: The linear portion of the cipher before the first round s-boxes (i.e. the input permutation, the copying of the right half, and the expansion permutation); the s-boxes for each round; the linear portion of DES between s-boxes for rounds 1-15 (i.e. the P permutation, xoring with the left half, swapping the input right half for the next round's left half, and the expansion permutation of the next round); and the linear portion of DES following the s-box in round 16. The following sections will discuss how each of these four portions are encoded and tied together to create a white-box version of DES.

5.3.2 White-Box DES

The high-level process of the white-box obfuscation of DES is to take the linear portions of the cipher, represent them first as ATs, then convert the ATs to lookup tables, and use non-linear input and output transformations to delinearize the tables, making them resistant to analysis. Similarly, take the non-linear s-boxes, partially evaluate them with respect to the round keys, and protect the keys by wrapping the s-boxes with input and output encodings chosen so as to enable networked encoding with the linear portions of the cipher.

The resultant cipher is, essentially, a large set of lookup tables that are cascaded through to produce the final output value. The steps taken to create those tables are described below.

Figure 5.3. DES as a sequence of table and matrix operations

5.3.2.1 White-Box DES – The T-Boxes

At the heart of DES lie its s-boxes — the non-linear substitution tables that provide the confusion aspect of the cipher. The same is true of the substitution boxes in the white-box version of the cipher, and perhaps even more so, as the encoded s-boxes, denoted T-boxes, have the round keys contained within them.

In unobfuscated DES, the s-boxes are lossy transformations, moving from 6-bit inputs to 4-bit outputs. As outlined in Section 5.2.3, if we were to simply wrap the lossy s-boxes with input and output transformations, it could leak some information to an adversary. For example, an adversary could always determine which T-box corresponded to s-box 4. To combat this problem, we convert the s-boxes into 6-bit to 6-bit (6×6) bijections by passing the two bits of the input thrown away by the s-box through as part of the output. Not only does this allow us to achieve local security for the T-boxes, but it also allows us to reconstruct the input to the s-boxes in the M_2 blocks (see Section 5.3.2.3).

In addition to converting the s-boxes to bijections, we also increase both the number and the size of the T-boxes so that there are now twelve 8×8 T-boxes. For reasons that are explained in Section 5.3.2.2, we need to carry the 32 bits of the left half of the data, as well as an extra 16 bits from the right half (denoted as REPL[7]) through the T-boxes in addition to the 48 bits of output from the DES expansion permutation. To do this, for each round r, eight of the twelve T-boxes are chosen at random during the construction of the previous block (see Sections 5.3.2.2 and 5.3.2.3) and will provide the functionality of the bijective versions of the s-boxes. These eight T-boxes will take as input six bits

[7]Because these 16 bits are *not* otherwise replicated by the DES expansion permutation, they must be explicitly called out.

from the expansion permutation, and two of the left half/REPL bits, and are encoded as ${}^{8}_{8}T_{r,j} = G_{r,j} \circ \left({}^{6}_{6}s'_i \| {}^{2}_{2}\mathbf{E}\right) \circ F^{-1}_{r,j}$, where ${}^{6}_{6}s'_i$ is the bijective version of s-box s_i. The other four "dummy" T-boxes simply pass eight bits of input from the left half/REPL bits through to the M_2 and M_3 transformations, and are encoded as ${}^{8}_{8}T_{r,j} = G_{r,j} \circ {}^{8}_{8}\mathbf{E} \circ F^{-1}_{r,j}$.

Finally, note that our input and output transformations are 8×4 bijections, i.e. $G_{r,j} = {}^{8}_{8}G_{r,j}$, where Chow et al. used two 4×4 transformations per T-box. Their use of the same input and output size when I/O-block encoding the ATs for DES could be seen as an oversight, failing to realize that the AT matrices could be split asymmetrically. More likely it was done to reduce the size of the encoded cipher. However, it resulted in a large negative effect on the security of the obfuscation, enabling the two attacks described in Sections 5.4.1 and 5.4.2.

5.3.2.2 White-Box DES – The M_1 Block

In unobfuscated DES, the linear portion of the cipher that precedes the first round's s-boxes consists of two operations: an input permutation that permutes the bits of each input block, and an expansion permutation (EP) that replicates 16 of the 32 bits in the right half of the input following the input permutation. The now 48-bit right half is then passed on to the s-boxes for further transformations. By representing these permutations as matrices, it is possible to convert these two steps into an AT consisting of a single 64×80 matrix, which can then be split into tables and encoded using the I/O-block and wide input encoding techniques described in Sections 5.2.2.3 and 5.2.2.7. However, more needs to be done to enable the later parts of the cipher to work properly, as well as to increase security of the obfuscation.

Most of what needs to be done to create the initial encoded section of the cipher (referred to as the M_1 block) is necessary because the left half of the input is not affected by the s-boxes[8]. A naive way to handle this situation would be to simply create a bypass channel for the left-half bits, and never pass them through the s-boxes. Unfortunately, this approach can leak information to an adversary.

To protect the left half we must pass both halves of the input through our encoded s-boxes. As explained in Section 5.3.2.1, this is done by using twelve 8×8 T-Boxes rather than the eight 6×4 s-boxes of unobfuscated DES. With the extra capacity of the T-boxes available, we prepare the data to pass into them as follows:

First we need to compute the EP and create the REPL bits, i.e. replicate the 16 bits of the right half of the input that were not replicated by EP. This allows us to later reconstruct the right half of the data to use as the left half input for the next round. We do this by using a 64×96 matrix, ${}^{96}_{64}R$, that takes in the output of the DES initial permutation (computed using a sparse 64×64 bit matrix, ${}^{64}_{64}IP$), and outputs a vector where the top 32 bits are the left half of the data, the next 16 bits are the REPL bits, and the bottom 48 bits are the result of the EP.

[8]When referring to the left or right halves of the input, we mean this in the standard cryptographic sense as the left and right halves after passing through the initial permutation. For most discussion of DES, its initial and final permutations are treated as if they did not exist.

The next item to change is to pass the bits from the left half of the input and the REPL bits through a bijective AT. We do this to mix the input entropy of these bits together, so that when the output bits are distributed amongst the twelve T-boxes, any single bit change in the left half of the DES input will likely cause a change in the output of all twelve T-boxes, rather than a localized change in just a single T-box. This mixing is done with a 96×96 bit matrix with a randomly selected bijection, ${}^{48}_{48}\rho_1$, in the upper-left quadrant, ${}^{48}_{48}\mathbf{I}$ in the lower-right quadrant, and zeros elsewhere (i.e. an identity bypass transformation of ${}^{48}_{48}\rho_1$).

Finally, a permutation matrix, ${}^{96}_{96}\delta_1$, is used to group the bits to be passed into the T-boxes.[9] That is, the 48-bit output of EP is partitioned into 6-bit chunks that will be passed into the appropriate T-boxes – more specifically, some random permutation, P, of the elements $[1 \ldots 12]$ is chosen, and bits $6i-5+1 \ldots 6i$ of the output of EP are positioned so they will be passed into T-box $P(i)$. The remaining bits from the left half and REPL are randomly permuted to fill the remaining 48 bits.

The product of these four matrices can be computed and encoded using the I/O-block and wide-input encoding techniques[10], resulting in an encoded transform, ${}^{96}_{64}M_1 = F \circ (\delta_1 \circ (\rho_1 \| \mathbf{I}) \circ R \circ IP) \circ G^{-1}$, where $F = F_{1,1} \wr F_{1,2} \wr \cdots \wr F_{1,12}$ with non-linear bijections ${}^{8}_{8}F_{1,i}$, and $G = {}^{64}_{64}\mathbf{I}$.[11] The twelve 8-bit blocks of output from M_1 are then passed into the first round's T-boxes where the input encodings are $F_{1,1}^{-1} \cdots F_{1,12}^{-1}$ in accordance with the networked encoding technique described in Section 5.2.1.

5.3.2.3 White-Box DES – The M_2 Blocks

The M_2 transformations are the encodings of the linear portions of DES that lie between the s-box substitutions. Much like M_1 and M_3, they are the I/O-block encoding of the product of a series of matrices. For round $1 \leq r < 16$, we have $M_{r,2} = F_{r+1} \circ M_r \circ G_r^{-1}$, where $G_r = G_{r,1} \wr \cdots \wr G_{r,12}$ is the output encoding from the preceding set of T-boxes, $F_{r+1} = F_{r+1,1} \wr \cdots \wr F_{r+1,12}$ with randomly selected non-linear bijections ${}^{8}_{8}F_{r+1,i}$, and ${}^{96}_{96}M_r$ the product of the following matrices:

- δ_r^{-1}, the inverse of the δ permutation from round r. Normally this permutation was selected in the construction of $M_{r-1,2}$, with the exception of δ_1 which was selected during the construction of the M_1 block.
- $\left(\rho_r \| {}^{48}_{48}\mathbf{I}\right)^{-1}$, used to reconstruct the left-half/REPL bits that were mixed in the previous block.
- ${}^{96}_{96}S$. This is a matrix transformation that permutes the output of the T-boxes and the left-half/REPL bits such that the output vector would have the first 32 bits be the left-half input to round r, the next 32 bits the right-half input, and the final 32 bits are

[9] Note that Jacob et al. define δ by combining two simpler permutations, γ, a permutation of the T-boxes, and μ, a permutation of the left half and REPL bits. Their definitions of these in [23] contain several errors.

[10] Using 8×8 bit block sizes, as seen in Section 5.3.2.1.

[11] G can actually be some other bijection, however, either the source of the plaintext must pre-encode the plaintext with G prior to encryption, or the resulting transformation will not compute true DES, but some related cipher instead.

the output of round r's s-boxes. This reconstruction is possible because the T-boxes were modified to pass the two bits of input that were duplicated by the EP as part of the output. These bits can then be combined with the REPL bits to reconstruct the right half of the input.

- $\binom{64}{64}\mathbf{I}\|^{32}_{32}P)$, which performs DES's P-box permutation, $^{32}_{32}P$, on the output of the s-boxes and leaves the left and right half inputs untouched.

- $^{64}_{96}X$, a matrix transform that moves the current round's right half into the left half, and replaces the right half with the xor of the input left half and the output of the P-box permutation. The output of this matrix corresponds to the output of round r of unobfuscated DES.

- $^{96}_{64}R$, this is the transformation that computes the EP and REPL bits. It is the same as the one from the M_1 block, described in Section 5.3.2.2.

- $^{48}_{48}\rho_{r+1}\|^{48}_{48}\mathbf{I}$, used to mix the input entropy of the left half and REPL bits of round r. This is similar to the transformation described in Section 5.3.2.2, only with a different randomly chosen bijective AT *rho*.

- δ_{r+1} a similar permutation to the one described in Section 5.3.2.2, only with different permutations of the T-boxes and left half/REPL bits.

And so we have

$$M_{r,2} = F_{r+1} \circ \left(\delta_{r+1} \circ \left(\rho_{r+1}\|^{64}_{64}\mathbf{I}\right) \circ X \circ \left(^{48}_{48}\mathbf{I}\|P\right) \circ S \circ \left(\rho_r\|^{48}_{48}\mathbf{I}\right)^{-1} \circ \delta_r^{-1}\right) \circ G_r^{-1}$$

.

5.3.2.4 White-Box DES – The M_3 Block

The M_3 transformation is the encoding of the final linear portion of DES that falls after the s-box substitution in round 16. The structure of M_3 is very similar to that of M_2 from the input encodings up through the X transformation. The difference between the two transformations is that the final three matrix components of M_2 have been replaced with two new matrices - one that swaps the left and right halves of the ciphertext to conform to the DES specification, and one that computes the DES final permutation. Additionally, the output encoding for M_3 will generally be an identity transformation, much like the input encoding of M_1 (see Section 5.3.2.2). A detailed description of the components of M_3 is as follows:

- δ_{15}^{-1}, the inverse of the δ permutation from round 15. This permutation was selected in the construction of $M_{15,2}$.

- $\left(\rho_{15}\|^{48}_{48}\mathbf{I}\right)^{-1}$, used to reconstruct the left-half/REPL bits that were mixed in the previous M_2 block.

- ${}^{96}_{96}S$. This is a matrix transformation that permutes the output of the T-boxes and the left-half/REPL bits such that the output vector would have the first 32 bits be the left-half input to round 16, the next 32 bits the right-half input, and the final 32 bits are the output of round 16's s-boxes.
- $\left({}^{64}_{64}\mathbf{I} \| {}^{32}_{32}P\right)$, which performs DES's P-box permutation, ${}^{32}_{32}P$, on the output of the s-boxes and leaves the left and right half inputs untouched.
- ${}^{64}_{96}X$, a matrix transform that moves the current round's right half into the left half, and replaces the right half with the xor of the input left half and the output of the P-box permutation. The output of this matrix corresponds to the output of round r of unobfuscated DES.
- ${}^{64}_{64}W$, which swaps the left and right halves of the output. This is equivalent to a 32-bit rotation of each of the rows of ${}^{64}_{64}\mathbf{I}$.
- ${}^{64}_{64}FP$ A sparse matrix that performs the DES final permutation.

And so we have

$$M_3 = F_{17} \circ \left(FP \circ W \circ X \circ \left({}^{64}_{64}\mathbf{I} \| P\right) \circ S \circ \left(\rho_{16} \| {}^{48}_{48}\mathbf{I}\right)^{-1} \circ \delta_{16}^{-1} \right) \circ G_{16}^{-1}$$

.

5.4 Attacks on Chow's White-Box DES

While Chow et. al.'s white-box implementation of DES is more secure than an unprotected implementation of key-impregnated DES, it still fails in the face of two known attacks: a statistical bucketing attack, and a fault injection attack. The statistical bucketing attack and a more powerful version of this attack will be described in Section 5.4.1, and the fault injection attack will be described in Section 5.4.2. Our improvements to the cipher and how they affect these attacks are covered in Section 5.5.

5.4.1 Attack on Split T-Box Output

Chow et al. describe a statistical bucketing attack on their white-box DES that exploits the nonlinearity of the s-boxes to expose the key in under ten seconds. Their attack tracks individual bit changes in the input to the second round of s-boxes, generating and comparing preimage sets of input to expose the key. For their T-box implementation, which divides the 8-bit T-box output into two separately encoded 4-bit halves, one of which is the obscured 4-bit s-box output, we observed that it is possible to generate more detailed partitions in the input and expose the key more efficiently. We implemented both attacks and found ours to be several times faster than that of Chow et al.

Our implementation of the statistical bucketing attack identifies the T-box corresponding to each s-box, exposes the first round key six bits at a time, and then uses brute force search to reveal the full DES key. Our software precomputes the 4-element sets of 6-bit

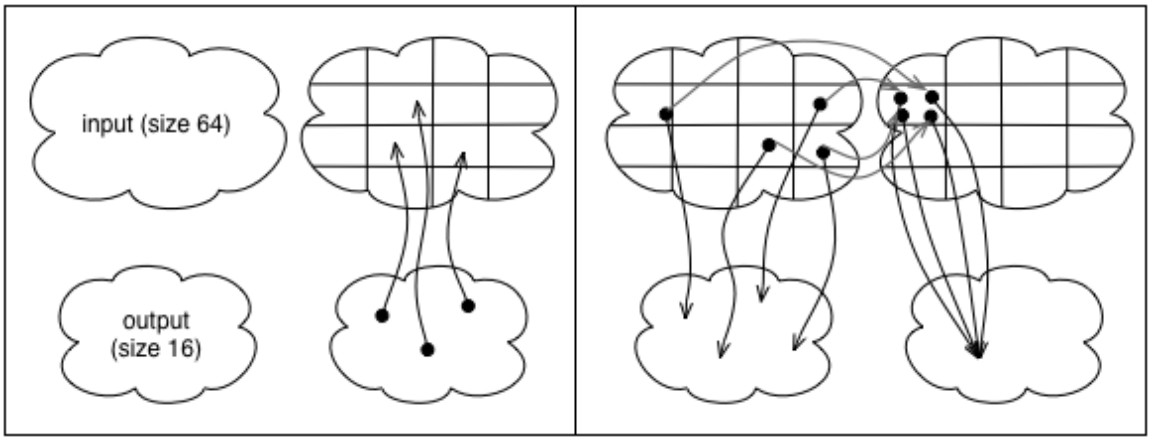

Figure 5.4. The structure of the domain (left) causes a correct round subkey (the effects of which are seen on the right) to produce recognizable preimage sets.

preimages for each s-box from the corresponding T-box using an all zero key. The 6-bit elements of each preimage are run backwards through the expansion permutation (EP) and the DES initial permutation (P1) to produce the corresponding zero-key preimages.

To identify the T-box corresponding to each s-box given a white-box DES, we encrypt 0 through the first T-boxes. We then turn on individual input bits and observe the changes in the T-box output to identify the 4-bit T-box output corresponding to each s-box. Next we begin testing the 6-bit hypotheses for each portion of the first round key. Hypotheses are reversed through EP and P1, and this value is xored with all elements of each zero-key preimage before being passed to the encryptor. If the processed preimages each map to a single 4-bit T-box output (i.e., the set of preimages remain the same), the hypothesis is correct and 6 bits of the 48-bit first round key have been identified (Figure 5.4). When the first round key is known, we record the full white-box DES encoding of an arbitrary input. This is used to perform brute force search of the remaining bits, comparing the recorded result with a conventional DES encoding for each potential key until a match is found.

In the pre-processing phase of the attack implementation, a reference DES is created that produces first-round s-box results. All 64 6-bit s-box inputs (corresponding to 6 bits of message encrypted with a zero key) are passed through all eight s-boxes and mapped backwards through EP and P1. The DES inputs corresponding to each of the 16 4-bit results for each s-box are recorded as preimage sets. Each s-box has 16 zero-key preimage sets with 4 elements.

The first step of the actual attack is to identify which of the twelve T-boxes correspond to the eight s-boxes. This is a matter of encrypting a zero block through the first round of T-boxes, and then encrypting individual bits and observing which T-box outputs change. For the bits $b_0b_1b_2b_3b_4b_5$ passed into an s-box, bits b_0 and b_5 are passed as "bypass bits" in the T-box to create a bijection, b_1 and b_4 are produced as bypass bits for other T-boxes, and b_2 and b_3 are replicated in an arbitrary T-box as part of the replicated bits (those bits

not already duplicated by EP) for R_r. The T-box that is changed by both b_1 and b_4 for an s-box is the matching T-box, and in particular the 4-bit T-box output block that changes corresponds to the 4-bit s-box output. Our attack uses this process to build s-box output bitmasks.

Once the s-box outputs have been identified, the program runs through each s-box and each of 64 corresponding candidate 6-bit portions of the first round key. For each candidate key, the 6 bits are reversed through EP and the DES initial permutation to get a plaintext that effectively cancels those key bits for that s-box. This key preimage is xored with the elements of a zero-key preimage set, and these values are passed into the white-box DES. The s-box output bitmask is used to isolate the results, and the four results are compared. If they are not all equal, then the candidate key bits are incorrect, and the algorithm moves on. If all four results are equal for all preimage sets, the candidate key bits are correct[12].

At this point we have two hypotheses for the first round key (see Footnote 12), and the algorithm is able to generate key schedules and perform brute force search of the 2^9 possible keys to find a plaintext/ciphertext pair matching one created with the white-box DES. From start to finish, a largely unoptimized version our attack takes approximately three seconds on an 800 MHz G4 processor.

Our statistical bucketing routine returns a DES key when given both a complete instance of Chow et al.'s DES function and a crippled version that returns the output of the first round of T-boxes. We supplied a crippled function for expediency, but this is not necessary; a function that returns T-box output can be automatically generated by analyzing the encryptor's control flow. We have not implemented this portion of a full attack for either our attack, or Chow's original version.

Because we record preimage sets considering the entire s-box output, we can reduce the hypothesis space for each 6-bit portion of the first round key to a single value. This makes the attack faster than that described by Chow et al., which considered only one bit of s-box output. With minimal algorithmic optimization, the entire process takes about three seconds on our reference platform. The statistical bucketing attack depends upon the separation of the permuted s-box results from the other T-box output bits. The s-boxes are lossy, and so the preimages have multiple elements and may be compared with one another even when the output values are renamed. When T-box output is a permuted 8-bit value, the T-boxes are bijections and preimage comparison is no longer useful.

5.4.2 Differential Fault Injection Attack

Jacob et al. describe an attack on the white-box DES implementation of Chow et al., in which faults are inserted to enable differential cryptanalysis of the embedded DES s-boxes. The attack is a very efficient one, requiring just dozens of calls to a decryption oracle and a similar number of encryptions using the white-box implementation to recover 48 bits of the embedded key. The other 8 bits can be recovered via brute force attack using a reference implementation of DES. We describe a simplified interpretation of the

[12]The sole exception is s-box 4, for which the preimage sets are the same as those for inputs that were xored an with 101111b prior to entering s-box 4. This property was first reported by Shamir in [44].

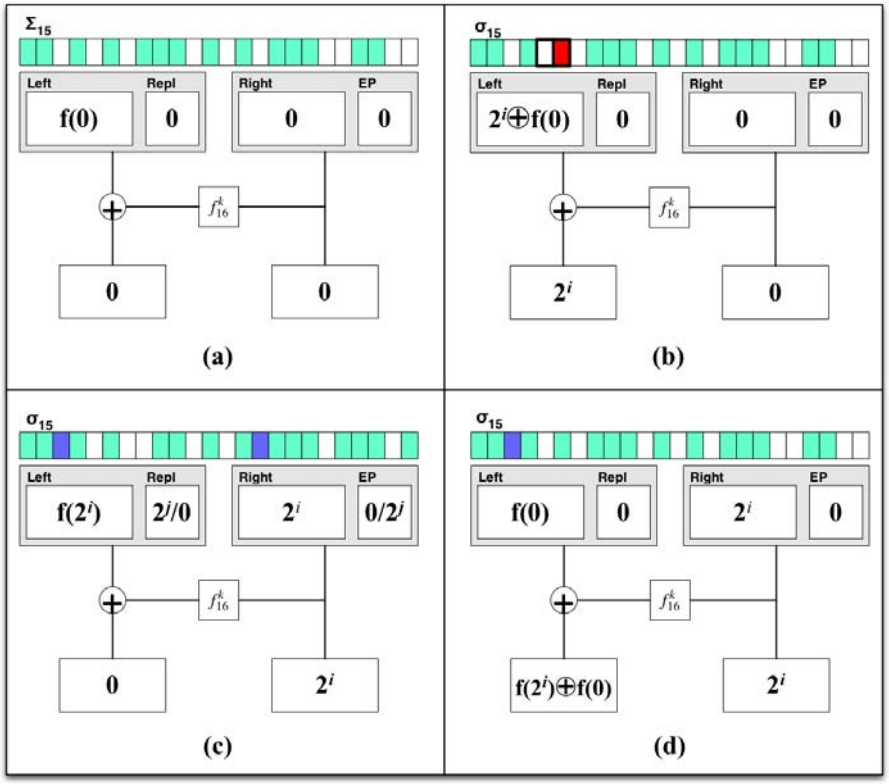

Figure 5.5. The attack proceeds by selecting ciphertexts, consulting a decryption oracle, recording the next to final results from encryption, and selectively overwriting these intermediate results prior to completing the encryption.

attack below. Throughout this description we assume the adversary has a white-box DES encryption function, and we will use the following notation:

$\mathcal{E}^k_{d_0,d_1}(l,r)$**:** This means to run the white-box encryptor, with embedded key k, for rounds d_0 through d_1 on the plaintext that consists of l in the left block and r in the right. If only d_0 is specified, then only that one round of encryption will be performed at that time. The resulting 96-bit value, σ_{d_1} will be the obfuscated input to the T-boxes of round $d_1 + 1$.

$\mathcal{D}^k(l,r)$**:** This means to call the decryption oracle with embedded key, k, on the ciphertext consisting of l in the left half, and r in the right. The resulting 64-bit value, l_0, r_0 will be the resulting plaintext.

We will also assume that the attacker is working on the ciphertext at the output of round 16 (denoted by the blocks L_{16} and R_{16}, i.e. before the final DES permutation. This is a valid assumption, as the final permutation is well known and easily invertible.

The fault injection attack proceeds in four stages:

- **Determine the representation of $L_{15} = f_{16}^k(0)$, $R_{15} = 0$**
  This is done by first computing $l_0, r_0 = \mathcal{D}^k(0,0)$, and then computing $\sigma_{15} = \mathcal{E}_{1,15}^k(l_0, r_0)$ (see Figure 5.5a). This is the obfuscated representation of $L_{15} = f_{16}^k(0)$, $R_{15} = 0$ and will be used later in the attack, where we will refer to it as Σ_{15}.
- **Determine which segments of σ_{15} affect which bits of L_{16}**
  This is done by first computing $l_i, r_i = \mathcal{D}^k\left(2^i, 0\right)$ and then computing $\sigma_{15}^i = \mathcal{E}_{1,15}^k(l_i, r_i)$ for $0 \leq i < 32$ (see Figure 5.5b). The σ_{15}^i can then be compared to Σ_{15} to determine which 4-bit blocks have changed for each i. This also tells the attacker which T-boxes are the "real" T-boxes, as well as to which s-box they correspond.
- **Determine $f_{16}^k(0) \oplus f_{16}^k(2^i)$**
  First compute $l_i, r_i = \mathcal{D}^k\left(0, 2^i\right)$, then compute $\sigma_{15}^i = \mathcal{E}_{1,15}^k(l_i, r_i)$ for $0 \leq i < 32$ (see Figure 5.5c). In the second step, it can be determined which T-box is affected by the bit that is set in the right half of the ciphertext, and in particular, which 4-bit block(s) in σ_{15}^i. The attacker can now swap in all of the blocks of Σ_{15} that will not overwrite the 4-bit block that is affected by 2^i, call this new value σ'^i_{15}.[13] This will have the effect of changing L_{15} from $f_{16}^k(2^i)$ back to $f_{16}^k(0)$ (see Figure 5.5d). Now, computing $\mathcal{E}_{16}^k(\sigma'^i_{15})$ results in $L_{16} = f_{16}^k(0) \oplus f_{16}^k(2^i)$, which will be used in the final step.
- **Perform differential cryptanalysis on the s-boxes**
  Given up to six different $f_{16}^k(0) \oplus f_{16}^k(2^i)$ for each s-box that the various 2^i will fall into, the attacker now performs differential cryptanalysis on each s-box to recover the 6-bits of the final round sub-key that goes into each s-box. Once the 48-bit sub-key is recovered, use brute force on a reference implementation of DES to recover the other 8 bits of the key. Note that the technique described in the previous step is guaranteed to produce at least four valid values of $f_{16}^k(0) \oplus f_{16}^k(2^i)$, which may result in the attacker recovering as few as 40 of the 48 bits of the final round sub-key, in which case the brute-force portion of the attack will require up to 2^{16} trial encryptions. Slightly more involved techniques in step three, however, can retrieve all 48 bits of the sub-key.

5.5 Implementation Improvements

While the changes we have made to Chow et al.'s version of white-box DES may seem small at first blush, they have a profound effect on the security of the resulting cipher. In particular, the changes completely nullify the statistical bucketing attack, and severely limit the effectiveness of Jacob et al.'s fault injection attacks. Analysis of these effects is provided in the following sections.

5.5.1 Statistical Bucketing Attack Resistance

Chow et al. admitted that their white-box DES implementation is vulnerable to a statistical bucketing attack on the input to the second round of T-boxes, but nevertheless they "recommend 4×4 blocking" for the matrices making up DES. This exposes their imple-

[13] This step will need to be repeated twice for those bits that are replicated by the expansion permutation, once for each 4-bit block that will be affected.

mentation to our more aggressive statistical bucketing attack, which exploits the separation of s-box output and supplemental output produced by the T-boxes.

Using an 8×4 block size rather than a 4×4 block size prevents both Chow's statistical bucketing attack, and our new attack. Although it would map well to the input and output of fully-delinearized T-boxes, 8×8 I/O-block encoding of the M_is unfortunately produces addition tables that map 16 bit values to 8 bit values, for a total of more than 250 MB of tables to implement a single encryption function! Using an 8×4 block size allows the T-box output to produce an 8-bit permuted value from the concatenation of two 4-bit values, and the multiplication and addition tables that make up the I/O-block encoded matrix become 8×4 tables. The result is a DES implementation whose T-box output does not leak information needed by the statistical bucketing attack. The new DES is the same size as Chow et al.'s, as the addition tables in both are the same size, and these tables make up the bulk of white-box DES.

Further discouraging analysis of the T-box output is possible by completely eliminating it as an accessible intermediate value. Once an M_i has been I/O-block encoded using an 8×4 block, the initial 8×4 multiplication tables accept the same size output as the 8×8 T-boxes produce, and the results of a T-box and a corresponding multiplication table can be precomputed to form a single 8×4 table. Each T-box result is fed into multiple multiplication tables, so ultimately this precomputation produces a replacement for each multiplication table and eliminates the T-boxes. (Chow et al. used the same approach when folding each pair of 4×4 multiplication tables into the subsequent addition table.) After this process, our entire DES implementation consists of 8×4 tables: multiplication tables (for the first AT only), fused T-box/multiplication tables, and addition tables. Chow et al.'s implementation uses 8×8 T-boxes (constructed with split input and output encodings, to take two 4-bit input halves and produce two 4-bit output halves), and 8×4 vector addition tables.

To prevent the original statistical bucketing attack on the input to the second round of T-boxes, we must disrupt the preimage of each input bit by intermixing the s-box input with L_r and R_r. We mix the 16 replicated bits of R_r with L_r by including a random AT, ρ, in the M_is[14], and delinearize the T-box input as a single 8-bit block. Moving to an 8×8 block size for the M_is would allow us to pass the T-boxes 8-bit blocks instead of two concatenated 4-bit blocks, but as we have said this would produce an intolerably large DES implementation. Once the 8×4 I/O-block encoding has been prepared, however, it is possible to use combined function encoding [11] on the final addition tables of each pair of trees (see Figure 5.1) in the I/O-block encoding, to generate the input for each T-box as a single 8-bit result. The combined function encoding replaces twenty-four 8×4 tables with twelve 16×8 tables before each round of T-boxes. Further optimization to use two 12×4 tables in place of each 16×8 table appears possible.

5.5.2 Differential Fault Injection Attack Resistance

In order to expose the final round subkey, this fault injection attack takes advantage of two of the design decisions made by Chow et al.: the splitting of the 8-bit T-box into two 4-bit

[14] According to [23], Chow et al. use a simple permutation μ within δ without an affine transform.

		Bits 4-6							
		000	100	010	001	110	101	011	111
Bits 1-3	000	0.00 0.00 0.00	2.68 3.31 5.00	2.42 3.32 5.00	2.42 3.24 5.00	3.68 5.42 6.00	3.68 5.51 6.00	3.42 5.40 6.00	5.00 5.88 6.00
	100	2.42 3.08 5.00	3.68 5.14 6.00	4.00 5.43 6.00	3.42 5.33 6.00	5.00 5.84 6.00	4.00 5.88 6.00	5.00 5.91 6.00	5.00 5.97 6.00
	010	2.19 2.99 5.00	4.42 5.40 6.00	3.68 5.37 6.00	3.42 5.14 6.00	5.00 5.91 6.00	4.42 5.93 6.00	4.00 5.84 6.00	6.00 6.00 6.00
	001	2.42 3.24 5.00	3.42 5.15 6.00	3.68 5.43 6.00	3.42 5.40 6.00	5.00 5.94 6.00	4.42 5.86 6.00	4.00 5.81 6.00	5.00 5.97 6.00
	110	3.68 5.39 6.00	5.00 5.78 6.00	5.00 5.94 6.00	4.00 5.84 6.00	5.00 5.97 6.00	5.00 5.97 6.00	6.00 6.00 6.00	6.00 6.00 6.00
	101	3.68 5.43 6.00	4.00 5.81 6.00	5.00 5.91 6.00	4.41 5.86 6.00	5.00 5.97 6.00	5.00 5.97 6.00	5.00 5.97 6.00	6.00 6.00 6.00
	011	3.68 5.38 6.00	5.00 5.88 6.00	5.00 5.97 6.00	5.00 5.91 6.00	6.00 6.00 6.00	6.00 6.00 6.00	5.00 5.97 6.00	6.00 6.00 6.00
	111	5.00 5.97 6.00	6.00 6.00 6.00	6.00 6.00 6.00	6.00 6.00 6.00	6.00 6.00 6.00	6.00 6.00 6.00	6.00 6.00 6.00	6.00 6.00 6.00

Cell entries show the minimum, mean, and maximum information gain given successful fault injection attacks on bits of s-box 1 input bits 1 through 6 as indicated by the row and column.

Table 5.1. Expected information gain in bits of final round subkey for s-box 1.

halves, and the specification of ρ as a permutation. This is done as follows:

1. The four bit blocking ensures that the "swapping in" of the representation of $f_{16}^k(0)$ will overwrite at most two right half bits in the "real" T-boxes (we are not concerned with the "dummy" T-boxes), thus guaranteeing at least four valid $f_{16}^k(0) \oplus f_{16}^k(2^i)$ data points per s-box – more than enough to carry out a differential attack on the s-boxes.
2. Because μ is simply a permutation of bits, a single bit change to L_{16} results in a single bit change in L_{15} (and vice versa) This means that only a small subset of the 4-bit blocks that represent $f_{16}^k(0)$ need to be "swapped in" in order to learn the value of $f_{16}^k(0) \oplus f_{16}^k(2^i)$ for any single s-box. Acting in concert with the note above, this makes it likely that an attacker can gain more than 4 valid $f_{16}^k(0) \oplus f_{16}^k(2^i)$ data points for each s-box, improving the efficiency of the attack.

Our improvements to white-box DES remove both of those leverage points. The first is affected by our introduction of 8-bit blocking at the inputs to the T-boxes. If a bit is flipped in the right half of the ciphertext, this change will be destroyed if the bit is anywhere in one of the T-boxes that are overwritten when $f_{16}^k(0)$ is swapped back into L_{15}. With the 4-bit blocking of Chow et al., four input bits are guaranteed to never be overwritten, thus four valid $f_n^k(0) \oplus f_n^k(2^i)$ data points are guaranteed to be computed in every DES s-box. In our implementation, there are no such guarantees.

The second leverage point is removed by augmenting the permutation μ with a random affine transform. Any single bit change to L_{16} will result in an expected change in half of the bits of L_{15} (as well as the replicated bits of the right half). Since each real T-box contains two of these bits, we can expect that 6 of the 8 real T-boxes (and 10 of the 12 total T-boxes) will be affected by a single bit change in L_{16}. What this implies is that in order to reset L_{15} so it contains $f_{16}^k(0)$ in at least the four bit positions that contain the s-box we are attacking, we will likely have to swap in all twelve 8-bit blocks of Σ_{15}.[15]

Unfortunately, the 8-bit blocking does not completely eliminate the threat of the fault injection attack. This is because each real T-box contains only two of the mixed left-half and replicated right-half bits. If the attacker always swaps in the eleven 8-bit blocks of Σ_{15} that do not overwrite the bit of interest set in the right half, there is still a chance that those two bits, which correspond to two bits of the representation of $f_{16}^k(2^i)$ may just be the same as the equivalent bits of $f_{16}^k(0)$. This event occurs with probability 0.25.

If this happens, L_{15} will be reset to $f_{16}^k(0)$ after completing the encryption, L_{16} will contain $f_{16}^k(0) \oplus f_{16}^k(2^i)$ for the s-box of interest, and the attacker will have gained a valid data point to use in differential cryptanalysis. The attacker can determine if this has happened because there will be at most two non-zero 4-bit blocks in the left half[16]. If this has not happened, most if not all of the 4-bit blocks of the left half will be non-zero with overwhelming probability.

In our improved implementation of white-box DES, an attacker still has a 0.25 probability of obtaining a valid data point to use in the differential cryptanalysis, with up to six data points per s-box. The amount of information gained varies according to the s-box being attacked, the position of the injected fault within the s-box, and the number of data points obtained. As an example, this information is provided for s-box 1 in Table 5.1. However, the attacker will gain an expected 1.22 effective bits[17] of the final round sub-key per s-box. Based on this measurement, we have improved the resiliency of white-box DES against this attack from surrendering a guaranteed 40 bits of the final round key (and all 48 with high probability), to surrendering an expected 9.8 bits of the final round key. While this is still not as strong as a black-box implementation, it is strong enough to make an attack on the triple-DES variant of our implementation infeasible.

5.5.3 Optimizing Construction

In principle, the condensed matrices, mixing bijections, and T-boxes can be combined to the desired end in many different ways. The resulting obfuscated implementation will look no different. Without consideration of the space and time required, however, the temporary data allocated during generation can vary substantially. In practice, a few optimizations

[15] The probability of this not occurring is approximately $2^{-8.7}$.

[16] The block corresponding to the s-box of interest plus one if the bit set in the right half was copied by EP. The adversary can tell which is which based on which s-boxes affect which bits in the final output.

[17] The attacker may not necessarily determine any actual bits of the sub-key, but is still able to reduce the search space of possible keys that need to be checked during brute force search.

[18] MiB is the number of mibibytes (2^{20} bytes) of tables if 4-bit values are stored as 8-bit characters (for efficiency of reference). Packed MiB is the number of mibibytes of tables if two 4-bit values are stored per 8-bit character.

	MiB[18]	Packed MiB
Chow et al. (4×4)	4.54	2.27
8×4	4.49	2.25
8×8	274.75	274.75
8×4, Joined Roots (JR)	16.40	14.20
8×4, JR, 3DES	48.83	42.42

Table 5.2. Comparison of white-box DES implementations

	Memory Allocation	Run Time
Before Optimization	287 MB	138 sec
Array Resource Pooling	264 MB	127 sec
Streamlining I/O-Block Encoding	67 MB	92 sec
Memoizing Inverse Input Encodings	64 MB	92 sec
Constant-Matrix Folding	62 MB	92 sec
Buffer Reuse During Network Encoding	60 MB	92 sec
Specialized Bit-Matrix Operations	58 MB	92 sec
Array Type Declarations	43 MB	92 sec
Non-consing Random Permutation Function	20 MB	92 sec
Using Compressed Bit-Matrices in T-box Construction	11.6 MB	49 sec
Compiled File Containing Obfuscated DES Function	4.9 MB	

Table 5.3. Run time and runtime memory allocation as compared to final DES function size, over the course of a series of optimizations

made substantial improvements in the time and space required to generate an instance of DES. Once our implementation was complete and correct, we performed a series of optimizations that took the temporary memory allocation when generating a 4.9MB implementation with 8×4 I/O-block encoding down from 287MB to a mere 11MB (including storage for the result).

Selected optimizations are summarized in Table 5.3. After some clear worst offenders were removed (a naive implementation of I/O-block encoding, most notably), optimization became a matter of tuning and refining several things throughout the system, in large part because most of the data were collections of 256-byte arrays. Precomputing constant matrices used in the ATs for DES reduced the AT multiplications that had to be done at runtime and saved some space. Declaring the types of our matrices to the Lisp compiler allowed elements to be packed more densely and manipulated with unboxed instructions by the compiled code, improving time and space[19]. Shrewd preservation and reuse of memory allocated to matrices, permutations, and temporary tables saved a great deal of space, and saved time spent initializing new empty structures. Finally we found that a surprising amount of time was devoted to creating and applying small random permutations; a tuned version of this common operation cut the space and time used in half one last time.

We implemented white-box DES using several approaches, including the 4×4 I/O-block encoding of Chow et al., to compare the time and space requirements (Table 5.2). The test platform was an 800Mhz PPC G4 running Mac OS X 10.2.6 and MCL 5.0b5. We implemented both our modified systems and Chow et al.'s original algorithm in Lisp in this environment.

5.6 Extensions Of These Techniques

There are a number of natural extensions of these white-boxing techniques, for example, the creation of a white-box version of triple-DES for added security, and the encoding of other ciphers such as AES. A brief discussion of these two extensions follows.

5.6.1 Application to triple-DES

When extending the technique to triple-DES, the final matrix from each DES portion (M_3M_2) is combined with the first matrix from the following DES (M_1). This eliminates intermediate values that would enable the implementation to be compromised in thirds. Using $M_1M_3M_2$ as a single I/O-block encoded matrix between the three DES portions requires the entire triple-DES be attacked as a whole. In addition it saves the encryptor both space and time, causing the whole of triple-DES to be less than three times as large and take less than three times the time as the encoded DES.

Our statistical bucketing attack applies to a triple-DES implementation constructed using Chow et al.'s white-boxing technique with split output T-boxes. Once we extract

[19]Lisp is dynamically typed so, typically, values carry tag bits to identify their type to the run-time environment. Declarations can be used to strip these bits off for the duration of several functions, eliminating the need for so-called boxing and unboxing sequences.

47 bits of the first key, leaving 9 bits of uncertainty (for the remaining 8 bits and the two possible 6-bit key portions associated with s-box 4), we have 512 hypotheses for what the first round encryption result could be. This equates to 512 hypotheses for what would result (implicitly) in a zero result after the first DES (or a zero result xored with a 6-bit first-round subkey for the second DES), giving 512 possible 48-bit first round keys (at most; it may be possible to eliminate some of these). Ultimately this leads to 2^{18} hypotheses for DES keys 1 and 2 when extracting a round key from the final iteration of DES. The resulting search space is 2^{27} to recover the full key. It is also worth noting that once the initial 512 hypotheses are acquired, they can be investigated independently, paralleling the attack.

We are inclined to recommend triple-DES rather than DES for any conceivable use case, because the malicious host is able to generate as many plaintext-ciphertext pairs as needed.

5.6.2 White-Box Encoded AES

In addition to obfuscated DES function generation, we have implemented an obfuscated AES encryption function generator as described by Eisen and van Oorschot [15]. In addition to providing the increased security of AES over DES, the implementation also has a smaller memory footprint, and is more flexible because it takes a transformed version of an encryption key as a parameter.

Like white-box DES, the AES implementation is made up of a large number of lookup tables and a function which operates on the tables. White-box AES consists of 8x8 and 8x32 tables grouped into "families" whose output can be xored using a simple machine instruction. This is done by randomizing the output of the tables with random affine transformations (ATs) that share a common linear component. Because familial output can be xored into a result suitable for a subsequent table's input decoding, white-box AES does not require obfuscated ATs to perform addition, and eliminates tables that can not be avoided in DES. They then add input and output encoding to the encryption function itself. Unlike published white-box DES generators, white-box AES generators are able to obfuscate the key schedule generation mechanism itself, and couple the output of an obfuscated key schedule function with an obfuscated encryption or decryption function.

Randomizing tables in this way provides perfect local security, but in the first and last rounds of AES the true plaintext and ciphertext are known and can be used to compromise the key using the following algorithm (which may be known but has not been published previously). Here we describe in pseudocode a simple attack that may be used to reveal the AES key with a trivial amount of computation, given the obfuscated key input to a dynamic-key white-boxed AES implementation. The attack depends on access to the external input and output encoding. Although Eisen and van Oorschot assume this external encoding is separated from the main white-box AES [10, 15], this attack demonstrates the vulnerability represented by even black-box access to the external encoding.

$k_{T(p)}$ = random $\in$ 0..255
$AT_{T(p)}$ = random bijective affine transform over $GF(2^8)$
$T(p) = AT_{T(p)}(S(p \oplus k_{T(p)}))$

using $T(p)$ as a table (without being given $AT_{T(p)}$), deduce k:

$\forall$ (hypotheses) $k' \in \{0..255\}$
let $p_i = k' \oplus S^{-1}(i)$, for $i \in \{0, 1, 2, 4, 8, 16, 32, 64, 128\}$
let $AT = Ax + c : c = T(p_0),\ col_1(A) = T(p_1) + c,\ \ldots\ col_8(A) = T(p_{128}) + c$

if $\forall\ p : T(p) = AT(S(k' \oplus p)) \rightarrow\ k' = k_{T(p)}$

Eisen and van Oorschot do not describe a means by which the encryption function can stand alone securely. To compensate, they add input and output encoding to the encryption function itself and require other components of a containing system to encode input destined for encryption and decode the encryption function's results to produce the final ciphertext. We do not believe this addresses the problem so much as removes it from the immediate vicinity of the white-box AES. We believe that further obfuscation (beyond the AT families) of the first and last rounds of AES will eliminate the vulnerability of the first AES round-key, and allow the locally secure inner rounds to remain as they are. This should not increase the total implementation size very much.

Our implementation and discussion pertains to the dynamic-key AES implementation [15], but it is important to note a new paper by Billet, Gilbert, and Ech-Chatbi [6] describing an efficient attack on the original static key white-box AES [10]. We have not yet analyzed their attack in sufficient depth to know how it affects the dynamic-key version of white-box AES, but their attack appears superior to our attack in that it does not seem to require even black-box access to the external encodings.

5.7 Future White-Boxing Work

While the encoding of ciphers such as DES and AES have been covered in great detail, there is still a great deal of work to be done in the field of obfuscation. Foremost among the open issues is the formalization of the ideas behind obfuscation, as well as the attack model that obfuscated code is claimed to be secure in, so that a better analysis of the security of these techniques may be performed. Additionally, much more cryptanalysis of the existing techniques is required before any faith should be expressed in the security of white-box encoding.

Other ongoing areas of research in the area are in the optimization of already existing implementations in order to create smaller, faster code, discussed below, and the implementation of a hardware version of these ciphers, removing the need for tamper-proof hardware.

Because addition table output is sent into twelve separate fused T-box/multiplication tables, we cannot combine the 16×8 addition tables with them. We recommend the combined function encoding of the root tables of each pair of addition tables that produce T-box input, and the creation of an intermediate value for T-box input that is obfuscated in 8-bit blocks. Once these modifications are made, each T-box input block includes six s-box input bits and two bits containing information about the left and replicated bits. This makes it more difficult to identify which T-box corresponds to which s-box in the second round and removes the association between preimages and individual bits of input, preventing statistical bucketing attacks.

Further improvements in the size and security of white-box DES may be possible. Join-

ing the roots of the vector addition trees to prevent statistical bucketing attacks on the T-boxes may not be necessary for any but the first and final few T-boxes, for example. Using split T-boxes in the interior of DES would greatly reduce the size of the implementation, and would lead to even greater savings in a triple-DES implementation. It is also possible to eliminate the dummy T-boxes in favor of eight 12-bit T-boxes. This would improve the security of the system further against the differential cryptanalysis attack of Jacob et al., reducing the adversary's expected gain to 0.05 effective bits per T-box, totaling 0.40 effective bits. This security comes at a substantial increase in size if 12-bit T-boxes are used throughout DES, so it is worth evaluating the potential of reverting to 8-bit, split-input T-boxes for the interior rounds to limit this growth.

Chow et al. speculates the possibility of matrix analysis enabled by "sparse" tables – multiplication tables containing only a few of the 2^b possible output values. We have not yet implemented our solution to this, but it should be possible to give individual values in the table many obscuring names instead of only one, and compensate for this when computing the contents of the following tables. Analysis of the utility in attacking the matrix before and after such modifications is necessary. Some analysis is also necessary to confirm that the new system is resilient to other forms of cryptanalysis. While we believe that the four 4-bit values passed to the joined root in our current implementation are not as susceptible to cryptanalysis as the original split T-box inputs were, we have not yet demonstrated this.

This additional work should ultimately result in a white-box implementation of triple-DES that is as compact as possible while providing near black-box level security.

Chapter 6

Cryptographic Approaches to Securing Mobile Code

Obfuscation methods such as the white-boxing described in Chapter 5 show promise, but nevertheless fail to have rigorous security proofs. In this chapter, we examine using cryptographic methods to obfuscate code. The resulting security is based on the security of the underlying cryptographic primitives used.

Encryption is meant to render plaintext unintelligible. Hence one would expect that in general, encrypting code does not produce anything executable, and, if it did, the code would have no meaningful relation to the original. However, in certain cases, encryption can be used to hide the data that is input into a function [16], or certain types of encryption schemes can be used to produce encrypted, yet executable instructions when applied to a restricted class of functions [40, 41, 42]. In Section 6.1 we examine the technique of working with encrypted data and in Section 6.2 we examine computing with encrypted functions.

We saw previously that it could be useful to have an obfuscated version of an encryption or decryption function such as triple-DES or AES. It would also be useful to have an encrypted or obfuscated version of a signature scheme to allow agents or hosts to sign messages or documents on behalf of a client. This however, introduces some complications such as the retainability of non-repudiation and source authentication. We address this problem in Section 6.3.

6.1 Computing with Encrypted Data

In [16], Feigenbaum suggests the notion of computing with encrypted data. The premise is that one does not care about the confidentiality of a program itself, but instead is interested in protecting the actual inputs and outputs. For example, suppose Alice wants to compute $f(x)$, but doesn't have the power to compute the function f, so she asks Bob to do it for her. She doesn't want Bob to learn the inputs or outputs of her particular computation; so instead of asking Bob to compute $f(x)$, she asks for $f(y)$ and somehow Alice is able to easily compute $f(x)$ from $f(y)$, but Bob is not. An example given in [16] is that of *blinding*

a problem instance of a discrete log computation: Suppose Bob can solve the discrete log problem. Alice wants to find the discrete log of $y = g^x \pmod{p}$ ($x = DL_g(y)$), but doesn't want Bob to learn x. So instead of asking Bob to find $DL_g(y)$, Alice generates a random secret r and asks for $DL_g(y * g^r) = DL_g(g^{r+x})$. When Bob returns the value k, then Alice computes $x = k - r$. Since r is secret, Bob cannot learn x. This idea is difficult to generalize and cannot be applied to all functions, but the notion of encrypting inputs can be tried on a case-by-case basis if the goal is not function confidentiality, but data confidentiality.

6.2 Computing with Encrypted Functions

In [40, 41, 42] Sander and Tschudin begin to tackle the problem of a software-only approach to securing mobile code. Their idea is to develop "executable" encrypted programs. In other words, programs that implement encrypted functions that can be run without first having to be decrypted.[1] They believe mobile agents should act autonomously and hence require that their protocols are non-interactive, but do allow a final computation to be done once the agent completes its work. Two approaches to "encrypted" functions are described. The first is via function composition and the second via *homomorphic* encryption functions. Both have limited application.

6.2.1 Function Composition.

The idea of encryption via function composition is similar to that of blinding inputs as described in Section 6.1, but applied to the function. As an example, suppose that f is a linear map. Then f can be represented by a matrix, F. To "encrypt" F, Alice chooses a random invertible matrix, R, and computes $E(F) = R \circ F$. She can then pass off a program implementing $E(F)$ to Bob and ask him to compute $y = E(F)(x)$ for her input x. Alice can recover her desired output Fx by applying the inverse of the matrix R to y: $Fx = R^{-1}y = R^{-1}RFx$. This notion can be generalized to apply to rational functions[2] instead of linear functions. A rational function f is encrypted via $E(f) = h = s \circ f$ where s is an invertible rational function. The security of the problem relies on the difficulty of decomposing functions: Given a multivariate rational function h that is known to be decomposable, find f so that there exists an s such that $h = s \circ f$. Although there is no known polynomial time algorithm to solve the decomposition problem, it must be studied in more detail as it applies to this problem. In particular, for this application it is necessary to find such an s with an inverse that is easy to compute. Thus far, all suggested candidates have been found to be insecure (see [40]).

6.2.2 Encrypting Functions via Homomorphic Encryption.

A second approach described by Sander and Tschudin [40, 41, 42] involves encrypting a function with an actual encryption algorithm. The method applies only to rational functions

[1] Note that this is different from the idea of encrypting code where the instructions must first be decrypted before they can be run.

[2] A rational function is one that can be written in the form $f = \frac{g}{h}$ where g and h are polynomials.

and the only type of encryption algorithms that can be used are *homomorphic* encryption schemes. The term homomorphic is used to describe a property of a mapping and needs to be used in conjunction with an operation. For instance, a mapping $E : A \to B$ is *additive* homomorphic if given $a_1, a_2 \in A$ and $E(a_1), E(a_2) \in B$, there is an efficient algorithm to compute $E(a + b) \in B$. An encryption scheme is *multiplicative* homomorphic if given $E(a_1), E(a_2)$ there is an efficient algorithm to compute $E(a_1 a_2)$; and *mixed-multiplicative* homomorphic if given $E(a_1), a_2$ there is an efficient algorithm to compute $E(a_1 a_2)$ that does not reveal a_1. Sander and Tschudin prove that an additive homomorphic encryption function on $\mathbf{Z}/n\mathbf{Z}$, the ring of integers modulo n, is also mixed multiplicative homomorphic. In particular, they describe a scheme based on the discrete logarithm problem that satisfies these properties (see [40]).

Suppose you have an encryption function, $E : A \to B$, satisfying the additive and mixed multiplicative homomorphic properties. Given a polynomial, $p = \sum a_{i_1 \ldots i_k} X_i^{i_1} \ldots X_k^{i_k}$, with coefficients in the domain, $a_{i_1 \ldots i_k} \in A$, Sander and Tschudin ([41], p.115) give the following method for computing with an encrypted version of p. Encrypt the coefficients to replace each $a_{i_1 \ldots i_k}$ by $E(a_{i_1 \ldots i_k})$. To compute $E(p)$ on input $x_1, \ldots, x_k$ do the following: Evaluate the monomials $(X_1^{i_1} \ldots X_k^{i_k})$ on the input $x_1, \ldots, x_k$. Each term of $E(p)$, $E(a_{i_1 \ldots i_k} x_i^{i_1} \ldots x_k^{i_k})$, can be evaluated using the mixed multiplicative property of the homomorphic encryption scheme, E. The sum, $E(p(x_1, \ldots, x_k))$ is then be computed using the additive property of E. This idea can be extended in the natural way to work on rational functions as well.

Now suppose that Alice has a rational function, f, that she wants Bob to compute for her, but she doesn't want Bob to learn f. Alice sends Bob a program to compute an encrypted version of f and then gives Bob her input $x_1, \ldots, x_k$. Bob returns $E(f(x_1, \ldots, x_k))$ and Alice applies the decryption algorithm to learn $f(x_1, \ldots, x_k)$.

Although this gives a provably secure method for encrypting certain functions, there are limitations. First note, that while encryption hides the coefficients, it does not hide the "skeleton" of the function - i.e., which monomials (which terms to which powers) occur in the function. This may be a problem if that must be secret (for example in an RSA algorithm, the skeleton reveals the key). Further, if the degree of the polynomial is low, this scheme is subject to interpolation attacks: if the output pairs $(x, f(x))$ are known, then given enough pairs, the coefficients of f can be solved for and f recovered. Finally, homomorphic encryption schemes are in general based on public key algorithms and the encrypted version of the function would be slow.

6.3 Digital Signatures

In the mobile agent scenario, it may be convenient to give an agent authority to sign particular inputs on behalf of a client. Further, public key algorithms in general and digital signature algorithms in particular are computationally intensive and can be prohibitive for a low power machine to compute. Hence, it would be useful to have an obfuscated or encrypted implementation of a signature algorithm that could be used by a mobile agent, or computed on a high power host on behalf of a low power client. However, even if the host could not extract a private key from the implementation, the host could potentially sign messages other than those intended by the client and pass them off as authentic. This

is counter to the notion of unique authentication and non-repudiation that is desired from public key algorithms. It is necessary to have a way to further link the message to the private-key owner or limit the type of messages that could be signed by the program on the untrusted host. We investigate two possible methods for doing this below. The first was proposed by Sander and Tschudin in [42] and examines the idea of coupling the signature algorithm with the output of a function that is to be signed. The second is a new idea proposed here that securely allows arbitrary messages to be signed when a second piece of information in addition to the message is provided to the host.

6.3.1 Undetachable Digital Signatures

The idea behind the "undetachable" signature scheme described in [40] is to link signatures to the output of a function in order to prevent the host from signing arbitrary messages. The scheme utilizes function composition (see Section 6.2.1). The assumption is that you have a rational function f, a rational signature function s, and a verification function, v, that can check the validity of a signature produced by the composition $f_{signed} = s \circ f$. The functions f and f_{signed} can be computed by a mobile agent to get an "undetachable" signature consisting of a pair $(f(x), z := f_{signed}(x))$. It can be checked via v, that $f(x)$ is a valid output of the function f. The security lies in the inability of an attacker to recover s from f and f_{signed}, i.e., to decompose a composite function. [40] presents a number of attacks on this system and attempts to fix them, but a detailed security analysis is not given.

This scheme can be used only if you have a mobile agent that you authorize to sign all outputs of a particular rational function - the originator can not specify particular inputs only. We next introduce a scheme that is more flexible and allows arbitrary messages to be signed.

6.3.2 Verifiably Linked Signatures

We introduce here the notion of verifiably linked signatures. Here the originator provides the agent with an obfuscated or encrypted signature scheme (to preserve the secrecy of the private key) and an input to be signed, together with a second set of information called *link data* that must be presented with the signature. The purpose of the link data is to bind the signature on the message to the originator as well as the message as proof that the message was intended for signature by the originator if the signature is challenged. Although the signing host could sign an arbitrary message, it does not have the ability to produce the proper corresponding link data.

The main application we envision is for a low power client to be able to ask a higher power host to sign an arbitrary message without giving the host the ability to sign messages without the client's approval. The message is verified as usual, however, if the client wishes to deny a message or if a signature is challenged, the client enters into a protocol to prove or deny the validity of the signature. Here the "prover" is the originating client.

Our goal is to create a scheme with the following properties:

1. The link data checks with a unique message.
2. The prover can verifiably deny a signature with invalid link data.
3. The prover is unable to deny a valid signature with valid link data.
4. No one but the client (prover) can produce valid link data.
5. For low power applications, the link data must be easy to compute.

In order to construct our design we draw from the notions of undeniable signatures [9] and chameleon hashes [25]. In undeniable signatures, a signature is produced which requires interaction for verification. The prover can deny an invalid signature, but cannot deny a valid one. A chameleon hash function is one for which collisions can be easily produced given knowledge of a trap door, but computationally infeasible otherwise. We incorporate these two ideas to produce our link data and the corresponding protocols to prove or deny linkability to the message and originator. In order to make this feasible for low power applications we require precomputation and storage of a number of values, but there may be other applications for which the data can be produced on the fly.

We call the client who wishes for a message to be signed the originator. The machine or agent enabled to produce the signature via an obfuscated or encrypted signature scheme is called the host. The signature scheme is denoted s, with corresponding verification scheme v. We assume the signature scheme is some obfuscatable or encryptable (e.g. rational) function. The public key consists of the public key for the signature scheme together with a prime p such that $p = 2q + 1$ with q a large prime, an element g of order q, and an element $y = g^x$. The secret key is the secret key associated with the signature scheme, together with x. The originator precomputes and stores 4-tuples of the form $(r, f, Z = g^{xr+rf}, T = g^{r^{-1}})$[3] where r is a secret, randomly chosen number. Such a 4-tuple will be needed for each signature.

Signature Protocol

1. To sign a message m, the originating client chooses a 4-tuple and computes $\ell = fm^{-1}$ $(f = m\ell)$.
2. The client computes $m' = (m + x\ell - xr - rf)x^{-1} \pmod q$.
3. The client sends the host m together with the link data, (ℓ, m', Z, T).
4. The host signs the message m, the signature is $(m, s(m), \ell, m', Z, T)$.

The message is verified as usual using the verification algorithm v on the signature piece $(m, s(m))$. However, in the case that the originator wishes the deny the signature or a challenge to the signature is made, the originator (prover) and a challenger (verifier) enter into the following protocol. There are two checks to perform. The second is interactive and allows the prover to verifiably deny a signature.

[3]Whenever there is an inverse in the exponent it will always be modulo q hence we will omit writing the "mod q" in the exponent for ease of notation.

Verification of Link Data Protocol

- Check 1: The verifier checks if $g^m y^\ell = y^{m'} Z \pmod p$. If not, then output invalid and stop.
- Check 2:
  1. The verifier chooses a random a, b and sends the challenge $C = Z^a g^b$ to the prover.
  2. The prover computes $R = C^{r^{-1}}$ and sends it back to the verifier.
  3. The verifier checks if $R = y^a g^{m\ell a} T^b$. If yes, then output valid and stop.
  4. If the output does not check, the verifier issues a second challenge to the prover: $C' = Z^e g^f$.
  5. The prover computes $R' = C'^{r^{-1}}$ and sends it back to the verifier.
  6. The verifier checks if $(RT^{-b})^e = (R'T^{-f})^a$. If so, the prover is consistent and the output is invalid, otherwise the verifier is cheating.

Check 1 should hold since $g^m y^\ell = y^{m'} Z \pmod p = g^{xm'} g^{xr+xm\ell}$ if and only if $m + x\ell = xm' + xr + rm\ell$ which holds by construction of m' in Step 2 of the Signature Protocol (recall $f = m\ell$). Check 2 is similar to the undeniable signature verification protocol in [9]. If the prover is honest and the link data is valid, then note that step 3 checks:

$$\begin{aligned} R = C^{r^{-1}} &= (Z^a g^b)^{r^{-1}} \\ &= ((g^{xr+rm\ell})^a)^{r^{-1}} (g^b)^{r^{-1}} \\ &= g^{xrar^{-1}} g^{rm\ell ar^{-1}} g^{br^{-1}} \\ &= g^{xa} g^{m\ell a} g^{r^{-1}b} \\ &= y^a g^{m\ell a} T^b. \end{aligned}$$

If step 3 does not check, then either the signature is invalid or the prover is lying. A lying prover is caught by the repeating of the challenge and comparison of answers in steps $4-6$. (Note that $(RT^{-b})^e = (Z^{ar^{-1}})^e = (Z^{er^{-1}})^a = (R'T^{-f})^a$.) The following two lemmas show that a prover cannot produce valid proof for invalid link data and that a prover trying to deny valid link data will be caught. The proofs of the following two lemmas follow directly from [9] with simple modifications.

Lemma 6.3.1. *The prover cannot provide a valid response to invalid link data with greater than negligible probability.*

Lemma 6.3.2. *The prover cannot avoid detection of inconsistency between two invalid responses to a valid signature with greater than negligible probability.*

We now show that our scheme satisfies all of our goals:

1. The link data checks with a unique message: Suppose there were two messages m_1 and m_2 with link data (ℓ, m', Z, T), then Check 1 implies that

$$\begin{aligned} g^{m_1} y^{\ell} &= y^{m'} Z \text{ and} \\ g^{m_2} y^{\ell} &= y^{m'} Z \\ &\Leftrightarrow \; g^{m_1} = g^{m_2} \\ &\Leftrightarrow \; m_1 = m_2 \pmod{q}. \end{aligned}$$

2. The prover can verifiably deny a signature with invalid link data: This is proved by Lemma 6.3.1.
3. The prover is unable to deny a valid signature with valid link data: This is proved by Lemma 6.3.2.
4. No one but the client (prover) can produce valid link data: The security of this scheme is based on the security of the discrete logarithm problem. Link data cannot be produced directly without knowledge of the secrets x and r (or by solving the discrete log). The data for Check 1 is created by knowing the trapdoor x, r and is infeasible to compute otherwise (see [25]).

   Both checks are required to prevent the host from being able to forge link data for a new message. If the host modifies a message m to be mk for some k, then in order to pass Check 1, he must modify Z, replacing it by Zg^k (modifying ℓ or m' properly requires knowledge of the secret key x). However, modification of Z causes Check 2 to fail. Nevertheless, Check 2 alone is not enough: Suppose the host modifies a message m to be mk. The host could pass Check 2 by modifying ℓ to be ℓk^{-1} (Z and T cannot be modified properly without knowledge of the secrets x, r), but this causes Check 1 to fail. Hence both checks together provide immunity against forgery.
5. For low power applications, the link data must be easy to compute: The data is precomputed and stored, at the time of a request to sign a message, the client need only compute a few multiplications if x^{-1} is also precomputed and stored.

Our protocols provide a mechanism for the originator to refute a signature that it did not intend the host to sign, but maintains accountability of the originator to stand by a valid signature, hence non-repudiation is retained while achieving the ability to allow a remote host to compute signature on behalf of the originator. We believe that these must be requirements for secure remote signatures. We note that the protocols to verify or deny link data are not entered into for every signature, only in the case of a dispute and so in general does not put undue demand on a low power client. However we also note that there are several undesirable properties associated with this scheme including the precomputation and storage required for the originator, the interactive nature and computation-heavy properties of the verification or deny protocols, and the long length of the signature and link data. Future work will focus on creating a scheme with fewer of these undesired properties. The main point here was to introduce the concept and requirements for verifiably linked signatures.

Chapter 7

Impossibility vs. Possibility Results for Circuit Obfuscation

In the previous chapters, we have examined a variety of new software obfuscation techniques that have been proposed in the past few years. Most of these techniques rely on informal notions of secure obfuscation, therefore little can be said about their security. In order to obtain provable security a formalized model of obfuscation needs to be constructed. The first to initialize a formal investigation into obfuscation models came from Barak, Goldreich, et al. in [4]. Using concepts from modern cryptography they introduced virtual blackbox obfuscation. They show in general it is impossible to construct an all-purpose program obfuscator under this model. Following this work, several alternative models were proposed [31, 47]. These models were the first to construct provably secure obfuscated programs.

Our goal in this chapter is to provide a technical summary of several different obfuscation models [4, 31, 47]. We will explore the various approaches and give a brief summary of their advantages and disadvantages. The organization of this chapter is as follows. In Section 7.1 we briefly review some basic notation and definitions. In Section 7.2 we discuss the virtual blackbox obfuscation model given in [4] and provide a revised proof that circuit obfuscators do not exist. We follow this discussion by reviewing Lynn, Prabhakaran, and Sahai, circuit obfuscation in the random oracle model [31]. We see that password identification schemes can be securely obfuscated if we allow the existence of random oracles. Finally, we finish by presenting the information-theoretic circuit obfuscation of Varnovsky and Zakharov [47] and see that password identification schemes are also obfuscatable under this model.

7.1 Preliminaries

We will use the notation PPT to stand for probabilistic polynomial-time Turing machine. If A is a PPT, B an oracle, and x an input to A, then by $A^B(x)$ we mean the algorithm that runs A on input x using oracle access to B. A *circuit* C is a directed acyclic graph in which every node, with in-degree 0 (*resp. out-degree* 0) is called an input (*resp. output*). Every other node is called a gate and is represented by an arbitrary 2 to 1 function $f : \{0,1\}^2 \rightarrow$

$\{0,1\}$. The size of the circuit $|C|$, is equal to the number of gates contained in the circuit.

If D is a distribution, then $\text{Supp}(D)$ will denote the measurable set of elements that have nonzero probability. When writing $x \stackrel{R}{\leftarrow} D$ we mean x is chosen according to the distribution of D. Whenever D is a set of binary strings of some fixed length, we will assume D has a uniform distribution unless stated otherwise. A function $\mu : \mathbb{N} \rightarrow \mathbb{R}^+$ is said to be *negligible*, if for any positive polynomial p there exists an integer N such that for any $k > N$, $\mu(k) < 1/p(k)$. We will sometimes use the notation $neg(\cdot)$ to denote an arbitrary negligible function.

A polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is called *one-way* if for every PPT A,

$$\Pr_{x \stackrel{R}{\leftarrow} \{0,1\}^k} [\, f^{-1}(f(x)) \ni y \leftarrow A(f(x), 1^k)] \leq neg(k).$$

Goldreich, Goldwasser, and Micali showed that if one-way functions exist, then so do pseudorandom functions [19]. A polynomial-time computable predicate $h : \{0,1\}^* \rightarrow \{0,1\}$ is said to be a *hard-core* predicate of a one-way function f if for every PPT A,

$$\Pr_{x \stackrel{R}{\leftarrow} \{0,1\}^k} [A(f(x), 1^k) = h(x)] \leq \frac{1}{2} + neg(k).$$

7.2 Obfuscators

In this section we examine several different obfuscation models [4, 31, 47]. We start our examination by highlighting the key properties of obfuscation. We will use these concepts to characterize the different types of models. Obfuscation in general has two main properties. The first property *functionality*, states that an obfuscated program is behaviorally equivalent to the original. Informally, this means the program is no less efficient and its input-output behavior is identical. Most authors define efficiency to mean that the obfuscated program's description length and running time are no more than polynomial larger than the original. The second obfuscation property *readability*, measures the unintelligibility of the obfuscation technique. This determines the type of information being protected and how its security is defined. Of the two properties, the second is often considered the most subjective, since unintelligibility has many forms of interpretation.

The first model we consider was proposed by Barak, Goldreich, et al. and is called virtual blackbox obfuscation [4]. Their work was the first to formalize the functionality and readability properties discussed above. The virtual blackbox model is an ideal form of obfuscation, in the sense that obfuscators behave like a virtual blackbox. This implies that everything that can be efficiently extracted from the obfuscated program can also be extracted by observing its input-output behavior. This type of security protects against all forms of information extraction and is regarded as a perfect form of obfuscation. Unfortunately, the main result of their paper proves that a general purpose program obfuscator does not exist (as we will soon show).

We begin by restating the definition of obfuscation as presented in [4]. We will restrict ourselves to circuit obfuscation throughout the rest of the section. The more restrictive case of Turing machine obfuscation is discussed in more detail in the original paper.

Definition 4. (Circuit Obfuscation in the Blackbox Model [4]) A probabilistic algorithm $\mathcal{O}$ is a circuit obfuscator if the following three conditions hold:

- (Functionality) For every circuit C, the string $\mathcal{O}(C)$ describes a circuit that computes the same function as C.
- (Polynomial Slowdown) There is a polynomial p such that for every circuit C, $|\mathcal{O}(C)| \leq p(|C|)$.
- (Virtual Blackbox) For any PPT A, there is a PPT S and a negligible function ν such that for all circuits C

$$\left|\Pr[A(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]\right| \leq \nu(|C|).$$

As you may have noticed the first two properties in the above definition satisfy the functionality requirement for obfuscation. The virtual blackbox condition measures the readability of the obfuscation technique. The PPT S simulates the advantage A has in determining some information about the obfuscated circuit. Their difference should be negligible with respect to the size of the circuit, when S is given oracle access to C. The definition calls for an obfuscator that works over all circuits. The main impossibility results prove that such an obfuscator does not exist. This condition may be weakened by considering circuit ensembles instead. Unfortunately even under this weakened assumption, it is still possible to show that circuit obfuscators do not exist, provided one-way functions exist. This result does not exclude the possibility of the existence of a securely obfuscatable circuit ensemble. Rather, it states that not all circuit ensembles can be obfuscated. In the following example we provide a construction of a class of circuits that are trivially obfuscatable under this model.

Example. A family of circuits $\mathcal{F}$ is said to be learnable, if there exists a PPT T such that for every $C \in \mathcal{F}$, the probability that $T^C(1^{|C|})$ returns a polynomial size circuit C^* that computes C, is 1. Let A be any PPT algorithm as defined in the virtual blackbox property of Definition 4 and $\mathcal{F}$ a family of learnable circuits. Since $\mathcal{F}$ is learnable there exists a PPT T such that,

$$\Pr[T^C(1^{|C|}) = C^*] = 1.$$

We define an obfuscator $\mathcal{O}$ as the algorithm that takes a circuit C and returns $C^* \leftarrow T^C(1^{|C|})$. Let S be the PPT algorithm that runs T using oracle C for the first half and A for the second. S outputs A's result. Based upon this construction we have,

$$\begin{aligned}
&\left|\Pr[A(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]\right| \\
&= \left|\Pr[\,(A \circ T^C)(1^{|C|}) = 1] - \Pr[\,(A \circ T^C)(1^{|C|}) = 1]\right| \\
&= 0.
\end{aligned}$$

The previous example serves as a reference point for obfuscatable circuits. Every other family or circuit ensemble is said to be non-trivial provided they are not learnable. It is still an open problem whether a non-trivial case exists. To help understand why this model fails in general we will construct a family of circuits that are unobfuscatable. Our construction will exploit the fact that physical access is a much more powerful concept then oracle access.

Definition 5. We define the circuit family of multi-point functions as the set of all circuits $C_{\alpha,\beta}$, $\alpha, \beta \in \{0,1\}^k$ such that,

$$C_{\alpha,\beta}(x) := \begin{cases} \beta, & \text{if } x = \alpha \\ 0^k, & \text{else} \end{cases}$$

In our construction we will assemble a family of circuits by gluing circuits together. Each circuit in the family will be the combination of two other circuits, such that it's domain and codomain are the coproducts of the other two. This will allow us to easily decompose the circuit in its respective halves. This combination is defined as follows.

Definition 6. Let $f : A \to B$ and $g : C \to D$ be two functions, we define their combination to be the function $f \amalg g : A \amalg C \to B \amalg D$, where

$$(f \amalg g)(x) := \begin{cases} f(x) \in B, & \text{if } x \in A \\ g(x) \in D, & \text{if } x \in C \end{cases}$$

The main idea of the proof is to construct a circuit $D_{\alpha,\beta}$ that takes circuits $C_{\alpha',\beta'}$ and checks if $C_{\alpha',\beta'}(\alpha) = \beta$. If it does $D_{\alpha,\beta}$ outputs 1, else it returns a 0. We construct a family of circuits $\mathcal{F} = \{F_k\}_{k\in\mathbb{N}}$ by defining F_k as the union of $C_{\alpha,\beta} \amalg D_{\alpha,\beta}$ and $C_{\alpha,0^k} \amalg D_{\alpha,\beta}$ for all $\alpha, \beta \in \{0,1\}^k$. Note we can decompose the two circuits more easily by inserting a single selection bit to determine the function we are using. Ideally we would like to decompose the circuits so that checking $D_{\alpha,\beta}(C_{\alpha,\beta}) = 1$ is easy, whenever direct access to the circuit is available. The main obstacle is that we cannot guarantee that the representation of the decomposition for either $C_{\alpha,\beta}$ or $\mathcal{O}(C_{\alpha,\beta})$ is small enough for an input into $D_{\alpha,\beta}$ or $\mathcal{O}(D_{\alpha,\beta})$. Therefore we will need a modification that will allow us to do this.

This modification will be in the form of a probabilistic polynomial-time algorithm, such that given oracle access $D_{\alpha,\beta}$ and input $C_{\alpha,\beta}$, it is easy to make the desired check. For physical access to the circuit this check can be performed with certainty, even for randomized choices in $\mathcal{F}$. However, for oracle access only, we cannot determine $D_{\alpha,\beta}(C_{\alpha,\beta})$ any better than guessing when α, β are chosen at random. This is stated formally in the following Lemma.

Lemma 7.2.1. *If one-way functions exist, then for every $k \in \mathbb{N}$ and $\alpha, \beta \in \{0,1\}^k$, there is a distribution $\mathcal{D}_{\alpha,\beta}$ on circuits such that:*

- *Every $D \in \mathrm{Supp}(\mathcal{D}_{\alpha,\beta})$ is a circuit of size poly(k).*
- *There is a polynomial-time algorithm A such that for any circuit C describing a multi-point function, and any $D \in \mathrm{Supp}(\mathcal{D}_{\alpha,\beta})$, $A^D(C, 1^k) = 1$ if and only if $C(\alpha) = \beta$.*
- *For any PPT S, $\Pr[S^D(1^k) = \alpha] \leq neg(k)$, where the probability is taken over $\alpha, \beta \xleftarrow{R} \{0,1\}^k$, $D \xleftarrow{R} \mathcal{D}_{\alpha,\beta}$, and the coin tosses of S.*

Proof: The core of the proof rests upon the construction of a circuit D that allows one to compute $C(\alpha)$ without exposing the values of either α or β. This is accomplished by encrypting the value α and simulating the outputs of each gate of the circuit C using a homomorphic function.

Consider the probabilistic encryption scheme $\mathrm{Enc}_K(b) := (r, f_K(r) \oplus b)$, where $r \overset{R}{\leftarrow} \{0,1\}^{|K|}$ and f_K is a pseudorandom function. The existence of f_K relies on our assumption that one-way functions exist. We define an algorithm Hom, that takes as input two private keys K, K', two ciphertext c and d, and a binary operation $\odot$. The binary operation $\odot$ is used to simulate the outputs of an arbitrary two to one gate contained in the circuit C and can be specified by a 2×2 truth table. Specifically we define,

$$\mathrm{Hom}_{K,K'}(c, d, \odot) := \mathrm{Enc}^*_{K'}(\mathrm{Dec}_K(c) \odot \mathrm{Dec}_K(d)).$$

where $\mathrm{Enc}^*_{K'}$ denotes encryption using $r = f_{K'}(r_c, r_d, \odot)$ and $f_{K'}$ is a pseudorandom function taking $2|K|+4$-bits to $|K|$-bits. The extra 4-bit input of $f_{K'}$ is used to specify the operation $\odot$. If we define the encryption of the k-bit string α as the k-tuple $(\mathrm{Enc}_K(\alpha_1), \ldots, \mathrm{Enc}_K(\alpha_k))$, then an adversary has no more than negligible success in determining α even when given oracle access to Hom. This is expressed in the following claim:

Claim 7.2.1. For any *PPT* A, the encryption of an arbitrary k-bit string α cannot be distinguished from a random string of the same size

$$\left|\Pr[A^{\mathrm{Hom}_{K,K'}}(\mathrm{Enc}_K(\alpha), 1^k) = 1] - \Pr[A^{\mathrm{Hom}_{K,K'}}(s, 1^k) = 1]\right| \leq neg(k),$$

$$\text{where } s \overset{R}{\leftarrow} \{0,1\}^{|\mathrm{Enc}_K(\alpha)|}$$

For a pair of keys K, K' and k-bit string $(r_1, \ldots, r_k)$ we construct the circuit family $\mathcal{D}_{\alpha,\beta}$ as the family of all circuits

$$D_{\alpha,\beta,K,K',(r_1,\ldots,r_k)} := \mathrm{Enc}_K(\alpha) \amalg \mathrm{Hom}_{K,K'} \amalg B_{K,\beta}$$

where $B_{K,\beta}$ is an algorithm which when fed a k-tuple of ciphertext $(c_1, \ldots, c_k)$ outputs 1 if for all i, $\mathrm{Dec}_K(c_i) = \beta_i$, and $\mathrm{Enc}_K(\alpha)$ is the encryption of α using r_i for each bit α_i. The distribution $\mathcal{D}_{\alpha,\beta}$ is defined as the family of circuits $D_{\alpha,\beta,K,K',(r_1,\ldots,r_k)}$ over uniformly selected keys K, K' and $(r_1, \ldots, r_k)$.

The first property of the Lemma states $D_{\alpha,\beta,K,K',(r_1,\ldots,r_k)}$ is a circuit with size polynomial in k. This is fairly straight forward since the pseudorandom function used in the probabilistic encryption scheme can be efficiently generated in poly(k). Similarly both $\mathrm{Hom}_{K,K'}$ and $B_{K,\beta}$ can be computed in poly(k). Hence the claim follows.

The second property states that there exists an efficient algorithm A that can check if $C(\alpha) = \beta$ using only oracle access to $D_{\alpha,\beta,K,K',(r_1,\ldots,r_k)}$. This is clear since we can use the homomorphic function to calculate the encrypted output at each gate in the circuit C, see Figure 7.1. Since C is a multi-point function this should take no more than poly(k) steps. We can then use $B_{K,\beta}$ to then check if the final output decrypts to β.

For the third and final claim we must show that for any *PPT* S, S has only negligible probability of finding α when given oracle access to $D_{\alpha,\beta,K,K',(r_1,\ldots,r_k)}$. Since β is independent of α,K,K', and $(r_1, \ldots, r_k)$ the probability that S queries $B_{K,\beta}$ at a point that is nonzero is negligible. Therefore we can remove oracle access to $B_{K,\beta}$ with only a negligible effect on the success of S. Based on our previous claim it follows S cannot determine α with any more than negligible success, because if it could it would be able to distinguish

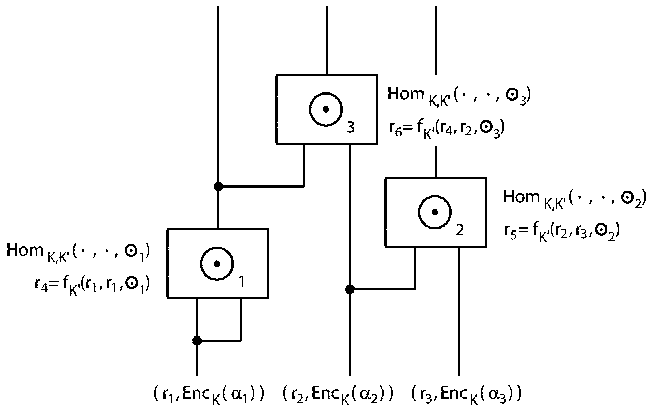

Figure 7.1. Gate by gate emulation of a circuit for $k = 3$.

the encryption of α from a random string. □

Formalizing our previous argument we now state the main Proposition.

Proposition 1. *If one-way functions exist, then circuit obfuscators do not exist.*

Proof: We will prove this by contradiction. Suppose there exists an obfuscator $\mathcal{O}$ and let $C_{\alpha,\beta}$ be a circuit computing a multi-point function. For each $k \in \mathbb{N}$ consider the following two distributions,

- $\mathcal{F}_k$: Choose α and β uniformly in $\{0,1\}^k$, $D \stackrel{R}{\leftarrow} \mathcal{D}_{\alpha,\beta}$. Return $C_{\alpha,\beta}$ II D.
- $\mathcal{G}_k$: Choose α and β uniformly in $\{0,1\}^k$, $D \stackrel{R}{\leftarrow} \mathcal{D}_{\alpha,\beta}$. Return $C_{\alpha,0^k}$ II D.

where $\mathcal{D}_{\alpha,\beta}$ is the distribution on circuits as given in Lemma 7.2.1. Let A be the PPT algorithm guaranteed by Property 2 in Lemma 7.2.1, and consider a PPT A' which on input a circuit F, decomposes $F = F_0$ II F_1 and evaluates $A^{F_1}(F_0, 1^k)$. Thus, when fed a circuit from $\mathcal{O}(\mathcal{F}_k)$ respectively $\mathcal{O}(\mathcal{G}_k)$, A' is evaluating $A^D(C, 1^k)$ where D computes the same function as some circuit from $\mathcal{D}_{\alpha,\beta}$ and C computes the same function as $C_{\alpha,\beta}$.

Let $\mathcal{H}_k$ be the ensemble of circuits that runs either $\mathcal{F}_k$ or $\mathcal{G}_k$ with $1/2$ probability. We claim that at least one of the differences

$$\left|\Pr[A'(\mathcal{O}(\mathcal{F}_k)) = 1] - \Pr[S^{\mathcal{F}_k}(1^k) = 1]\right| \text{ and } \left|\Pr[A'(\mathcal{O}(\mathcal{G}_k)) = 1] - \Pr[S^{\mathcal{G}_k}(1^k) = 1]\right| \quad (7.1)$$

is not bounded above by a negligible function. If we can show for any PPT algorithm S

$$\left|\Pr[S^{\mathcal{F}_k}(1^k) = 1] - \Pr[S^{\mathcal{G}_k}(1^k) = 1]\right| \leq neg(k).$$

then our claim will follow since Equation 7.1 becomes,

$$\left|1 - \Pr[S^{\mathcal{F}_k}(1^k) = 1]\right| \quad \text{and} \quad \left|2^{-k} - \Pr[S^{\mathcal{G}_k}(1^k) = 1]\right|.$$

Let E be the event that at least one of the queries to the left hand of the decomposition is non-zero for $C_{\alpha,\beta}$. Note, this probability depends only on the choice of α and β and not on $\mathcal{F}_k$ or $\mathcal{G}_k$. Then,

$$\begin{aligned}
&\Big|\Pr[S^{\mathcal{F}_k}(1^k) = 1] - \Pr[S^{\mathcal{G}_k}(1^k) = 1]\Big| \\
&= \big|\Pr[S^{\mathcal{F}_k}(1^k) = 1 \mid E] \cdot \Pr[E] + \Pr[S^{\mathcal{F}_k}(1^k) = 1 \mid \overline{E}] \cdot \Pr[\overline{E}] \\
&\qquad\qquad - \Pr[S^{\mathcal{G}_k}(1^k) = 1 \mid E] \cdot \Pr[E] - \Pr[S^{\mathcal{G}_k}(1^k) = 1 \mid \overline{E}] \cdot \Pr[\overline{E}]\big| \\
&= \Big|\Pr[S^{\mathcal{F}_k}(1^k) = 1 \mid E] - \Pr[S^{\mathcal{G}_k}(1^k) = 1 \mid E]\Big| \cdot \Pr[E].
\end{aligned}$$

since $\Pr[S^{\mathcal{F}_k}(1^k) = 1 \mid \overline{E}] = \Pr[S^{\mathcal{G}_k}(1^k) = 1 \mid \overline{E}]$. From the previous Lemma $\Pr[E]$ must be negligible and hence the above claim follows.

Choose any increasing positive polynomial p. By Property 3 of the virtual blackbox model there exists an integer N and a PPT S, such that for every circuit C with $|C| > N$,

$$\Big|\Pr[A'(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]\Big| < \frac{1}{p(|C|)}.$$

In our construction of the circuit ensemble $\mathcal{H}_k$, we will assume each circuit has size k^t for some integer $t \geq 1$. If it doesn't we can add gates so that it does. Without loss of generality suppose that the first equation in Equation 7.1 is not negligible. We can replace the input into S from 1^k to 1^{k^t}, since there is an efficient one-to-one relationship. Therefore there exists an infinite number of k satisfying,

$$\Big|\Pr[A'(\mathcal{O}(\mathcal{F}_k)) = 1] - \Pr[S^{\mathcal{F}_k}(1^{k^t}) = 1]\Big| > \frac{1}{p(k)}.$$

But,

$$\begin{aligned}
&\Big|\Pr[A'(\mathcal{O}(\mathcal{F}_k)) = 1] - \Pr[S^{\mathcal{F}_k}(1^{k^t}) = 1]\Big| \\
&= \Big|\sum_{C \in \mathcal{F}_k} \big(\Pr[A'(\mathcal{O}(C)) = 1 \mid C \leftarrow \mathcal{F}_k] - \Pr[S^C(1^{|C|}) = 1 \mid C \leftarrow \mathcal{F}_k]\big)\Pr[C \leftarrow \mathcal{F}_k]\Big| \\
&\leq \sum_{C \in \mathcal{F}_k} \Big|\Pr[A'(\mathcal{O}(C)) = 1 \mid C \leftarrow \mathcal{F}_k] - \Pr[S^C(1^{|C|}) = 1 \mid C \leftarrow \mathcal{F}_k]\Big| \Pr[C \leftarrow \mathcal{F}_k].
\end{aligned}$$

which implies there exists a circuit C such that $\big|\Pr[A'(\mathcal{O}(C)) = 1] - \Pr[S^C(1^{|C|}) = 1]\big| > 1/p(k)$. Thus for $k > N$, we have a contradiction. □

To remove the one-way function condition we will show how to construct a one-way function out of an obfuscator. This will prove circuit obfuscators do not exist unconditionally.

Lemma 7.2.2. *If efficient obfuscators exist, then one-way functions exist.*

Proof: Suppose that $\mathcal{O}$ is an efficient obfuscator. For $\alpha \in \{0,1\}^k$ and $b \in \{0,1\}$, define $C_{\alpha,b} : \{0,1\}^k \to \{0,1\}$ as the circuit that outputs b if the input $x = \alpha$, and 0 otherwise. We

claim that the collection of functions $f_k(\alpha, b, r) := \mathcal{O}_r(C_{\alpha,b})$, $k \in \mathbb{N}$ and using coin tosses r is one-way.

It is sufficient to show that returning α can be achieved with no more than negligible success, since α must be part of any preimage of $\mathcal{O}_r(C_{\alpha,b})$. But this is equivalent to showing b is a hard-core bit, since if we can extract α then we can also find b. Observe for any PPT S,

$$\Pr[S^{C_{\alpha,b}}(1^k) = b] \leq \frac{1}{2} + neg(k), \quad \alpha \xleftarrow{R} \{0,1\}^k, b \xleftarrow{R} \{0,1\}$$

Using Property 3 of Definition 4 it follows that for any PPT A,

$$\Pr[A(f_k(\alpha, b, r)) = b] = \Pr[A(\mathcal{O}_r(C_{\alpha,b})) = b] \leq \frac{1}{2} + neg(k).$$

over random choices of α, b, and r. Therefore b is a hard-core bit of f_k, which implies it is one-way. □

Using the previous Lemma, we have the following main result.

Corollary 1. *Efficient circuit obfuscators do not exist in the virtual blackbox model.*

It is clear based on the above result that a single method of obfuscation will not work over all circuits. Therefore in order to get positive results, obfuscators need to be more focused by considering specific applications. We examine this approach by looking at the feasibility of obfuscating symmetric-key encryption schemes. This example stresses the difficulty of obfuscation in general.

Ordinarily there are considerable differences between obfuscating an encryption scheme and an arbitrary program. For one, we can usually assume the details of the algorithm for the encryption scheme is publicly available. Therefore obfuscating the algorithm itself would serve very little purpose. The only unknown piece of information is the secret key. Thus an obfuscator only needs to provide protection against key extraction. This approach is different than the approach taken by the virtual blackbox model, since it protects against all forms of information extraction, including key recover. By limiting ourselves to specific types of information hiding, we increase the tools available for obfuscation. We believe at least in part, that obfuscation should be application specific, since different types of applications call for different types of protection.

Definition 7. (Key Recovery) A symmetric-key encryption scheme $\mathcal{SE} = (\mathcal{K}, E, D)$ is said to be obfuscatable if the following conditions hold

- (Functionality) For each private key K, there exists a circuit E_K that computes the encryption scheme $\mathcal{SE}$.
- (Readability) For any PPT A,

$$\Pr_{K \xleftarrow{R} \mathcal{K}(k)}[A(E_K) = K] \leq neg(k).$$

Unfortunately in [4] the authors prove the above definition is unattainable for certain symmetric-key encryption schemes. This negative result shows just how powerful physical

access to a circuit is. So far we have focused our attention on the impossibility results of several different obfuscation models. We now switch directions and look at ways to relax these model. We will see under certain conditions obfuscation is possible for specific applications.

The next model we look at was proposed by Lynn, Prabhakaran, and Sahai and is called the random oracle model [31]. This obfuscation model is a variant of virtual blackbox obfuscation, with one distinct difference. The difference is that, circuits now have the capability of accessing random oracles. This is a more powerful version of blackbox obfuscation, since any obfuscatable circuit in the virtual blackbox model is also obfuscatable by adding random oracles. The idea of using an oracle for obfuscation is quite powerful, since it assumes that part of the program is inaccessible. From a hardware perspective, the oracle may be viewed as a tamper resistant random function. Circuit obfuscation in the random oracle model is defined as follows.

Definition 8. (Circuit Obfuscation in the Random Oracle Model [31]) A probabilistic algorithm $\mathcal{O}$ is a circuit obfuscator for a family of circuits $\mathcal{F} = \{\mathcal{F}_k\}_{k\in\mathbb{N}}$ if the following three conditions hold:

- (Approximate Functionality) There exists a negligible function μ such that, for all k and $C \in \mathcal{F}_k$ we have

$$\Pr[\,\exists\, x \text{ such that } \mathcal{O}^{\mathcal{R}}(C)(x) \neq C(x)] \leq \mu(k).$$

- (Polynomial Slowdown) There is a polynomial p such that, for all k and $C \in \mathcal{F}_k$, $|\mathcal{O}(C)| \leq p(k)$.
- (Virtual Blackbox) For any PPT A, there is a PPT S and a negligible function ν such that, for all k and $C \in \mathcal{F}_k$

$$\left|\Pr[A^{\mathcal{R}}(\mathcal{O}^{\mathcal{R}}(C)) = 1] - \Pr[S^{C}(1^k) = 1]\right| \leq \nu(k).$$

The main result of [31] shows that password identification schemes are obfuscatable under this model. A password identification scheme is a family of point functions $\{F_k\}_{k\in\mathbb{N}}$, such that each function $f_\alpha \in F_k$, outputs a 1 if $x = \alpha \in \{0,1\}^k$ and 0 else. A point function is just a multi-point function with a binary output. We can construct an obfuscator in the random oracle model for this family using the following procedure. Let $\mathcal{R}$ be a random oracle taking k-bits to $2k$-bits and $\mathcal{O}^R(f_\alpha)$ the circuit that stores $r \leftarrow R(\alpha)$, such that on input $x \in \{0,1\}^k$ checks if $R(x) = r$. If equality holds $\mathcal{O}^{\mathcal{R}}(f_\alpha)$ return a 1, else a 0. Based on this construction we have the following result.

Proposition 2. *Password identification schemes are obfuscatable in the random oracle model.*

This result is possible, because of the addition of the random oracle. The oracle acts as a blackbox thereby protecting the password. Other oracle obfuscation models can be constructed using these same ideas. The main disadvantage in using this model is the oracle implementation. If an external source is not available, then the oracle has to be

implemented internally. This puts the burden of security on constructing an obfuscated random function. The advantage in this approach is that the security has been reduced to solving a single problem. The disadvantage is that a solution may not exist.

The final model we look at is called the information-theoretic model and was proposed by Varnovsky and Zakharov [47]. In this model, the authors take a more pragmatic approach by specifying a secret predicate for obfuscation. The secret predicate is chosen based on the application and is used to define the type of information to be hidden. This is similar to the key recovery obfuscation model discussed earlier. Further, this model abandons the notion of blackbox security in favor of information theory. Specifically they define a secret predicate P over an ensemble of programs $\{F_k\}_{k\in\mathbb{N}}$. The predicate P can be considered a random variable on F_k, by defining $\Pr_{f\leftarrow F_k}[P(f) = b]$, $b \in \{0, 1\}$. An obfuscator $\mathcal{O}$ is said to be secure, if for any adversary A, the mutual information between P and $A(\mathcal{O})$ is negligible for random choices $f \in F_k$.

The mutual information between two random variables is a measurement of the amount of information one random variable contains about the other. The smaller the value the less dependence they share. The mutual information between two random variables is defined as follows.

Definition 9. Let X and Y be two random variables with a joint probability mass function $p(x, y)$ and marginal probability mass functions $p(x)$ and $p(y)$. The mutual information $I(X;Y)$ between X and Y is defined as,

$$I(X;Y) := \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x,y)}{p(x)p(y)}$$

Intuitively, mutual information security measures the difficulty in determining some information about X, given Y only. If X is a predicate P and Y an obfuscator $\mathcal{O}$, this means on average it is difficult to determine $P(C)$, given only $\mathcal{O}(C)$ for random choices of C. This motivates the following definition.

Definition 10. (Information-Theoretic Circuit Obfuscation [47]) A probabilistic algorithm $\mathcal{O}$ is a circuit obfuscator for a circuit ensemble $\mathcal{F} = \{\mathcal{F}_k\}_{k\in\mathbb{N}}$ and secret predicate P, if the following three conditions hold:

- (Functionality) For every k and circuit $C \in \mathcal{F}_k$, the string $\mathcal{O}(C)$ describes a circuit that computes the same function as C.
- (Polynomial Slowdown) There is a polynomial p such that, for all k and $C \in \mathcal{F}_k$, $|\mathcal{O}(C)| \leq p(k)$.
- (Mutual Information Security) For any PPT A, there is a negligible function ν such that for all k,
$$I(P; A(\mathcal{O})) \leq \nu(k).$$

Since mutual information only measures average security, we cannot say for certain whether all circuits are securely obfuscatable. It may be the case that certain circuits are easy to break, while others are very difficult. This is not a particularly attractive quality for many types of applications. Nonetheless it is useful tool for measuring security. As

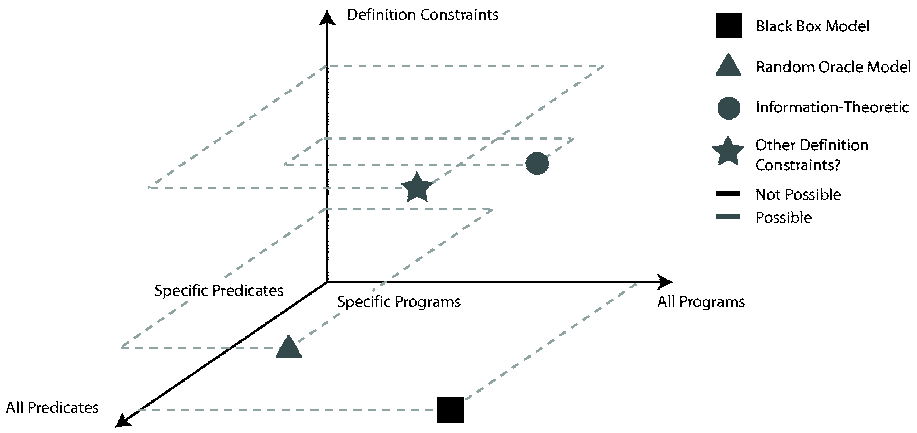

Figure 7.2. Possibility results for circuit obfuscation.

an example the authors show that password identification schemes are obfuscatable. They start by constructing an ensemble of programs, that contain both a simple identification scheme and free access simulator. The predicate P is chosen as a distinguisher between the two programs. Given that one-way permutations exist we have the following main result.

Proposition 3. *If one-way permutations exist, then the secret predicate P can be securely obfuscated for password identification schemes.*

The above result was the first possibility result for a formalized obfuscation model.

In figure 7.2 we have a visual description of the obfuscation models we have discussed in this section. Based on the results, it is unclear if any of these models will survive as a security standard. We believe there is still a great deal more research that can be achieved. For future work, we plan to incorporate the various ideas presented in each model into our own. We will focus on developing models that are geared towards specific types of applications. The predicate based information hiding approach given by the information-theoretic model holds promise and we will investigate further into extending these ideas. Given the relative immaturity of obfuscation, we are optimistic in developing obfuscation models that expand our current understanding of the field.

Chapter 8

Future Obfuscation Work

As we have already noted a theory of obfuscation is not available today. If and when we have one, we will have more than intuition at our disposal when we compare two obfuscated programs. We would like to be able to determine which program provides more obfuscation. This would enable us to rank order obfuscations. The goal, unfortunately, is more grandiose: we would like to be able to determine the lower bounds in time for an adversary to break the obfuscation. In this chapter we consider different possibilities for such a theory.

Collberg [12] presents a number of techniques. It is logical to presume that if one of these techniques is good, then two might be better. But it is possible that some sets of techniques work against each other, weakening the set, possibly reducing the obfuscation, maybe even making the program easier to understand. We have no guide here – we are flying blind, so to speak, so we do not know what will happen. Instead of applying a set of techniques, perhaps we could iteratively obfuscate and deobfuscate, choosing obfuscation techniques at each iteration. Given the reasonable assumption that deobfuscators do not always generate the original program, this approach might mimic the "wearing out" of code that makes legacy code eventually unmaintainable. This appears to use entropy to our advantage – an unusual arrangement. For this application we would like to know the relationship between number of iterations and obfuscation strength. But again we have no guide. Worse, the inevitable bugs in the obfuscators and deobfuscators work against us. What confidence do we have that the final obfuscated product has the same input/output behavior as the original program? This line of reasoning suggests that for a theory of obfuscation we need to get "below" the techniques and look for an ideal, for a perfect obfuscation system.

Cryptography has a model of a perfect cryptosystem, namely the Vernam cipher, also known as the one-time pad: a random and never re-used key stream of zeros and ones is xored with the input, which also consists of zeros and ones. Presuming that the key stream is random and is never re-used, then the cipher stream is perfectly secure. This system has actually been used, though it is impractical for most purposes. The value of the scheme is primarily its ideal nature. Is there a similar ideal scheme for obfuscation that would serve as a starting point?

Yes, as a matter of fact, there is such an ideal scheme for obfuscation. Unfortunately it is even less practical than the Vernam cipher. Here is the scheme: ask the adversary to

run all possible programs for n steps - the length of our program - on all possible inputs, reporting the output whenever a program halts;[1] when the input and program we want halts, then we have our answer. At some point thereafter we tell the adversary to stop charging us for compute cycles. We could call this "Vernam obfuscation," to suggest that this is a perfect method of obfuscation. Surprisingly there is no obfuscation involved! Our program is run as is. We protect our program by hiding it amongst many other programs. Although this is a completely impractical approach it does provide us with a starting point.

We can make Vernam obfuscation increasingly efficient[2] by limiting the size and the range of the input or the number of programs executed, but as we do so we provide the adversary with more information, narrowing the adversary's search space. Does this process narrow the adversary's search space faster than what we gain in efficiency? What is the shape of the function that describes the trade-off between efficiency and security?

Unfortunately, the Vernam obfuscation approach seems to lack a "workload advantage." We want to show two results simultaneously. First, we want to show that for a given program with n steps that the adversary has to pay an exponential price to break the obfuscation. That is, the adversary has to consider mn programs, where $m > 1$. We could call this "possible program explosion." Second, we want to show that for the same program we have to pay only a linear price, or maybe only a polynomial price, to create the obfuscation and execute the resulting obfuscated program. If we can show these two results, then we have the adversary "over a barrel," as the expression is, and we are on our way to a theory of obfuscation.

Perhaps we could construct Vernam obfuscation by building a program that uses p instructions for each instruction in the original program. Rivest [38] presents a cryptographic approach, called "winnowing," that provides privacy via integrity that is similar to this. The idea of winnowing, in the extreme, is that the sender sends both a zero and a one for each bit in the message. For each zero and for each one, the sender includes a Message Authentication Code (MAC) such that for each pair of bits only one MAC will authenticate. The receiver recovers the message by "winnowing," by discarding as "chaff" the bit in each pair that does not authenticate. The message, like an obfuscated program, is in plaintext, but the message is afforded privacy because it is hidden, in a sense: the adversary does not know which bit of each pair is part of the message. So the adversary has to consider all 2^n possible messages. This is the workload advantage. Can we make this approach work for obfuscation?

We could begin by considering an approach that for simplicity uses two instructions for each of the n instructions in the original program. We obfuscate by adding a "phony" instruction for each real instruction. The adversary is forced to winnow the real instructions from the phony ones.

The problem we have that the winnowing message-sender does not have is state: state persists between instructions. The adversary in the winnowing case has to consider all 2^n possible messages because either the zero or the one of each pair can be in the message. But this is not the case with our program. For example, the following shows pairs of instructions

[1] Of course any reasonable adversary would run the programs using dovetailing so that he is not caught trying to complete a program that does not halt.

[2] Or perhaps it is better to say, "decreasingly inefficient."

for each step in the obfuscated program:

step i:	load register 1 from address ...
	(another instruction that does not use or load register 1)
step i+1:	load register 1 from address ...
	(another instruction)

The adversary knows that the instruction in step i from the original program is not the load, simply because that load is immediately overwritten in the next step.[3] The first load is "dead" code.

Our task is to construct a program such that any (or at least enough) of the instructions at step i could be part of a program that uses any (or at least enough) of the instructions at step i+1 (or some subsequent step), for any (or at least enough) i in the range of n. This is the first piece of the puzzle.

The second piece of the puzzle is the use of a key. Like cryptography the same obfuscation algorithm operating on the same program should produce a different obfuscated program given a different key. Using Kerckhoff's assumption, the security of the scheme should rest as much as possible on the security of the key alone. If we can align obfuscation with that assumption, then we can use results from cryptography to help with a theory of obfuscation.

Assuming the simplest approach, namely that the obfuscator adds one instruction for each instruction in the original program, the key could be used to determine whether the original or the phony (i.e., the added instruction) instruction comes first, as suggested in Table 8.1.

We presume that the key would be necessary to extract the results of program execution. This would suggest that a superset of the output of the original program should be sent back as a result of execution to the obfuscators computer. Perhaps that superset could be something like a trace [36]. Although this suggests that this approach could provide execution integrity, and privacy of execution, code, and data,[4] it is not clear what output should be generated.

The key could, like a one-time pad, have as many bits as the program has instructions and thus be just as random as keys for the Vernam cipher. Unlike the Vernam cipher the key never has to leave the owners control.

Unfortunately we have not addressed how we get a sufficient possible program explosion via this approach. That is, which instructions do we use for the phony ones? Is either puzzle piece possible? To our knowledge these are open questions.

However, before we leave this topic, consider another twist. Rivest points out that what we consider "chaff" could actually be another message.[5] In fact there could be m messages

[3] Unless, of course, the instruction at step i in the original program is not the equivalent of a no-op.

[4] This approach is too loosely defined to determine if it precludes the adversary from violating execution integrity by returning bogus output. Since some of the instructions inserted for obfuscation will execute, perhaps they could also serve to generate a result that provides a check on execution integrity.

[5] That is, one mans wheat is another mans chaff.

Step	Sample Key Bit *	Instruction Sequence
0	0	(real instruction 0)
		(phony instruction 0)
1	1	(real instruction 1)
		(phony instruction 1)
2	1	(real instruction 2)
		(phony instruction 2)
3	0	(real instruction 3)
		(phony instruction 3)
...	...	...

Table 8.1. Initial Part of an Obfuscated Program (* If the ith bit of the key is 0, then the real instruction for the ith pair is the first in the pair, otherwise it is the second in the pair.)

all interleaved in some random way known only to the sender.[6] Each of the m recipients, using its unique key, can separate its wheat (the bits in the message intended for that recipient) from the chaff (the bits intended for some other recipient, or the bits that really are chaff, included to confuse the adversary). As the broadcast stream continues, some messages complete while new ones begin as the quantity of true chaff waxes and wanes. Applying this to obfuscation, can we combine two programs such that it is infeasible for the adversary to untangle them?

The problem again is state. The state of a program includes the contents of some registers (at least the program counter) and some portion of memory (at least the data that is contributing to the output). We are tempted at this point to appeal to functional programming [32] because of referential transparency: a functional programming function depends only on its inputs. The same function always returns the same results given the same inputs; it is independent of state. This is a step toward messages that consist of zeros and ones that also carry no state. This suggests that instead of considering instructions as our basic unit of obfuscated programs, as we have in the discussion above, we should consider functional programming functions.[7] This is counter-intuitive because these objects are at a higher level of abstraction than instructions and thus more likely to be easier to

[6]The collecting of many programs in order to protect each has some similarities to the Crowds system for anonymity of web transactions [36].

[7]There is a subtlety here. What the adversary sees could consist of assembly language statements generated by a compiler, just as we have tacitly assumed in the discussion further above. But if we are using functional programming functions, then alternatively and without loss of protection what the adversary sees could consist of high-level language statements since we presume that the adversary could decompile that assembly language into a high-level language. If the obfuscation is done properly, then the high-level language form would still be too difficult to understand.

understand.[8] But such functions are more mutually independent and thus more amenable to winnowing. Recall that the zeros and ones in winnowing are by themselves perfectly easy to understand. Is there light ahead?

Finally we need to consider steganography. This is the study of hidden messages. The goal is for the adversary to be unaware of even the existence of a hidden message. Would this work for obfuscation? That is, could there be such things as hidden programs?

[8]Indeed, the intent of functional programming is to make programs easier to understand, not harder.

Chapter 9

Verifying Remote Execution

In this chapter, we discuss Remote Execution: A computing task (or portion) is sent to a remote computer, which sends back the answer. This is a slightly broader classification than our definition of mobile code where the code has to move by itself. The results here apply to any execution that is done on an untrusted host.

One reason for doing Remote Execution would be to farm out a computer job. Perhaps special hardware or software is available at the remote site; or the remote site has greater computing power; or we have little computing power (e.g. a smart card); or we want to subdivide a task among many computers. Depending on the parties' interests, there are various incentives to cheat. We consider the rather simple motivation that the remote site might simply want to shirk the job, and report back a fake or approximate answer, while collecting full payment. Or the remote site might want to lie so we will make use of its bad answer. We will assume the remote site is a least pretending to be honest. Depending on the value of the computation and its consequences, we may be satisfied with various probabilities of detecting cheating: a 50% chance of detection would prevent a well-known remote site from making a career of cheating; but we might want near-certainty of detection in other cases. The amount of loss/gain from undetected cheating, and the penalty for getting caught, will be factors in this decision.

We won't address some of the interesting metaphysical problems: Perhaps we want to mislead a remote site into thinking it had successfully lied to us, or even try to entrap a remote site by offering an opportunity to cheat. If there's payment involved for doing computation, we might need to provide evidence of cheating to a judge. This might require us to disclose information we don't want to reveal.

9.1 GIMPS

There are a fair number of cooperative distributed computations being done over the net, usually by volunteers. And there are companies trying to make a business of brokering spare compute cycles for big computations like weather and rendering of movies.

As an example, we'll briefly review one distributed computation, The Great Internet

Mersenne Prime Search (GIMPS). GIMPS is looking for prime numbers P such that $2^P - 1$ is also prime. The computation has been going on for nearly a decade, finding a new prime every eighteen months on average. There are tens of thousands of participants, and perhaps a hundred thousand computers involved. Volunteers download the software, which runs "unobtrusively", at low priority, on their computer. The software contacts a central server for an assignment, a particular prime P to test. The project is presently working on primes P in the range $20M$ - $40M$, although a few users are working on much larger numbers. An individual prime P takes a few weeks or months to test. Typically, the prime P will generate a non-prime $2^P - 1$, and the remote machine reports the failure and a small 64-bit datum from the end of the computation as a kind of checksum. About 25% of the project effort is devoted to double-checking. Each prime is P assigned a second time, and a homomorphic version of the prime test is run, which should produce the same final checksum. The checksums are compared at the server. They usually match, but about 2% of the cases don't match, and a triple-check is assigned. In the rare, happy occasion when a new Mersenne prime is located, the exponent P is kept secret for a few weeks while the GIMPS managers run two independent checks with different hardware and different software. There have been a couple of false alarms, probably caused by undetected hardware problems. (The client software runs checks during the computation, and catches most hardware errors.)

The GIMPS project shows the best case for remote distributed computation. The volunteers are mostly honest, and the hardware mostly works. There's not much in the way of valuable secrets, and not much incentive to cheat. Each test is independent, so there's no grand final total to be corrupted if any input is wrong. Any computation can be rechecked. Errors are low consequence, at worst embarassing. As machines get faster, past computations are repeated independently, so any error will eventually be detected.

In this best case, enough assurance is provided by simply running each prime twice, doing the double-checking process with the 64-bit check value. Most of the project computation goes into testing primes P and reporting "$2^P - 1$ is not prime". The high-value results, the occasional "$2^P - 1$ *is* prime" answers, are double and triple checked on independent, different, hardware and software. But the low-value chaff primes are usually double-checked on the same hardware (most machines are late model Intel Pentiums), with the same software (GIMPS is the best program around, and others are less efficient). Double-checking is one or two years behind primary testing, since most volunteers prefer to work on the frontier of testing fresh primes.

Let's think about cheating.

1. The most obvious cheat is to fake a report of a new Mersenne prime. This will cause great excitement for a few days, until the double and triple checks are completed. When the software nears the end of testing a particular prime P, it saves some near-the-end state information. When P is lucky, the state information is returned to the server. This lets the managers run a pre-double-check by repeating the last snippet of the checking computation, providing a quick forecast of how the full double and triple checks will come out.

2. The next most obvious cheat is to reserve primes from the server and never report an answer back. This is handled by timing out reserved primes and reassigning them.

(Any volunteer project has lots of dropouts.) There are the expected problems with slow machines taking too long, and needing extensions of the timeouts.

3. An opponent could "gum up the works" by reserving a block of primes and reporting false answers, claiming to have completed tests that weren't actually run. This would go undetected for two years, until the double-checks began rolling in. Presumably the project managers would try to locate and discard results contributed by the offender, and schedule the suspect primes for priority retesting. If the opponent were clever enough to lie about the identity of the machines contributing bad results, it could be difficult to isolate his bad data. The project would be forced to allocate most of its resources to double checking past reports, and ultimately to keep double-checking current. This would significantly slow the search.
4. An opponent could trap a report of a new Mersenne prime. He might simply pretend to have dropped out, or might actually send in a fake checksum. In the latter case, he'd delay rediscovery of the new prime for two years, until the double checkers caught up. He could conceal the prime for many years by getting the double-check assigned to himself or a cohort, and reporting the same fake checksum. This would go undetected until a GIMPS successor came along and repeated the old computations. (People with new software always recheck old results, both to check the new software and to check the old results. Often some minor errors are found in the old stuff.) Of course, the opponent would have to be very lucky to have picked a prime P that generates a new Mersenne Prime, a one-in-a-million chance. Or he might have superior computing resources and have already found such a prime, and contrived to have the GIMPS server assign it to him. Or he might have a new software or hardware improvement to search more effectively, or new mathematics that suggested which primes are more likely to be winners.

The GIMPS project is in a relatively benign environment: No obvious opponents, and no particular incentives to cheat. In such an environment, the most efficient safegaurd is simply to repeat each computation, and trust to luck that the errors that do occur will be independent. Doing homomorphic repeats, instead of exact repeats, helps. The cost of this approach is "only" a factor of 2 in repeated computations.

9.2 Distributed Search, with Opponents

Now consider a hypothetical distributed search. The goal is to discover a secret 64-bit encryption key, that maps a particular known plaintext value to a particular known ciphertext value. Volunteers will be assigned (or reserve) portions of the key space to search, and report back with a checksum of all the encryptions in the range. In this scenario, an opponent who knew the key and wanted to thwart the search could get the magic key range assigned to his own machine, and report a fake checksum. If the project has double-checking, he would need to get the double-check assignment too. If the project managers can't be sure that their volunteers are independent (and identities are hard to check over the internet), they could have a problem. One safegaurd would be to have the server assign key ranges in a random order, making it hard for the opponent to get the magic range assigned to himself. This would still be vulnerable if there were many opponents: If half of the "volunteers" are actually zombies from a well-organized opposition, they could still get the magic

range assigned by luck. A more advanced counter measure is to make the key searching random. Instead of assigning key ranges, each volunteer independently generates random keys and tries them. This approach is unaffected by opponents and fake volunteers, since they are equivalent to dead machines. The drawback is that the key space is multiply covered. There's no way to know that every key has been tried, and reports of completed effort can't be trusted since they might be from zombies. Double checking is pointless; the effort may as well go to extra primary searching. The degree of multiple coverage isn't too bad: A plain search would expect, on average, to find the key after covering .5 of the key space. A random search, with random repetitions, will expect to find the key after $log_2 = .693$ trials of the key space, a 39% penalty. This argues that, even in a benign environment, double checking is a waste of time. (This conclusion depends on whether, in the event of an unsucccessful search, we need a firm believable report of failure. Things also change slightly if there might be several keys, and we must find all of them.)

9.3 Modular Exponentiation

We consider the problem of remotely computing modular exponentiation. In the simplest case, we want to package up a problem $W^X \pmod Y$, and ship it to a remote machine, and the machine will report back the answer. We presume we are computationally challenged, and find the problem too hard to tackle directly. (Notation: every exponentiation in this section is mod Y, but sometimes we'll leave off saying mod Y to keep the formulas simpler.] Our problem is how to accomplish this result when we don't especially trust the remote machine. Our approach will be to send out several problems, knowing that the answers must be interrelated. We hope that we can use the relationships to detect any cheating by a dishonest remote machine. The idea is that the interrelationships between the answers are secret, known only to us, and the remote opponent can't make a goal directed change in the answers while preserving the checking relations. Of course it could send back garbage, or simply refuse to reply. This isn't very helpful to us, but we'll have no confusion about the validity of any answers.

Most of our results also apply to the elliptic curve case, where the problem is to compute $K * P$, where K is an integer and P is a point on a non-secret elliptic curve. Any important differences will be noted. In most of our modular exponentiation examples, we assume that the modulus Y is prime, or that its primality doesn't matter. Some additional opportunities arise if Y is a composite, and we know the factors (and hence $\phi(Y)$, the order of the multiplicative group) while the opponent doesn't. However, in this case we may need to protect the value of $\phi(Y)$.

9.3.1 Exponentiation with Multiple Remote Machines

We first dispose of a simple example. If we have multiple independent machines we can send our problems to, then double-checking is the cheapest approach. If we wish to conceal the value of W, we can split the problem in two with the technique of Blinding. We generate a random number R in the range $[2, Y-2]$. If Y is composite, we also require $GCD(R, Y) = 1$. We compute $V = W/R \pmod Y$, and ask for independent remote machines to compute

$R^X \pmod Y$ and $V^X \pmod Y$. We compute the modular product $(R^X) * (V^X) \pmod Y$, which is our answer, since $W = R * V$. [We assume we have enough computing power to do some modular arithmetic, but not very much. The modular reciprocal required for computing W/R could be a nuisance, but there are mitigating methods. When Y is prime, we can compute $V' = RW \pmod Y$ instead of V, and ask for remotes to compute V'^X and R^{Y-1-X}. Then $W^X = V'^X * R^{Y-1-X}$.]

If we wish to conceal the exponent X, we can generate a random S in the range $[1, S-1]$ and issue the two problems W^S and W^{X-S}. The answers are then multiplied to obtain W^X. If Y is prime, we can instead choose S from the range $[1, Y-2]$, and replace the exponent $X - S$ with $X - S \pmod Y$. This has the virtue of concealing even the rough magnitude of X.

If we are willing to use a multi-round scheme, where we get back some exponentials and then send out more problems, there's another technique to conceal X: We choose random T in $[2, Y-3]$ and relatively prime to $Y-1$, and compute $T' = X/T \pmod{Y-1}$. Then we ask for remote computation of $W^T \pmod Y$ and then, in a second phase, $(W^T)^{T'} \pmod Y$.

We can conceal both W and X by first choosing R and then $S1$ and $S2$, or first choosing S and then $R1$ and $R2$.

We need to impose a "no subcontracting" condition on our remote machines, to prevent an opponent from collecting our problems and deducing the original W and/or X. A no-sub condition can be hard to enforce, and even hard to check. There was an analogous real-world incident in the late 1980s. A phone company substation near Chicago burned down, taking out substantial cross-country bandwidth. Some companies had provided for bandwidth disruption by purchasing backup bandwidth from independent vendors. Sadly, because of subcontracting, the independent bandwidth was actually traveling on the same wires as their primary bandwidth, and the independent backup was worthless.

We can achieve modest confidence in the confidentiality of W, X if we subdivide the problem in many pieces for many remote machines. This sets the opponent a harder data collection problem. He doesn't need to collect all the data, since he may be able to infer one of W or X from repeated values in the problems. We can make his inference problem harder by issuing some dummy problems. We must wait for all the answers to come back, even though we are planning to ignore some of them, to avoid giving an eavesdropper hints about which problems are the dummies.

Each of these steps means additional modular multiplies when we carry out the reconstruction phase. We need to take care that we're actually saving work. We've also turned one problem into many, but we assume the remotes are more capable, so this is less of a consideration.

We've ignored the problem of concealing the modulus Y. A modest amount of obfuscation can be achieved by multiplying Y by a large prime Z, or several large primes $Z' * Z'' * Z'''$ etc. The base is randomized by adding a random multiple of Y, with the multiplier being uniform in $[1, Z]$ (or $[1, Z'Z''...]$). The exponent may be randomized by adding a random multiple of $Y - 1$, ($\phi(Y)$ for composite Y), with the multiplier uniform in $[1, Z-1]$ (or $[1, (Z'-1)(Z''-1)...]$). The modulus Y must always be concealed in the same multiple (such as YZ). If two different multiples of Y are used, such as YZ and YZ',

the opponent can usually expose the number Y by taking the GCD of the two concealing moduli: $GCD(YZ, YZ') = Y * GCD(Z, Z')$, and usually $GCD(Z, Z') = 1$. For us to remove concealment, we simply take any residue computed mod YZ and reduce it mod Y.

9.3.2 Exponentiation with Just One Remote Machine

Now we consider a more complicated case, where all of our remote exponentials must be computed on one remote machine. We assume the remote machine is pretending to be helpful, but might be malicious. We want to limit the possible damage, by detecting if the remote lies about some of its results, and possibly by concealing the base and/or exponent we actually need.

9.3.2.1 Using Exponent Consistency Checks

We wish to compute $W^X \pmod{Y}$. We assume W and Y are not secret, and X is optionally to be kept secret. We give an example with 16 generated problems; for 64-bit cryptographic strength, the example can be generalized to use 128 generated exponents. We use the Knapsack Problem to make it difficult for the remote to cheat undetected.

Select 16 exponents $a, b, ..., Q$.

abcd
efgH
ijkL
mNPQ

Lower case letters are first selected uniform randomly in $[1, Y-2]$. Upper case letters are then calculated to make various sums $= 1 \pmod{Y-1}$. If X is not to be concealed, simply set $a = X$. If X is to be concealed, choose 8 of the lower case letters at random and require that they sum to $X \pmod{Y-1}$. (This makes the 8th letter a calculated value.) The letters in each of the following patterns must add up to $1 \pmod{Y-1}$.

abcd
efgH
....
.... determines H.

abcd
....
ijkL
.... determines L.

ab..
ef..
ij..
mN.. determines N.

a.c.

e.g.
i.k.
m.P. determines P.

Q is determined by requiring all sixteen letters to sum to 1 $\pmod{Y-1}$. Now we randomly permute the sixteen values (but remember the permutation). The sixteen problems $W^a, W^b, ...$ are then shipped off to the remote in the permuted order, so the remote doesn't know which exponent is a, which is b, etc. The remote replies with sixteen answers. We undo the permutation, and check the five products corresponding to the five patterns that determine H, L, N, P, Q.

$$W^a * W^b * W^c ... W^H = W^{a+b+c+...+H} = W^{1 \pmod{Y-1}} = W \pmod{Y}.$$

and so on for the other patterns. If all five checks are passed, we assume that the remote hasn't cheated. Depending on whether $X = a$, or $X =$ sum of 8 values $\pmod{Y-1}$, we either take W^a as the value for W^X, or multiply together the appropriate 8 powers of W.

For the remote to cheat, he must be able to modify some or all of the sixteen exponentials $W^a, ...$, while preserving the five product relationships. Since he doesn't know which of the exponents he receives is a, which is b, and so on, he must try to recover this information by solving a Knapsack Problem: He must locate an eight-element subset of the sixteen exponents that adds up to 1 $\pmod{Y-1}$. (He must actually locate 4 orthogonal subsets.) In this small toy example, the Knapsack Problem isn't very hard, and the remote could solve it. But a 128-element version of the problem takes effort roughly 2^{64} to solve, making it a difficult problem. If the remote is able to solve the Knapsack Problem, this allows him to fudge the values of the exponential he returns while still passing the verification checkproducts. In the open $X = a$ case, he could now fudge $W^a = W^X$ arbitrarily. But in the concealed X case, he doesn't know which 8 values are multiplied to calculate W^X. In this case his best strategy is to still fudge some single value, say W^a, and hope it's included in the product for W^X.

We can strengthen this scheme by using a few small numbers as the sums that determine the upper case letters (instead of making everything add up to 1 $\pmod{Y-1}$.) We might use Fibonaccis, or powers of 2, or simply small random numbers. These will give easy-to-compute small powers of W for the final checkproducts. But if the remote doesn't know the small numbers for the exponent sums, his knapsack problems are harder – he doen't know his target sums, only approximate values.

The total checking work for the 128-exponent case is substantial: There will be 8 checks, each requiring 63 modular multiplies, and possibly a ninth long product to calculate W^X. This is 500 modular multiplies. If we are using 1024 bit exponents (and presumably W, X, Y are all about this size), then a straight calculation of W^X would only require about 1200 multiplications. So the savings isn't dramatic. The suggestions below may reduce the problem size and amount of checking work somewhat.

If we need to compute several exponentials at a time, we can embed all of them into one grid of exponential problems. This amortizes the checking work.

Some areas for further research: Explore using small coefficients for the lower case letters in the sums that make up the capital letters: The first constraint changes from

$a+b+c+d+e+f+g+H = 1 \pmod{Y-1}$ to $2a+3b+c+d+3e+f+2g+H = 1 \pmod{Y-1}$. This makes life even harder for the remote, since he doesn't know the coefficients, and he must examine a wider range of knapsack problems. We can also include negative coefficients cheaply: We avoid the need for modular reciprocals by reading the constraint as $2a+c+d+f+2g+H = 1+3b+3e \pmod{Y-1}$, and doing the multiplications for both sides in the check. In the elliptic curve case, point negation, doubling, and halving are very cheap operations. If we are using Koblitz curves, then the space of easily computed coefficients expands enormously: any point can be multiplied by a power of $\tau = (-1+i\sqrt{7})/2$ simply by rotating the coordinates' bit patterns. This expands the search space that the remote must check enormously, and allows us to use smaller Knapsacks (fewer variables, less work for the checkproducts) with an acceptable security level.

9.3.2.2 Using Base Consistency Checks

This scheme assumes that W is concealed but X is not. We must know one modular exponential such as $2^X \pmod{Y}$ ahead of time. It's best if 2 is a primitive root of Y. We choose 10 variables $A, B, ..., J$. We set $A = W, B = 2, C = D = E = 1$, and choose F, G, H, I, J randomly in $[1, Y-1]$. A Simple Random Fibonacci Step (SRFS) is: multiply A by a randomly selected choice of $B, ..., J$. (This is a modular multiplication, mod Y.) Then the variables are rotated one step, $A \leftarrow B \leftarrow C \leftarrow ...I \leftarrow J \leftarrow A$. After 100 SRF steps, the variables are suitably scrambled. We permute them randomly, and ask the remote to calculate $A^X, B^X, ..., J^X$ (permuted). When the answers are returned, we undo the permutation. We calculate the 10 reciprocals $W^{-A}, W^{-B}, ..., W^{-J}$. (Use Montgomery's trick to only calculate one actual reciprocal.) Then we are prepared to undo our sequence of Simple Random Fibonacci Steps, naturally in reverse order. Undoing steps requires using the reciprocals to undo the modular multiplications. And we must also keep the reciprocals updated with appropriate multiplications, so they are available when needed to undo further steps. We need about 100 multiplications in the forward direction, and about 200 in the reverse direction. At the end, if the remote has not cheated, we have W^X in the A position, 2^X in the B position, and $1^X = 1$ in the C, D, E positions. If the remote cheats randomly, at least one of the C, D, E positions (probably all three) will be worng. If the remote cheats by using a different X, then the 2^X value in the B position will be wrong. For the remote to cheat in a directed fashion, he must uncover the latent product relations in the set of bases he is presented with. If we run the SRFS process with symbolic values (just keeping the letters $A...J$ to represent the initial values) and calculate the expressions after 100 steps, the coefficients are roughly 25-bit numbers. The remote must somehow guess these values to find the hidden relationships and make an undetectable change to the exponentials that he is supposed to calculate. The coefficient growth rate can be increased in several ways, reducing the number of SRF steps required to randomize the problem. We can use small random coefficients in the Fibonacci process. We can do a more complex step: select 1/3 of the variables randomly, multiply them together perhaps with small powers, multiply the product into another 1/3 of the variables selected at random. If we are using elliptic curves, we have $-1, 2$, and $\pm 1/2$ available as cheap coefficients. If we are using Koblitz curves, we also have powers of τ available. This is a very attractive option, since a lot of randomization is introduced in every Random Fibonacci Step.

Some areas for further research: Try making the bases and exponents jointly variable. If we are willing to use a sequential protocol for the exponentials, where we release the problems one at a time and the remote must answer one problem at a time, we can get an even harder-to-defeat system. The remote doesn't know enough information to cheat safely while he's computing the early exponentials.

9.3.3 Checking a Generic Computation

Next we explore the idea of checking a generic computation made on a remote machine. This is possible in theory, as outlined below. We have made some progress in the direction of making it practical, but there's still a long way to go.

Any mathematical proof can be formalized – converted into a (long) character string, with pieces consisting of axioms, variable substitutions, deductions, etc. The rules for verifying that each statement follows from the preceding statements can be boiled down to verifying that various substrings are equal, or that one string is a substitution of another. Such a proof can be mechanically checked by a computer program. The QED project was a higher level version of this idea, trying to get a good piece of mathematics into machine verified form, with the eventual goal that anyone offering a theorem would pre-check it in the mechanical verifier. They had worked from the basic axioms of set theory up to verifying the prime number theorem, but haven't been heard from in a few years.

There's not a lot of interest in proof checking, but a few people are working on it. Most of the work is converting intuitive proofs into machine syntax, and expanding implicit things that everyone knows, or "it's obvious" into the intervening lemmas.

The idea of Probabilistically Checkable Proofs is to take an existing machine checkable proof, at the level of "character 37625 equals character 74289" and expand it with error-correcting codes, so that any error in the original proof is magnified enough to pervade the entire proof. A good result is that if the original proof fails any check, the new proof will likely show a problem from inspecting only three bits. I.e., a verifier that's checking the expanded proof can examine three bits of the proof, and if the proof is bad the verifier will reject it with decent probability - maybe 25%. If the proof is good, the verifier typically will say ok. The verifier can repeat with another test of three bits (presumably at least one of them will be in a different place in the proof) and get another independent check. So a bad proof will be detected with near certainty after examining a few hundred bits of the proof – a bounded amount of inspected data, even though the original proof had no particular size bound.

The catch is the amount of expansion: polynomial in the size of the original proof. This makes it theoretically cute, but of no practical value.

We've been trying to beat on this idea to make it useful for our problem: our remote machine would supply a proof that it had executed our program, by providing an execution trace: each instruction in the program, $A = B + C$, or $ADD\ \%AH, 27$, would produce a line in the proof that the current values of B and C are xxx, or that location 27 contains xxx, and register $\%AH$ contains xxx, and the appropriate list of bit values to prove the addition. ($Abit0 = Bbit0 \otimes Cbit0, Carrybit0 = Bbit0$ and $Cbit0, \&c$). The remote machine

would produce the expanded proof and send it to us, and we'd inspect the necessary few hundred bits to verify the correct execution. Somehow we deal with the impractical size of the proof. And we somehow arrange to only transmit the bits we actually plan to inspect, but without giving the remote machine a chance to fudge them. We have partially solved these problems.

The remote machine must still calculate the polynomially expanded proof. [This alone makes the scheme impractical.] However, instead of transmitting the proof for us to use, he calculates a tree-hash of the proof (explained below), and sends us that tree-hash, along with the size of the expanded proof. Our verifier decides which triples of bits it wishes to inspect, and sends queries to the remote machine requesting the values of the particular bits. The remote machine responds with the bit values, along with the hash-tree proofs linking each bit into the master tree-hash value. If the verifier likes the triple of bits, and the tree-hash proofs are ok, that bit triple is accepted. The verifier moves on to the next check. After a hundred good checks, we're done. The proof, and hence the computation, is accepted. Of course this only verifies the computational aspect of the remote process. If the remote machine is supposed to read a temperature input as part of the computation, we have no way to check up on it.

(A tree-hash is a binary tree of data. Each leaf is hashed. Each parent node is the hash of its two children's hashes. The root hash is the value for the tree. To prove that any particular value is in the tree at a particular spot, the prover provides the location, value, and the hashes of all the brother-of-ancestor nodes along the path to the root. The verifier checks that hashing the value, and then hashing that with each of the uncle nodes along the path-to-root, produces the known full-tree hash value. The prover must keep track of the whole tree and all the intermediate hashes, but it's only a moderate amount of work per leaf-change. This provides a way to distribute an authenticated phone book, where the tree-hash is broadcast, and individual subscribers are provided with their list of intermediate hashes for their path-to-root; they can prove their phone number to a third party.)

We've made a little more progress to reduce the impracticality. As part of the proof of execution, each memory reference (such as to location 27 above) must include an implicit assertion that the contents of location 27 haven't changed since the last store into 27, which was at time-step 45623 of the proof. Checking that time-step 45623 stored the alleged value into location 27 isn't too hard, we just look back in the proof, but checking that nothing has modified location 27 between then and now is hard: the direct approach requires examining all intervening instructions to make sure thay didn't modify location 27.

The new idea is that each instruction which references or modifies memory will be accompanied (in the execution proof) with a tree-hash of all of memory, with the tree expanded to show, for example, location 27's value before (and after, if there's a modification), and any updated value for the total memory tree-hash (if location 27 is changed). The amount of checking required to verify the hashes is still pretty awful, but it's now only proportional to the tree depth, which is log(memory size).

The ultimate goal is to reduce the amount of proof-size expansion to not-much-worse-than linear. Perhaps we could trade off additional checking work and a more complicated checking protocol, to make a smaller proof. This is still an open problem.

References

[1] Mikhail J. Atallah, K. N. Pantazopoulos, John R. Rice, Eugene E. Spafford, "Secure Outsourcing of Scientific Computations." Advances in Computers, Vol. 54, pp. 215-272. 2001.

[2] David Aucsmith, "Tamper Resistant Software: An Implementation." Lecture Notes in Computer Science. Vol. 1174. Springer-Verlag, Berlin. pp. 316-333. 1996.

[3] "Avalon Bus Specification: Reference Manual, ver. 2.3," www.altera.com, July 2003.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, "On the (Im)possibility of Obfuscating Programs". In J. Kilian, editor, Advances in Cryptology-CRYPTO '01, Lecture Notes in Computer Science. Springer-Verlag, 2001, August 2001.

[5] D. Baker, "Fortresses Build Upon Sand."" Proceeding of the New Security Paradigms Workshop, 1996.

[6] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi, "Cryptanalysis of a White Box AES Implementation", in Proceedings of Selected Areas in Cryptography 2004, pp. 251-264, 2004.

[7] Philip L. Campbell, "An Introduction to Software Obfuscation." SAND2004-2198. Printed June 2004.

[8] Philip L. Campbell, et. al, "Principles of Faithful Execution in the Implementation of Trusted Objects," SAND2003-2328, Sandia National Laboratories, Albuquerque, NM, 2003.

[9] David Chaum and Hans van Antwerpen, "Undeniable Signatures", Proceedings of Crypto - Crypto '89, LNCS 435, pp. 212-216, 1989.

[10] Stanley Chow, Phil Eisen, Harold Johnson, and Paul van Oorschot, "A white-box cryptography and an AES implementation", in Proceedings of SAC 2002 - 9th Annual Workshop on Selected Areas in Cryptography, volume 2595 of LNCS, pages 250–270. Springer, 2002.

[11] Stanley Chow, Phil Eisen, Harold Johnson, and Paul van Oorschot "A white-box DES implementation for DRM applications." in Proceedings of DRM 2002 - 2nd ACM Workshop on Digital Rights Management, 2002.

[12] Christian Collberg, Clark Thomborson, Douglas Low, "A Taxonomy of Obfuscating Transformations." Technical Report 148, Department of Computer Science, University of Auckland. 1997.

[13] http://www.dvdcca.org/css.

[14] Joan Daemen and Vincent Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard", Springer, 2002.

[15] Phil Eisen and Paul C. van Oorschot, "Dynamic-key white-box cryptography", preprint, 2003.

[16] Joan Feigenbaum, "Encrypting Problem Instnaces Or..., Can you Take Advantage of Someone Without Having to Trust Him?." Proceedings of Crypto – '84, LNCS 218, pp. 477-488, 1986.

[17] "Data encryption standard", Number 46-2 in Federal Information Processing Standards Publications. U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1993.

[18] Flask Home Page: http://www.cs.utah.edu/flux/fluke/html/flask.html.

[19] O. Goldreich, S. Goldwasser, S. Micali, "How To Construct Random Functions". J. of the ACM, Vol. 33, 792-807, 1986.

[20] Robert S. Gray, David Kotz, George Cybenko, Daniela Rus, "D'Agents: Security in a Multiple-Language, Mobile-Agent System." Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 154-87, 1998.

[21] S. Hada, "Zero-knowledge and Code Obfuscation". Advances in Cryptology-ASIACRYPT '00, Lecture Notes in Computer Science. Springer-Verlag, 443-457, 2000.

[22] Fritz Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts." Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 92-114, 1998.

[23] Matthias Jacob, Dan Boneh, and Edward Felten, "Attacking an obfuscated cipher by injecting faults", in ACM CCS-9 Workshop DRM, 2002.

[24] Cees J. A. Jansen, "Investigations on Nonlinear Stream Cipher Systems: Construction and Evaluation Methods," Philips usfa B.V., the Netherlands, 1989.

[25] Hugo Krawczyk and Tal Rabin, "Chameleon Hashing and Signatures", in Proceedings of NDSS 2000, pp. 143-154, 2000.

[26] Lindholm and Yellin, "The $Java^{TM}$ Virtual Machine Specification, Second Edition," java.sun.com.

[27] Linux Security Modules homepage. http://lsm.immunix.org/lsm_modules.html.

[28] Peter Loscocco, Stephen Smalley, "Meeting Critical Security Objectives with Security-enhanced Linux." http:// www.nsa.gov/selinux.

[29] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, John F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." Proceedings of the 21st National Information Systems Security Conference, pages 303-314, October 1998.

[30] Sergio Loureiro, "Mobile Code Protection." Ecole Nationale Superieure des Telecommunications. Paris, France. January 26, 2001.

[31] B. Lynn, M. Prabhakaran, A. Sahai, "Positive Results and Techniques for Obfuscation". In C. Cachin, J. Camenisch, editors, Advances in Cryptology-EUROCRYPT '04, Lecture Notes in Computer Science. Springer-Verlag, 2004, May 2004.

[32] Bruce J. MacLennan, Functional Programming: Practice and Theory. Addison-Wesley, New York. 1990.

[33] Spencer E. Minear, "Providing Policy Control Over Object Operations in a Mach Based System." Proceedings of the Fifth USENIX Unix Security Symposium. June 5-7, 1995. Salt Lake City, UT. pp. 141-55. (see also http:// www.securecomputing.com/randt/HTML/technical-docs.html)

[34] Sau-Koon Ng, "Protecting Mobile Agents Against Malicious Hosts," Masters Thesis. Division of Information Engineering. The Chinese University of Hong Kong. June 2000.

[35] "NIOS 3.0 CPU Data Sheet, ver. 2.1," www.altera.com, March 2003.

[36] Mark K. Reiter, Aviel D. Rubin, "Crowds: Anonmymity for Web Transactions." ACM Transactions on Information and System Security, June 1999.

[37] James Riordan, Bruce Schneier, "Environmental Key Generation Towards Clueless Agents," Mobile Agents and Security. G. Vigna, editor. Lecture Notes in Computer Science, Volume 1419, pp. 15-24, 1998.

[38] Ronald L. Rivest, "Chaffing and Winnowing: Confidentiality without Encryption." MIT Lab for Computer Science. March 18, 1998. (http://theory.lcs.mit.edu/ rivest/chaffing.txt).

[39] Ronald L. Rivest, Leonard Adleman, and M. Dertouzos, "On data banks and privacy homomorphisms." In Demilllo, Dobkin, Jones, and Lipton, editors, Foundations of Secure Computation, pp. 169-80. New York: Academic Press, 1978.

[40] Tomas Sander, Christian F. Tschudin, "Towards Mobile Cryptography." 1998 IEEE Symposium on Security and Privacy. pp. 215-24, May 3-6, 1998.

[41] Tomas Sander, Christian F. Tschudin, "On software protection via function hiding." Lecture Notes in Computer Science, V1525, p.111-123, 1998.

[42] Tomas Sander, Christian F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", Mobile Agents and Security, LNCS 1419, p. 44-60, 1998.

[43] Security-Enhanced Linux Project Page: http://www.nsa.gov/selinux/.

[44] Adi Shamir, "On the security of DES", in Hugh C. Williams, editor, *CRYPTO*, volume 218 of LNCS, pages 280–281. Springer, 1985.

[45] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies." Proceedings of The Eighth USENIX Security Symposium, pages 123-139, August 1999.

[46] Thomas D. Tarman, et. al. , "On the Use of Trusted Objects to Enforce Isolation Between Processes and Data," in Proceedings, 2002 International Carnahan Conference on Security Technology, IEEE, Piscataway, NJ, pp.115-119, 2002.

[47] N. Varnovsky, V. Zakharov, "On the Possibility of Provably Secure Obfuscating Programs". In M. Broy, A. Zamulin, editors, Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, Lecture Notes in Computer Science. Springer-Verlag, 2003, July 2003.

[48] Giovanni Vigna, "Protecting Mobile Agents Through Tracing." 3rd ECOOP Workshop on Mobile Object Systems, Jyvälskylä, Finland, June 1997.

[49] Chenxi Wang, "A Security Architecture for Survivability Mechanisms," Ph.D. Dissertation, University of Virginia, Department of Computer Science, October 2000.

[50] Robert N. M. Watson, "TrustedBSD: Adding Trusted Operating System Features to FreeBSD".

[51] Gregory Wroblewski, "General Method of Program Code Obfuscation." (draft) Ph.D Dissertation. Wroclaw 2002.

[52] A. C. Yao, "Protocols for secure computations." IEEE Symposium on Foundations of Computer Science 82, pp. 160-4, Chicago, Illinois, 1982.

DISTRIBUTION:

2 MS 0785
W. Anderson, 5514

3 MS 0785
C. Beaver, 5514

2 MS 0785
W. Neumann, 5514

2 MS 0785
R. Schroeppel, 5514

2 MS 0785
P. Campbell, 5516

2 MS 0455
H. Link, 5517

2 MS 0806
L. Pierson, 9336

1 MS 0785
T. McDonald, 5514

1 MS 0785
R. Hutchinson, 5516

1 MS 0455
R. Tamashiro, 5517

1 MS 0806
L. Stans, 9336

1 MS 0785
R. E. Trellue, 5501

1 MS 0451
S. G. Varnado, 5500

1 MS 9018
Central Technical Files, 8945-1

2 MS 0899
Technical Library, 9616

2 MS 0612
Review & Approval Desk, 9612 For DOE/OSTI

1 MS 0123
LDRD Program Office, Attn: Donna Chavez