KEY RECOVERY FROM DECAYED MEMORY IMAGES AND OBFUSCATION OF CRYPTOGRAPHIC ALGORITHMS

ROGER ZAHNO

A THESIS

IN

CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

JULY 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Roger Zahno**

Entitled: **Key Recovery From Decayed Memory Images and Obfuscation of Cryptographic Algorithms**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Information Systems Security

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Simon Li (CIISE) _____ Chair

Dr. Walaa Hamuda (ECE) _____ Examiner

Dr. Benjamin Fung (CIISE) _____ Examiner

_____ Examiner

Dr. Amr Youssef (CIISE) _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin Drew, Dean

Faculty of Engineering and Computer Science

ABSTRACT

Key Recovery From Decayed Memory Images and Obfuscation of Cryptographic Algorithms

Roger Zahno

A cold boot attack is a type of side channel attacks which exploit the data remanence property of Random Access Memory (RAM) to retrieve contents that remain readable for a short time after power is disconnected. Specialized algorithms have been proposed to recover cryptographic keys from decayed memory images. However, these techniques were cipher-dependent and certainly uneasy to develop and to fine tune. On the other hand, for symmetric ciphers, the relations that have to be satisfied between sub-round key bits in the key schedule always correspond to a set of nonlinear Boolean equations.

In the first part of this thesis, we investigate the use of an off-the-shelf SAT solver (CryptoMiniSat), and an open source Gröbner basis tool (PolyBoRi) to solve the system of Boolean equations in the algebraic step of the cold boot attack. We also compare the pros and cons of both approaches and present simulation results for the extraction of AES and Serpent keys from decayed memory images using these tools.

Because of its simplicity, ease of implementation, and speed, RC4 has become one of the most widely used software oriented stream ciphers. It is used in several popular

protocols such as SSL and it is integrated into many applications and software such as Microsoft Windows, Lotus Notes, Oracle Secure SQL and Skype.

In the second part of this thesis, we present an obfuscated implementation of RC4. In addition to investigating different practical obfuscation techniques that are suitable for the cipher structure, we also compare the performance of these different techniques. Our implementation provides a high degree of robustness against attacks from execution environments, where the adversary has access to the software implementation, such as in the case of digital right management applications.

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor Dr. Amr Youssef. Without his support and encouragement, this thesis would not be possible.

I would also like to express special thanks to my parents and my sister, back in Switzerland, who showed deep support and understanding regarding my decision to pursue graduate studies in Canada. To all members of my family: I am very grateful and I highly appreciate the support that I had the privilege to receive.

Finally, I would like to express my thanks to my colleagues in the *Cryptography and Data Security Laboratory* at CIISE: You made my journey through the master program an interesting, enjoyable and unforgettable one.

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard

CRT Chinese Remainder Theorem

DDR SDRAM Double Data Rate Synchronous Dynamic Random Access Memory

DES Data Encryption Standard

DRM Digital Rights Management

RAM Random Access Memory

SDRAM Synchronous Dynamic Random Access Memory

Chapter 1

Introduction

In today's computer systems, memory resident data and swap areas contain important information about current running processes as well as terminated ones. Existing reverse engineering tools and techniques can be used to search through the Random Access Memories (RAMs) and swap areas for recoverable secret information such as cryptographic keys which are left unprotected and directly accessible. In this thesis we investigate the recovery of the AES [29] and Serpent [13] keys from decayed memory images and present a possible approach to protect such sensitive data for the case of the RC4 cipher.

Cold boot attacks [33] [34] exploit the data remanence property of RAM to retrieve its contents which remain readable for a short time after power has been disconnected. At run time, cryptographic systems such as disk encryption utilities (e.g., Truecrypt [9]) keep the key information for encryption and decryption in the memory. Retrieving this information breaks the cryptographic system without directly attacking the underlying cryptographic primitives in an intensive way. In the first part of this thesis, we present a systematic approach to perform the algebraic step of the cold boot attack against AES and Serpent.

Different approaches that address the cold boot attack were presented in the literatures. For example, *Tresor* [8] is a kernel patch for Linux based operating systems which loads and manipulates key related data directly in the microprocessor and its registers. However,

in this work, we are interested in a more general approach that can be adapted for other applications and ciphers. In the second part of the thesis, we present an obfuscated implementation of RC4. In addition to investigating different practical obfuscation techniques that are suitable for the cipher structure, we also perform a comparison between the performances of these different techniques. Our implementation provides a high degree of robustness against attacks from execution environments where the adversary has access to the software implementation such as in Digital Right Management (DRM) applications.

1.1 Motivation

Techniques to attack cryptographic primitives can be classified into two main categories: pure mathematical attacks and side channel attacks. Pure mathematical attacks exploit the inner structure of the cipher and rely only on known or chosen input-output pairs of the cryptographic function in order to reveal its secret information. On the other hand, in side channel attacks, the attacker is assumed to have physical access to the cryptographic device. Timing information [39] and power consumption [40] are two well known side channels which can be utilized by attackers to leak critical information related to the secret keys involved in the cryptographic operations.

The remanence effect of Random Access Memory (RAM) is another highly critical side channel which exploits the fact that traces of sensitive data remain in the computer memory, even after its power is removed. Experiments confirming this data remanence property were reported in 2002 by Skorobogatov [49]. However, Halderman *et al.* [33] were the first to practically exploit the remanence of memory modules to recover cryptographic keys where they presented a proof of concept experiment applying a cold boot attack to recover secret keys of DES, AES and RSA. In contrast to the general belief, Dynamic RAM (DRAM) retain some of its content for seconds up to several minutes after its power is disconnected even if the memory module is removed from the motherboard. Cooling the memory module

can significantly extend this time frame.

Several authors (e.g., [51] [12] [34]) further improved Halderman *et al.*'s proof of concept and presented algorithms for recovering the private keys with higher decay factors. However, techniques presented by these authors were cipher-dependent and certainly uneasy to develop and to fine tune. On the other hand, for symmetric ciphers, the relations that have to be satisfied between sub-round key bits in the key schedule always correspond to a set of nonlinear Boolean equations which lend itself naturally to well studied algebraic problems such as the SAT problem and the Gröbner basis reduction problem. In this work we investigate the use of an off-the-shelf SAT solver (CryptoMiniSat [4]), and an open source Gröbner basis tool (PolyBoRi [16] [5]) to solve the resulting system of Boolean equations. We also compare the pros and cons of both approaches and present some simulation results for the extraction of AES and Serpent keys from decayed memory images using these tools.

Software obfuscation addresses the requirements of several recently developed applications which demand a higher degree of robustness against attacks from the execution environment where adversaries have access to the system with its hardware and software implementation of key instantiated cryptographic functions. DRM is an example for such applications where one of the main objectives is to control the access to digital contents stored on different medias. White-box implementations for AES and DES [20] [19], achieve this by hiding the encryption keys within the implementation of the cipher through the use of different obfuscation techniques.

Because of its simplicity, ease of implementation and robustness, RC4 [43] has become one of the most commonly used stream ciphers. In its software form, implementations of RC4 appear in many protocols such as SSL, TLS, WEP and WPA. Furthermore, it has been integrated into many applications and software including Windows, Lotus Notes, Oracle Secure SQL, Apple AOCE, and Skype. Although the core of this two-decade old cipher is

just a few lines of code, the study of its strengths and weaknesses as well as its different software and hardware implementation options is still of a great interest to the security and research communities.

Directly applying the techniques developed for white-box implementation of block ciphers to stream ciphers does not seem to work, mainly, because normal operation of stream ciphers requires us to always maintain the inner state of the cipher. Recovering the inner state of the stream cipher usually compromises the security of the cipher even if the attacker is not able to recover the key. In the second part of the thesis, we present an obfuscated implementation of RC4.

1.2 Contributions

Our contributions can be summarized as follows:

- Application of two off-the-shelf algebraic tools for extraction of cryptographic keys from corrupted memory images: We investigate the use of an off-the shelf SAT solver (CryptoMiniSat), and an open source Gröbner basis tool (PolyBoRi) to solve the system of equations produced by the cold boot attack in order to recover the secret key. We also provide the pros and cons of both tools and present some experimental results for the extraction of AES and Serpent keys from decayed memory images.

- An obfuscated implementation of RC4: We investigate several obfuscation techniques that are applicable to array-based stream ciphers such as RC4. We also perform a comparison between the performances of these different techniques when applied to RC4. Although our proposed implementation does not provide the same level of theoretical security provided by white-box implementations for block ciphers, it still provides a high degree of robustness against attacks from execution environments where the adversary has access to the software implementation such as

in DRM applications.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Relevant background and literature are briefly reviewed in Chapter 2. Chapter 3 presents our results of applying the CryptoMiniSat and PolyBoRi for the extraction of AES and Serpent keys from decayed memory images. In Chapter 4, we present our obfuscated implementation of RC4. Finally, Chapter 5 presents our conclusions and suggestions for future work.

Chapter 2

Preliminaries

In this chapter we briefly review some of the background information required for the understanding of the following two chapters.

2.1 Cryptanalytic Attacks

Cryptanalysis [43] can be defined as the study of methods and techniques that allow unauthorized access to secrets protected by cryptographic methods (such as encryption algorithms and signature schemes) and devices. Depending on the assumptions regarding what is accessible to the cryptanalyst, cryptanalysis models can be classified into three categories: Black-box, Grey-box and White-box models.

2.1.1 Black-Box Models

Black-box models offer the least amount of information to adversaries. In particular, in the black-box model, only input, corresponding output, and the algorithm in use are assumed to be known by the attacker. No further information about the system implementation details or inner structure is made available to the adversary. In other words, the black-box model refers to the traditional cryptanalysis scenario where adversaries can only manipulate the

input, observe the resulting output, and apply their observations to the cryptographic algorithm in use. Mathematical attacks such as differential cryptanalysis and linear cryptanalysis [35] are well known attacks in this model which are applied against block and stream ciphers. In particular, differential cryptanalysis is a known plaintext attack where the adversary knows a set of plaintext (P) and ciphertext (C) pairs. A fixed ΔP with $\Delta P = P_1 \oplus P_2$ is defined (e.g., $\Delta P = 0000000011010000$ for a 16 bit block cipher) so that a specific output $\Delta C = C_1 \oplus C_2$ occurs with a high probability for the given ΔP . The distribution of $\Delta C's$ for the given ΔP results into a differential characteristic that reveals bit information about the key in use. Exhaustive search over the remaining bits of the key accomplishes the attack [35]. On the other hand, linear cryptanalysis is a known plaintext attack where the adversary knows a set of plaintext (P) and ciphertext (C) pairs. The idea is to find linear approximations (modulo 2) between some of the P and C bits which hold with probability $0.5 \pm \varepsilon$. In a fully randomized system, the bias value ε would be almost 0. The expression to be chosen has to have a high bias value in order to result into a linear characteristic that reveals bit information about the key in use. Exhaustive search over the remaining bits of the key completes the attack [35].

2.1.2 Grey-Box Models

In grey-box models, an adversary has some additional information related to the targeted cryptographic system. In particular, the input, resulting output, the algorithm in use, and parts of the system to be analyzed are known and accessible by the attacker. *Timing analysis*, *Power analysis*, and *Fault analysis* are well known side channel representatives for the grey-box model. Timing analysis and power analysis were first introduced by Paul Kocher [39] [40]. *Fault analysis* attacks introduced by Boneh *et al.* [15] provide another example of this class of attacks.

In timing attacks, the attacker attempts to compromise a cryptosystem by analyzing the

time needed to execute cryptographic algorithms. Every logical operation in a computer takes time to execute, and the time can differ based on the input. With precise measurements of the time for each operation, an attacker can work backwards to the secret information [39]. Power analysis exploits the correlation between the cryptographic circuit power consumption and the value of the processed secret information during the execution of different cryptographic operations. Power analysis attacks can be divided into *Simple Power Analysis (SPA)* and *Differential Power Analysis (DPA)*. While SPA directly interprets the measured power consumption, DPA is a more advanced form of power analysis which allows attackers to compute the intermediate values within cryptographic computations by statistically analyzing data collected from multiple cryptographic operations [40].

Fault analysis exploits the misbehavior of cryptosystems in response to carefully crafted fault injections. Probably, the best known demonstration of this attack is fault analysis of RSA cryptosystems which use the Chinese Remainder Theorem (CRT). By manipulating intermediate results, an adversary can easily find the factors of the modulus with the help of the greatest common divisor Euclidian algorithm [15] [36].

Cold boot attacks are examples of powerful side channel attack which exploit the remanence effect of memory modules. More details about this class of attacks will be described later on in this chapter.

2.1.3 White-Box Models

In contrast to both the black-box and the grey-box models, in white-box models, an adversary has full access to the system in use. In other words, in a white box model, the input, resulting output, the algorithm in use, and the whole system are known and accessible. In particular, the attacker can access and manipulate the memory and other storage devices of the system, and observe intermediate calculation results.

Because of the ability to access the whole system, in the white-box model, adversaries

have further techniques at their disposal which extend those already available in the black-box and grey-box attack models. Additionally, gaining access to the memory and CPU registers allows the adversary to observe, manipulate intermediate results of the cryptographic evaluations, and access the binary file for disassembling and debugging the target programs. This small but incomplete list of examples reveals how powerful an adversary is in a white-box model. Such powerful attacks demand new approaches to secure secrets on targeted system.

White-box implementations for AES and DES [20] [19] are designed to resist attacks launched in a white-box environment. The white-box implementation is a powerful technique based on software obfuscation. In what follows, we provide a description for the general idea of white-box implementations. White-box implementations of AES and DES ciphers are based on lookup tables following a specific scheme to creates their content. The rest of this section explains the idea behind these lookup tables and show how they are related to each other. We start with two definitions that build the foundation for this approach.

Definition 1 (Encoding) *Let X be a transformation from m bits to n bits. Choose an m-bit bijection F and an n-bit bijection G. We call* $X^{'} = G \circ X \circ F^{-1}$ *an encoded version of X. F is an input encoding and G is an output encoding [20].*

Definition 2 (Networked Encoding) *A networked encoding for computing* $Y \circ X$ *(i.e. transformation X followed by transformation Y) is an encoding of the form* $Y^{'} \circ X^{'} = (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) = H \circ (Y \circ X) \circ F^{-1}$ *[20].*

Now consider the case where $Y^{'}, X^{'}$ are lookup tables representing the different execution steps. The concatenation $Y \circ X$ represents the internal computation of $Y^{'} \circ X^{'}$, and F, G, H are hidden bijections. The lookup tables contain an encoded version of the values required for each step, i.e., iteration, of the cipher. They are built by concatenating the two

bijections for input and output encoding and storing the resulting value into the table. In networked encoding, input and output encoding are designed such that the input encoding of the following iteration eliminates the effect given by the output encoding.

Following this scheme would result in an encoded cipher with remaining input and output encoding H and F. To eliminate the effect of the resulting input and output encoding, the first and the last lookup table in the chain have to be handled differently. A straightforward solution applies only one of the two bijections to the two tables, so that in the resulting computation no input and output encoding occurs. Based on the intended purpose of the application and the infrastructure in use, different variations might be possible to eliminate the effects of the input and output encoding. For example, on the transmitter side, one might have an input that is concatenated with an output encoding related to the input encoding of the first table, and therefore compensates the effect given by the output encoding. Applying definitions 1 and 2, together with the described considerations to eliminate the effects of the input and output encoding to the 10 rounds of AES-128, we get

$$X_1^{'} \circ X_2^{'} \circ \cdots \circ X_{10}^{'} = (X_1 \circ F_1^{-1}) \circ (G_2 \circ X_2 \circ F_2^{-1}) \circ \cdots \circ (G_{10} \circ X_{10}) = X_1 \circ X_2 \circ \cdots \circ X_{10}$$

where $X_r^{'}, r = 1, \cdots, 10$, denotes the lookup table for the r^{th} round with the associated sub-round keys. Only the bijection F_1^{-1} is applied to table $X_1^{'}$ to eliminate the effect of G_1. Alternatively, G_1^{-1} can be applied to the input so that $input \circ G_1^{-1}$ presents the new input. In $X_2^{'}, ..., X_9^{'}$, both bijections G_{rd} and F_{rd}^{-1} are applied. For $X_{10}^{'}$, only bijections G_{10} is applied to avoid an obfuscated output.

Concatenating the lookup tables $X_1^{'} \circ X_2^{'} \circ \cdots \circ X_{10}^{'}$ results in the internal computation $X_1 \circ X_2 \circ \cdots \circ X_{10}$ which represent the uncoded AES-128 cipher. In case of the AES-128 cipher, the lookup tables map 128 input bits to 128 output bits. Following a naive approach, where one table for each iteration of the cipher contains the 128 $\times$ 128 bijections from plaintext to ciphertext for a given key, requires huge tables that cannot be handled by any

computer system (table size of 5.4×10^{39} bytes [20]).

The structure of AES and DES allows dividing the large sub-round tables into a subset of tables. In case of AES, four different table types (see Figure 1) are needed to break down this huge table into roughly 3,100 smaller lookup tables with a total cumulated size of 770,048 bytes. Further details on how these tables are constructed can be found in [20].

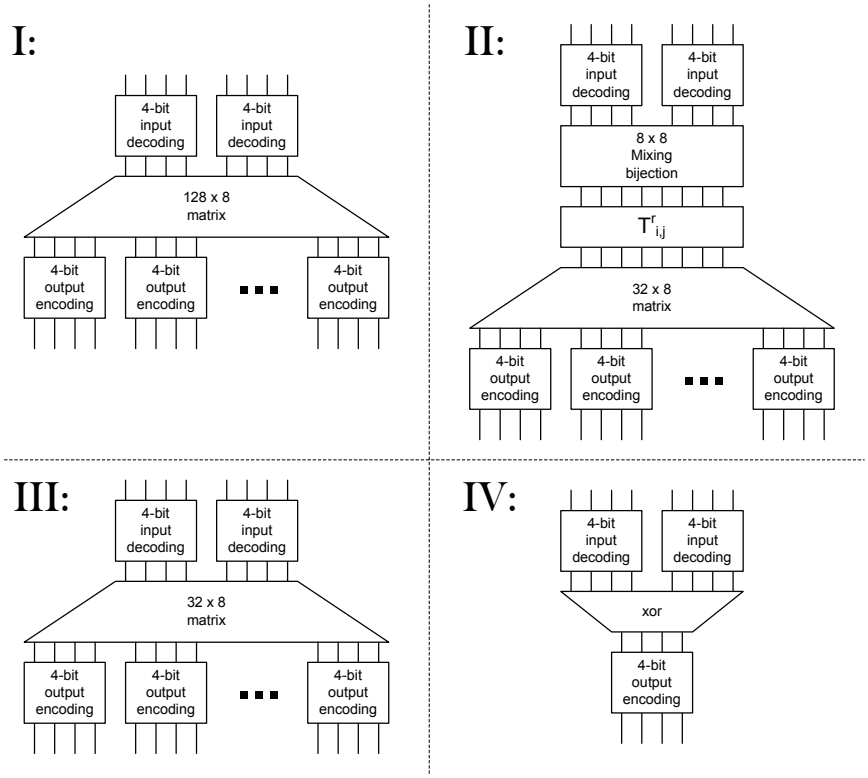

Figure 1: Types of tables for an AES white-box implementation [20]

2.2 Cold Boot Attacks

A cold boot attack [33] is a side channel attack that exploits the fact that data loss of a non-powered random access memory can be delayed by cooling it down. In 2002, Skorobogatov [49] performed some experiments to study the temperature dependency of data retention time in static RAM devices. The reported experimental results indicated that many chips

may preserve data for relatively long periods of time at temperatures above $-20°$C which contradicted the common thought that was widely believed at that time. The temperature at which 80% of the data remained for one minute varied widely between different devices. While some devices required cooling to at least $-50°$C, others, surprisingly, retained data for this period at room temperature. Memory retention time also varied between devices of the same type from the same manufacturer, most likely, because controlling data retention time is not a part of the chip manufacturing quality process. Table 1 presents the evaluated systems and RAM technologies used for the cold boot attack in [33]. Three different RAM technologies in six different systems were used to form the test probe. System *A* works with Synchronous Dynamic Random-Access Memory (SDRAM), the oldest of the compared RAM technologies. Systems *B* and *C* work with Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM) and the systems *D*, *E* and *F* work with Double Data Rate Synchronous Dynamic Random-Access Memory (DDR2 SDRAM) of the second generation.

Table 2 shows the effect of cooling the memory modules to $-50°C$ on the percentage of bit errors compared with uncooled memory modules for systems *A* to *D*. As shown in the table, cooling the memory modules can dramatically reduce the decay effect for a relatively long time and therefore reduce the expected decay factor. A smaller decay factor increases the amount of known bits that can be used to launch the algebraic step of a cold boot attack against the target cryptographic primitive. Thus, one way to launch a cold boot attack is to remove the memory module, after cooling it, from the target system and immediately plug it in another system under the adversary's control. This system is then booted to access the memory. Another possible approach to execute the attack is to cold boot the target machine by cycling its power off and then on without letting it shut down properly. Then a lightweight operating system is, instantly, booted where the content of memory is dumped to a file. Further analysis can then be performed on the information that is retrieved from

	Memory Type	Chip Maker	Memory Density	Make/Model	Year
A	SDRAM	Infineon	128Mb	Dell Dimension 4100	1999
B	DDR	Samsung	512Mb	Toshiba Portégé	2001
C	DDR	Micron	256Mb	Dell Inspiron 5100	2003
D	DDR2	Infineon	512Mb	IBM T43p	2006
E	DDR2	Elpida	512Mb	IBM x60	2007
F	DDR2	Samsung	512Mb	Lenovo 3000 N100	2007

Table 1: Test Systems used in [33]

	Seconds w/o power	Error % at operating temp.	Error % at $-50°C$
A	60	41	(no errors)
	300	50	0.000095
B	360	50	(no errors)
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

Table 2: Effect of cooling the memory module on the error rates [33]

memory in order to find sensitive information such as cryptographic keys or passwords.

2.2.1 Countermeasures

The following list of recommendations, extracted and supplemented from [33], gives an overview of countermeasures that can be used to defend against cold boot attacks.

- Scrubbing the memory: Do not store cryptographic keys longer than needed in the memory. Overwrite the keys after their usage and clean the memory during the boot process (shutdown and boot) by overwriting the memory content.

- Limiting booting from network or removable media: Limit the boot options to *boot only from installed hard disk*, which limits the options of an adversary to replace the

memory modules or the boot medium (hard disk). Protect the BIOS configuration for unintended changes of the boot options.

- Suspending the system safely: When a computer system is in *Sleep* mode or *Hibernation* mode, the memory content must be maintained in a way that allows a quick reproduction of the running state of the system. When using these modes, encrypt the memory data during suspension and decrypt the data during the revoke process after successfully entering the password by the user.

- Avoid auto-mount of encrypted file systems: Auto-mount of encrypted file systems requires the password accessible in the memory even if the file system is not in use which contradicts the previously mentioned memory scrubbing recommendation.

- Avoid pre-computation: While pre-computations speed up cryptographic operations, they require maintaining the key relevant values in the memory. Repeated computation of the key primitives and clearing its values from the memory may hurt performance. However, it would respect the first recommendation in this list.

- Key Expansion: Apply transformations to the keys in the memory to complicate the reproduction of the keys during a cold boot attack.

- Physical defenses: Limit the physical access to the memory modules. Monitor the system case and erase the memory content when the case is opened.

- Architectural changes: Change the behavior of the used hardware by designing memory modules with a smaller remanence and design systems that reset the state of their modules during shutdown and boot processes.

- Encrypting in the disk controller: The approach used in *tresor* [8] is to relocate the computation of key initiated cryptographic functions into the CPU and its debug

registers or other controllers that are capable of computing and storing cryptographic keys.

Based on our work developed in chapter 4, *software obfuscation* can be added to the list above. Software obfuscation can be seen as a superordinate concept of the *key expansion* described above.

2.3 Obfuscation Techniques

Software obfuscation represents a set of techniques to transform a program code or binary file into a new secured representation of the program that is hard to read, analyze and reverse engineer. In this section we briefly review different categories of obfuscation techniques [21]. Obfuscation techniques relay on the transformation of readable information into a form that is hard to interpret and to reverse engineer. The available techniques highly depend on the used programming language, the used compiler, the operating system, and the programming technique (e.g., object-oriented programming vs. structured programming). Desirable properties for such transformations are minimal cost (w.r.t execution time and program size) and the stealthiness of the obfuscation. Stealthiness is relevant where only a part of the target program is obfuscated; it prevents adversaries from finding the obfuscated code and concentrating their effort on it.

2.3.1 Complicating the Control Flow

The remaining of this section describes several techniques that can be used to complicate the analysis of control flows by transforming the control flow into a more complex structure while retaining the intended functionality.

- Opaque expressions: Opaque predicates are opaque expressions evaluated at execution time. The expected result (always true, always false, or one of them) is known

in advance during the implementation, but hard to figure out by an attacker. Simple examples for this techniques are:

$x^2 + x \bmod 2 = 0$ always returns true.

$x \bmod 2 = 0$ sometimes returns true and sometimes false.

- Flattening the control-flow: Flattening removes the structure of the control flow, while maintaining the original control flow, with multiple jumps. An example of this approach is the replacement of a *for loop* (position 1) with a *switch/case structure* (position 2).

  1. $for(int\ i = 0; i < 10, i++)\{...\}$
  2. $int\ i = 0;$
     $while(1)\{$
     $switch(i)\ \{$
     $case\ 0 : \cdots ;\ break;$
     $case\ 1 : \cdots ;\ break;$
     $\cdots$
     $case\ 9 : \cdots ;\ break;$
     $default : \cdots ; goto\ end;\ \}$
     $i++;\}$
     $end : \cdots ;$

  Further techniques for control-flow flattening may include *goto*, *throw/catch*, and other language depended instructions.

- Aliasing: An Alias represents an alternative path to manipulate the value, respectively the memory location, of a variable. It forces adversaries to analyze how variables might be modified in several different ways. Missing one of these paths might

result into an unexpected behavior for the adversary.

- Inserting bogus control flows: A bogus control flow interrupts the sequence of a code segment by introducing additional branches to the code. Combined with opaque predicates, these branches can simply split the sequence and after evaluating the result resume the sequence or they can introduce additional execution paths.

- Jump through branch functions: Replacing unconditional jumps such as goto statements, which jump to an address in the code given by a label, with branch functions. The return address of the branch function stored in the *ebp* register is replaced with the start address to the desired functionality. The function branch below shows an example where the return address of the function in the *ebp* register is replaced by the value in the parameter *addr*.

$$
\begin{aligned}
&void\ branch(unsigned\ int\ addr)\{\\
&\_\_asm\{mov\ ebp,\ addr;\}\\
&\}
\end{aligned}
$$

2.3.2 Opaque Predicates

We already introduced the basic idea of opaque predicates as a tool to complicate the control flow. For the sake of completeness, in what follows we provide a description of some specialized opaque predicates.

- Opaque predicates from pointer aliasing: Define a set of pointer structures $G = \{G_1, G_2, \cdots\}$ where the elements in G are called *graphs* that contain a set of nodes and further define a set of of pointers $PTR = \{ptr_1, ptr_2, \cdots\}$ pointing to some nodes in the *graphs*. Modify in multiple iterations the graph structures by randomly splitting the graphs, inserting, deleting, or moving nodes and reassigning the assigned pointers to new nodes in the same graph. Based on the new structures define opaque

predicates related the the resulting graphs (e.g., $(ptr_1 \neq ptr_2) == true$) and at runtime, randomly reassign the pointers in the graphs to different nodes.

- Opaque values from array aliasing: Define an array (A[]) and fill the array with values following a specific scheme. Evaluate opaque predicates based on different entries in the array. At runtime randomly modify the values in the array without violating the defined scheme. An example for this technique is:

  if$((A[6] \ mod \ A[2]) == A[1])$ { $\cdots$ }

  As seen in the example above, the scheme to assign and change values in the array depends on the chosen relations for the opaque predicates.

- Opaque predicates from concurrency: Opaque predicates from concurrency work with concurrent threads utilizing race conditions and maintaining the same data structure. Define the opaque predicates according to the expected values in the modified data structure as a function of the thread winning the race.

2.3.3 Data Encodings

We can compare *data encodings* with encryption and decryption techniques from cryptography. An encoding function *Enc()* transforms the variable into an obfuscated representation and a decoding function *Dec()* undo this transformation. The transformed value must be able to represent all the possible values that the entity can take which requires additional functionality to guarantee this required behavior.

- Encoding integers: We can define a set of obfuscated operations in form of functions (*ENC(), DEC(), ADD(), SUB(), MULT(), LT(),* $\cdots$) that neutralize each others effect. Several approaches for transforming an integer exist. These approaches range from number-theoretic tricks to applying cryptographic algorithms such as the AES

or DES to the target integer value. The following example, extracted from [21], illustrate the above idea.

typedef int T;

T ENC (int e) {return $e + 1$;}

int DEC(T d) {return $d - 1$;}

T ADD(T a, T b) {return $a + b - 1$;}

T MULT(T a, T b) {return $a * b - a - b + 2$;}

BOOL LT(T a, T b) {return $a < b$;}

Another approach to encode integers is splitting the value in question. At this point we only mention two of the many possible approaches. First, split the bit stream representing the integer value into two pieces and assign the two values to different variables. For the second approach, generate a random value, assign it to one variable and assign the original value Xored with the random value to a second variable.

- Encoding Booleans: Similar to *encoding integers*, a set of functions representing the required and transformed boolean operations can be defined. These approaches include splitting of the Boolean value, defining our own true table, e.g., an even integer value represents False and an odd value represents True.

- Encoding literal data: *Literal data* can be regarded as a specialized set of integer values and all transformations related to *encoding integers* can also be applied to literal data. Due to the enhanced nature of literals, further approaches such as specialized state machines further extend the applicable techniques.

- Encoding arrays: In contrast to the encoding approaches above, the obfuscation techniques for arrays are somewhat limited and fall into two categories. First, the arrays

can be reordered by applying a permutation or a homomorphic function to the indices of the array. The second category modifies the structure of arrays. An array can be split into several sub-arrays, different arrays can be merged together, a multi-dimensional array can be flattened into a lower dimension, and an array can be folded into a new array of higher dimension.

It is not hard to imagine how different techniques can be combined together. For example, after encoding an integer array, the values stored in the array can be subjected to additional obfuscation techniques.

2.3.4 Breaking Abstractions

To provide a readable and maintainable code, programming tasks are broken down into several parts that are implemented separately. Based on the resulting abstraction and programming environment these parts can be represented by packages, modules, classes, functions, structures, loops, frameworks/libraries, which may reveal information to the adversaries. *Breaking abstractions* eliminates such structures. Additional abstractions to be considered are given by the hardware and the operating system of the targeted computer system.

- Merging function signatures: This technique unifies the function signatures in order to prevent analysis based on function semantics. For example,

  $$void\ foo(int\ x, float\ y)\{\cdots;\}$$

  $$int\ bar(char\ x)\{\cdots;\}$$

  $$void\ flu(float\ x, float\ y)\{\cdots;\}$$

  can be transformed into the following three functions with a uniform signature

  $$int\ foo(int\ x, float\ y, float\ z, char\ a)\{\cdots;\}$$

  $$int\ bar(int\ x, float\ y, float\ z, char\ a)\{\cdots;\}$$

$$int\ flu(int\ x, float\ y, float\ z, char\ a)\{\cdots;\}$$

- Modifying instruction encodings: This technique introduces its own interpreter as an additional layer between the executing machine and the programm to be executed. It also defines how the machine instructions are selected and interpreted. This approach is based on a decoding binary tree to select the required instructions. At run time, the structure of the tree is modified such that the same instruction uses different bit patterns as a selection criterion. Due to the dynamic structure of the decoding tree, this approach can only be applied on hard coded functionality which is called as a bit structure representing a machine code from the program itself.

Chapter 3

Application of Two Off-the-shelf Algebraic Tools for Extraction of Cryptographic Keys from Corrupted Memory Images

Cryptographic key recovery from memory or memory dumps, for malicious or forensic purposes, has attracted great attention of security professionals and cryptographic researchers. In [48], Shamir and van Someren considered the problem of locating cryptographic keys hidden in large amount of data, such as the complete file system of a computer system. In addition to efficient algebraic attacks locating secret RSA keys in long bit strings, they also presented more general statistical attacks which can be used to find arbitrary cryptographic keys embedded in large files. This statistical approach relies on the simple fact that good cryptographic keys possess high entropy. Areas with unusually high entropy can be located by searching for unique byte patterns in sliding windows and then selecting those windows with the highest numbers of unique bytes as a potential places for the key. Moe *et al.* [42]

developed a proof of concept tool, *Interrogate*, which implements several search methods for a set of key schedules. To verify the effectiveness of the developed tool, they investigated key recovery for systems running in different states (live, screen-saver, dismounted, hibernation, terminated, logged out, reboot, and boot states). Another proof of concept tool, *Disk Decryptor*, which can extract Pretty Good Privacy (PGP) and Whole Disk Encryption (WDE) keys from dumps of volatile memories was presented in [38].

All the above techniques and tools took another dimension after the publication of the cold boot attack by Halderman *et al.* [33]. While the remanence effect of RAM has already been known since decades [49], it attracted greater attention in cryptography only after Halderman *et al.* work in 2008, which explicitly exploited those observations to recover cryptographic keys from the memory . They developed tools which capture everything present in RAM before power was cut off and developed proof of concept tools which can analyze these memory copies to extract secret DES, AES and RSA keys. In particular, Heninger *et al.* [34] showed that an RSA private key with small public exponent can be efficiently recovered given a 27% fraction of its bits at random. They have also developed a recovery algorithm for the 128-bit version of AES (AES-128) that recovers keys from 30% decayed AES-128 Key Schedule images in less than 20 minutes for half of the simulated cases. Tsow [51] further improved upon the proof of concept in Halderman *et al.* and presented a heuristic algorithm that solved all cases at 50% decay in under half a second. At 60% decay, Tsow recovered the worst case in 35.5 seconds while solving the average case in 0.174 seconds. At the extended decay rate of 70%, recovery time averages grew to over 6 minutes with the median time at about five seconds. In [12], Albrecht *et al.* proposed methods for key recovery of ciphers (AES, Serpent and Twofish) used in Full Disk Encryption (FDE) products where they applied a method for solving a set of non-linear algebraic equations with noise based on mixed integer programming. To improve the running time of their algorithms, they only considered a reduced number of rounds.

Applying their algorithms, they obtained satisfactory success rates for key recovery using the Serpent key schedule up to 30% decay and for the AES up to 40% decay.

Cryptanalytic attacks can be classified into pure mathematical attacks and side channel attacks. Pure mathematical attacks, are traditional cryptanalytic techniques that rely only on known or chosen input-output pairs of the cryptographic function, and exploit the inner structure of the cipher to reveal secret key information. On the other hand, in side channel attacks, it is assumed that the attacker has some physical access to the cryptographic device through one or more side channel. Well-known side channels, which can leak critical information about the encryption state, include timing information [39] and power consumption [40].

In addition to these commonly exploited side channels, the remanence effect of random access memory (RAM) is a highly critical side channel that has been recently exploited by cold boot attacks [33] [34] to retrieve secret keys from RAM. Although dynamic RAMs (DRAMs) become less reliable when its contents are not refreshed, they are not immediately erased. In fact, contrary to popular belief, DRAMs may retain their contents for seconds to minutes after power is lost and even if they are removed from the computer motherboard. A cold boot attack is launched by removing the memory module, after cooling it, from the target system and immediately plugging it in another system under the adversary's control. This system is then booted to access the memory. Another possible approach to execute the attack is to cold boot the target machine by cycling its power off and then on without letting it shut down properly. Upon reboot, a lightweight operating system is instantly booted where the targeted memory contents are dumped to a file.

Experimental results in [49] show how data are retained for a relatively long time in computer memories after a system power off. However, the first work explicitly exploiting those observations to recover cryptographic keys from the memory was reported by Halderman *et al.* [33] where they presented proof of concept experiments which showed that

it is practically feasible to perform cold boot attacks exploiting the remanence effect of RAMs to recover secret keys of DES, AES and RSA. After the publication of Halderman *et al.* [33], several other authors (e.g., [51] [12] [34]) further improved upon this proof of concept and presented algorithms that solved cases with higher decay factors. However, almost all these previously proposed techniques were cipher-dependent and certainly uneasy to develop and fine tune. On the other hand, for symmetric ciphers, the relations that have to be satisfied between the subround key bits in the key schedule always correspond to a set of nonlinear Boolean equations. In this chapter, we investigate the use of an off-the-shelf SAT solver (CryptoMiniSat [4]), and an open source Gröbner basis tool (PolyBoRi [16]) to solve the resulting system of equations. We also discuss the pros and cons of both tools and present some experimental results for the extraction of AES and Serpent keys from decayed memory images.

3.1 Modern Algebraic Tools and Their Applications to Cryptography

The use of SAT solvers and Gröbner basis in cryptanalysis has recently attracted the attention of cryptanalysts. Courtois *et al.* [26] demonstrated a weakness in KeeLog by presenting an attack which requires about 2^{32} known plaintexts. For 30% of all keys, the full key can be recovered against a complexity of 2^{28} KeeLoq encryptions. In [25], 6 rounds of DES are attacked with only a single known plaintext/ciphertext pair using a SAT solver. Erickson *et al.* [32] used the SAT solver and Gröbner basis [17] attacks against SMS4 on equation system over GF(2) and GF(2^8). In [18], a practical Gröbner basis [17] attack using Magma was applied against the ciphers Flurry and Curry, recovering the full cipher key by requiring only a minimal number of plaintext/ciphertext pairs.

SAT solvers and Gröbner basis have also been applied to the cryptanalysis of stream

ciphers. Eibach *et al.* [31] presented experimental results on attacking a reduced version of Trivium (Bivium) using exhaustive search, a SAT solver, a binary decision diagram (BDD) based attack, a graph theoretic approach, and Gröbner basis. Their result implies that the usage of the SAT solver is faster than the other attacks. The full key of Hitag2 stream cipher is recovered in a few hours using MiniSat 2.0 [27]. In [28], the full 48-bit key of the MiFareCrypto 1 algorithm was recovered in 200 seconds on a PC, given 1 known initial vector (IV) from one single encryption. In [52], Velichkov *et al.* applied the Gröbner basis on a reduced 16 bit version of the stream cipher Lex.

Mironov and Zhang [44] described some initial results on using SAT solvers to automate certain components in cryptanalysis of hash functions of the MD and SHA families. De *et al.* [30] presented heuristics for solving inversion problems for functions that satisfy certain statistical properties similar to that of random functions. They demonstrate that this technique can be used to solve the hard case of inverting a popular secure hash function and were able to invert MD4 up to 2 rounds and 7 steps in less than 8 hours. In [50], Sugita *et al.* used the Gröbner basis to improve the attack on the 58-round SHA-1 hash function to 2^{31} computations instead of 2^{34} in Wang's method [54].

3.1.1 Gröbner Basis and PolyBoRi

A Gröbner basis is a set of multivariate polynomials that have desirable algorithmic properties. In what follows, we briefly review some basic definitions and algebraic preliminaries related to Gröbner basis as presented in [47].

Let K be any field (in here we are interested in the case where $K = \mathbb{F}_2$.) We write $K[x_1, ..., x_n]$ for the ring of polynomials in n for the variables x_i having its coefficients in the field K.

Definition 3 *A subset $I \subset K[x_1, ..., x_n]$ is an ideal if it satisfies:*

1. $0 \in I$.

2. *if f,$g \in I$, then $f + g \in I$.*

3. *if $f \in I$ and $h \in K[x_1, ..., x_n]$, then $hf \in I$.*

Definition 4 *Let $f_1, ..., f_m$ be polynomials in $K[x_1, ..., x_n]$. Define the ideal $\langle f_1, ..., f_m \rangle = \{\sum_{i=1}^{m} h_i f_i \ : \ h_1, ..., h_m \ \in \ K[x_1, ..., x_n] \ \}$. If there exists a finite set of polynomials in $K[x_1, ..., x_n]$ that generate the given ideal, we call this set a basis.*

Definition 5 *A monomial ordering on $K[x_1, ..., x_n]$ is any relation $>$ on $\mathbb{Z}^n_{\geq 0}$, or equivalently, any relation on the set of monomials x^α, $\alpha \in \mathbb{Z}^n_{\geq 0}$, satisfying:*

1. *$>$ is a total ordering on $\mathbb{Z}^n_{\geq 0}$.*

2. *if $\alpha > \beta$ and $\alpha, \beta, \gamma \in \mathbb{Z}^n_{\geq 0}$, then $\alpha + \gamma > \beta + \gamma$.*

3. *$>$ is a well ordering on $\mathbb{Z}^n_{\geq 0}$. That is every nonempty subset of $\mathbb{Z}^n_{\geq 0}$ has a smallest element with respect to $>$.*

An example of monomial ordering for our application is lexicographic order which is defined as follows:

Definition 6 *(Lexicographic Order (lex)). Let $\alpha = (\alpha_1, ..., \alpha_n)$, and $\beta = (\beta_1, ..., \beta_n) \in \mathbb{Z}^n_{\geq 0}$. We say $\alpha >_{lex} \beta$ if, in the vector difference $\alpha - \beta \in \mathbb{Z}^n$, the left-most nonzero entry is positive. We will write $x^\alpha >_{lex} x^\beta$ if $\alpha >_{lex} \beta$.*

Definition 7 *Let $f = \Sigma_\alpha a_\alpha x^\alpha$ be a non-zero polynomial in P and let $>$ be a monomial order. The multidegree of f is $multideg(f) = max_>(\alpha \in \mathbb{Z}^n_{\geqslant 0} : a_\alpha \neq 0)$.*

Definition 8 *(leading term of a polynomial). Let $f(x) = \sum_{i=1}^{m} c_\alpha x^\alpha : c_\alpha \in K$ is non-zero and $>$ is the order relation defined for the monomials of the polynomial $f(x)$. The greatest monomial in $f(x)$, regarding to the order relation $>$, is called the leading monomial for the polynomial $f(x)$ and is represented by $LM(f) = x^{multideg(f)}$. Also the set $M(f)$ consists*

of all monomials of $f(x)$ and $T(f)$ denote the set of all terms of $f(x)$. The coefficient of the leading monomial is represented by $LC(f) = a_{multideg(f)} \in K$ and called the leading coefficient. The term containing both the leading coefficient and leading monomial is called the leading term, represented by $LT(f) = LC(f) \cdot LM(f)$.

The idea of Gröbner basis was first proposed by Buchberger [17] to study the membership of a polynomial in the ideal of the polynomial ring.

Definition 9 *(Gröbner basis) Let an ideal I be generated by $G = g_1, ..., g_m$, where g_i, $1 \leq i \leq m$ is a polynomial. G is called the Gröbner basis for the ideal I, if:*

$$\langle LT(I) \rangle = \langle LT(g_1), ..., LT(g_m) \rangle,$$

where $\langle LT(I) \rangle$ denotes the ideal generated by the leading terms of the members in I.

One can view Gröbner basis as a multivariate, non-linear generalization of the Euclidean algorithm for computation of univariate greatest common divisors, Gaussian elimination for linear systems, and integer programming problems. In this work, we use Gröbner basis as an algebraic tool that allows us to solve non-linear Boolean equations by using the PolyBoRi framework.

The following example explains the main involved steps and commands for the PolyBoRi framework in Sage [2] to solve a given system of nonlinear Boolean equations.

Example 3.1.1 *Consider the following system of non-linear Boolean equations*

$$\begin{aligned} x_1x_2 \oplus x_3x_4 &= 1, \\ x_1x_3x_5 \oplus x_4x_5 &= 0, \\ x_1x_2x_5 \oplus x_3x_5 &= 0, \\ x_2x_3 \oplus x_3x_4x_5 &= 1, \end{aligned} \tag{1}$$

Figure 2 shows the steps to be executed in PolyBoRi to obtain the Gröbner basis of the systems of equations in (1). As shown in the figure, the function ideal() *in step 2 takes the corresponding homogeneous system of equations as a calling parameter. The resulting Gröbner basis is given by* $[x_1 + x_4 + 1, x_2 + 1, x_3 + 1, x_4^2 + x_4, x_5]$. *In this notation,* x_i *appearing in a separate term by itself implies that the system of equations under consideration can be solved by setting* $x_i = 0$. *Similarly,* $x_i + 1$ *implies that* $x_i = 1$. *Also, the notation* $x_i + x_i^2$ *implies that* x_i *can be assigned a 0 or a 1. Thus the above basis corresponds to the following two independent solutions:* $\{x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 0\}$ *and* $\{x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1, x_5 = 0\}$.

```
c    Lines starting with 'c' are comments
c    Step 1 defines the Polynomial Ring; where GF(2) defines the Galois field (GF) of two elements
c    as the base ring, 5 is the number of variables and order = 'lex' sets the order to lexical order
c    Step 2 defines the Ideal taking a set of homogeneous equations as calling parameter
c    Step 3 combines the ideal I with the field ideal; limiting the solution range to F2
c    Step 4 executes the Gröbner basis returning the result

sage: PR.<x1,x2,x3,x4,x5> = PolynomialRing(GF(2), 5, order='lex')
sage: I = ideal([x1*x2 + x3*x4 + 1, x1*x3*x5 + x4*x5, x1*x2*x5 + x3*x5, x2*x3 + x3*x4*x5 + 1])
sage: J = I + sage.rings.ideal.FieldIdeal(PR)
sage: J.groebner_basis()

[x1 + x4 + 1, x2 + 1, x3 + 1, x4^2 + x4, x5]
```

Figure 2: Working with PolyBoRi to solve the systems of equations in (1)

3.1.2 The SAT problem and CryptoMiniSat

The Boolean satisfiability (SAT) problem [24] is defined as follows: Given a Boolean formula, check whether an assignment of Boolean values to the propositional variables in the formula exists such that the formula evaluates to true. If such an assignment exists, the formula is said to be satisfiable; otherwise, it is unsatisfiable. For a formula with m variables, there are 2^m possible truth assignments. The conjunctive normal form (CNF) is frequently used for representing Boolean formulas. In CNF, the variables of the formula appear in literals (e.g., x) or their negation (e.g., $\overline{x}$). Literals are grouped into clauses, which represent

a disjunction (logical *OR*) of the literals they contain. A single literal can appear in any number of clauses. The conjunction (logical *AND*) of all clauses represents a formula. For example, the CNF formula $(x_1) \wedge (\overline{x}_2 \vee \overline{x}_3) \wedge (x_1 \vee x_3)$ contains three clauses: x_1, $\overline{x}_2 \vee \overline{x}_3$ and $x_1 \vee x_3$. Two literals in these clauses are positive (x_1, x_3) and two are negative ($\overline{x}_2$, $\overline{x}_3$). For a variable assignment to satisfy a CNF formula, it must satisfy each of its clauses. For example, if x_1 is true and x_2 is false, then all three clauses are satisfied, regardless of the value of x_3.

While the SAT problem has been shown to be NP-complete [24], efficient heuristics exist that can solve many real-life SAT formulations. Furthermore, the wide range of target applications of SAT have motivated advances in SAT solving techniques that have been incorporated into freely-available SAT solvers such as the CryptoMiniSat.

When preparing the input to the SAT solver, the terms of quadratic and higher degree are handled by noting that the logical expression

$$(x_1 \vee \overline{T})(x_2 \vee \overline{T})(x_3 \vee \overline{T})(x_4 \vee \overline{T})(T \vee \overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3 \vee \overline{x}_4) \tag{2}$$

is tautologically equivalent to $T \Leftrightarrow (x_1 \wedge x_2 \wedge x_3 \wedge x_4)$, or the $GF(2)$ equation $T = x_1x_2x_3x_4$. Similar expressions exist for higher order terms. Thus, the system of equations obtained in this step can be linearized by introducing new variables as illustrated by the following example.

Example 3.1.2 *Suppose we would like to find the Boolean variable assignment that satisfies the following formula*

$$x_0 \oplus x_1x_2 \oplus x_0x_1x_2 = 0.$$

Then, using the approach illustrated in (2), we introduce two linearization variables, $T_0 = x_1x_2$ *and* $T_1 = x_0x_1x_2$. *Thus we have*

$$
\begin{aligned}
&x_0 \oplus T_0 \oplus T_1 = 0, \\
&(\overline{T}_0 \vee x_1) \wedge (\overline{T}_0 \vee x_2) \wedge (T_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1, \\
&(\overline{T}_1 \vee x_0) \wedge (\overline{T}_1 \vee x_1) \wedge (\overline{T}_1 \vee x_2) \wedge \\
&(T_1 \vee \overline{x}_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1.
\end{aligned} \tag{3}
$$

Since the CryptoMiniSat expects only positive clauses and the CNF form does not have any constants, we need to overcome the problem that the first line in (3) corresponds to a negative, i.e., false, clause. Adding the clause consisting of a dummy variable, d, or equivalently ($d \wedge d \cdots \wedge d$) would require the variable d to be true in any satisfying solution, since all clauses must be true in any satisfying solution. In other words, the variable d will serve the place of the constant 1.

Therefore, the above formula can be expressed as

$$
\begin{aligned}
&d = 1, \\
&x_0 \oplus T_0 \oplus T_1 \oplus d = 1, \\
&(\overline{T}_0 \vee x_1) \wedge (\overline{T}_0 \vee x_2) \wedge (T_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1, \\
&(\overline{T}_1 \vee x_0) \wedge (\overline{T}_1 \vee x_1) \wedge (\overline{T}_1 \vee x_2) \wedge \\
&(T_1 \vee \overline{x}_0 \vee \overline{x}_1 \vee \overline{x}_2) = 1.
\end{aligned}
$$

Applying the same logic to the system of equations in (1), we obtain

$$
\begin{aligned}
& d = 1, \\
& T_1 \oplus T_2 = 1, \\
& (\overline{T_1} \vee x_1) \wedge (\overline{T_1} \vee x_2) \wedge (T_1 \vee \overline{x_1} \vee \overline{x_2}) = 1, \\
& (\overline{T_2} \vee x_3) \wedge (\overline{T_2} \vee x_4) \wedge (T_2 \vee \overline{x_3} \vee \overline{x_4}) = 1, \\
& T_3 \oplus T_4 \oplus d = 1, \\
& (\overline{T_3} \vee x_1) \wedge (\overline{T_3} \vee x_3) \wedge (\overline{T_3} \vee x_5) \wedge \\
& (T_3 \vee \overline{x_1} \vee \overline{x_3} \vee \overline{x_5}) = 1, \\
& (\overline{T_4} \vee x_4) \wedge (\overline{T_4} \vee x_5) \wedge (T_4 \vee \overline{x_4} \vee \overline{x_5}) = 1, \\
& T_5 \oplus T_6 \oplus d = 1, \\
& (\overline{T_5} \vee x_1) \wedge (\overline{T_5} \vee x_2) \wedge (\overline{T_5} \vee x_5) \wedge \\
& (T_3 \vee \overline{x_1} \vee \overline{x_2} \vee \overline{x_5}) = 1, \\
& (\overline{T_6} \vee x_3) \wedge (\overline{T_6} \vee x_5) \wedge (T_6 \vee \overline{x_3} \vee \overline{x_5}) = 1, \\
& T_7 \oplus T_8 = 1, \\
& (\overline{T_7} \vee x_2) \wedge (\overline{T_7} \vee x_3) \wedge (T_7 \vee \overline{x_2} \vee \overline{x_3}) = 1, \\
& (\overline{T_8} \vee x_3) \wedge (\overline{T_8} \vee x_4) \wedge (\overline{T_8} \vee x_5) \wedge \\
& (T_8 \vee \overline{x_3} \vee \overline{x_4} \vee \overline{x_5}) = 1,
\end{aligned}
$$

Figure 3 shows the CryptoMiniSat input file corresponding to the above system of equations. As shown in the figure, a negative number implies that the variables assumes a value = 0 *and a positive number implies a value* = 1*. Lines starting with 'x' denote an XOR equation and each lines is terminated with '0'.*

From the above examples, its is clear that, compared to PolyBoRi, preparing the input for the CryptoMiniSat requires relatively longer pre-processing steps. Also, unlike the Gröbner basis approach which returns the general form of the solution, CryptoMiniSat returns one valid solution. To find the other solutions, the already found solutions have to

```
c    Lines starting with 'c' are comments
c    The first line in the SAT file is in the form: 'p cnf # variables # clause'
c    Each line ends with '0' and lines starting with 'x' denote XOR equations
c    True variables are denoted by positive numbers and False variables are denoted by
c    negating the number; example: x1 → 2; (x̄2 → -3)
c    d → 1, x1 → 2, x2 → 3, … , T6 → 12, T7 → 13, T8 → 14

p cnf 14 32
1 0
x 7 8 0
-7 2 0
-7 3 0
7 -2 -3 0
-8 4 0
-8 5 0
8 -4 -5 0
x 9 10 1 0
-9 2 0
-9 4 0
-9 6 0
9 -2 -4 -6 0
-10 5 0
-10 6 0
10 -5 -6 0
x 11 12 1 0
-11 2 0
-11 3 0
-11 6 0
11 -2 -3 -6 0
-12 4 0
-12 6 0
12 -4 -6 0
x 13 14 0
-13 3 0
-13 4 0
13 -3 -4 0
-14 4 0
-14 5 0
-14 6 0
14 -4 -5 -6 0
```

Figure 3: CryptoMiniSat input file corresponding to the system of equations in (1)

be negated and added to the SAT solver input file. In the example above, the first solution returned by the CryptoMiniSat ({$1, -2, 3, 4, 5, -6, -7, 8, -9, -10, -11, -12, 13, -14$*}) is negated ({*$-1, 2, -3, -4, -5, 6, 7, -8, 9, 10, 11, 12, -13, 14$*}) and added to the SAT solver input file as a new entry. When running the SAT solver again, this added entry forces the SAT solver to eliminate this as a possible solution and search for a new one that solves the SAT problem. When doing so, the SAT solver returns the second possible solution ({*$1, 2, 3, 4, -5, -6, 7, -8, -9, -10, -11, -12, 13, -14$*}).*

3.2 Structure of the AES-128 and Serpent Key Schedules

In this section, we briefly review the relevant details of the AES-128 and Serpent key schedules.

3.2.1 Key Schedule of AES-128

In the following we describe the AES-128 key scheduler [29] [1]. AES-128 works with a user key (Master Key) of 128 bits (16 bytes) represented by a 4x4 array $K^0_{i,j}$, the AES state matrix; with $0 \leq i, j \leq 3$ where i and j denote the row and column indices, respectively. $K^{r+1}_{i,j}$ denotes the bijective mapping of the user key to the 10 sub-round keys, where $0 \leq r \leq 9$ denotes the number of the rounds. The r^{th} key schedule round is defined by the following transformations:

$$\begin{aligned} K^{r+1}_{0,0} &\leftarrow S(K^r_{1,3}) \oplus K^r_{0,0} \oplus Rcon(r+1) \\ K^{r+1}_{i,0} &\leftarrow S(K^r_{(i+1)mod4,3}) \oplus K^r_{i,0}, 1 \leq i \leq 3 \\ K^{r+1}_{i,j} &\leftarrow K^{r+1}_{i,j-1} \oplus K^r_{i,j}, 0 \leq i \leq 3, 1 \leq j \leq 3 \end{aligned} \tag{4}$$

where $Rcon(\cdot)$ denotes a round-dependent constant and $S(\cdot)$ represents the s-box operations based on the 8×8 Rijndael S-box [29]. Figure 4 shows the transformations given by equation 4.

3.2.2 Key Schedule of Serpent

Serpent [13] is a 32 round block cipher based on a substitution permutation network (SPN) structure with an Initial Permutation (IP) and a Final Permutation (FP). It has 32 rounds, each consists of a key mixing operation, a pass through S-boxes, and (in all but the last round) a linear transformation. In the last round, this linear transformation is replaced by an additional key mixing operation. The cipher accepts a variable user key length that is

always padded up to 256 bits by appending one bit-value '1' to the end of the most significant bit followed by bit-values '0'. To obtain the 33 128-bit subkeys $K_0, ..., K_{32}$, the user key is divided into eight 32-bit words $w_{-8}, w_{-7}, ..., w_{-1}$, from which the 132 intermediate keys or pre-keys ($w_0 ... w_{131}$) are derived as follows:

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11 \tag{5}$$

where ϕ is a constant formed by the fractional part of the golden ratio $(\sqrt{5} + 1)/2$ or 0x9e3779b9 in hexadecimal.

The round keys k_i are evaluated from the pre-keys by first calling one of the eight 4×4 S-boxes in bit slice mode. In bit slice mode, each input of the S-box comes from a different 32-bit word and each output goes to a different 32-bit word. The 4x32 bits per round are all handled by the same S-box. A group of four input or four output words defines a unit that is handled together. The transformation from pre-keys w_i into words k_j of round keys is performed as follows:

$$\begin{aligned}
\{k_0; k_1; k_2; k_3\} &= S_3(w_0; w_1; w_2; w_3) \\
\{k_4; k_5; k_6; k_7\} &= S_2(w_4; w_5; w_6; w_7) \\
\{k_8; k_9; k_{10}; k_{11}\} &= S_1(w_8; w_9; w_{10}; w_{11}) \\
\{k_{12}; k_{13}; k_{14}; k_{15}\} &= S_0(w_{12}; w_{13}; w_{14}; w_{15}) \\
\{k_{16}; k_{17}; k_{18}; k_{19}\} &= S_7(w_{16}; w_{17}; w_{18}; w_{19}) \\
\vdots \quad & \vdots \quad \vdots \\
\{k_{124}; k_{125}; k_{126}; k_{127}\} &= S_4(w_{124}; w_{125}; w_{126}; w_{127}) \\
\{k_{128}; k_{129}; k_{130}; k_{131}\} &= S_3(w_{128}; w_{129}; w_{130}; w_{131})
\end{aligned} \tag{6}$$

where S_i denotes the i^{th} S-box of Serpent. The round keys K_i are then formed by regrouping the 32-bit values k_j as 128-bit sub-keys K_i (for i $\in$ 0,.., r) as follows:

$$K_i := \{k_{4i}; k_{4i+1}; k_{4i+2}; k_{4i+3}\} \tag{7}$$

Finally, we apply IP to the round keys K_i in order to place the key bits in the correct column, i.e., $\hat{K}_i$ = IP(K_i). Figure 5 depicts the described key scheduler of Serpent.

By exploiting the asymmetric decay of the memory images and the redundancy of key material inherent in the key schedule of both algorithms above, rectifying the faults in the corrupted memory images of the the key schedule is formulated as a Boolean satisfiability problem which can be solved efficiently for relatively large decay factors.

3.3 Simulation Results

Because of the nature of the cold boot attack, it is realistic to assume that only a corrupted image of the contents of memory is available to the attacker, i.e., a fraction of the memory bits will be flipped from its charged state. Halderman *et al.* [33] observed that, within a specific memory region, the decay is overwhelmingly asymmetric, i.e., either $0 \rightarrow 1$ or $1 \rightarrow 0$. When trying to retrieve cryptographic keys, the decay direction for a region can be determined by comparing the number of 0's and 1's since in an uncorrupted key, the expected number of 0's and 1's should approximately be equal.

Similar to the previous work in [33] [51], throughout our experimental results, we assume an asymmetric decay model where bits overwhelmingly decay to their ground state rather than their charged state. Using this model, only the bits that remain in their charged state are useful to the cryptanalyst since one cannot be sure about the original values of the 0 bits, i.e., whether they were originally 0's or decayed 1's. Let β denote the fraction of decayed bits. If the percentage of 0's and 1's in the original key schedule bits is p_z and $1 - p_z$, respectively, then the fraction, f, of key bits that can be assumed to be known by

examining the decayed memory of the key schedule is given by

$$f = 1 - (p_z + \beta \times (1 - p_z)) = (1 - p_z) \times (1 - \beta).$$

Since in an uncorrupted key schedule key, we expect the number of 0's and 1's to be approximately equal, i.e., $p_z \approx 1/2$, then we have $f \approx (1 - \beta)/2$.

In our experiments, the input files for the CryptoMiniSat contained 5,144 and 18,500 clauses for AES and Serpent, respectively. For PolyBoRi, 1,280 equations with 1,728 variables were defined for AES and 8,448 equations with 8,704 variables were defined for Serpent.

Tables 3, 4, 5 and 6 show statistics for the run time required to recover the key of AES and Serpent from the corresponding corrupted memory images for different decay factors. These runtime statistics were obtained using PolyBoRi and CryptoMiniSat running on a Dell Precision 370 workstation with a 3.0 GHz Intel Pentium 4 CPU and 1 GB of RAM. Examining the results in the tables reveal the following observations:

- While the resource requirements of both tools (time for CryptoMiniSat, and time and memory for PolyBori) seem to grow exponentially with the decay factor, for practical values of the decay factor, both tools require reasonably short time to recover the secret keys from corrupted memory images.

- The simple and high redundancy in the AES key schedule allows for faster recovery of the key from corrupted memory images. This makes AES more prone to these attacks as compared to other AES finalist such as Serpent. In fact, our initial experiments with Twofish [10] indicate that its relatively more complex key schedule limits the practical applications of these tools to very small values of the decay factor.

- CryptoMiniSat seems to be more suitable for applications in this type of attacks. In particular, every time we tried to push the decay factor higher than the values reported

in Table 4, the PolyBoRi tool always crashed after few minutes due to the excessive memory consumption. This behavior also persisted on a 64 bit Linux operating systems with a freshly compiled PolyBoRi/sage system and 8 GB RAM. The question remains if solutions for a higher decay factor can be achieved in a reasonable time if this memory limitations is fixed in the tool.

Table 3: Run-time statistics using Gröbner basis for AES.

Decay	10%	20%	30%	40%	50%	60%	70%
Min	0.4	0.6	0.8	1.3	2.2	3.5	7
Max	0.7	0.9	1.1	2.1	3.6	7.6	45
Avg.	0.6	0.7	0.9	1.7	2.9	5.6	21
St.Dev	0.1	0.1	0.1	0.3	0.5	1.2	13
Med.	0.5	0.7	0.9	1.7	2.9	5.3	15

Table 4: Run-time statistics using Gröbner basis for Serpent.

Decay	10%	20%	30%	40%	50%	60%	70%
Min	8	9	17	56	114	417	-
Max	9	34	50	2075	2812	578	-
Avg.	8	15	36	328	399	507	-
St.Dev	0.3	7	11	656	764	47	-
Med.	8	12	40	107	131	504	-

Table 5: Run-time statistics using SAT-solver for AES [37].

Decay	30%	40%	50%	60%	70%
Min	0.046	0.046	0.062	0.062	0.078
Max	0.593	0.140	0.187	0.593	207.171
Avg.	0.064	0.066	0.074	0.102	1.233
St.Dev	0.009	0.007	0.008	0.028	4.899
Med.	0.062	0.062	0.078	0.093	0.359

Table 6: Run-time statistics using SAT-solver for Serpent.

Decay	10%	20%	30%	40%	50%	60%	70%
Min	0.4	0.4	0.5	0.5	0.7	0.9	4
Max	0.7	0.8	1.2	1.6	8.0	69	35282
Avg.	0.6	0.6	0.7	0.9	1.9	8	1278
St.Dev	0.05	0.07	0.10	0.22	1.30	11	4402
Med.	0.15	0.17	0.20	0.35	1.18	9	27706

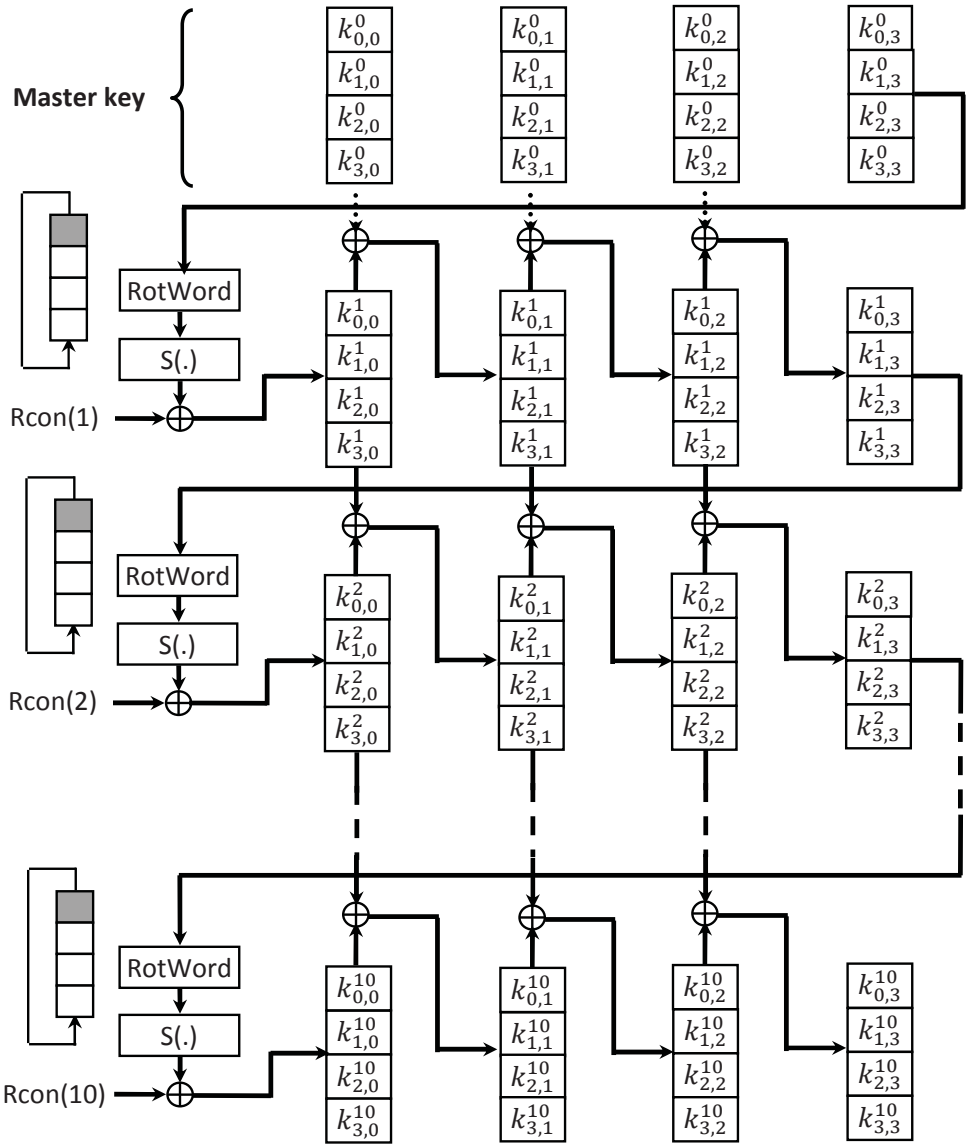

Figure 4: Key Schedule of AES

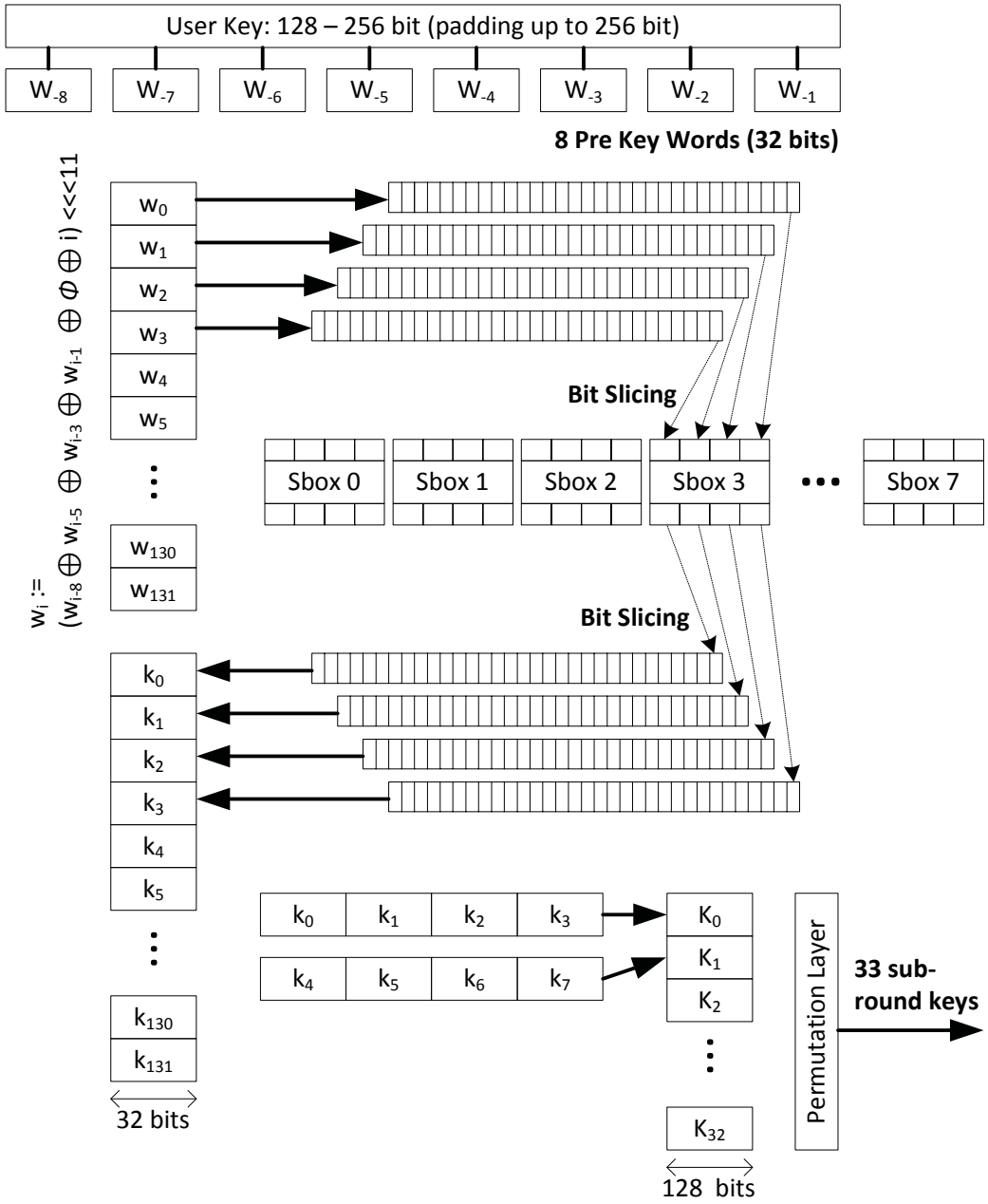

Figure 5: Key Schedule of Serpent

Chapter 4

An Obfuscated Implementation of RC4

Cryptographic techniques are traditionally implemented to protect data and keys against attacks where the adversary may observe various inputs to and outputs from the system, but has no access to the internal details of the execution. On the other hand, several recently developed applications require a higher degree of robustness against attacks from the execution environment where the adversary has closer access to the software implementation of key instantiated primitives. Digital Rights Management (DRM) is an example of such applications where one of the main design objective is to control access to digital media content. This higher degree of robustness can be achieved through white-box implementations where encryption keys are hidden, using obfuscation techniques, within the implementation of the cipher. White-box implementations for block ciphers, such as AES and DES, are widely available [20] [19].

On the other hand, obfuscation can be used to transform a program from an easily readable format to one that is harder to read, trace, understand and modify. This offers an additional layer of security as it protects the program by increasing the required human and computational power that is needed to reverse engineer, alter, or compromise the obfuscated program. Obfuscation techniques can be categorized into automated and manual methods. The former systematically, using specific tools, modify the source code (e.g. [3]) or the

binary executable file (e.g. [11] [6]). The latter techniques rely on the programmer to follow obfuscation techniques during coding. These techniques are governed by the programming language in use. For example, languages like C and C++, which are commonly used in the implementation of cryptographic primitives due their high performance, are flexible in syntax and allow the use of pointers.

Obfuscated programs can be reverse engineered using techniques such as static and dynamic analysis. Static analysis techniques analyze the program file by performing control flow and data flow analysis ([21] [53]) without running the program. Dynamic analysis, on the other hand, takes place at runtime and addresses the followed execution path.

In this chapter, we investigate several obfuscation techniques that are suitable for applications to array-based stream ciphers such as RC4. We also perform a comparison between the performance of these different techniques when applied to RC4. Although our proposed implementation does not provide the same level of theoretical security provided by white-box implementations for block ciphers, it still provides a high degree of robustness against attacks from execution environments where the adversary has access to the software implementation such as in digital right management applications.

4.1 The RC4 Cipher

In this section, we briefly review the Key Scheduling Algorithm (KSA), and the Pseudo-random Generation Algorithm (PRGA) of RC4. We also describe the Skype attempt to provide an obfuscated software implementation for RC4.

4.1.1 Standard RC4 Implementation

While traditional feedback shift register based stream ciphers are efficient in hardware, they are less so in software since they require several operations at the bit level. The design

of RC4 avoids the use of bitwise operations as it requires only byte manipulations which makes it very efficient in software. In particular, RC4 uses 256 bytes of memory for the state array, $S[0]$ through $S[255]$, k bytes of memory for the key, $key[0]$ through $key[k-1]$, and two index pointers: a sequential index i, and quasi random index j.

Algorithm 1 shows the KSA of RC4, where the permutation S is initialized with a key of variable length, typically between 40 to 256 bits. Once this is completed, the key stream is generated using the PRGA shown in Algorithm 2. The generated key stream is combined with the plaintext, usually, through an XOR operation.

```
1: for i = 0 → 255 do
2:     S[i] := i
3: end for
4:
5: j := 0
6: for i = 0 → 255 do
7:     j := (j + S[i] + key[i mod keylength]) mod 256
8:     swap values of S[i] and S[j]
9: end for
```

Algorithm 1: RC4 Key Scheduling Algorithm (KSA) [43]

```
1: i := 0
2: j := 0
3: while GeneratingOutput do
4:     i := (i + 1) mod 256
5:     j := (j + S[i]) mod 256
6:     swap values of S[i] and S[j]
7:     K := S[(S[i] + S[j]) mod 256]
8:     output K
9: end while
```

Algorithm 2: RC4 Pseudo-Random Generation Algorithm (PRGA) [43]

Analyzing the KSA and PRGA algorithms of RC4 yields the following observations:

1. As with most stream ciphers, an adversary does not have to find the key in order to break the cipher. In other words, recovering the initialized inner-state S allows the

adversary to efficiently generate the keystream output of the cipher and decrypt the target ciphertext even without knowing the key array.

2. An adversary who is able to observe the values of the index pointer j in the PRGA can efficiently recover the whole inner state S.

4.1.2 Skype's RC4 Implementation

The only reference to obfuscated RC4 implementation in the open literature appeared in a Blackhat publication that describes the leaked implementation used in Skype [7] [14]. By analyzing this leaked implementation, we observe the following:

- Regarding the cipher itself, the cryptographic key used is 80 bytes in length whereas standard implementations use a key of 40 to 256 bits in length (i.e. 5 to 32 bytes).
- Regarding key management, the Skype's implementation selects a key from a pool of 2^{32} keys.
- Regarding the use of the cipher, RC4 itself is used as an obfuscation technique to hide the network layer.

The implementation utilizes a macro called *RC4_round* that is used in both the KSA and the PRGA. Therefore, we first describe this macro. This macro is shown in Algorithm 3. When called, the macro is passed the following parameters:

i**:** the sequential index

j**:** the quasi random index

$RC4$**:** an array corresponding to S in the standard implementation

t**:** a variable for swapping the i^{th} and j^{th} element in S

k**:** the cryptographic key used in this iteration

Lines 1, 3 and 4 describe the swapping operation, line 2 evaluates the new quasi random index j, and line 5 evaluates the key for this iteration. The main difference in the use of this macro between KSA and PRGA is in the key value passed and the action based on the return value. In the PRGA, the value for k is always zero, whereas in the KSA, the key used in this iteration is passed. Furthermore, in the KSA, the returned value of this macro is discarded, whereas in the PRGA the returned value is used as part of the key stream.

```
1: t :=RC4[i],
2: j := (j + t + k) mod 256,
3: RC4[i] := RC4[j],
4: RC4[j] := t,
5: RC4[(RC4[i] + t) mod 256] {Output to be returned}
```

Algorithm 3: Round Macro: RC4_round(i,j,t,k,RC4) [7]

The KSA is shown in Algorithm 4. In this implementation, the inner-state of the cipher is stored in a data structure called rc4. This structure contains an array (representing the array S from Algorithms 1 and 2) which holds the random ordered values, a sequential index i, and quasi random index j. The ”for” loop in lines 1 through 6 initializes the array rc4.s sequentially from 0 to 255. The array rc4.s is allocated exactly 256 sequential bytes in memory. The implementation takes advantage of the array structure in memory by initializing 4 bytes in each iteration rather than a single byte. Therefore, the index j in the loop is incremented in each iteration by 4 bytes ($0x04040404$), and is assigned to the array at the i^{th} position. Consequently, the index i has to be incremented by 4 in each iteration. The reordering step (lines 9 through 11) is done by the macro *RC4_round*.

The PRGA is shown in Algorithm 5 where the ”for” loop iterates over the data stream (buffer, of size bytes) performing the encryption/decryption operation. This is done by XORing the data (in buffer) and the key stream generated by the macro *RC4_round*. Furthermore, the indices i and j in the data structure rc4 are updated.

```
1: j := 0x03020100
2: for i = 0 → 255 do
3:     i := i + 4
4:     j := j + 0x04040404
5:     rc4.s[i] := j
6: end for
7:
8: j := 0
9: for i = 0 → 255 do
10:    RC4_round(i, j, t, byte(key,i%80), rc4.s)
11: end for
```

Algorithm 4: Skype Implementation for the RC4 KSA [7]

```
1: for (; bytes; bytes − −) do
2:     i := (i + 1) mod 256
3:     buffer++ {positioning the pointer to the new value to be en-/decrypted}
4:     *buffer = *buffer XOR RC4_round(i, j, t, 0, s)
5:     rc4.i := i
6:     rc4.j := j
7: end for
```

Algorithm 5: Skype Implementation for the RC4 PRGA [7]

Clearly, the Skype implementation described above does not provide enough level of protection for the inner state of the cipher. It should be noted, however, that Skype uses the RC4 cipher itself as an obfuscation technique for the network layer but the effective data stream (voice, chat, video) is encrypted with AES [14].

4.2 Proposed Implementation

In this section we present our obfuscated implementation of RC4. Throughout our work, we assume that the cipher is implemented as a standalone module, i.e., the implemented code contains only the functionality of the cipher. Another approach would be to mix the implementation of the cipher with parts of a larger application. Such a needle in the haystack approach can add more security as the containing application offers more obfuscation space. However, since an implementation of this approach is highly dependent on

the containing application, it is therefore not considered in this work.

In our proposed obfuscated implementation, we first eliminate the use of the array S by using independent set of variables. To improve the efficiency of this approach, we use function pointers. Following that, we utilize multithreading to provide security against dynamic analysis attacks. Finally, we present other generic techniques used to further obfuscate the proposed implementation. Throughout the remaining of this chapter, for illustrative purposes, we use a toy implementation of RC4 (with an array of size $N = 4$). However, performance measures have been made based on a RC4 with standard parameters (i.e. with an array of size $N = 256$).

4.2.1 Eliminating the S Array Data Structure

As the KSA and PRGA algorithms show, standard RC4 implementation requires only a few data objects, namely two index pointers i and j, and an array S.

As a first step towards obfuscation, we substitute the array data structure S by N independent variables, where N is the number of elements in the array S. Unlike the elements of an array which are stored in consecutive memory locations, these independent variables can be scattered throughout the program memory. On the other hand, working with such independent variables eliminates our ability to dynamically address them using a loop structure since we no longer have an array index that can be related to loop counters. To address these variables, we use the loop unrolling technique, also known as loop unwinding. This is illustrated for the toy implementation in Figure 6. As depicted in the figure, the implementation is based on two nested switch/case structures, where the outer structure operates over the index i and the inner structure operates over the index j. It is worth noting that the inner switch/case structures are almost identical for various outer switch/case structures. However, since the i^{th} element in array S has been substituted with an independent variable, this variable has to be correctly referenced in each inner switch/case structure. This

nested structure cannot dynamically evaluate expressions such as $S[S[i] + S[j]]$. For such expressions, a dedicated switch/case structure, $Switch(Output - Index)$, is used. This structure is passed an intermediate value, $Output - Index = S[i] + S[j]$, and evaluates the expression above. In an 8 bit implementation with an array of size $N = 256$, the number of possible combinations is given by $N \times N = 2^8 \times 2^8 = 2^{16}$. This implies that a total of 2^{16} case statements are needed to represent all the possible combinations. Thus, despite its conceptual simplicity, the use of nested switch/case structures results in a prohibitively large program (e.g. for $N = 256$, the program size exceeds 12 MB). In the next subsection we show how this obfuscation approach can be enhanced to yield a more practical program size.

4.2.2 The Use of Function Pointers

Function pointers are pointers that hold addresses of functions, and can be used to execute them. Depending on the address assigned to the pointer, a single function pointer can be used to call multiple functions. Normally, a designated array is used to hold the addresses of the functions and when a function is to be called, its address is assigned to the pointer and the function is executed. A visualization of function pointers is shown in Figure 7 where the array $fctArrJ[]$ is the designated array that holds the addresses of functions $jX(), jY(), jZ()$. As the code fragment shows, the address of the desired function in loaded into the function pointer $fctPtr$, and then executed. T. Ogiso *et al.* [45] analyze the use of function pointers in software obfuscation. Specifically, they prove that when using arrays of function pointers, determining the address a pointer points to is NP-hard.

Arrays of function pointers can be used to replace the inner switch/case structures in our obfuscation technique presented in the previous subsection. In addition to the security advantage resulting from the difficulty of determining the address a function pointer points to, implementing function pointers requires much less space than switch/case structures

described in the previous section. This enables us to maintain the complexity introduced by the nested switch/case structure while reducing the program size.

The use of function pointers as a replacement of the inner switch/case statement requires the following:

1. For each index j, there exists a function in which the variable that substitutes $S[j]$ is hard coded. Furthermore, the variable that substitutes $S[i]$ is passed as a parameter to this function. With these variables, the functionality of RC4 can be easily realized.

2. There exists a designated array that holds the addresses of the functions described above.

The array of function pointers can be directly used to replace the inner switch/case structures operating on index j. The inner switch/case statements are replaced by functions in which the variable representing $S[j]$ is hard coded. To evaluate the inner structure, we first compute the new j value. This is used to retrieve the corresponding function address from the designated array. Finally, the retrieved function is called using a function pointer and the variable that substitutes $S[i]$ is passed as a parameter.

Figure 8 shows the use of function pointers as a replacement of the inner switch/case structures of our obfuscated toy implementation of RC4. As shown in the figure, the loop over index i remains unrolled using the switch/case structure proposed initially. The setup for the array of function pointers requires:

1. The declaration of two function pointers ($*output$ and $*jN$)

2. The definition of output functions $oS0(), oS1(), os2()$ and $oS3()$ that return the value $S[x]$

3. The definition of functions $j0(sX), j1(sX), j2(sX)$ and $j3(sX)$ that replace the inner switch/case structure

4. The definition of arrays used to hold the addresses of the functions stated in steps 2 and 3 above

Next, we illustrate the combination of switch/case structures with array function pointers. The outer structure used to unroll the loop over the index i remains unchanged. However, the inner switch/case structure is replaced by using the function pointers concept. To do this, we first compute the new j value. This is used to retrieve the corresponding function address from the designated array. Finally, the retrieved function is called using a function pointer where the variable that represents $S[i]$ is passed as a parameter. With these variables, the functionality of RC4 can be realized. When implemented, this approach while improving the obfuscation level, significantly reduces the obfuscated program size. In comparison to the initial attempt (in section 4.2.1), the program size is reduced from 12 MB to about 450 KB.

The techniques used so far have mainly increased the resilience of the implementation against static analysis. We, next, introduce further obfuscation with the objective of increasing its resilience against dynamic analysis.

4.2.3 Multithreading

Traditionally, multithreading allows various parts of a program to run simultaneously. Each such part is called a thread and although these are functionally independent, they share some resources such as processing power and memory, with other threads. Shared resources, such as data, code, and heap segments allow communication and functional synchronization between threads. Furthermore, constructs such as *critical sections*, and *semaphores* enable the realization of atomic units which, in turn, prevent corruption of shared resources. Because of this parallelism and the randomness in order of execution, analyzing multithreaded programs is much harder than their single threaded counterparts [22]. In this section, we capitalize on this and utilize the randomness in the order of execution

introduced by multithreading to further obfuscate our implementation. To do so, we require the following:

1. There exists a multithreaded environment where each thread implements the RC4 functionality (key stream value) for a specific subset of index values i. That is, each thread contains an implementation of a subset of switch/case statement for the corresponding values of i. Furthermore, for each implemented switch/case statement, let the thread implement the function pointer concept for all values of j, as described in 4.2.2.

2. The sets of index values i are assigned to the threads such that each value of i is assigned randomly to at least two threads. This introduces randomness in the threads that have the capability of implementing the RC4 functionality for a given value of i. Since at least two threads have this capability, the execution path cannot be determined with certainty which introduces an additional layer of obfuscation.

If one uses only 2 threads, requirement 2 above would result in identical functionality for both threads, which simplifies conducting static analysis on the cipher. Thus, in our implementation, the minimum number of threads used is set to 3. This ensures that the implementations of various threads differ. The specific number of threads to be used is left as a design parameter.

To compute the key stream value for the current value of i, the running thread enters a critical section and retrieves i. If this thread does not implement the switch/case statement for this value of i, the critical section exits without affecting the cipher inner state and without producing any new keystream words. On the other hand, if the thread implements the switch/case statement for this value of i, the key stream value is evaluated and returned.

Figure 9 illustrates a toy implementation, with $N = 4$, of the multithreading implementation described above. In this example, the sequential index i can have the values

$\{0, 1, 2, 3\}$, and 3 threads are used. The first thread implements the switch/case statements for $i \in \{1, 2, 3\}$; the second thread implements the statements for $i \in \{0, 2, 3\}$, and the third thread implements the statements for $i \in \{0, 1, 3\}$. For standard RC4 parameters, this implementation increases the program size to 650 KB but offers an additional obfuscation layer and enhances the implementation's resilience to dynamic analysis attacks.

4.2.4 Handling the Key Scheduling Process

As shown in section 4.1.1, RC4 runs two main algorithms, the PRGA, and the KSA. While the implementation of the KSA can be obfuscated using the same techniques discussed above, one weakness of this approach is that the cryptographic key has to be passed in the clear to the KSA algorithm. In this section, we discuss a possible extension of the above implementation in order to mitigate this vulnerability.

In the white-box implementations of AES [20] and DES [19], the cryptographic key is integrated into the lookup tables of the implemented algorithms. Furthermore, the lookup tables are pre-created outside the users' environment. Applying this off-line generation technique to the inner states of RC4 can be used to eliminate the need for a KSA algorithm and consequently mitigate the vulnerability described above. This shifts our objective from protecting the key and key scheduling algorithm to protecting the process of securely assigning the off-line generated values to the inner-state.

Assume a setup where the user receives some encrypted data stream, and pre-created inner-state for the cipher from some service provider. In this case, the inner-state can be transferred from the provider to the customer in the form of an array. Instead of generating the inner-state by the KSA, the inner-state is initialized by directly assigning the values from the array to the corresponding variables. In other words, the array from the provider contains 256 values corresponding to S and the two index pointers *i* and *j* in a random order. Those 258 values are directly assigned to the variables representing the inner-state

on the customer side. It should be noted that as long as the order of the values in the array is not known, an adversary cannot gain any useful information about the inner state of the cipher. Furthermore, assuming that the service provider knows the memory structure of the user's cipher, the service provider can produce a formatted memory dump that can be loaded directly into the user's cipher.

To this point, our obfuscation approaches structurally altered the implementation of the cipher. Additional obfuscation techniques, that are deployable on a smaller scale can also be utilized. These techniques do not significantly change the implemented structure and are easily applied. In the next subsection, we briefly explore the application of such techniques to our RC4 implementation.

4.2.5 Generic Obfuscation Techniques

In this section, we introduce a set of standard techniques that can be used to further obfuscate our implementation of RC4. These techniques are independent of the structure of the implemented program and do not significantly change the structure of the resulting obfuscated program.

Order and Dimension Change of Arrays

When using arrays of function pointers, assigning the pointers to the array in a sequential order introduces a $1 : 1$ mapping between the j value and the index of the array. This mapping can be further obfuscated using a random allocation table or by modifying the array structure. In [55], Zhu, *et al.* address this problem by changing the index order and the arrays' dimension. Transforming an array $A[N]$ into an array $B[M]$, where$M > N$ and M is relatively prime to N, can be done by applying the mapping $B[i] = A[i \times N \bmod M]$. We have used this methodology to obfuscate the array of function pointers in our implementation.

Variable Aliases

When two or more variables address the same memory location, they are called aliases. Introducing aliases to a program reduces the effectiveness of static analysis techniques as they increase the data flow complexity [21] since the attacker has to identify and track all aliases that manipulate a specific memory location. The larger the program, the harder it is for the attacker to identify and keep track of all the aliases. The pointers used in our implementation are an extensive form of aliasing, and therefore, introduce an additional level of obfuscation.

Scattering the Code for the Swap Operations

The RC4 cipher makes extensive use of swapping in both the KSA and the PRGA algorithms. In a standard implementation, monitoring the swapping function easily reveals the position of j and consequently, its value, which compromises the security of the implementation. To address this problem, in our implementation, we scatter the steps of the swap functionality throughout the program. In addition, we use a pool of temporarily swap variables rather than a single variable.

Opaque Constructs

Opaque predicates are expressions that evaluate either to true or false upon a given condition, but their outcome is known/controlled in advance. These constructs introduce confusion and are widely used in obfuscation [21] [23]. Opaque predicates can be classified based on their possible outcome into two types. In the first type, the outcome is always either 'true' or always 'false'. An example of such predicate would be $j > 255$, which is always evaluated 'false' in an 8 bit environment. In the second type, the output could be either 'true' or 'false', but is controlled by adjusting the statement that it evaluates. To increase the complexity of control flow analysis, we implemented a similar approach where

either the real function or some other dummy one is executed.

Evaluation of the Index Pointer j

Normally, the index pointer j, at step i, is calculated as

$$j_i = j_{i-1} + S[i] \tag{8}$$

where j_i denotes the value of the index pointer j at iteration i and $S[i]$ denotes the value of the inner state array during the i^{th} iteration of the cipher. Monitoring the value of j_i, while knowing the value of i, allows reproducing the inner state. In our implementation, we obfuscate the computation of j_i by introducing three intermediate variables (a, b, c) that are initialized as follows:

$$\begin{aligned} b &= \text{random}, \\ a &= b - j_{i-1}, \\ c &= 2a - b - S[i+1]. \end{aligned}$$

Then we calculate the value of j_i as $j_i = a - c$.

4.3 Performance Evaluation

In this section we compare the execution costs (i.e., program size and execution time) for various combinations of obfuscation techniques proposed in the previous sections.

The reported timing, as summarized in Table 7, are measured on an HP PC with a quad core Intel 2.67 GHz processor, and 8 GB of RAM. The prototype was implemented in C using Microsoft Visual C++ that was running on Windows 7 Enterprise platform. As expected, obfuscated implementations impose a penalty on the resulting program size and execution speed.

RC4 Implementation options	Program Size (KB)	Throughput (KB/sec)	a	b	c	d
No obfuscation	8	288,700	-	-	-	-
Configuration 1	514	17,850	x	-	-	-
Configuration 2	518	15,450	x	x	-	-
Configuration 3	537	11,500	x	x	x	-
Fully obfuscated	969	600	x	x	x	x

Table 7: Program size and throughout for different obfuscation options

a: Loop unrolling over index pointer i, Array of function pointers, Variable aliases, Opaque constructs

b: Order and dimension change of arrays

c: Evaluation of index pointer j

d: Multithreading

From Table 7, the slowdown factor varies highly between the obfuscation configurations. For configuration 1, the slowdown factor is about 16, whereas the slowdown factor when implementing all the described obfuscation techniques is about 481. The slowdown factors for the white-box implementations of AES and DES found in the published literature are 55 for AES and 10 for DES [46] [41].

In the following subsections, we highlight the main causes of this performance impairment.

4.3.1 Multithreading

Although multithreading is typically used to enhance the performance of a program, in our implementation it is used only as an obfuscation technique. Multithreading, as implemented, significantly enhances the programs resilience against dynamic analysis attacks but it also slows down the resulting program because of the following reasons:

1. The threads are essentially executed in sequence as the key stream value is evaluated only within a critical section. Therefore, when this section is in use by a given thread,

other threads remain idle.

2. The use of a critical section introduces an additional overhead. Furthermore, as the design of RC4 dictates, only a single index can be evaluated at a time, which offers no room for parallelism.

3. Threads that do not implement the switch/case statement for the given value of the index i introduce an additional delay. These threads lock the critical section and consequently prevent other threads from operation and, yet, produce no keystream words.

4.3.2 Excessive Use of Context Switches

The proposed implementation extensively uses branch statements to switch between real and dummy functionality. This context switching introduces a delay since each switch/case structure is evaluated before the real functionality is executed. In addition, each function, whether dummy or real, introduces further delay as its parameters have to be pushed or pulled from the stack. Furthermore, due to the use of arrays of function pointers, in each iteration, two additional functions have to be handled.

4.3.3 Additional Calculations Overhead

Many mathematical calculations are required to obfuscate the value of j and the array indices. This includes the additional calculations used to obfuscate the index pointer j. Furthermore, when obfuscating the order of arrays by changing their dimensions, additional computations are used to select the correct index for the address of each function.

```
while(NOT_END_OF_DATA_STREAM)
{
    i++;
    Switch(i) {
        Case 0: //i = 0
            Switch(j) {
                j = (j + S0); //j = (j + S[i]) mod N;
                Case 0:   //j = 0
                          //swap(S[i], S[j]);
                          swap(S0, S0);
                          //Output S[S[i] + S[j] mod N];  → Output S[Output-Index];
                          Output-Index = (S0 + S0);
                Case 1: swap(S0, S1); Output-Index = (S0 + S1);
                Case 2: swap(S0, S2); Output-Index = (S0 + S2);
                Case 3: swap(S0, S3); Output-Index = (S0 + S3);
            } //end switch(j)
        Case 1: //i = 1
            Switch(j) {
                j = (j + S1); //j = (j + S[i]) mod N;
                Case 0: swap(S1, S0); Output-Index = (S1 + S0);
                Case 1: swap(S1, S1); Output-Index = (S1 + S1);
                Case 2: swap(S1, S2); Output-Index = (S1 + S2);
                Case 3: swap(S1, S3); Output-Index = (S1 + S3);
            } //end switch(j)
        Case 2: //i = 2
            Switch(j) {
                j = (j + S2); //j = (j + S[i]) mod N;
                Case 0: swap(S2, S0); Output-Index = (S2 + S0);
                Case 1: swap(S2, S1); Output-Index = (S2 + S1);
                Case 2: swap(S2, S2); Output-Index = (S2 + S2);
                Case 3: swap(S2, S3); Output-Index = (S2 + S3);
            } // end switch(j)
        Case 3: //i = 3
            Switch(j) {
                j = (j + S3); //j = (j + S[i]) mod N;
                Case 0: swap(S3, S0); Output-Index = (S3 + S0);
                Case 1: swap(S3, S1); Output-Index = (S3 + S1);
                Case 2: swap(S3, S2); Output-Index = (S3 + S2);
                Case 3: swap(S3, S3); Output-Index = (S3 + S3);
            } // end switch(j)
    } //end switch(i)
    Switch(Output-Index) {
        Case 0:  Output S0;//→ Output S[S[i] + S[j] mod N];
        Case 1:  Output S1;
        Case 2:  Output S2;
        Case 3:  Output S3;
    } //end switch(Output-Index)
} //end while
```

Figure 6: The implementation of the PRGA when replacing the array data structure by independent variables

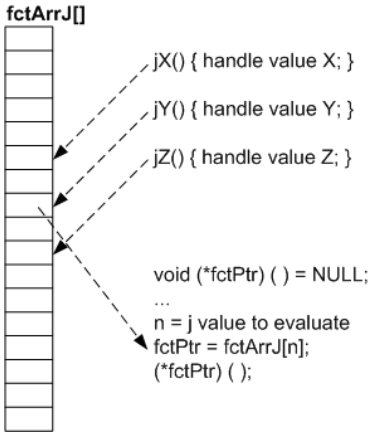

Figure 7: Array of Function Pointers

```
//Declaration of function pointers
unsigned int (*output)() = NULL; //for functions returning the value of S[x]
void (*jN)(unsigned int) = NULL; //for functions handling a specific value of index j

//function returning the value of S[x]
oS0() { return S0; }
oS1() { return S1; }
oS2() { return S2; }
oS3() { return S3; }

// functions handling a specific value of index j
// The variable representing S[j] is know in advance and hardcoded
// The variable that handles S[i] has to be passed as function parameter to sX
J0(sX) { swap(S0,sX); output = arrayOutput[S0+sX]; Output output(); }
J1(sX) { swap(S1,sX); output = arrayOutput[S1+sX]; Output output(); }
J2(sX) { swap(S2,sX); output = arrayOutput[S2+sX]; Output output(); }
J3(sX) { swap(S3,sX); output = arrayOutput[S3+sX]; Output output(); }

// Assign addresses of the functions to the arrays
// Access to the functions via its position in the array
arrayJN[N] = {&J0, &J1, &J2, &J3};
arrayOutput[N] = {&oS0, &oS1, &oS2, &oS3};

while(NOT_END_OF_DATA_STREAM)
{
    i++;

    Switch(i){

        Case 0:   //i = 0
                  j = (j + S0); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S0); //Execute function via the function pointer

        Case 1:   //i = 1
                  j = (j + S1); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S1); //Execute function via the function pointer

        Case 2:   //i = 2
                  j = (j + S2); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S2); //Execute function via the function pointer

        Case 3:   //i = 3
                  j = (j + S3); //j = (j + S[i]) mod N;
                  jN = arrayJN[j]; //Select function address from array
                  jN(S3); //Execute function via the function pointer

    } // end switch(i)

} //end while
```

Figure 8: Implementation of the PRGA using switch/case for i and array of function pointer for j

```
Thread1() {
    while(NOT_END_OF_DATA_STREAM)
    {
        EnterCriticalSection; //enter the coherent, not interruptible code sequence
        Switch(i) {
            //Case 0: i = 0 is not handled in this thread
            Case 1:  //i = 1
                     j = (j + S1); //j = (j + S[i]) mod N;
                     jN = arrayJN[j]; //Select function address from array
                     jN(S1); //Execute function via the function pointer
                     i++;
                     //Leave the coherent, not interruptible code sequence
                     LeaveCriticalSection;
            Case 2:  //i = 2
                     j = (j + S2); //j = (j + S[i]) mod N;
                     jN = arrayJN[j]; //Select function address from array
                     jN(S2); //Execute function via the function pointer
                     i++;
                     //Leave the coherent, not interruptible code sequence
                     LeaveCriticalSection;
            Case 3:  //i = 3
                     j = (j + S3); //j = (j + S[i]) mod N;
                     jN = arrayJN[j]; //Select function address from array
                     jN(S3); //Execute function via the function pointer
                     i++;
                     //leave the coherent, not interruptible code sequence
                     LeaveCriticalSection;
            default: //Handles the non-available 'cases'
                     //Leave the critical section
                     LeaveCriticalSection;
        } // end switch(i)
    } //end while
} //end Thread1

Thread2() {
    //same structure like Thread1(), but
    //Case 0: i = 0 is handled
    //Case 2: i = 2 is handled
    //Case 3: i = 3 is handled
    //and
    //Case 1: i = 1 is not handled
}

Thread3() {
    //same structure like Thread1(), but
    //Case 0: i = 0 is handled
    //Case 1: i = 1 is handled
    //Case 3: i = 3 is handled
    //and
    //Case 2: i = 2 is not handled
}
```

Figure 9: Implementing the PRGA using multithreading

Chapter 5

Conclusion and Future Work

5.1 Summary and Conclusions

Many computer systems and applications store working copies of sensitive data in the memory where they remain in cleartext even after their usage. Storing sensitive data unprotected in the memory makes them prone to attacks in scenarios where attackers have access to the memory content or even a corrupted version of this content.

In the first part of our work, we investigated the suitability of two off-the-shelf algebraic tools (CryptoMiniSat and PolyBoRi) for extraction of cryptographic keys from corrupted memory images. Based on our experimental results, it is clear that while the CryptoMiniSat requires a slightly longer preprocessing step to prepare its input file, this step is done only once and the tool runs much faster than the Gröbner basis tool, PolyBoRi. Furthermore, CryptoMiniSat does not require a large amount of memory during run time. However, if several solutions were possible for the SAT problem in question, only one result is returned by the solver and the additional solutions have to be explicitly searched again by re-running the tool after appending some extra constraints to exclude already found solutions. On the other hand, Gröbner basis returns a general form representing all possible solutions. However, PolyBoRi requires large memory and usually crashes when the memory requirements

is exceeded which limits its applications for solving large problems. It should also be noted that, given the high redundancy of the key schedules of the considered ciphers, the advantage of being able to return all possible solutions does not seem to be very significant since in all the instances we considered, only one possible solution exists.

In the second part of our work, we investigated and compared different practical obfuscation techniques for the protection of the RC4 implementations in the white-box attack model. Our obfuscated implementation of RC4 provides a high degree of resiliency against attacks from the execution environment where the adversary has access to the software implementation such as digital right management applications. Furthermore, while the focus of this work was RC4, these techniques can be applied to other array-based stream ciphers such as HC-128, HC-256 and GGHN.

5.2 Future Work

While the use of algebraic tools automate the process of solving the system of equations required by the cold boot attack process, producing and pre-processing of these equations are still very tedious tasks that are prone to errors if performed by a non-specialist. Automating this part of the process using a tool with a suitable GUI (e.g., one that allows the user to draw the key schedule using drag-and-drop of different components such as s-boxes, bit rotation, bit shift, bit permutation, XOR module, linear transformation, modular addition, modular subtraction, modular multiplication, finite field operations as well other possible user defined modules) would be of a great help to the forensics community.

Providing a white box implementation with more theoretical foundations for different cryptographic primitives, including RC4, is an interesting research problem. Exploring how to use multithreading to speed up the obfuscated RC4 implementation is another challenging research problem.

Bibliography

[1] Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard (AES). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[2] Boolean Polynomials. Sage Reference Manual V4.7.2. http://www.sagemath.org/doc/reference/sage/rings/polynomial/pbori.html, Accessed April 2012.

[3] C++ Obfuscator - Obfuscate C and C++ Code. http://www.stunnix.com/prod/cxxo/overview.shtml, Accessed April 2012.

[4] CryptoMiniSat. http://www.msoos.org/cryptominisat2/, Accessed April 2012.

[5] PolyBoRi. http://polybori.sourceforge.net/, Accessed April 2012.

[6] SecuriTeam - Shiva, ELF Encryption Tool. http://www.securiteam.com/tools/5XP041FA0U.html, Accessed April 2012.

[7] Skype's Obfuscated RC4 Algorithm Was Leaked. http://www.reddit.com/r/technology/comments/cn4gn/skypes_obfuscated_rc4_algorithm_was_leaked_so_its/, Accessed April 2012.

[8] TRESOR Runs Encryption Securely Outside RAM. http://www1.informatik.uni-erlangen.de/tresor, Accessed April 2012.

[9] TRUECRYPT, Free Open-Source On-The-Fly Encryption. http://www.truecrypt.org/, Accessed April 2012.

[10] Twofish. http://www.schneier.com/twofish.html, Accessed April 2012.

[11] UPX: the Ultimate Packer for EXecutables. http://upx.sourceforge.net/, Accessed April 2012.

[12] M. R. Albrecht and C. Cid. Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In J. Lopez and G. Tsudik, editors, *ACNS 2011*, volume 6715 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2011.

[13] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. http://www.cl.cam.ac.uk/~rja14/serpent.html, Accessed April 2012.

[14] P. Biondi and F. Desclau. Silver Needle in the Skype. http://www.secdev.org/conf/skype_BHEU06.pdf, Accessed April 2012.

[15] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT'97*, pages 37–51, 1997.

[16] M. Brickenstein and A. Dreyer. PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. *Journal of Symbolic Computation*, pages 1326–1345, Sep. 2009.

[17] B. Buchberger. *Gröbner-Bases: An Algorithmic Method in Polynomial Ideal Theory.*, chapter 6, pages 184–232. Reidel Publishing Company, Dodrecht - Boston - Lancaster, 1985.

[18] J. Buchmann, A. Pyshkin, and R.-P. Weinmann. Block Ciphers Sensitive to Gröbner Basis Attacks. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2006.

[19] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. A White-Box DES Implementation for DRM Applications. In J. Feigenbaum, editor, *Digital Rights Management Workshop*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.

[20] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot. White-Box Cryptography and an AES Implementation. In K. Nyberg and H. M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.

[21] C. S. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison Wesley, 2010.

[22] C. S. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, 1997.

[23] C. S. Collberg, C. D. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL*, pages 184–196, 1998.

[24] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158. ACM, 1971.

[25] N. Courtois and G. Bard. Algebraic Cryptanalysis of the Data Encryption Standard. In S. Galbraith, editor, *Cryptography and Coding*, volume 4887 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2007.

[26] N. Courtois, G. V. Bard, and D. Wagner. Algebraic and Slide Attacks on KeeLoq. In K. Nyberg, editor, *FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.

[27] N. Courtois, S. O'Neil, and J.-J. Quisquater. Practical Algebraic Attacks on the Hitag2 Stream Cipher. In P. Samarati, M. Yung, F. Martinelli, and C. Ardagna, editors, *Information Security*, volume 5735 of *Lecture Notes in Computer Science*, pages 167–176. Springer, 2009.

[28] N. T. Courtois, K. Nohl, and S. O'Neil. Algebraic Attacks on the Crypto-1 Stream Cipher in MiFare Classic and Oyster Cards. Cryptology ePrint Archive, Report 2008/166, 2008.

[29] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[30] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion Attacks on Secure Hash Functions Using satSolvers. In J. Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 377–382. Springer, 2007.

[31] T. Eibach, E. Pilz, and G. Völkel. Attacking Bivium using SAT solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 63–76. Springer, 2008.

[32] J. Erickson, J. Ding, and C. Christensen. Algebraic Cryptanalysis of SMS4: Gröbner Basis Attack and SAT Attack Compared. In D. Lee and S. Hong, editors, *Information,*

Security and Cryptology - ICISC 2009, volume 5984 of *Lecture Notes in Computer Science*, pages 73–86. Springer, 2010.

[33] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In P. C. van Oorschot, editor, *USENIX Security Symposium*, pages 45–60. USENIX Association, 2008.

[34] N. Heninger and H. Shacham. Reconstructing RSA Private Keys from Random Key Bits. In S. Halevi, editor, *CRYPTO'09*, volume 5677 of *LNCS*, pages 1–17. Springer, Aug. 2009.

[35] H. M. Heys. A Tutorial on Linear and Differential Cryptanalysis. Technical report, Centre for Applied Cryptographic Research, Department of Combinatorics and Optimization, University of Waterloo, 2001.

[36] M. Joye, A. K. Lenstra, and J.-J. Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *J. Cryptology*, pages 241–245, 1999.

[37] A. Kamal and A. Youssef. Applications of SAT Solvers to AES Key Recovery from Decayed Key Schedule Images. In *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on*, pages 216 –220, July 2010.

[38] B. Kaplan. RAM is Key, Extracting Disk Encryption Keys From Volatile Memory. Master's thesis, Carnegie Mellon University, May 2007.

[39] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[40] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[41] H. E. Link and W. D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box Encoding. In *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, 2005.

[42] C. Maartmann-Moe, S. E. Thorkildsen, and A. Årnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, pages 132–140, 2009.

[43] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[44] I. Mironov and L. Zhang. Applications of SAT Solvers to Cryptanalysis of Hash Functions. In A. Biere and C. P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[45] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, pages 176–186, 2003.

[46] J.-Y. Park, O. Yi, and J.-S. Choi. Methods for practical whitebox cryptography. In *Information and Communication Technology Convergence (ICTC), 2010 International Conference on*, pages 474–479, Nov. 2010.

[47] A. Segers. Algebraic Attacks from a Groebner Basis Perspective. Master's thesis, Technische Universiteit Eindhoven, Oct. 2004.

[48] A. Shamir and N. van Someren. Playing Hide and Seek With Stored Keys. In M. Franklin, editor, *Financial Cryptography*, volume 1648 of *Lecture Notes in Computer Science*, pages 118–124. Springer, 1999.

[49] S. Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, 2002.

[50] M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic Cryptanalysis of 58-Round SHA-1. In A. Biryukov, editor, *FSE 2007*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[51] A. Tsow. An Improved Recovery Algorithm for Decayed AES Key Schedule Images. In M. Jacobson, V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 215–230. Springer, 2009.

[52] V. Velichkov, V. Rijmen, and B. Preneel. Algebraic cryptanalysis of a small-scale version of stream cipher Lex. *Information Security, IET*, pages 49–61, June 2010.

[53] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical Report CS-2000-12, Univ. of Virginia, 2000.

[54] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In V. Shoup, editor, *CRYPTO'05*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2005.

[55] W. Zhu, C. D. Thomborson, and F.-Y. Wang. Obfuscating arrays by homomorphic functions. In *IEEE International Conference on Granular Computing*, pages 770–773, 2006.