Executive Summary

Side channel cryptanalysis is a technique that can be used to break cipher algorithms by exploiting side channel information of the algorithm implementation. Side channel information can be any pattern of data leakage from the working system depending on the secret key of the cipher. In multitasking systems, one source of data leakage can come from CPU micro-architectural shared components such as cache memory and branch predication unit. There are many ways to exploit these shared components in order to discover the secret key. One technique is running a malicious program to spy on the footprints of the victim (cipher) program in one of these shared micro-architectural components. This malicious program is often called micro architectural spy process. It is believed that this model is practical and can be used to breach the security of the cryptosystems even for the most today's strong ciphers such as Advanced Encryption Standard (AES). Therefore, there is a growing demand to stop breaching cryptosystems particularly in those environments which are believed to be most vulnerable for such attacks such as cloud computing.

In this thesis, I have alleviated the danger of spy process for cache memory micro-architectural component through the following points:

1. I analyzed the most relevant CPU assembly instructions in x86 architecture and the behavior of cache based spy process in practice.
2. I devised two novel detection methods to classify cache based spy process. I named these methods: Counting Method and Program Flow Method.
3. I implemented one of these methods which is Counting Method.
4. I integrated the implemented version with Bash shell GNU 4.2 source code to provide a demonstration for an Intrusion Detection System (IDS) for cache based spy process.
5. I evaluated the accuracy and performance for the produced detection system.
6. I introduced two examples for potential countermeasures that can be used to bypass the produced detection system.
7. I introduced possible mitigations for these countermeasures.

Acknowledgment

I would like to acknowledge and express my heartfelt thanks for the following persons who have supported me during my research and have provided me with valuable advice and continuing encouragement. Without him, I could not have completed this project:

- My supervisor Dan Page for his vital ideas, notes, corrections and the great guidance.
- My family for their regular support and prayers.
- My friends, for their encouragement and great advice.
- My flat mates in Winkworth House, for their essential comments particularly for the demo of the poster day.

And to God, who made all things possible.

Contents

Chapter 1. Introduction 6
1.1 Aim and Objectives 7
1.2 Added Value 7
1.3 Thesis Scope 8
Chapter 2. Technical Basis 9
2.1 Computer Architecture 9
2.1.1 Cache Memory 9
2.1.2 Brach Predication Unit 11
2.2 Process 12
2.3 Cryptographic Primitives 15
2.3.1 Advanced Encryption Standard 15
2.3.2 RSA algorithm 18
2.4 Cache Based Side Channels 19
2.4.1 Cache Attacks Strategies 20
2.4.2 OST Synchronous Attacks 21
2.4.3 OST Asynchronous Attacks 23
2.5 Branch Prediction Side Channels 24
2.5.1 Deterministic Branch Prediction Attack 24
2.5.2 Synchronous Attack 25
2.5.3 Asynchronous Attack -Eviction 25
2.5.4 Asynchronous Attack -Trace Driven 26
2.6 Malware Detection System 26
2.6.1 Automatic Learning Approach 27
2.6.2 Manual Learning Approach 27
2.6.3 Disassembly Algorithms: 28
2.6.4 Code Analysis 29
2.7 Evaluation Criteria 30
2.7.1 Accuracy 30
2.7.2 Performance 35
Chapter 3. Design and Implementation 36
3.1 Disassemble CBSP 36
3.2 Analyze CBSP 37
3.2.1 Reading Time Stamp Counter 37
3.2.2 Serialized Instruction 38

3.2.3 No Cache Access 39
3.2.4 Less System calls 39
3.3 CBSP Signature based Detection Methods 40
3.3.1 Counting based Method 40
3.3.2 Program Flow Method 41
3.4 Implementation 42
3.4.1 Version I 42
3.4.2 Version II 43
3.4.3 Kernel Space Version 44
Chapter 4. Evaluation and Discussion 46
4.1 Accuracy 46
4.2 Performance 48
4.3 Countermeasures 48
4.3.1 Code Obfuscation 48
4.3.2 Fake System Calls 49
4.4 Other Approaches 50
Chapter 5. Conclusion 51
5.1 Future Work 52

Figures

Figure 1.1: Side Channel Cryptanalysis 6
Figure 2.1: 2- Way Set Associative Cache Structure 11
Figure 2.2: Process Statuses Diagram 14
Figure 2.3: Protection Rings 14
Figure 2.4: SubBytes 16
Figure 2.5: ShiftRows 16
Figure 2.6: MixColumns 17
Figure 2.7: AddRoundKey 17
Figure 2.8: Illustration for Synchronous OTS prime+ probe cache attack 23
Figure 2.9: ROC Space Analysis 34
Figure 3.1: Part of the disassembled of CBSP 37
Figure 3.2: Secure Terminal 44

Tables

Table 2.1 Process Classification Classes 31
Table 2.2: Confusion Matrix for Malware Classifier. 32
Table 4.1: Counting Method Accuracy Results over 2844 binaries in Ubuntu 10.04 with $\boldsymbol{\alpha 1 = 2,\ \alpha 2 = 1,\ \beta = 3}$ 47
Table 4.2: Counting Method Accuracy Results over 2665 binaries in Windows Vista with $\boldsymbol{\alpha 1 = 2,\ \alpha 2 = 1,\ \beta = 3}$ 47
Table 4.3: Counting Method Accuracy Results over 5509 binaries in Ubuntu 10.04 and Windows Vista with $\boldsymbol{\alpha 1 = 2,\ \alpha 2}$ 49

Chapter 1. Introduction

There are two broad approaches for cryptanalysis: the first approach is based on analyzing the theoretical weaknesses in the structure of the cipher algorithm; the second approach is analyzing the weaknesses in the physical implementation of the cryptosystem. The latter one is called Side Channel Cryptanalysis. In this approach, the attack occurs based on "Side Channel Information" which can be retrieved from the device in which the encryption or decryption process is running. Examples of side channel information are timing information, power consumption and electromagnetic waves. Figure 1.1: Side Channel Cryptanalysis shows the basic idea of side channel attacks.

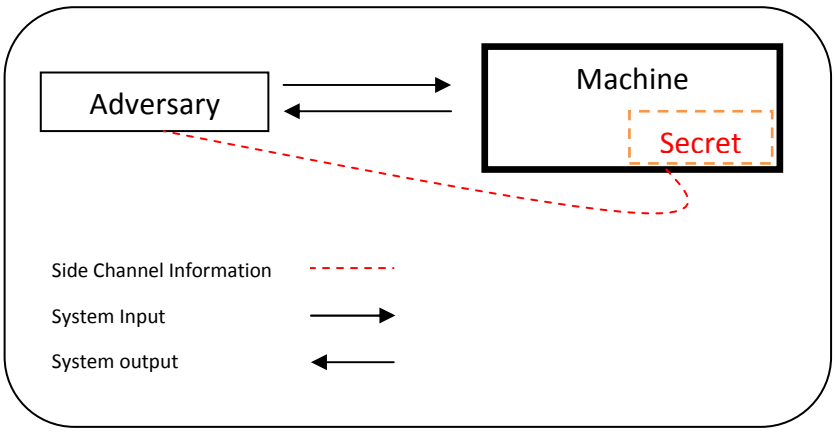

Figure 1.1: Side Channel Cryptanalysis

Recent studies have showed that there is a way to perform side channel cryptanalysis based on leaking information from Micro-Architectural (MA) components of processors such as Cache memory and Branch Prediction Unit (BPU). The risk of MA side channel was pointed out more than 20 years ago in [1]. The author considered the possibility of leaking unprivileged data and establishing a covert channel using cache memory. One model to which this kind of cryptanalysis can be applied is the examination of footprints on the operations that are preformed during the execution. A typical way to do this is by running a

process in parallel a so-called spy process with the target process (encryption or decryption process). The spy process then traces the footprints on MA components and obtains information that can lead to security breach. As we will be seen, there are many attacks that have succeeded in breaking the security and recovering the secret key using this model of spy process.

1.1 Aim and Objectives

Since there is an increase demand to stop spy process based attacks, the aim of this project is to alleviate the danger of spy process by introducing an Intrusion Detection System (IDS). This system should be able to capture the malicious process (spy process) that attempts to trace the footprints on MA components of another process (target process). The objective of this project is the following:

1. Provide a detailed overview of the most recent MA attacks that use inter process leakage model i.e., attacks that use spy process to reveal unprivileged data.
2. Survey various types of detection approaches that are used to classify malicious processes (malware).
3. Devise a novel detection method to classify spy processes.
4. Evaluate the accuracy and performance of the new detection method.
5. Optionally, provide potential strategies that could be used to prevent the detection functionality.

1.2 Added Value

Introducing intrusion detection system that classifies MA spy process is a novel work. Until the time of writing this report, we did not find any hint indicate that this kind of intrusion detection system has been introduced before.

Since there is a grown demand to stop information leakage through MA components, the potential detection system is expected to attract several big markets. The first market is the Anti-virus software companies given that there is more than 12 million people are suffering from malicious attacks [2]. Deploying such detection system will increase the overall security level of their customers in particularly those customers who will be affected negatively if there is a security breach has occurred in their systems such as financial organizations. The second market is the cloud computing. According to [3], the adversary can leak information from another process once he is successfully be placed co-resident with a target virtual machine in the cloud. This occurs because of the sharing of MA resources between the two virtual machines.

1.3 Thesis Scope

In this section, we will be presenting the structure of this thesis and topics that will be covered.

First, this project overlaps with many different fields. Therefore, we have attempted to cover all these fields and have a fair knowledge for each of them with reasonable level of details. The first field is the processor architecture and its components. It is not realistic to go through all processors components during this project. However, we have selected two examples of these components which are used in MA side channel attacks. These components are cache memory and Branch Predication Unit BPU and they will be explained in 2.1. In addition, the concept of the process and examples of cryptographic algorithms will be introduced in 2.2 and 2.3 respectively.

Since the ultimate goal is to introduce a mechanism to detect example of MA spy process, it is necessary to understand and analyze the behavior of spy process by studying different examples of MA side channel attacks based on inter-process leakage. In section 2.4 and section 2.5, several examples for both cache memory and BPU MA side channels attacks will be presented.

Since spy process can be treated as a malicious program, it is essential to examine different approaches and techniques that is often used in malware detection systems. Thus different approaches and techniques used in malware detection systems will be explained in section 2.6 with the evaluation criteria that should be kept in mind.

In chapter 3, the design and the implementation of the introduced detection system will be presented. This will include an explanation for various stages before the system is implemented. In addition, two versions of the detection systems will be provided.

The evaluation of the produced detection system will be presented in chapter 4. Furthermore, we will critically analysis the introduced system and will be providing examples of approaches that can be used, by the adversary, to bypass the detection system. Moreover, we will discuss the possibility of employing other approaches for developing MA detection algorithms. At the end, a conclusion will be provided in chapter 5 with some ideas for an extended work.

Chapter 2. Technical Basis

In this chapter, we will be presenting some introduction for various topics which are related for the present project. One can note that the major characteristic in this project requires a broad knowledge for various computer science branches. It overlaps with many areas in computer architecture, operating systems, cryptography and computer security. The most relevant topics will be introduced in this chapter.

2.1 Computer Architecture

Computer architecture is a very large topic. It witnesses a dramatic improvement over the last decades and this improving is expecting to be continuing in the future producing a significant influence in our life. There is considerable number of research that has been introduced new design, algorithms or components led to noteworthy improvement in the overall computer systems. In this section, we will be focusing on some terminologies in the computer architecture that are mostly relevant to the present project. They are: cache memory, branch prediction unit and process.

2.1.1 Cache Memory

In the earliest days of microprocessor system, system designers and engineers noticed that the main memory has high latency to retrieve data to the CPU. Thus, the demand of having a high speed memory works with CPU has been arisen to reduce the bottleneck between CPU and bus bandwidth. It was believed that, this demand will be continuing to be increased. Therefore, the CPU cache memory was introduced to solve this problem.

Cache memory is a fast memory area which stores copies of data that is frequently used or needed by CPU in the near future. It is designed to be mapped to the main memory in a way so that the CPU can load or store data from or to the main memory in a very short time. They are three mapping technique can be used to map the main memory with the CPU cache:

1. **Directed Mapping.** In this technique, each memory location in the main memory mapped only to one location in the cache. The common way to do this is using appropriate function with Cache size modulo. For example f(x) = x mod <Cache Size>. The negative side of this technique is the less flexibility in terms of where data can be resided in the cache.

2. **Fully Associative Mapping.** This technique is very complex but much flexible. The new entry of the cache can be placed anywhere. If cache is full, then the replacement algorithm decides which current entry in the cache should be evicted.

3. **Set Associative Mapping.** This technique is in the middle between the previous techniques. For every address in the memory, there is a set of addresses in the cache on which can be placed. The number of addresses for one set is called degree of associatively. More degree we have, more flexibility will get.

The majority of cache memories structure is designed according to Set Associative Mapping. The structure of this mapping is as follows. Cache is split into S sets of caches. In each set, there is a specific number L of a cache line or block. Some textbooks termed this lines “ways of cache”. Each line consists of B data bytes. Hence, the total cache size can be calculated by S*L*B bytes. Date word is stored only in one set which is determined by the LSB of the main memory address. When data is requested by CPU, the corresponding set in the cache only will be searched. Thus, one can conclude that the data should be placed in one of the L lines, given that data is already exist in the cache. Figure 2.1, which is quoted from [4], shows the general structure of 2- ways Set Associative Mapping.

There are two cases with regard to the existence of data exist in the cache memory: c*ache hit* and *cache miss*. When data is needed by the CPU, it checks cache first and see if the data is exist there. If data is already found in the cache, then CPU will read and use it directly. This case is called *cache hit*. On the other hand, if data is not found in cache, then CPU will request this data from the main memory and fetch a copy for its use. This case is called *cache miss*. The retrieved copy will be kept in cache hopefully to be wanted and used by CPU in the near future.

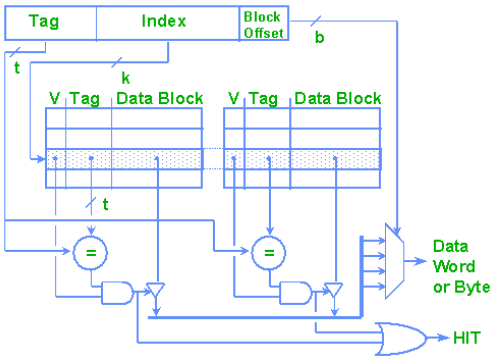

Figure 2.1: 2- Way Set Associative Cache Structure

The Cache memory can be on-chip or off-chip. The former one is very fast but due hardware limitation is very small and expensive while the latter is larger and less expensive but with less speed. In some architecture, cache is designed to be in multiple levels in order to balance between the performance and the cost.

2.1.2 Brach Predication Unit

Throughout the instruction stream of a program, a branch instruction is a point such that the next instruction is not necessary the executed instruction [5]. There are two types of branch instructions. The first one is unconditional branch such as goto statement. The second one is the condition branch such as for or while loop and if-else condition clause. In the conditional branch, the decision to take (execute) the branch depends on certain conditions to be evaluated or calculated during the execution time. In pipelined microprocessors, jumping from small branches to another may kill the performance.

One approach to circumvent the issue of conditional branch is to predict the most likely branch or path to be executed and to which target (address) the branch will go if it was taken. Therefore, a set of algorithms to predict the branch predictor and branch target are introduced. The component that is implementing these algorithms is Branch Predication Unit BPU. It is a micro architectural component responsible to predicate the most likely branch or path to be executed and to which address the branch will go if it was taken. It is beneficial for different types of microprocessors particularly superscalar processors which is a high performance CPU that has the ability to initiate several instructions simultaneously i.e. during the same clock cycle.

When a program reaches a branch instruction point, there will be two cases. Either the predication is taken or is not taken. If the prediction has been taken, then a considerable number of cycles will be saved. If the predication has not been taken, the microprocessor will pay a penalty of mispredication. It is clear that the predication algorithm plays an essential role to avoid the *mispredication* which is resulting in increasing the clock cycles.

The addresses that have been taken are placed in special buffer which is called *Branch Target Buffer (BTB)*. Keeping the target addresses in BTB will make these addresses be executed without any need to be fetched for the next time if they needed. However, since BTB has limited size, there is no guarantee that these addresses will be kept for long time.

2.2 Process

The term process is fundamental to any Operating System OS. It can be defined as an instance of a program in execution [6]. It contains the program machine code and the needed data for the execution. In some operating systems, the process can be made up of multiple threads which execute instructions concurrently.

Multitasking OS is a system which is capable to perform more than one task (process or thread) during the same period of time. In fact, in a single core CPU, there is no way to execute more one CPU instruction at the same time. However, in this case, concurrent executions for processes are done in a way that gives the appearance of executing multiple tasks simultaneously. In multitasking OS, it is essential to allocate resources in order to be used by processes. Due to the limitation of these resources, multitasking OS needs to share these resources among processes. Therefore, processes share common resources such as CPU and main memory.

The OS controls executing multiple processes in the CPU by a scheduler. The scheduler usually applies particular scheduling algorithm. The algorithm determines which process should be executed in the CPU and which one should be paused and resumed later. In order to apply a smooth switching among processes, OS have to maintain a particular data structure to hold the state or context of each process in CPU. This data structure is usually called Process Control Block (PCB) and the process of switching a CPU from one task (process or thread) to another one is called context switch.

There are usually five primary statuses for each process in multitasking OS and seven statuses in OS that supports virtual memory. These statuses are needed to increase the efficiency of multitasking operations. For example, a CPU may be occupied by a process that is waiting for some input for a user. In this case, it will be much efficient to block this process and start or resume another process from processes queue. These statuses are as follows:

1. **Created or New**: The process is created by being loaded from the secondary storage device and placed into the memory. In this state, the process should have been assigned a Process ID PID.

2. **Ready or Waiting**: The process is being awaiting the execution on a CPU.

3. **Running**: The process is being executed on a CPU.

4. **Blocked**: The process being awaiting particular event such as input/output operation or specific signals.

5. **Suspended Ready**: The ready process is being moved from the main memory to the virtual memory.

6. **Suspended Blocked**: The blocked process is being moved from the main memory to the virtual memory.
7. **Terminated**: The process is exit either after completing the task or has been killed by OS.

Figure 2.2: Process Statuses Diagram shows a state diagram that illustrates these different process statuses.

There are many modes of CPU execution for any process. Most CPUs have at least two modes of process execution which are restrictive or user mode and unrestrictive or kernel mode. The latter one allows the CPU to execute any instruction including the ones that are communicating directly with the hardware. On the other side, in the unrestrictive mode, not all instructions are allowed to be run on the CPU. The user space code can only call an instruction of the kernel space through the system calls. This hierarchy is called the Protection Rings and it aims to protect the system and data from faults and malicious behavior. Figure 2.3 shows the hierarchy of protection rings. Code from the least privileged rings can access code of the most privileged ring through a specific system call.

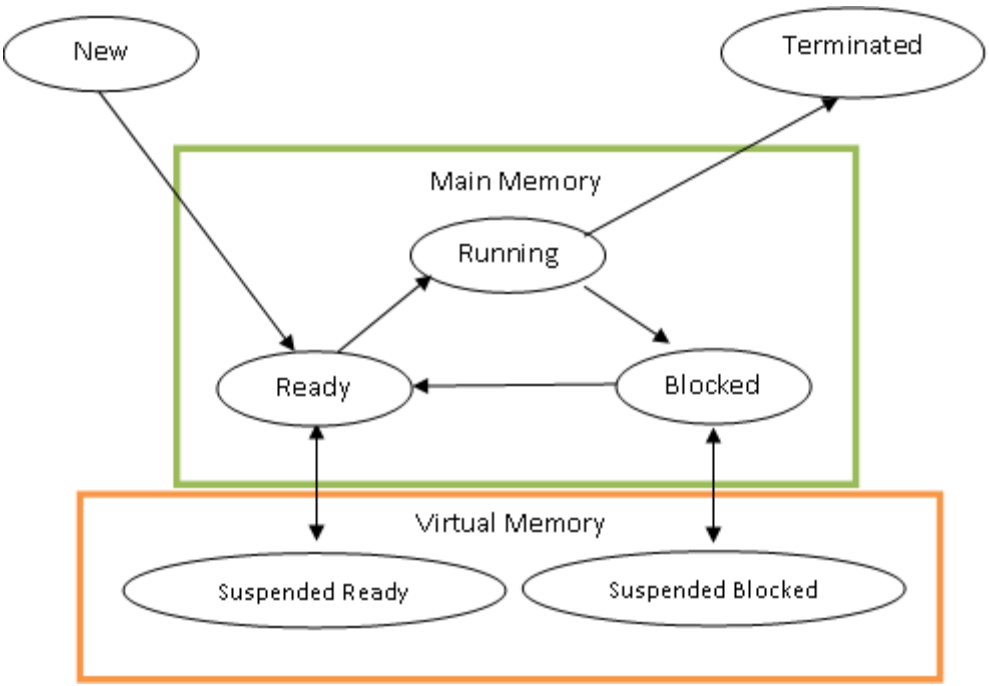

Figure 2.2: Process Statuses Diagram

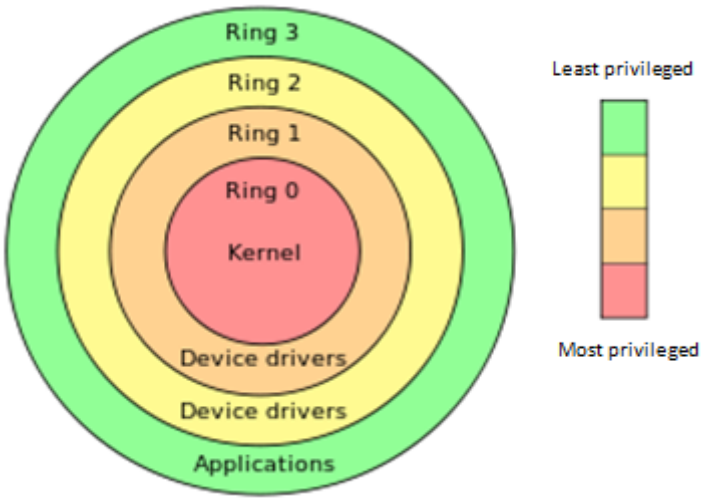

Figure 2.3: Protection Rings

2.3 Cryptographic Primitives

In this section, some cryptographic algorithms will be presented. These algorithms are provided as an example of cipher algorithms that have been broken by the side channel techniques such as cache based side channel attacks and branch prediction side channel attacks.

2.3.1 Advanced Encryption Standard

In 1997, National Institute of Standard and Technology (NIST) in the United States announced an international competition to replace their previous encryption standard, Digital Encryption Standard (DES), with a new one which they called Advanced Encryption Standard (AES). In 2002, Rijndael which is a cipher of the two Belgium cryptographers Joan Daemen and Vincent Rijmen, has won the competition and their cipher has become the Advanced Encryption Standard. Since that time AES has been implemented in many systems and platform and become much popular cipher worldwide.

AES is a block cipher meaning that it encrypts and decrypts the data in known fixed blocks size. The block size is 128 bits. AES is a symmetric key cipher meaning that the encryption and the decryption algorithms have only one key. Thus, this key should be secret. AES key size can be 128, 192, 256 bits.

There are four basic operations in AES. They are: AddroundKey, SubBytes, ShiftRows and MixColumns. These operations are as follows.

- **AddRoundKey**: is a normal binary XOR operation with the round's sub-key. The sub-keys are driven from the actual key by Rijndael's key schedule process.

- **SubButes**: is a byte substitution process with a pre-defined lookup table. This table can be generated during the run time or it can be placed static in the memory. Choosing which approach should be used depends on the available memory and needed performance.

- **ShiftRows**: is a process of permutation in the byte level. Bytes are placed in 4X4 matrix and each row is cyclically shifted with a pre-defined number.

- **MixColumns**: is a specific matrix multiplication operation with a pre-defined vector.

Figure 2.4, Figure 2.5, Figure 2.6 and Figure 2.7, which are quoted from [7], show diagrams for the basic operations SubBytes, ShiftRows, MixColumns and AddRoundKey respectively.

The high level description for the encryption process is as follows:

1. Key Schedule to generate the subkeys.

2. First Round:
    a. Addround Key: the plaintext with the actual key

3. Middle Rounds: Iterating the following operations 9, 11 or 13 times depending on the key size 128, 192 or 256 respectively.
    a. SubBytes
    b. ShiftRows
    c. MixColumns
    d. AddRoundKey

4. Final Round: it same as the middle rounds but with no MixColumns operation. It is as follows:
    a. SubBytes
    b. ShiftRows
    c. AddRoundKey

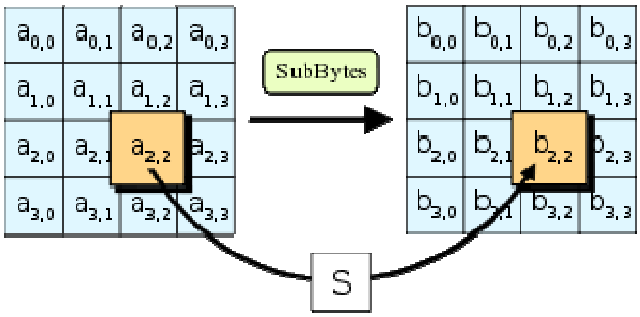

Figure 2.4: SubBytes

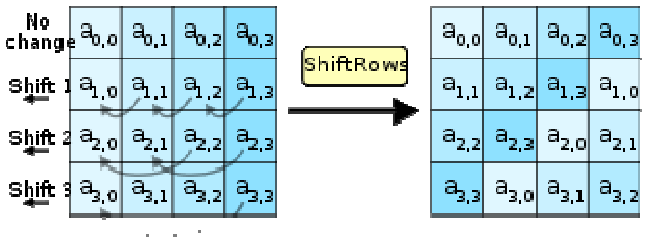

Figure 2.5: ShiftRows

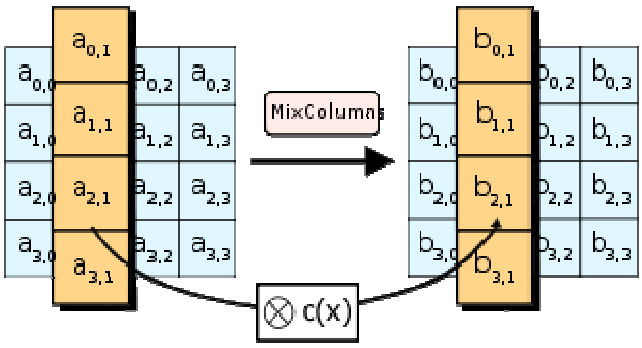

Figure 2.6: MixColumns

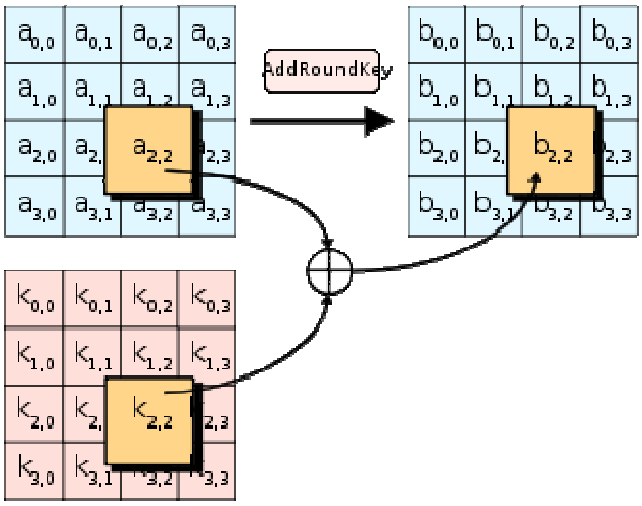

Figure 2.7: AddRoundKey

They are two general ways to implement AES. We can name them: the straightforward implementation and the lookup table implementation.

1. **Straightforward Implementation**: is implementing the aforementioned steps of the encryption in the same way as it described above. This implementation is suitable in 8 bits processors such as different types of smart cards. This implementation does not suffer from the cache time side channel attacks which we will be presenting in the next sub-section. However, it might be slightly slower than it counterpart, i.e. table lookup implementation, particularly if it is implemented in platforms with 32 bits processors.

2. **Table Lookup Implementation**: is combining the following steps SubBytes, ShiftRow and MixColumns into four different lookup tables. The lookup tables $T_0, T_1, T_2\ and\ T_3$ for this implementation and the round transformation $e_j$are defined below. It is obvious that this implementation requires more memory than the straightforward implementation. However, it is much faster and much suitable for 32 bits machines.

$$T_0[a] = \begin{pmatrix} s[a].0x02 \\ s[a] \\ s[a] \\ s[a].0x03 \end{pmatrix} T_1 = \begin{pmatrix} s[a].0x03 \\ s[a].0x02 \\ s[a] \\ s[a] \end{pmatrix}$$

$$T_2[a] = \begin{pmatrix} s[a] \\ s[a].0x03 \\ s[a].0x02 \\ s[a] \end{pmatrix} T_3 = \begin{pmatrix} s[a] \\ s[a] \\ s[a].0x03 \\ s[a].0x02 \end{pmatrix}$$

$$e_j = T_0[a_{0,j}]\ XOR\ T_1[a_{1,j-c_1}]\ XOR\ T_2[a_{2,j-c_2}]\ XOR\ T_3[a_{3,j-c_3}]\ XOR\ k_j$$

The lookup table implementation is not secure against cache time attacks which we will discuss in the next sub section. More details about AES and about its implementation can be found in [8].

2.3.2 RSA algorithm

In 1977, a major advance in cryptography occurred when Ron Rivest, Adi Shamir and Leonard Adleman introduced RSA algorithm. It stands for their sure names Rivest, Shamir and Adleman [9]. It is one of the most popular among the public key algorithms and it is widely implemented in different applications particularly for confidentiality and authenticity. Unlike AES, RSA is an asymmetric encryption algorithm which means that the encryption key and the decryption key are different. In confidentiality settings, the sender uses the public key to send a digital message to the receiver. For decrypting the message, the receiver uses the private key.

The high level description for the RSA operations is as follows.

Key Generation:

- The receiver chooses p, q sufficient large prime numbers.
- Compute n = pq.
- The receiver chooses e and d such that:
  $ed = 1\ mod\ (p-1)(q-1)$ where mod is the modular arithmetic.
- The private key i.e. the receiver's key is (d,p,q)
- The public key i.e. the sender's key is (e,n)

Encryption:

- The sender encrypt the message m with the public key (e, n)

$$c = m^e\ mod\ n$$

Decryption:

- The receiver decrypt the message c with the private key (d, p,q)

$$m = c^d \ mod \ n$$

- For better performance, the decryption operation can be performed by Chinese remainder algorithm described in [10] and [11]. In this case p and q will be used directly by the receiver in the decryption operation.

Since the numbers involved in RSA operations are significantly large, multiple precision arithmetic algorithms should be employed. For efficient implementation for exponentiation, left to right binary exponentiation algorithm can be used [10]. The pseudocode for this algorithm is as follows:

INPUT: $g \in G \ and \ d = (d_t d_{t-1} \dots d_1 d_0)_2 \ is \ a \ positive \ integer$
OUTPUT: g^d

Algorithm:

$1. A \leftarrow 1$

$2. For \ i \ from \ t \ down \ to \ 0 \ do \ the \ following$:
$2.1 \ A \leftarrow A \cdot A$
$2.2 \ If \ d_i = 1, then \ A \leftarrow A \cdot g$

$3. Return(A)$

Step 2.2 in the pseudocode of left to right binary exponentiation algorithm can be an appropriate source of information leakage if d was the actual private key. In step i , if 2.2 was performed, this will mean that d_i was set. Otherwise, it was zero. Any side channel information can leak weather 2.2 was performed or not during RSA decryption can be used to recover the private key of RSA. There are many side channel attacks are discovered due to this vulnerability such as [12] and [13] We will see in section 2.5 examples of micro architectural side channel attacks based on branch predication unit employing this disadvantage.

2.4 Cache Based Side Channels

MA cryptanalysis witnesses dramatic advances over the last 10 years. By June 2002, [14] expanded the notes of [15] and drew the attention to the possible danger of cache memory by producing the first cache attack on the symmetric algorithm DES. Even though the attack is a theoretical, it reveals certain strategies which have been used later in developing practical attacks in DES such as [16] and in AES such as [17]. These strategies are termed time driven and access driven, see section Cache Attacks Strategies.

In 2005, Bernstein tried to present a remote cache attack by comparing the timing behavior of the target machine with a known referenced behavior. This requires a prior knowledge of the target machine profile under a known key before running the attack. However, [18] provided an analysis for this attack and showed that it is not realistic remote attack. Concurrently with Bernstein, [19] presented synchronous and asynchronous attack for full AES in reasonable requirements.

In this chapter we discuss general strategies of Cache based attack as well as two examples of cache based inter-process leakage attacks.

2.4.1 Cache Attacks Strategies

Some literatures, such as [14] and [17] classified the cache based attack strategies into three different classes:

1. **Trace Driven.** The adversary is able to have a profile of the entire cache activity and the outcome for every cache access. This includes the cache miss and cache hit.

2. **Time Driven.** The adversary can only have a summarized profile of the cache memory showing, either explicitly or implicitly, the total number of cache miss and cache hit. Thus, this class is limiting the adversary abilities.

3. **Access Driven.** The adversary is able to access the cache sets and modify it. By doing so, the adversary can infer which address has been accessed by another objects.

As described in previously, cache MA attacks exploit the cache states, for example cache hit and cache miss, in order to leak a secret data. We briefly showed various strategies of cache attacks above. One strategy is access driven attack which can be carried out in multi processing environment. [20] has presented an essential example for covert channel communication by means of inter-process communication through cache memory. He introduced the concept of Trojan Process and Spy Process. Then, based on this model, he produced cache based attack on RSA. It is the first public key cryptography attack based on exploiting the cache behavior.

Concurrently but independently, the first efficient inter-process cache leakage attack is demonstrated on AES in [19]. It is the first work for efficient cache based attack on AES in a real life setting. In their work, they have considered two modes of attacks. These modes are: synchronous and asynchronous. A description of these attacks will be introduced in the next two sub sections.

2.4.2 OST Synchronous Attacks

The synchronized attack consists of two stages: online and offline. In the first stage, a set of random samples of known plaintext is collected using cache side channel. i.e., cache hit and cache miss. Then, a cryptanalysis of this data is carried out offline.

The intuition of this mode of attack as follows. Let us denote that the key byte k_i and the plaintext bytes p_i. In the lookup implementation of AES, we have 16 lookup operations per round. This can be concluded because we have 4 lookup tables T_i such that $i \in \{0, 1, 2, 4\}$ and each table is accessed 4 times per round. In the online stage, the spy process fills the cache with its own data, and then triggers the target process to encrypt or decrypt one block (i.e. 16 bytes). Next, the spy process tries to read its own data again in order to examine the cache buffer. If there is a cache miss, it can be implied that the target process has accessed the cache set. Otherwise, target process has not touched this set. i.e., cache hit.

In the offline stage, the attacker needs to know which lookup tables have been accessed in the cache of cache miss. The idea is by using the cache hit to eliminate the non accessed set. Since $p_i \oplus k_i$ with known plaintext p_i, the attacker can discards a large number of candidates fork_i. Doing so in the first round, he can recovers only half of the key bits i.e., the most significant bits. To complete the full key, the attacker needs to analyze the second round of AES and exploit the relationship between the first round and the second round. This relation can be driven from the Rijndael specification [8].

So far, the reader may notice that there is something important is missing. How the attacker can infer that there is a cache miss or cache hit. The authors provided two methods to answer this question.

1. The first method is called Evict+Time and it can be described as follows:

    a. The attacker runs the target process with known plaintext p and waits until the process completes.

    b. Spy process accesses single cache set and fills it with its own data. This will evict any data has been written before by the previous process (i.e. target process)

    c. The attacker re-runs the target process with the known plaintext p and times it. If the average time is higher, the attacker can conclude that the target process suffers a cache miss since its own data has been evicted by the spy process in step b.

The authors termed this method Evict+Time due to the eviction operation in step (b) and the timing operation in step (c).

It is clear that this method is very sensitive to various type of external influence, in particular in the noisy environments. If, for example, the encryption process is called by kernel system call, additional code will be added such as instruction scheduling and page table misses resulting in a significant noise. The authors have realized this and therefore another method is introduced by them.

2. The second method is called Prime+Probe which proceeds as follows:

    a. Spy process fills either the entire cache or the relevant cache sets with its own data.

    b. The attacker runs the target process with known plaintext p and waits until it completes.

    c. Spy process re-reads its own data in the cache set and measure the total time. If the target process has accessed this set, high time will be observed. This indicates the cache miss state.

    This method is termed Prime+Probe due to prime operation in step (a) and probe operation in step (b). Figure 2.8 shows an illustration for this method.

The authors have tested the synchronized attack against AES implementation of OpenSSL library and dm-crypt device. In the former one, with full knowledge of memory addresses mappings and using Prime+Probe, they discovered the full 128-bit AES key after 300 encryptions on Athlon 64 and 16,000 encryptions of Pentium 4E.

However, the efficiency of this attack is degraded when they run it without information of addresses mappings in Athlon 64. This results in the need to perform 8,000 encryptions. No results have been given for Pentium 4 or any other processors in this case of lack of memory information. The entire experimental results can be found in [19].

It is important to mention that this attack is triggered only under AES-ECB mode of operation and thus demonstrating this attack under any other mode is luckily much harder.

2.4.3 OST Asynchronous Attacks

This attack differs from the synchronous one in many aspects. First, the attacker has no prior knowledge with regard to the actual plaintext or ciphertext. Second, this is not an inter-process communication in the same way aforementioned in the previous attack. It uses, instead, Hyper-Threading technology. As a result, the authors added constrains on the hardware and the plaintext characteristics in which the attack can be mount. More precisely, the attack requires hardware that supports simultaneous multithreaded processor. Furthermore, the plaintext should follow a known distribution. The latter constrain implies that the key will be revealed by analyzing the frequency distribution of the plaintext. The authors uncovered 45.7 bits of information about the key in 1 minute after running many processes which is encrypting English text.

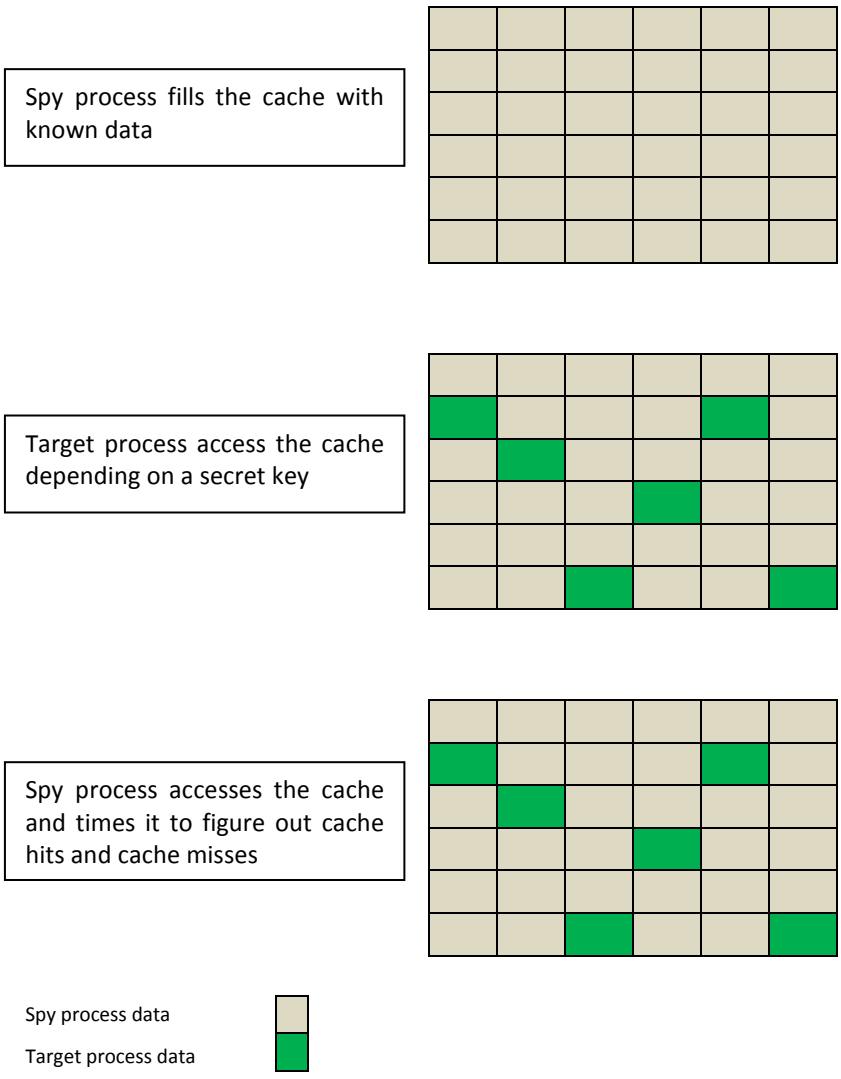

Figure 2.8: Illustration for Synchronous OTS prime+ probe cache attack

2.5 Branch Prediction Side Channels

A few years ago, [5] pointed out another component in many processors that can be used as a source of information leakage. This component is the Branch Prediction Unit (BPU). A Brief description of this component is introduced in section 2.1.2 and further details can be found in [21]. Generally speaking, it appeared from the first paper that this new class of MA attacks is powerful and easy to be mounted. Several attacks have been demonstrated into different platforms against several cryptographic systems since 2006. These attacks can be found in [5] and [22]. Since the birth of this vulnerable unit, many cryptographic implementations are becoming suffering from this kind attack. In general, we can say that any cryptosystem consists of conditional branches depending on the key running in concurrent environment is vulnerable to this risk if the appropriate countermeasures have not been applied seriously. In this section, we will shed the light on different types of these attacks. Further explanation can be also exist in [5].

2.5.1 Deterministic Branch Prediction Attack

The authors termed this attack a direct timing attack. It can be triggered in any processors implementing deterministic branch prediction algorithm. The attack is against RSA implementation that applies right to lift Square and Multiply (SM) exponentiation and Montgomery Multiplication (MM) without applying Chinese Reminder Theorem. Further details about RSA implementation algorithms can be found in [10].

The idea of this attack is derived from [12] which explains timing attack under the same RSA implementation. The attack is based on simulating the i intermediate steps of SM operations and it proceeds as follows. Let us take a large number of samples, say 10,000 and more is better, of ciphertext (if we want to carry it out under digital signature scheme). Based on the final reduction of Montgomery and whether the branch prediction outcome will be taken or not taken the key will be revealed.

The attacker will measure the execution time of the samples under one single key. Assume that the attacker knows the i secret bits of the key. It is clear that the first most signification bit is one. Now, the attacker needs to guess the $(i+1)th$ key. We denote it d_{i+1}. Then, he breaks the samples into 2 sets i.e., 5,000 for each sample. The first set is under the assumption that the bit is one and the second set is under the assumption that the bit is zero. Now, to figure out which set is the right one i.e. which guess is correct, the attacker needs to break each set a second time into two additional sets. This will yield to have a total number of four sets. In this time, the attacker is looking for figuring out if the branch prediction taken or not taken. It is obvious that we have four cases during MM as follows.

C_1: misprediction under the assumption d_{i+1} = 1
C_2: No misprediction under the assumption d_{i+1}= 1
C_3: misprediction under the assumption d_{i+1}= 0
C_4: No misprediction under the assumption d_{i+1}= 0

If the average time of C_1 and C_2 is more significant than the average time of C_3 and C_4 then the actual $(i+1)th$ bit is one. Otherwise, it is zero. It is not clear that what the authors mean by more significant. However, [12] provided a statistical approach to determine the significance based on performing certain statistical tests.

2.5.2 Synchronous Attack

The authors of [5] considered also the synchronous case as [19] did in their work. This attack should be succeeded if the adversary is able to synchronize the spy process with the intermediate steps of the target process.

For example, in step no i in SM operation, spy process should access the BTB in order to analyze the behavior of the target process. In each step i, the adversary needs to measure the time of the target process twice before and after running the spy process. In the first time, no modification for BTB is applied by spy process. However, in the second time, the spy process should manipulate BTB of the conditional branch of SM which is depending on the ith key bit.

It is obvious that any cipher algorithm suffers a conditional instruction that depends on the secret key is vulnerable to this attack. However, it appeared that it is difficult to synchronize the spy process with the encryption steps of the target process in the real environments. Thus, we can conclude that this attack is less practical.

2.5.3 Asynchronous Attack -Eviction

This attack runs on a simultaneous multi threaded environment. The attacker runs spy process to spy on specific behavior in BTB. The idea of this attack is to bring BTB to a known state and then perform cryptanalysis according to that. More precisely, spy process clears the BTB then the target process will suffer mispredication when it accesses BTB. The idea of simulating the intermediate exponentiation is also considered in this attack with the same separation basis of the previous attack in section 2.5.1.

In order that the spy process evicts the previous addresses in BTB, the authors considered three strategies:

1. **Full Eviction Method.** Spy process clears all addresses. It is clear that this method is appropriate when the adversary does not know the target branch addresses.

2. **Partial Eviction Method.** Spy process clears partial addresses.

3. **Single Eviction Method**. Spy process clears a specific set in BTB which is holding the target address of the target branch. It seems that this method is better in terms of the performance. However, it requires that the adversary knows exactly the actual address of any branch.

2.5.4 Asynchronous Attack -Trace Driven

This attack differs from the previous asynchronous attack by using different measuring approach. In this attack, the adversary measures the time of spy process rather than the timing of the target process. The idea of this attack as follows. The spy process executes unconditional branches such that they maps to the same BTB set of the branch under attack in the target process. The spy process should be triggered before and after the execution of the target process. Then, the adversary analyzes the gathered time. The average time of both the first and the second execution for the spy process should reveal the secret key.

It becomes much visible that these two asynchronous attacks are efficient and easy to implement. Therefore, these attacks may be a good candidate to be used for an adversary if she wishes to reveal the secret keys of a particular cipher process though the branch prediction unit.

2.6 Malware Detection System

The goal of this project is to introduce a detection mechanism to classify any of the spy (malicious) processes which are used in intercrosses leakage based MA attacks. Therefore, we decided to think about this spy processes as a malicious program (malware) which should be detected. Thus, it was necessary to study different types of detection approaches which are used to detect the malware. In this section, we discuss different approaches which are used to detect malwares. Then, we shed the light on code analysis and disassembly algorithms which are required in some techniques such as static analysis.

We have found that there are many different categorizations for the detection approaches. Some of these are based on traditional network based IDS techniques, which are presented in [23], such as misuse detection, anomaly detection and specification based detection. Other categorization such as those presented in [24] divided the detection techniques into signature based and anomaly based where the specification based techniques are subclass

of anomaly techniques. In this report we present a new categorization based on the learning model of the detection system. Our categorization is automatic learning approaches and manual learning approaches. Each of them can also be divided into two subclasses which are static and dynamic. In this section, we will discuss each approach in some details.

2.6.1 Automatic Learning Approach

The idea in approach is based on examining if the Process under Investigation (PUI) has normal characteristics or abnormal characteristics. The definition of normal or abnormal characteristics should be defined in advance before running the detection system. Based on these characteristics we categorized this approach into two sub-classes: static and dynamic. We can define the static one by saying that it is any characteristics that can be used to analyzed the PUI offline i.e., before the execution time whereas the dynamic one is any characteristics which can be used to analyze the PUI during the execution time. Each of these characteristics should be specified in advance based on finite rules or specification.

It is clear that it is necessary to train the detector what is the normal characteristics should look like. Thus, there are two main phases in this approach. First, the training (or learning) phase in which the detector will learn the properties of the normal characteristics. Second, the monitoring phase by which the detector monitors the PUI and judge, based on the knowledge gathered from the learning phase, whether it is doing normal or abnormal activity.

The main advantage of this approach is the ability to detect zero day attacks. Zero day attacks are the attacks which are recently created and they are not known by any anti malware system. However, it is still quite difficult for the detection systems which are applying this approach to overcome the issue of the false detection error which is named by the literature the false positive or the false alarm. The error occurs when the detector reports the non-malicious process as a malware. We believe that this issue related to the quality of the learning phase. Therefore, better features are used in the training phase; better rate of false detection will be obtained.

2.6.2 Manual Learning Approach

In this approach, the detection is based on a previous knowledge of the characteristics for known malware. These known characteristics are called attack signature in the literature. Like the previous approach, we divide the methods that belonging to this approach into two classes based on the type of characteristics. They are static methods and dynamic methods.

The detector in static approach looks for specific sequences of instructions. These sequences should be determined in the lab once the malware is known. Usually, this step

requires disassembling the binary representation and analyzing the resulting code. One possible technique that is most potentially to be used by the attackers is obfuscating the code of the malware program. As a result, people who analyze the malware should take into accounts using techniques that able to disassemble the obfuscated codes. There are many of these techniques presented in [25].

The second class of the signature scheme is runtime methods or the dynamic techniques. In this technique, the PUI is analyzed during the run time (online) by investigating its behavior. Like the static techniques, laboratory work should be carried out once the malware has been announced. In this case, the malware analysts will be looking for the sequence of specific behaviors rather than set of the instructions. Thus, the attacker will not be able to circumvent by obfuscating the code in order to bypass the malware detection system.

Unlike the Automatic approach, it is obvious that this approach does not suffer the false alarm issue. This is because the detection system depends on a much specified knowledge about the malware. On the other hand, some disadvantages are arisen. Since this approach is relying on extracting a signature of the malware in the lab, human effort is the key factor. In other words, code analyst play crucial role to examine the malware and extract its signature every time malware is known. Unfortunately, this step is not subject to be automated in the context of this approach. As an immediate consequence of this step, the set of signatures of the malicious processes should be stored in repository. Thus, this approach has a space requirement and it subject to be increased over the time with the increase number of malwares. Another limitation, which is spawn from the main advantages, is the lack of the ability of predicting the novel attacks i.e. the unknown attacks before they are performing any malicious activity.

As we have discussed in the previous section, the manual approaches require laboratory work to extract specific set of instruction sets practically in static class. Since we may need this phase when developing our potential detection method, it will be fine to have some overview of main steps that involved in this phase. These steps are binary disassembly and code analysis. In this section we will discuss different algorithm that are used to disassemble the regular binary representation of the program as well as the code analysis techniques.

2.6.3 Disassembly Algorithms:

They are two major algorithms can be used to disassemble the regular binaries [26]. Here, several basic algorithms will be introduced.

1. **Linear sweep disassembly.** In this technique the analyzer starts to decode everything in the machine code segment sequentially i.e. one instruction after another. It is obvious that this approach is straightforward and very simple.

However, it is unable to distinguish the control flow of the program in order to avoid misinterpretation [26]. For example, assume that our program contains jump instruction to specific address. In such case, the analyzer will disassemble the jump instruction and anything immediately after jump even if it will never be reached at any time during the run time or it was only NULL bytes. This will result in interpreting non-executable bytes to executable bytes. Therefore, the second technique is introduced.

2. **Recursive traversal disassembly.** This technique tries to fix the drawback of the previous one by starting at the beginning of the machine code section and looking for any branch instruction like jmp. Once it is found, the following possible control addresses will be determined. Consequently, the analyzer disassembles the instructions at those addresses and it continues to do so at each branch. Thus, this method follow the control flow of the program only and avoid any decoding operation for any data which is exist in the text segment. However, this technique also suffers when the branch, such as jmp, is indirect i.e. not static. One example of this case when the branch depends on piece of data that will be given during run time. As a result of this problem, the program may not be disassembled precisely leading to incorrect code and thus the hybrid technique is provided.

3. **Speculative disassembly (Hybrid Disassembly).** This algorithm is introduced by [27] and it applies the two previous algorithms together. This algorithm starts with the same way of the recursive traversal disassembly and it use special algorithm to determine if the following code is valid or not. That is, it disassemble only what is believed to be valid code. Further details about this algorithm can be found in [28].

2.6.4 Code Analysis

After disassembling the binary code, it is analyzed by carrying out one of the following approaches. The first approach is the static analysis by which the program is analyzed based on combination of the instructions sets. The second approach is the dynamic analysis where the program is sanitized based on the behavior upon certain objects.

Static Analysis. Static analysis is the process of examining the code without executing it. In this process, the entire code is usually examined in order to draw conclusions about the program functionality. In general, there are two major steps in this approach. First, the analyzer specifies the undesirable (malicious) behaviors. Then, it scans the whole program and look for the presence of any instruction sequences, which is called signature, inside the code that implement these negative behaviors. Most of the techniques are using control flaw and data flow analysis during this approach. A number of these techniques can be found in [29], [28] and [30].

The main advantage of this approach is that it covers the entire code and it tends to be much faster. However, the major drawback is that it cannot analyze and detect complex malicious behavior. Since the attack is written by human adversary, it can be written to be hard to be detected and some attackers tend to use code obfuscation techniques in order to make the detection mechanism much harder. Therefore, in some cases, it is better to use the dynamic counterparts i.e. runtime analysis.

Dynamic Analysis. In contrast to static analysis, dynamic analysis or runtime analysis is examining the code behavior during the execution time. The main advantage of this approach is its immunity to the matters of code obfuscation or self modifying programs [31]. Since the code analysis should be carried out on machine, it is necessary to build special environment when running the experiments. This is because running the malware directly may harm the underline machine and be travelled over the internet to harm other machines. Therefore, the analysis tends to run in offline virtual machine. However, it turns out that the malware can hide itself under the virtual machine to prevent any kind of analysis. Therefore, a special tools is produced to mitigate this problem such as TTAnalyse in [31].

2.7 Evaluation Criteria

The research in malware detection has focused for many years on the automatic (anomaly) detection and manual (signature) approach. Even though the later seems to be preferable on the commercial sector due to its high level of detection rate, the former sometimes considered as more powerful due to the ability of detecting the infant or novel attacks [23]. However, the primary challenge in the academic community is how to evaluate the automatic detection systems under different environment in order to analyze the strength and the limitation of these kinds of techniques. In this section, we discuss number of different criteria that can be taken into consideration to evaluate malware detection system.

2.7.1 Accuracy

The accuracy word here should answer the question of how much the introduced detection method can detect malware correctly [23]. For example, the system that has 60% accuracy means that it able to correctly classify 60 instances out of 100 instances. Generally speaking, the system which has more than 80% accuracy level is considered a good detection system [23]. However, the remaining percentage i.e. 20% may play a significant role to determine whether this system is desirable or not. The incorrect classified instances can be categorized into two parts:

1. False Positive Instances. They are the normal instances which are classified as abnormal (malicious) instances.
2. False Negative Instances. They are the abnormal (malicious) instances which are classified as normal instances.

The impact of these misclassification types on the user differs according to application nature and users preferences. For example, managers who use spam filter may not prefer those filters which have high percentage of false positive rate because the impact of missing true email might be higher than receiving spam email. On the other hand, those organizations that keep sensitive information about their customers, such as banks, may not prefer anti-malware systems which have high proportion of false negative rate.

Table 2.1 Process Classification Classes shows four different types of classifications for malware detection system. When the system detects malicious process or normal process correctly, that means it achieved true positive or true negative result. In contrast, when it detects malicious process or normal process incorrectly, that means it achieved false positive or false negative result.

Actual Process	Detection System Decision	Classification Result	Name in Literature
Normal	Normal	Correct	True Negative (TN)
Normal	Malicious	Incorrect	False Positive (FP)
Malicious	Normal	Incorrect	False Negative (FN)
Malicious	Malicious	Correct	True Positive (TP)

Table 2.1 Process Classification Classes

Accordingly, the main goal for every malware detection system is to produce as high percentage of TN and TP as possible and as low percentage of FP and FN as possible. These four variables TN, TP, FP & FN are used in a big majority of accuracy models of intrusion detection systems. In the next paragraphs I will shed the light on different methods that are used to model the accuracy.

Confusion Matrix

Confusion Matrix is a specific table layout that is proposed by [32] to visualize the outcome of the classification problem. In the context of malware detection, we will have two actual classes in both column and row i.e., 2x2 matrix. To judge the accuracy of the systems, the non-zero numbers should be located only in the main diagonal of the matrix. The ideal detection system is that which have zero values in the whole matrix unless the diagonal. Table 2.2 shows the layout of the confusion matrix.

Confusion Matrix		Predicated Class	
		Normal (Negative)	Malicious (Positive)
Actual Class	Normal (Negative)	TN	FP
	Malicious (Positive)	FN	TP

Table 2.2: Confusion Matrix for Malware Classifier.

The main diagonal (↘) consists of TN and TP. Good detection systems should avoid having high values in the non diagonal cells. The ideal systems are those which have diagonal matrix.

From Table 2.2, the following metrics is defined as follows.

Accuracy (AC) = $\frac{TN+TP}{total} = \frac{TN+TP}{TN+TP+FP+FN}$ *(Eq 2.1)*

Error Rate (ER) = $\frac{FN+FP}{total} = \frac{FN+FP}{TN+TP+FP+FN}$ *(Eq 2.2)*

True Positive Rate (TPR) = $\frac{\text{TP}}{\text{actual positive}} = \frac{TP}{\text{FN}+TP}$ = Sensitivity *(Eq 2.3)*

True Negative Rate (TNR) = $\frac{TN}{\text{actual negative}} = \frac{TN}{\text{TN}+FP}$ = Specificity *(Eq 2.4)*

False Positive Rate (FPR) $= 1 - TPR$ *(Eq 2.5)*

False Negative Rate (FNR) = $1 - TNR$ *(Eq 2.6)*

In the next paragraphs, we focus on the classification of the positive (malicious) instances i.e. TP, FN & TP which are generated by the classifier.

Precision, Recall and F- measure

There are many metrics which are introduced to analyze the accuracy in terms of the positive instances [23]. The first one is the *Precision* which shows how many instances of the actual malicious instances were successfully predicted over the total predicted malicious instances. In other words, in the context of malware detection system, it evaluates how much the system correctly detects the actual malwares over the total number of processes which are detected as a malware. For example, assume that the system detects 10 processes as a malware and only 6 of them were truly malware. Then the Precision of this system will be 0.6. The formula of this metric as follows.

Precision (P) = $\frac{\text{TP}}{\text{predicted positive}} = \frac{TP}{\text{FP}+TP}$ *(Eq 2.7)*

The disadvantage of using only the Precision metric is the fact that it omits the total number of the actual positive instance that is covered by the classifier. As a result, the system may have a high Precision even if it detects small part of the malicious instances. Therefore, there is another metric can be used for this purpose which is termed *Recall*.

Recall metric shows the missing part of Precision. The formula of this metric is:

$$\text{Recall (R)} = \text{TPR} = \frac{\text{TP}}{\text{actual positive}} = \frac{TP}{\text{FN}+TP} \qquad \textit{(Eq 2.8)}$$

If the classifier classified only 4 instances out of 10 actual positive, Recall value will be 0.4. One may notice that using this metric only is not a good approach as well. This is because it does not consider the number of false alarms instances i.e. FP. For example, if the total actual malware is 10 and 8 of them were correctly positively predicted (i.e. TP = 8). That is mean Recall value is 0.8 which is high. Suppose that the total of the predicted instances as malware was 15. That is mean the false alarms is 7 (FP = 15 – TP) which is also high. Thus, we can conclude that high Recall value may not also means good detection system.

From the previous paragraphs, we pointed out that high Precision is not a good indicator to judge that the detection system has a good accuracy level and we have illustrated the same issue for Recall. What we need is a classifier with high value in both Precision and Recall. Therefore, it is essential to have a metric that combine these two metrics.

Due to what we have discussed, *F-Measure* metric is introduced in 1992 to evaluation tasks of information extraction technology [33]. This metric combines the two previous metrics i.e. Recall and Precision. It can be calculated by the following formula:

$$\text{F- Measure} = \frac{2}{\frac{1}{\text{P}}+\frac{1}{\text{R}}} = 2 \times \frac{P \times R}{P+R} \qquad \textit{(Eq 2.9)}$$

When a classifier has 100% Precision and Recall, this metric will be 1.0 meaning that the classifier has 0% false alarm and detect 100% of the positive instances. Thus, F-measure is preferred to be used when one accuracy metric is required to evaluate the detection mechanism.

ROC Curves

The Receiver Operating Characteristics (ROC) analysis is one of the important tools that can be used not only in intrusion and malware detection systems, but also in a large number of practical problems such as medical diagnosis, radiology as well as machine learning and data mining [34]. Historically, it is introduced during World War II to analyse the radar signals [35].

In the context of the malware detection systems, ROC curves have two main applications. First, it can be used to visualise the relationship between the proportion of malwares that are correctly detected (TPR) and one minus the proportion of normal process that are correctly classified (FPR). The literature usually terms the former Sensitivity or intrusion detection rate and the latter false alarms rate. The second application is to compare the accuracy between two or more classifier or detection method. Conventionally, X-axis in ROC curve accommodates FPR while the Y-axis depicts the TPR. Figure 2.9 shows ROC space areas [36].

To explain ROC space interpretation, let us describe the meaning of the following points in Figure 2.9:

- (0,0). The classifier always classifies all instances as a negative (normal). It is clear that there are no false alarms. However, the classifier is unable to detect any positive (malicious) instance which is undesirable.
- (1,0). This point is also unwelcome since it indicates that the classifier always produces false alarms and it unable to detect any instance correctly. It is the worst point.
- (1,1). This point indicates that the classifier has detection rate 100%. In other words, it is able to classify any positive instance correctly, but with 100% false alarms rate.
- (0,1). This point shows that the classifier is able to detect any positive instances with 0% false alarms rate. This point is the ideal one. Thus, the closer classifier to this point, the more accurate classifier is.
- Any classifier placed on the diagonal line between the upper and the lower area is called random classifier.

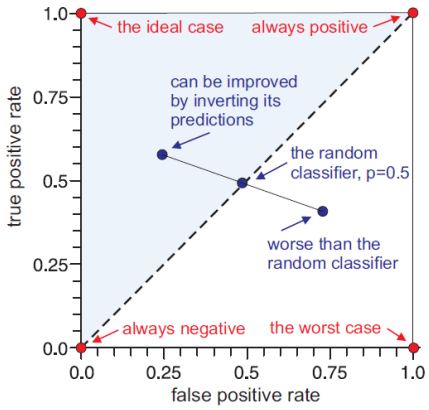

Figure 2.9: ROC Space Analysis

2.7.2 Performance

The evaluation of the intrusion and malware detection systems performance is an important matter. The additional issue on this evaluation involves many factors related to the underline platform (hardware and operating system) in which the detection system works [23]. Examples of these factors are the CPU speed, the memory capacity and the operating system. Generally, performance evaluation criterion for commercial anti-malware detection system is the scanning time while in Network based IDS it can be measured by evaluating the ability to process the traffic data in a high speed link with minimum packet loss [23].

In the context of spy process detection system, we believe that the main factor in achieving high performance depends on the approach that is employed to detect spy process. For example, in the signature based scheme, the performance depends on the scanning time of the PUI's binary if we employ the static techniques. In the case of behaviour techniques it depends on the ability of the detector to monitor the behaviour of PUI on specific resources.

We found out that they are many other metrics in the literature. For example timely response, completeness and const sensitivity which are used to evaluate Network IDSs. Since our work is more relevant to malware detection system, we will not cover these metrics.

Chapter 3. Design and Implementation

In this project, we have focused on devising a detection system for cache based Spy Process (CBSP). Signature based detection is the technique which we have used for designing the detection methods. This means that we have to reverse engineer the spy program by disassembling the program's binary. Then, carefully analyze its own CPU instructions and determine the list of the instructions that are essential to be used in CBSP. It is also possible that some specific mostly frequent instructions are not employed in CBSP. Therefore, it makes sense to also consider the absence of particular instructions as an indicator to the possibility of the program to be classified as CBSP. We will see example of such instructions shortly.

3.1 Disassemble CBSP

Spy process instances are rare, and it is difficult to be found in public resources. The only example of CBSP that we could find publicly available over the time of writing this report is found in [37]. Therefore, we have used this instance to discover the signature of CBSP then design and develop the detection system for it.

The first step of finding out any program signature is to disassemble its binary, which is in our context the CBSP. This should be done offline and online. The offline analysis is to disassemble the binary before implementing the detection system. The goal of this step is to recognize the appropriate signature. This step should be performed by engineers who have a fair knowledge of the spy process. The online analysis occurs during the run time of the detection system. The goal of this analysis is to decide whether the program under inspection holding the signature of CBSP or not.

There are plenty of commercial and non-commercial disassembles. The one which we have used in the offline analysis is objdum -d while we have used libdisasm library in the online inspection. We were interested in using a mature open source disassembler with globally support community. In Figure 3.1: Part of the disassembled of CBSP, a part of the

disassembled code of CBSP in presented. It is the main section that forms the signature. We used objdum –d to generate this disassembled code.

```
80485ae:        0f af 1b                imul   (%ebx),%ebx
80485b1:        0f af 9b 00 08 00 00    imul   0x800(%ebx),%ebx
80485b8:        0f af 9b 00 10 00 00    imul   0x1000(%ebx),%ebx
80485bf:        0f af 9b 00 18 00 00    imul   0x1800(%ebx),%ebx
80485c6:        0f 31                   rdtsc
80485c8:        29 f0                   sub    %esi,%eax
80485ca:        0f c3 04 39             movnti %eax,(%ecx,%edi,1)
80485ce:        01 c6                   add    %eax,%esi
80485d0:        0f af 5b 40             imul   0x40(%ebx),%ebx
80485d4:        0f af 9b 40 08 00 00    imul   0x840(%ebx),%ebx
80485db:        0f af 9b 40 10 00 00    imul   0x1040(%ebx),%ebx
80485e2:        0f af 9b 40 18 00 00    imul   0x1840(%ebx),%ebx
80485e9:        0f 31                   rdtsc
80485eb:        29 f0                   sub    %esi,%eax
80485ed:        0f c3 44 39 01          movnti %eax,0x1(%ecx,%edi,1)
80485f2:        01 c6                   add    %eax,%esi
80485f4:        0f af 9b 80 00 00 00    imul   0x80(%ebx),%ebx
80485fb:        0f af 9b 80 08 00 00    imul   0x880(%ebx),%ebx
8048602:        0f af 9b 80 10 00 00    imul   0x1080(%ebx),%ebx
8048609:        0f af 9b 80 18 00 00    imul   0x1880(%ebx),%ebx
8048610:        0f 31                   rdtsc
8048612:        29 f0                   sub    %esi,%eax
8048614:        0f c3 44 39 02          movnti %eax,0x2(%ecx,%edi,1)
8048619:        01 c6                   add    %eax,%esi
804861b:        0f af 9b c0 00 00 00    imul   0xc0(%ebx),%ebx
8048622:        0f af 9b c0 08 00 00    imul   0x8c0(%ebx),%ebx
8048629:        0f af 9b c0 10 00 00    imul   0x10c0(%ebx),%ebx
8048630:        0f af 9b c0 18 00 00    imul   0x18c0(%ebx),%ebx
8048637:        0f 31                   rdtsc
8048639:        29 f0                   sub    %esi,%eax
804863b:        0f c3 44 39 03          movnti %eax,0x3(%ecx,%edi,1)
8048640:        01 c6                   add    %eax,%esi
8048642:        0f af 9b 00 01 00 00    imul   0x100(%ebx),%ebx
8048649:        0f af 9b 00 09 00 00    imul   0x900(%ebx),%ebx
8048650:        0f af 9b 00 11 00 00    imul   0x1100(%ebx),%ebx
8048657:        0f af 9b 00 19 00 00    imul   0x1900(%ebx),%ebx
804865e:        0f 31                   rdtsc
8048660:        29 f0                   sub    %esi,%eax
8048662:        0f c3 44 39 04          movnti %eax,0x4(%ecx,%edi,1)
```

Figure 3.1: Part of the disassembled of CBSP

3.2 Analyze CBSP

After having the disassembled binary of the process under investigation, it is essential to examine the CPU instructions that are employed in the program. This will determine the most significant instructions that are required by CBSP leading to the list of the CPU instructions that can form the CBSP. After careful analysis, we found that the following instructions that can be form the signature of CBSP. In the following sections, these instructions are presented.

3.2.1 Reading Time Stamp Counter

In CBSP, it is essential to measure the access time for a particular location in the memory in order to conclude the cache hit and cache miss. Since this operation is very fast, using the

ordinary C functions of measuring the execution time such as gettimeofday(), time() or clock() are not precise enough to obtain the correct results. Therefore, reading CPU time stamp counter is used to calculate the execution time of accessing a memory. According to Intel Developer Guide, which can be retrieved from [38], rdtsc is x86 CPU instruction that returns the current value of the processor's time stamp counter. It loads the value into EDX:EAX registers. The low-order 32 bits are loaded into EAX register while the high-order 32 bits are loaded into EDX register. Every clock cycle, the processor monotonically increases the counter. In other words, it counts the number of processors clocks since the last processor reset. To measure the performance of particular code, say memory access for example, the user get the CPU time stamp counter before and after the code, then subtract the two values to get the number of CPU clocks that was consumed for executing the code.

It is important to say that rdtsc is not a serialized instruction which means that it will not wait until the pervious instructions in order to be executed. Therefore, among the processors that support out of order execution, one should use a serialized instruction such as cpuid before running rdtsc in order to obtain precise results. In the next subsection, this issue will be discussed.

3.2.2 Serialized Instruction

Executing instructions one after the other may lead to poor performance in the processors. Some instruction dependencies may not be ready at the time of the instruction execution. As a result, the processor will need to spend more cycles until their dependencies being available. This usually called the stall. Out of order execution is a paradigm used in most of high performance microprocessors to reduce the number of the stalls that occur when executing the instructions in order.

The idea is when an instruction dependency, say data for example, is not ready; microprocessor may decide to execute another instruction after the current instruction if it does not depend on the current instruction and put its output in the result queue. As a result, the source code may not be executed in order and one may find some later instructions have been executed before earlier instructions. This issue is particularly important when the programmer are concerned to use some CPU instruction for monitoring the performance such as reading the time stamp counter instruction rdtsc.

With that in mind, there is a serious issue when using rdtsc instruction in out of order execution processors. The problem is rdtsc may be executed before or after its location in the source code. This potentially will give inaccurate cycle count leading to misleading determination for cache hit and cache miss.

In order to solve this problem and avoid rdtsc from being executed out of order, it is essential to use a serialized instruction before using rdtsc. The serialized instruction forces the microprocessor to complete the previous instructions before being executed. Thus, the program will not continue to execute rdtsc before the serialized instruction is complete.

One example of a serialized instruction in x86 architecture is cpuid. The instruction name is driven from CPU Identification. This instruction can be used to determine the processor information such as its type and the presence features. Therefore, if the spy process measuring the execution time in out of order execution processor, this instruction may be used.

3.2.3 No Cache Access

In CBSP, it is important in cryptanalysis stage to retain the cycle number that returns from rdtsc instruction in the memory without polluting the cache. This can be performed by using movnti instructions. This instruction is useful for accessing non temporal data which is any data that will not be used before it gets evicted from the cache. In other words, movnti is an instruction to access any data on the memory without cache it. The source operand of this instruction is a general purpose register and the destination is a 32-bit memory. More information about this instruction can be found in [38].

Due to the accuracy level that is needed by the spy program in order to be able to leak cache hits and cache misses precisely, it is essential for the adversary to write spy program that does not pollute the cache memory with any kind of unnecessary data. Thus, it is necessary for the adversary to find a way to know the cache states without being affected with this data. As a consequence, movnti instruction is quite useful for the spy process programs writers. It allows moving the number of CPU clocks that was consumed for accessing the memory into a memory location without accessing the cache. As a result, the presence of movnti instruction in the program under investigation is a strong indicator for the possibility of that program to be a spy process. For that reason, this instruction is classified as a basic instruction for CBSP.

3.2.4 Less System calls

As we have seen in 2.2, any process running in a CPU should be executed under specific execution mode. There are usually four modes of execution ranging from the least privileged to the most privileged. The kernel is the most privileged and can access all system resources. If there is a process needs to access particular hardware resources, it should request this service from the kernel using what is called system call.

Most of the normal programs request a service from the kernel in order to execute hardware related operations such as reading or writing from the hard disk or printing into the standard output. CBSP does not normally need to use much system calls since its activities are only concentrated on measuring the memory access time. Thus, the number of the system calls that is exist in a program may be used as an additional factor in the detection procedure for CBSP. For example, the absence or a few number of the system calls will be increase the suspicions of a program to be spy process.

Keeping in mind the previous CPU instructions, we can find a good signature for the CBSP. For simplicity, we will call all instructions that are listed in this section the primary instructions for CBSP; for short, we will say primary instructions.

3.3 CBSP Signature based Detection Methods

Based on the analysis that has been mentioned in the previous section, two signature based detection methods for CBSP have been introduced during this project. They are: Counting method and Program Flow method. In this section, these two methods will be explained.

3.3.1 Counting based Method

In this method the classification for the program under investigation is based on how many number of primary instructions exist in the disassembled code. It counts the number of each primary instruction and takes the decision based on specific parameters which are called $\alpha_1, \alpha_2, \beta$. The main advantage of this method is the performance since there is no much complex work needed for this method to be carried out.

The steps of this method are as follows:

Inputs:

Binary file, $\alpha_1, \alpha_2, \beta$

Output:

Classifications for binary file either CBSP or non CBSP

Steps:

1. Read the binary under investigation
2. Disassemble the binary.
3. Count the primary instructions as follows:

3.1 rdtsc_n ← $count\ of\ rdtsc\ instructions$
3.2 cpuid_n ← $count\ of\ cpuid\ instructions$
3.3 $movnti\_n \leftarrow count\ of\ movnti\ instructions$
3.4 $sys\_call\_n \leftarrow count\ of\ systemcalls\ insturctions$

4. Compare if
   4.1 rdtsc_n = $cpuid\_n \geq \alpha_1$ AND
   4.2 $movnti\_n \geq \alpha_2$ AND
   4.3 $sys\_call\_n \leq \beta$ ⇒ Classify the binary as CBSP and exit

5. Classify the binary as non-CBSP

$\alpha_1 \& \alpha_2$ are the minimum expected number of rdtsc and movnti instruction instructions in the source code of the successful CBSP binary. β is the maximum expected number of the sys_call CPU instructions in the source code of the successful CBSP binary. We named $\alpha_1 \& \alpha_2$ the existence parameters for the primary instructions while β is the absence parameter for the primary instructions.

The obvious advantage of this method is its simplicity. There is no much complex work required to run this method; therefore, high performance is obtained. On the other hand, it may produce false positive or false negative results particularly if inappropriate values are set to either the existence parameters or the absence parameter.

3.3.2 Program Flow Method

In this method, the classification decision is based on examining the program flow of the binary under investigation to determine some essential blocks for CBSP in the disassembled code. One example of these blocks is the loop of filling the cache with the adversary data. The loop in this case should also contain the primary instructions rdtsc and movnti. The description of this method is as follows:

Inputs:

Binary file, $\alpha_1, \alpha_2, \beta$ cache parameters (set, lines, size)

Output:

Classifications for binary file either CBSP or non CBSP

Steps:

1. Read the binary under investigation.

2. Disassemble the binary.

3. Detect the loop primary instructions such as cmp, jge, jmp etc. If no go to 9.
4. Check if number of iterations equals to one of the cache parameter. If no go to 9.

5. Check if the primary instructions rdtsc, cpuid & movnti inside the loop block. If no go to 9.

6. Count the frequency of the following primary instructions inside the loop block
   6.1 rdtsc_n $\leftarrow count\ of\ rdtsc\ instruction$
   6.2 cpuid_n $\leftarrow count\ of\ cpuid\ instruction$
   6.3 $movnti\_n \leftarrow count\ of\ movnti\ instruction$

7. Count outside the loop $sys\_call\_n \leftarrow count\ of\ systemcalls\ insturction.$

8. Compare if
   8.1 rdtsc_n $= cpuid\_n \geq \alpha_1$ AND
   8.2 $movnti\_n \geq \alpha_2$ AND
   8.3 $sys\_call\_n \leq \beta \Rightarrow$ Classify the binary as CBSP and exit

9. Classify the binary as non-CBSP

It seems that this method has less number of both false positive and false negative instances than the previous method. This is because scanning of the primary instructions is restricted only inside the loop block. However, tracing the program flow and detecting CBSP loop block will add an additional overhead leading to less performance outcome for the overall detection system.

3.4 Implementation

Signature based IDS in this project is implemented based on the Counting Method which is introduced in the previous section. Two versions of IDS for CBSP have been developed. The first version of IDS for CBSP is developed to be used to evaluate the Counting Method. The second version is an embedded IDS for CBSP in bash shell of the Linux systems. This version is named secure terminal. The Two versions will be explained in the next subsections.

3.4.1 Version I

In this IDS version, Counting Method for CBSP is implemented as a single program. The aim of this version is to evaluate the Counting Method in different systems and determine the appropriate values for the existence parameters and the absent parameter that are

mentioned in the previous section. The evaluation process involves the error rate as well as measuring the performance. More details about the evaluation results can be found in chapter 4.

3.4.2 Version II

Version II or secure terminal is a modified version of GNU Bash Shell 4.2 software with the ability to stop running any process if it classified as CBSP. That is, the user (adversary) will not be allowed to run any program that has been classified as CBSP. Therefore, if the adversary is using this secure terminal, he or she will not be able to spy on micro-architectural cache component when running spy program. This version is developed to provide a clear demonstration for what IDS for micro-architectural spy process should look like. Figure 3.2 shows a snapshot for secure terminal demo in Ubuntu 10.04 system. One may notice that ls command works fine since it is allowed to be run by the terminal because it is not classified as a spy process. On the other hand, cache_spy process is classified as a spy process. Therefore, it is not allowed to be executed.

Deploying this version in the real environments has advantages with regard to the ease of dealing with the false positive instances and less cost when update is needed. However, secure terminal may not provide the security required in some environments. Since the secure terminal is a user space application, attackers can find a way to bypass it if they have enough privileges. For example, the attacker can use his own bash shell if he has the right to install programs and use them! This scenario can occur if there are, for instance, two virtual machines for two users run in one single physical machine. As a result, this version may not be appropriate in Platforms such as a Service PaaS or Infrastructure as a Service Iaas in cloud computing. Much robust solution for this problem is to integrate the IDS for CBSP in the operating system kernel instead of integrated with the terminal. Therefore, we have recommended this to be an extended work for this project.

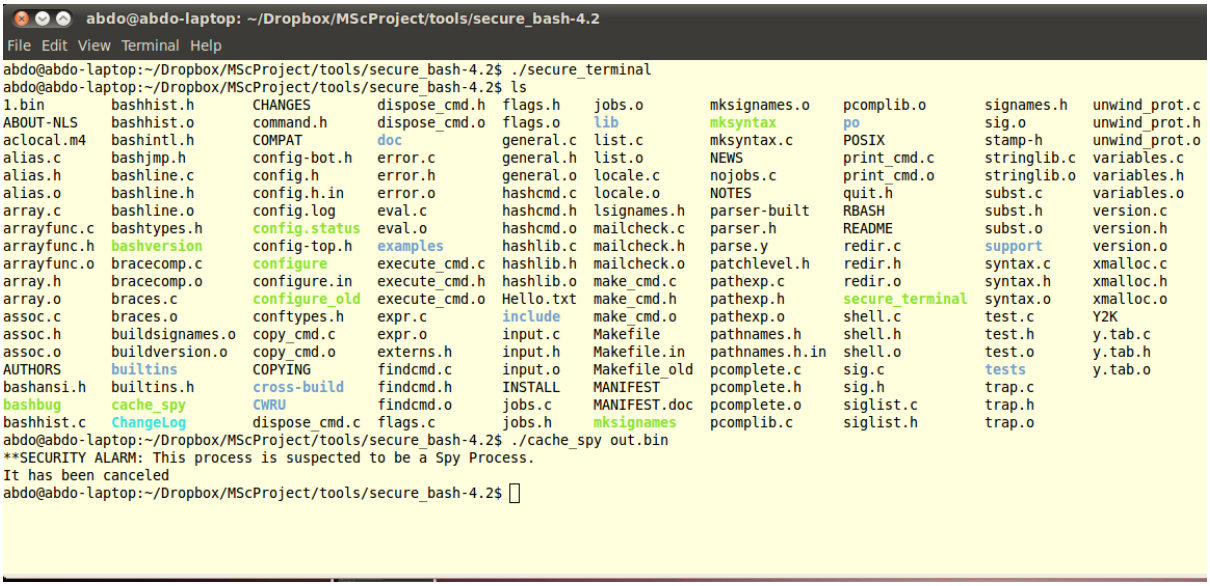

Figure 3.2: Secure Terminal

3.4.3 Kernel Space Version

For the reasons that we have discussed in 3.4.2, we have thought about how the introduced detection system can be integrated in the Linux kernel. Generally speaking, the kernel must classify the program before allowing it to be loaded in the CPU and being executed. Technically, let us trace the kernel functions when a program in x86 architecture is being executed under Linux system. When a user runs a program in the terminal, execve() function is invoked to hold the program location and its arguments and pass it to the kernel. execve() is the system call that is responsible for loading the program into the kernel space in order to be executed in the CPU. sys_execve() in arch/i386/kernel/process.c file is the first function in the kernel space receives the program. Inside this function, do_execve() which is called to open a file to perform some check operations. If the program has passed the check operations, shearch_binary_handler() in fs/exec.c is called to find out the format of the executable program to fills struct linux_binfmt with the appropriate data that is related to the program format. If the program is in ELF format, then load_elf_binary() is invoked to load the binary and give the control to the code segment and now the program is being executing in the CPU.

If the static signature approach is used, one challenge for integrating IDS with the kernel is how the program file will be fully scanned before reaching load_elf_binary() function. The normal sense for doing that is using basic file operations such open and read. However, opening or reading a file in the kernel space is not recommended operations by many Linux kernel experts [39]. One solution is scanning the file during the loading operation itself.

However, this option is need to be investigated thoroughly in the terms of security, reliability and performance.

Another issue that should be considered is disassembling the process in the kernel space. To do so, essential files in the kernel that is developed for this purpose should be examined such as arch/x86/lib/insn.c file. This file can be a good starting point since it contains a list of functions that can be used to analysis the x86 instructions. One example of kernel patches that uses these functions is an in-kernel disassembler which is released in April 2012 for kernel debugging purposes [40].

Chapter 4. Evaluation and Discussion

In this chapter, we will shed light on some experimental results which have been conducted to evaluate the produced IDS of CBSP. In addition, analysis and discussion for some related issues will be presented.

4.1 Accuracy

Many experiments have been conducted for the produced detection system to evaluate the accuracy in both Linux and Windows operating systems. It involved scanning all standard and some installed binaries for both systems. The targeted binaries in Linux system are all binaries that are located in the following directories: /bin, /sbin, /usr/bin and /usr/sbin. In Windows system, the targeted binaries are located in C:\Windows\System32 directory. Based on these experiments, recommended values for both existence and absence parameters $\alpha_1, \alpha_2, \beta$ have been set. The experiments have been conducted in Ubuntu 10.04 Linux distribution and Windows Visa operating systems. The total number of binaries that are scanned in both systems is more than 5500 binary.

At the beginning, we were interested in examining how many binaries usually use rdtsc CPU instruction more than 2 times in their source code. This has given a rough estimation of the percentage of the binaries that use rdtsc instruction. It also could lead to an approximate number of the false positive instances if we relayed only on rdtsc in the detection system. After conducting the experiments, we found out that 8% of the standard binaries are using rdtsc instruction at least two times. In addition, we noted that there are no binaries that use cpuid instruction jointly with rdtsc. Therefore, we did not consider cpuid in this evaluation process.

Secondly, we moved to investigate the binaries that do not use system calls heavily. It is clear that using this characteristic only will not be useful. Therefore, we have evaluated the accuracy rate when the detection system method scans for both time stamp counter and

system calls. We found out that the accuracy of the detection system has been increased and the number of the false positive instances has been decreased significantly from 8% to 1% in Windows system.

Finally, we were aiming to obtain as lowest false positive instances as possible in order to reduce the error rate. After evaluating the detection system when adding movnti instruction, the accuracy is increased and we have obtained no false positive instances in both Linux and Windows systems.

The values that are used for $\alpha_1, \alpha_2, \beta$ are very restrictive. They are 2, 1, and 3 respectively. Obtaining zero number of false positive instances is a positive indicator to the ability of the produced system to work in real environment without reporting much number of false alarms. The blow tables summarize the experiments results for both Linux and Windows systems.

Primary Instruction(s) (Signature)	False Positive Instances	True Negative Instances	Error Rate %
Rdtsc	31	2813	1%
Rdtsc & int 0x80	0	2844	0%
Rdtsc, int 0x80 & movnti	0	2844	0%

Table 4.1: Counting Method Accuracy Results over 2844 binaries in Ubuntu 10.04 with $\alpha_1 = 2,\ \alpha_2 = 1,\ \beta = 3$

Primary Instruction(s) (Signature)	False Positive Instances	True Negative Instances	Error Rate %
Rdtsc	237	2428	8%
Rdtsc & int 0x80	4	2661	1%
Rdtsc, int 0x80 & movnti	0	2665	0%

Table 4.2: Counting Method Accuracy Results over 2665 binaries in Windows Vista with $\alpha_1 = 2,\ \alpha_2 = 1,\ \beta = 3$

Finding other instances of spy processes is a matter. We did not find any public source of cache based spy process in order to conduct experiments for evaluation the ability of the system to detect the unseen spy processes. We have tried to build a generator for building spy process but we did not find any appropriate method or algorithm that can be used to ensure that the generator is suitable enough as a source for spy processes. However, since we have developed signature based static detection system; this issue may not be a big problem. The main advantage of the signature based systems is the ability to classify instances with high accuracy rate since it is looking only for specific instructions that should be shared among all instances. We believe that we have almost covered all these characteristics.

4.2 Performance

Experiments for evaluating the produced detection system performance are also conducted. These experiments are performed in Intel Core 2 Duo CPU in Ubuntu 10.04. Since the implemented method, which is Counting Method, is not complex, very good performance result is obtained. The introduced system could scan 1 MB of binary bytes in 0.43 seconds on average.

It can be noted that this result can be improved by applying some High Performance Computing Techniques. In Counting Method, scanning for the primary instructions can be performed easily in parallel which is an additional advantage for Counting Method over Program Flow Method.

4.3 Countermeasures

It is important to think as an attacker to be aware of the possible countermeasures that can be applied by the adversary to bypass the produced detection system. In this section, we will shed the light on some of these countermeasures that are we believed that they can be used to bypass the system. One of these countermeasures is writing an obfuscation code for CBSP. Another one is trying to add fake system calls. Each of these countermeasures will be discussed and recommendations for mitigating the provided countermeasure will be presented.

4.3.1 Code Obfuscation

Code Obfuscation is anti reverse engineering techniques which can be used to hide the source code and make it unreadable by human being. Most of the signature base detection systems suffer from this technique since it will make the disassembly process quite difficult. IDSs for MA spy process are not an exception. The adversary can also obfuscate the spy

process code in order to make the disassembly procedure as hard as possible. As a result, both the offline and the online disassembly stage will become very difficult.

One way for mitigating this problem is designing behavior base detection method for MA spy process. This approach will be discussed in section 4.4 and we will conclude that it might be difficult as well. Another way for handling this problem is using dissemblers with the ability to reverse engineer the binaries even if they are obfuscated or encrypted. They are many commercial disassemblers which their authors claimed that they have this feature. One example is Interactive DisAssembler IDA provided from Hex Ray. More information about IDA disassembler can be found in [41].

4.3.2 Fake System Calls

It is obvious that there is a possible manner for the attacker to bypass the detection system by adding number of fake system calls more than β value. We mean by fake system calls is any system calls which is not needed by a program to achieve its functionality. We have noted this counter step; and consequently, we have investigated the accuracy of the introduced detection system when applying Counting Method with rdtsc and movnti instructions only without taking into consideration the number of system calls. That is, the system will not take into accounts the number of the system calls when classifying a given process. After evaluating the accuracy, we found out that the detection system gives a reasonable error rate which is 3% in the same environment of evaluating the Counting Method with system call version. Table 4.3 summarizes the evaluation result and compares the Counting Method with system calls version and Counting Method without system call version.

Metric	False Positive Instances	Error Rate %
Counting Method with NO sys_calls	166	3%
	18 (Linux)	
	148 (Windows)	
Counting Method with sys_calls	0	0

Table 4.3: Counting Method Accuracy Results over 5509 binaries in Ubuntu 10.04 and Windows Vista with $\alpha_1 = 2,\ \alpha_2$

4.4 Other Approaches

After introducing the signature based detection system for CBSP, ideas for using other approaches have been discussed. Examples of these ideas are applying behavior base or machine learning base detection system. Let us discuss why we do not choose either of these approaches.

As we have seen in previous chapter, the decision of classification in the behavior base detection systems is taken based on the activities (behavior). In the case of cache based spy process, one should have a tool to monitor the behavior of the cache memory. To develop the detection method properly, we need to determine the main behavior that would lead to the correct classification.

Based on the survey of cache based spy process attacks, filling the entire cache with data can be considered as the primary behavior for CBSP. Monitoring such behavior requires a support from the hardware. Unfortunately, until the time of writing the present report there is no direct x86 instruction can be used for knowing whether the cache is fully occupied or not. Supporting this kind of instructions from the processor manufactures will be a good starting point for developing behavior based detection system for CBSP.

Machine learning techniques is a branch of artificial intelligence which is concerned with design and develop algorithms that can take input empirical data and produce predictions based on the provided data. The main advantages of these algorithms are their ability to recognize complex patterns leading to intelligent decisions. One disadvantage of these algorithms is that they need a considerable large number of training (empirical) data in order to provide much accurate results.

In the context of CBSP, we have tried to apply machine learning techniques to classify CBSP and non CBSP. Unfortunately, since there is a very rare number of CBSP, machine learning approach might be inappropriate to be used for classifying CBSP at the moment.

Chapter 5. Conclusion

In this thesis, we have presented some technical background that would be needed for developing micro - architectural detection system. We have surveyed the current basic CPU components that are vulnerable to be employed to breach the cryptosystems. We have presented the basic techniques that are usually used to classify malicious programs.

We have targeted cache memory MA spy process and we have analyzed the basic instructions that are usually needed by any cache based spy process. With that in mind, we could devise two methods to classify CBSP and non-CBSP. In addition, an implementation for one of these methods is developed and a new version of the bash shell is developed with the feature of the embedded IDS for CBSP. Furthermore, an evaluation and critical analysis have been presented accompanied with possible number of countermeasures.

Through this project, we have concluded with number of different aspects that need to be considered when developing IDS for MA spy process. First, understanding the selected micro architectural component is a significant step for building successful detection system. Based on the work in CBSP IDS, the component states, related CPU instructions and the common and the rare operations are important information. The second step is to understand a considerable number of inter-process based side channel attacks that are able to break the security through the selected MA component. Good implementation and demonstration is a useful approach. This will lead to better analysis for the behavior and the characteristics of spy process. It is also important to mention that this step may require a further inspection for CPU instructions.

Third, the choice of which detection method approach should be used is essential. As we have seen, there is signature base or behavior base detection system. Signature base usually provides most accurate classification leading to less number of false alarms and high number of true negative instances. However, it can be bypassed by using some techniques such as code obfuscation. Behavior based system might be difficult to be circumvented by the adversary but it may need hardware support to be implemented successfully.

Finally, the evaluation of the detection method for MA spy process is a significant issue due to the rare number of spy processes instances. It can be easy to determine the approximate number of false positive instances for the produced MA spy process IDS. However, the predication of the true positive cases might be a big challenge. In other words, how can we be sure that the produced IDS can classify the other unseen samples of spy processes? We believe that this problem might be solved by finding a method for building spy process generator for evaluation purposes. Further research for this open problem will be appreciated.

5.1 Future Work

Designing algorithms or methods to detect micro architectural spy process is an interesting area of research. The advances in this area will be influenced positively in securing most publicly environments such as cloud computing.

There are many projects that can be carried out in the area of the protection of micro architectural spy process attacks. One project is designing an algorithm to detect branch prediction base spy process. As we have seen, these kinds of attacks can threaten the security of the systems that use any cipher algorithms which have branches depending on the key such as RSA.

Another project is implementing the cache or BPU spy process IDS as a kernel module in order to be part of the operating system kernel. As we have seen, user space's IDS may not be secure enough in some settings. Therefore, developing kernel space's IDS for MA will most likely increase the overall security of IDS and makes it much harder for the adversary to bypass the system.

Bibliography

[1] W. M. Hu, "Lattice scheduling and covert channels." pp. 52-61.

[2] U. Khan. "12 million people suffered a computer virus attack in the last six months," 14-05, 2012; http://www.telegraph.co.uk/technology/news/5317908/12-million-people-suffered-a-computer-virus-attack-in-the-last-six-months.html.

[3] T. Ristenpart *et al.*, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds." pp. 199-212.

[4] S. Devadas. "Multilevel Memory (Improving Performance using little "cash")," 11-05, 2012; http://people.csail.mit.edu/devadas/6.004/Lectures/lect18/sld013.htm.

[5] O. Acıiçmez, Ç. Koç, and J. P. Seifert, "Predicting secret keys via branch prediction," *Topics in Cryptology–CT-RSA 2007*, pp. 225-242, 2006.

[6] D. P. Bovet, and M. Cesati, *Understanding the Linux kernel*: O'Reilly Media, 2005.

[7] Wikipedia. "Advanced Encryption Standard," September, 2012; http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[8] J. Daemen, and V. Rijmen. "AES Proposal: Rijndael."

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM,* vol. 21, no. 2, pp. 120-126, 1978.

[10] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, pp. 591-630: CRC, 1997.

[11] N. R. Wagner, "The Laws of Cryptography with Java Code," *Available online at Neal Wagner's home page*, 2003.

[12] J. F. Dhem *et al.*, "A practical implementation of the timing attack." pp. 167-182.

[13] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems." pp. 104-113.

[14] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *Department of Computer Science, University of Bristol, Tech. Rep. CSTR-02-003*, 2002.

[15] J. Kelsey *et al.*, "Side channel cryptanalysis of product ciphers," *Computer Security—ESORICS 98*, pp. 97-110, 1998.

[16] Y. Tsunoo *et al.*, "Cryptanalysis of DES implemented on computers with cache," *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 62-76, 2003.

[17] O. Acıiçmez, and Ç. Koç, "Trace-driven cache attacks on AES (short paper)," *Information and Communications Security*, pp. 112-121, 2006.

[18] M. Neve, J. P. Seifert, and Z. Wang, "A refined look at Bernstein's AES side-channel analysis." pp. 369-369.

[19] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," *Topics in Cryptology–CT-RSA 2006*, pp. 1-20, 2006.

[20] C. Percival, "Cache missing for fun and profit," *BSDCan 2005*, 2005.

[21] M. Milenkovic, A. Milenkovic, and J. Kulick, "Microbenchmarks for determining branch predictor organization," *Software: Practice and Experience,* vol. 34, no. 5, pp. 465-487, 2004.

[22] O. Aciiçmez, S. Gueron, and J. P. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures." pp. 185-203.

[23] A. A. Ghorbani, W. Lu, and M. Tavallaee, *Network intrusion detection and prevention: concepts and techniques*: Springer Publishing Company, Incorporated, 2009.
[24] N. Idika, and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, pp. 48, 2007.
[25] G. Vigna, "Static disassembly and code analysis," *Malware Detection*, pp. 19-41, 2007.
[26] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited." pp. 45-54.
[27] C. Cifuentes, and M. Van Emmerik, "UQBT: Adaptable binary translation at low cost," *Computer,* vol. 33, no. 3, pp. 60-66, 2000.
[28] M. Christodorescu *et al.*, "Semantics-aware malware detection." pp. 32-46.
[29] M. Christodorescu, and S. Jha, *Static analysis of executables to detect malicious patterns*, DTIC Document, 2006.
[30] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis." pp. 91-100.
[31] U. Bayer *et al.*, "Dynamic analysis of malicious code," *Journal in Computer Virology,* vol. 2, no. 1, pp. 67-77, 2006.
[32] R. Kohavi, and F. Provost, "Glossary of terms," *Machine Learning,* vol. 30, no. June, pp. 271-274, 1998.
[33] Y. Sasaki, "The Truth of F-measure," *Teaching, Tutorial materials, Version: 26th October*, 2007.
[34] T. Fawcett, "An introduction to ROC analysis," *Pattern recognition letters,* vol. 27, no. 8, pp. 861-874, 2006.
[35] D. M. Green, and J. A. Swets, *Signal detection theory and psychophysics*: Wiley New York, 1966.
[36] V. Hlavác. "Classifier performance evaluation," Septermber, 2012.
[37] B. B. Brumley, "Covert timing channels, caching, and cryptography," 2011.
[38] Intel. "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, M-Z," September, 2012; http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf.
[39] Johnson, and Richard. "Open file in kernel Mode," September, 2012; http://lkml.indiana.edu/hypermail/linux/kernel/0005.3/0061.html.
[40] HiramatsuMasami. "[RFC PATCH -tip 00/16] in-kernel x86 disassember," September, 2012; http://www.phoronix.com/scan.php?page=news_item&px=MTA4MTI.
[41] Hex-Rays. "Interactive DisAssembler IDA," Septermber, 2012; http://www.hex-rays.com/products/ida/index.shtml.