Design and Analysis of Memory Access Pattern Protection

仲野, 有登

Design and Analysis of Memory Access Pattern Protection

Yuto Nakano

December, 2018

Department of Infomatics

Kyushu University

Content

Summary **1**

Summary (Japanese) **3**

1 Introduction **6**
- 1.1 Background 6
- 1.2 Contribution of This Paper 6
- 1.3 Organisation of This Paper 7

2 Security Issues in Software Protection **9**
- 2.1 Threats against Software and its Data 9
  - 2.1.1 Memory Dump 9
  - 2.1.2 Cold Boot Attack 10
  - 2.1.3 Cache Timing Attack 11
  - 2.1.4 `ptrace` System Call 12
  - 2.1.5 Vulnerabilities 13
- 2.2 Related Works 14
  - 2.2.1 Oblivious RAM 14
  - 2.2.2 Oblivious Data Structures 22
  - 2.2.3 Oblivious Parallel RAM 23
  - 2.2.4 Private Information Retrieval 23
  - 2.2.5 Difference between ORAM and PIR 27
  - 2.2.6 Hardware-assisted Control Flow Obfuscation 28

3 Key Extraction Attack **32**
- 3.1 Introduction 32
- 3.2 Background: RSA and AES 34
  - 3.2.1 RSA Description 34
  - 3.2.2 AES Description 35
- 3.3 Attack Scenarios and Key Extraction Attack 36
  - 3.3.1 Attack Scenarios 36
  - 3.3.2 Attack against RSA 37

3.3.3 Attack against AES 38
3.4 Process Peeping Tool (PPT) 39
3.5 Key Extraction from Memory Dump 40
3.5.1 RSA 43
3.5.2 AES 45
3.6 Conclusion of This Chapter 47

4 Lightweight Access Pattern Protection 48
4.1 Introduction 48
4.2 Access Pattern Protection Problem 51
4.3 Our Scheme for Memory Access Pattern Protection 53
4.3.1 Assumptions 53
4.3.2 Set-up 54
4.3.3 Instance 1: Construction with Secure Memory 55
4.3.4 Instance 2: Construction without Secure Memory 58
4.3.5 Applying to Other Structures 58
4.4 Security Analysis 59
4.4.1 Access Pattern Hiding 59
4.4.2 Parameters: Size of Secure Memory and History Table 61
4.4.3 How much dummy data should we use? 62
4.4.4 Access Timing Hiding 62
4.4.5 Copying More Than Two Blocks 63
4.5 Comparison 64
4.6 Small Example of Proposed Scheme 67
4.7 Implementation Issues 68
4.7.1 Managing Data in Buffer Using Flags 68
4.7.2 Constructing Secure Region 69
4.7.3 Reducing Storage Overhead 72
4.7.4 Implementation Result 73
4.8 Performance Comparison with Path ORAM 74
4.9 Conclusion of This Chapter 75

5 Active Attack against ORAM 77
5.1 Introduction 77
5.2 Preliminaries 79
5.2.1 Side Channel Attack against ORAM 79
5.2.2 Ren *et al.*'s Active Attack 79
5.3 Identifying Dummy Blocks 80
5.3.1 Identifying Dummy Blocks in Square Root ORAM 82
5.3.2 Identifying Dummy Blocks in Our Proposal 82
5.4 Identifying Access Pattern 84

5.4.1 Identifying Access Pattern in Square Root ORAM 85
5.4.2 Identifying Access Pattern in Hierarchical ORAM 86
5.4.3 Identifying Access Pattern in Our Proposal 86
5.4.4 Identifying Access Pattern in Path ORAM 87
5.5 Experiment 87
5.6 Countermeasures 93
5.6.1 Parallelism 93
5.6.2 Integrity Checking 94
5.7 Conclusion of This Chapter 94

6 Application of ORAM **96**
6.1 Volume Encryption 96

7 Conclusion **98**

Acknowledgements **101**

References **103**

Index **116**

List of Publications **117**

List of Figures

2.1 Heartbleed Description 13
2.2 Overview of ORAM 15
2.3 Data Structure of Path ORAM 20

3.1 List of libraries used by RSA sample program. The boxed library is for cryptographic operations. 41
3.2 Values referred from RAM when we iterate RSA encryption and decryption ten times 42
3.3 Recovered value of the RSA key 45
3.4 Recovered value of the AES secret key 46

4.1 The number of blocks (x-axis) and ϵ (y-axis) when $m = 128$, $\ell_h = 512$ and $\delta = 20$ 65
4.2 Our Scheme 67
4.3 Constructing Secure Area Using Obfuscation 69
4.4 Performance Evaluation in Various Settings 73

5.1 Modifying bucket 4 88

List of Tables

2.1 ptrace 13
2.2 Comparison of ORAM and related schemes 30
2.3 Comparison of PIR 31

3.1 Experimental environment 42
3.2 The watch list of functions for RSA decryption and AES encryption . . . 43
3.3 Private key for the sample program 44

4.1 Summary of Difference between ORAM and Our Proposal 51
4.2 Performance of Path ORAM 74
4.3 Performance of Proposed Scheme 75

5.1 Probabilities that the adversary can detect dummy blocks 81
5.2 Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is Encrypted 90
5.3 Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is NOT Encrypted 91

Summary

When program is executed, data that is necessary for the execution is temporary stored on RAM. There are several known attacks against data on memory and they are major concern to software protection. Another concern is bugs which can cause information leakage. Oblivious Random Access Machine(ORAM) can mitigate these threats and protect sensitive data on memory. ORAM can hide an access pattern made by a program and it can protect data from an adversary who can observe memory. However, ORAM incurs large overhead and does not suit for the practical use. In this thesis, we propose more efficient scheme and evaluate the security of ORAM against stronger adversary.

Chapter 1 describes background of the research as well as main contributions and organisation of the thesis.

Chapter 2 describes threats against software and summarises ORAM researches.

In Chapter 3, we consider an attack using memory dump. When the adversary tries to extract secret data from dumped data, it is required to search entire data. Our proposed attack utilises the access pattern, more precisely the number of accesses, to efficiently extract secret data. We take RSA as an example and demonstrate the private key can be extracted when encryption and decryption of random data are iterated 10 times with fixed key. We also demonstrate the secret key of AES can also be extracted when encryption of random data is iterated with fixed key.

In Chapter4, we propose a lightweight access pattern protection scheme. The large overhead of ORAM is mainly caused by the re-shuffling of data blocks. The proposed scheme achieves its lightweightness by omitting this re-shuffling. Even when the re-shuffling is not applied, accesses occur only once can be hidden. However, accesses occur multiple times need to be hidden. The proposed scheme records history of accesses and operates dummy accesses based on the history. These dummy accesses makes it possible to hide the access pattern without re-shuffling. Moreover, we propose solutions of issues for the efficient management of data, the construction of secure region and the efficient usage of storage. We implement the propose scheme and show it outperforms the existing scheme by around 5 times.

In Chapter 5, we consider the security of ORAM under a stronger adversary. In general, the security is evaluated under the adversary who only observes the access and does not modify data on memory. This type of adversary is called passive. Another stronger type is called active who can also modify data on memory and we propose an active attack against ORAM. In the attack, the adversary first saves the initial state of the program and executes the program step-by-step while recording all transition of the internal states and output. Then, he modifies one of the data block and executes the program step-by-step. During this execution, the adversary compares the internal state and output to those when the modification is now applied. If any difference to the behaviour of the program is detected, the adversary records in which step the difference is occurred and the modification. By iterating this procedure to all data blocks, the adversary can reveal the access pattern even if ORAM is applied. We apply this attack to AES implemented with Path ORAM and show where the secret key is stored.

In Chapter 6, we consider the application of ORAM to a storage encryption. The encryption can provide confidentially to data stored in storage, but it alone cannot hide the existence of the encrypted data. There is a solution to hide the existence of encrypted volume however, it fails against an adversary who can observe the access pattern to the volume. When access to the encrypted volume is made through ORAM, its pattern can be hidden from the adversary.

In Chapter 7, we conclude the thesis.

Summary (Japanese)

メモリ・アクセスパターン保護の設計と評価

ソフトウェアの実行に必要なデータはメモリ上に一時的に格納されるため、これを狙った攻撃がいくつか知られており、ソフトウェアに対する大きな脅威となっている。また、ソフトウェアの脆弱性によって重要なデータが漏洩することも懸念される。ソフトウェアが扱う機密情報を保護するためには、これらの脅威への対策が必要であり、その一つとして Oblivious Random Access Machine(ORAM) が提案されている。ORAM を利用することで、攻撃者がメモリを監視できたとしても、メモリアクセスのパターンを秘匿することが可能であり、ソフトウェアが扱う情報を保護することが可能となる。しかし、既存の ORAM はオーバーヘッドが大きいという課題があった。そこで、オーバーヘッドを削減した手法を提案するとともに、従来よりも強い攻撃者に対する ORAM の安全性評価を行う。

第 1 章では、背景について述べ、成果および構成について説明する。

第 2 章では、ソフトウェアに対する脅威について述べ、対策として提案されている ORAM の研究動向を整理する。

第 3 章では、メモリダンプを用いた攻撃の効率化手法を検討する。メモリダンプによってソフトウェアが利用するデータを取得することが可能であるが、メモリ上のすべてのデータの検索が必要であるという課題があった。そこで、ソフトウェアのアクセスを観測することでデータの検索を不要とする攻撃を示す。具体的にはソフトウェアが行うメモリアクセスについて、各アドレスへのアクセス回数を数え上げることで検索を不要とする。例として、RSA を対象に、鍵を固定し乱数の暗号化と復号を繰り返し実行す

る。この時、鍵が格納されているアドレスに対してアクセスが集中することを示す。実験では暗号化・復号の処理を 10 回繰り返した場合に、鍵の特定が可能である。また、AES を対象とした実験では鍵を固定し、乱数の暗号化を行った場合に、秘密鍵の特定が可能である。

第 4 章では、従来の ORAM に比べて高速なメモリ・アクセスパターン保護手法の提案を行う。既存手法は ORAM ストレージ内のデータを一定周期でシャッフルする必要があり、処理負荷が高いという課題があった。そこで、一定周期のシャッフルを省略することで高速化を実現する。シャッフルなしの場合でも、ソフトウェア実行中に 1 度しかアクセスされないデータについては、安全性を確保可能である。しかし、ソフトウェア実行中に複数回アクセスされるデータは、ソフトウェアにとって必要なアクセスである可能性が高く、保護する必要がある。そこで、提案手法では、アクセスの履歴を記録しておき、各アクセスにおいて、履歴として登録されているデータに対してダミーのアクセスを実行する。これによって、定期的なシャッフルを行わずに、アクセスパターンを秘匿することを可能にし、高速化を実現する。さらに、提案手法を実装する際の課題となる、データの効率的な管理手法、安全な領域の構築手法、ストレージの効率的な利用手法、について解決策を示す。さらに、既存手法に比べて最大約 5 倍高速であることを実験的に示す。

第 5 章では、より強い攻撃者に対して ORAM の安全性評価を実施する。一般的に ORAM の安全性は、アクセスパターンを監視するだけで、データの変更は行わない攻撃者 (受動的な攻撃者) を想定して評価されている。しかし、攻撃者はデータの変更も可能であることが多く、データを変更可能な攻撃者 (能動的な攻撃者) に対する安全性評価が課題であった。そこで、能動的な攻撃に対する ORAM の安全性を評価する。攻撃者は、ソフトウェアの初期状態を記録し、さらに内部状態と出力を記録しながら 1 ステップずつ処理を実行する。その後、ソフトウェアを初期状態に戻し、データを 1 つ選択し、それを変更する。その後、ソフトウェアの挙動に変化が生じるかどうかを確認しながら、1 ステップずつ実行する。挙動に変化が生じた場合、どのデータを変更した場合にどのステップで変化が生じたかを記録し、初期状態に戻す。この操作をすべてのデータに対して繰り返すことで、各データがどのステップで利用されるかが特定可能である。PathORAM を適用した AES に対して本手法を適用し、鍵の格納先を特定可能であることを示す。

第 6 章では、ORAM の適用先として PC 等のストレージの暗号化を挙げ、研究の動向をまとめる。単にストレージ全体を暗号化しただけでは、暗号化領域の存在を秘匿することはできず、何らかの情報が保管されている可能性を攻撃者に対して秘匿できない。そこで暗号化領域の存在そのものを秘匿する機能を実現したソフトウェアが公開されているが、アクセスのパターンを攻撃者が観測可能な場合は、暗号化領域の存在を検知可能であることが指摘されている。暗号化領域の存在を攻撃者から秘匿するためには、暗号化領域へのアクセスパターンを秘匿する必要があり、このために ORAM が活用されている。

最後に、第 7 章で得られた成果に関するまとめを行う。

Chapter 1

Introduction

1.1 Background

Attacks against software are now major threats. In these attacks, the sensitive information can be leaked even if underlying cryptographic primitives are secure. Moreover, as the software becomes more and more complex, the chance of critical bugs being included inside the software becomes higher. These bugs sometimes can be exploited by an adversary.

As cloud storage services are getting common, many users outsource their data to a cloud server.

1.2 Contribution of This Paper

We discuss current status of security issues in software protection in Chapter 2. Especially software that deals sensitive information.

In order to show these threats are critical, we propose key extraction attack against RSA and AES in Chapter 3. In the proposed attack, we closely observe the memory accesses performed by RSA and AES. During the attack, we set the private key for RSA and the secret key for AES remain fixed while input messages are randomly generated. Then, iterate encryption and decryption and check the number of accesses to each address. The result shows that in both cases some of the addresses are accessed a lot more often than the others. We extract values from these addresses and confirm that the extracted values match the keys.

Oblivious RAM (ORAM) schemes can mitigate these threats, however, they impose large computational overhead and not realistic in practical use. To reduce the overhead of ORAM, we propose lightweight scheme in Chapter 4. The main idea behind the proposed scheme is to use the history of accesses. In ORAM schemes, it is important all blocks are only accessed once during an epoch (between oblivious shuffle) otherwise the information can be leaked. Our proposal arrows any block to be accessed more than once by performing dummy accesses to blocks which have been before. This allows us to omit oblivious shuffle and achieve a lightweight scheme. We also apply some techniques to improve the performance of our scheme. Then we evaluate the performance with various parameter settings and compare with one of the most efficient scheme.

The security of ORAM schemes are usually considered against passive adversaries, whose ability is limited only to observe the access. Considering a scenario in which ORAM scheme is used for software protection, the user himself can be the adversary and tries to extract sensitive information embedded inside the executable binary of software. In this scenario, the user (*i.e.* adversary) has more ability than the passive adversaries, that is modifying data stored on ORAM. We consider the security of ORAM under active adversaries and propose two attacks in Chapter 5.

We discuss two applications of ORAM schemes in Chapter 6. The first one is to secure multi-party computation and the second one is to volume encryption. In both applications, hiding access pattern, which is realised with ORAM, is the key to achieve the security requirements. Finally, we summarise research results in Chapter 7.

1.3 Organisation of This Paper

The background and purpose of this research are shown in Chapter 1. Trends and issues in recent ORAM research are shown in Chapter 2. An efficient key extraction attack is shown in Chapter 3. A lightweight memory access pattern protection scheme is addressed in Chapter 4. Some techniques to improve empirical performance of the proposed memory access pattern protection scheme and performance evaluation are also given in Chapter 4. Active attacks against ORAM schemes are shown in Chapter 5. As an application of ORAM, disk encryption is discussed in Chapter 6. Finally, a summary of my research results is given in Chapter 7.

Chapter 2

Security Issues in Software Protection

2.1 Threats against Software and its Data

Attacks against the privacy and the protection of sensitive data in memory are an area of on-going research. In this section, we study various attacks covered in the existing research that are strongly related to our work.

2.1.1 Memory Dump

Memory dump is usually used for debugging programs especially to detect buffer overflows. It also can be used to attack programs. During the execution of the program, its data is temporally stored in RAM and any process with the same privilege as the user executing the program or a root privilege can access that region. If the program is an encryption/decryption program, the decryption key must be stored in RAM and it is possible for the adversary to dump the contents in the RAM and search for the key. On Linux systems, `dd` or `ptrace()` can be used to dump memory. We can also access the memory of a running process that runs with different privileges than the user, assuming root is not involved, by using Linux capabilities. By giving the user a capability for the particular program, the user will be able to execute it without the normally required privilege. However, setting capabilities to a program/process requires, initially, root privileges or an appropriate capability.

On operating systems which use virtual memory, part of or entire memory contents of a program are sometimes moved from main memory to secondary storage (i.e. the hard disk drive). If the adversary can access the region of a disk where the pages are stored, it is easy to read the content of memory. Another possibility that the memory content can be stored on the disk is core dump. When a process is unexpectedly stopped, the memory image of that process is saved as a core file in order for debugging. The adversary can access the core file and try to analyse the memory image.

Maartmann-Moe *et al.* [58] presented a key recovery attack with memory dumping. Their attack exploits the fact that key representation in memory has a certain structure: the 128-bit original key is followed by the remaining 1280-bit expanded key in case of AES-128. Once the memory is successfully dumped, the attacker can search 128-bit original key candidates. Then the attacker applies the key schedule algorithm for all candidates. Only the correct candidate returns 1280-bit values which is identical to the values on memory. The authors also applied the same idea to attack Serpent and Twofish.

2.1.2 Cold Boot Attack

The cold boot attack is very similar to the memory dump. It involves reading an image of the RAM without needing root privilege. When RAM module is subject to cold temperatures (e.g., -50°C), it takes several minutes (5-10 minutes) for its image to degrade – this is a property that can be used to recover the contents of the memory even when power to the memory module has been switched off. Halderman *et al.* [48] demonstrated by using this low temperature characteristic that it is possible to extract the disk encryption key from a target computer. The attack proceeds in the following steps:

1. freeze the DRAM module while the computes is on or in sleep mode,

2. power off the computer,

3. boot the computer with a custom OS that extracts entire data in DRAM to an external drive,

4. Identify the decryption key from acquired RAM image, and

5. decrypt the hard drive.

Müller *et al.* [65] demonstrated the cold boot attack against an Android smartphone. Android version 4.0 and upwards support disk encryption so that users can encrypt their personal data. Their tool, called the Frost (Forensic Recovery of Scrambled Telephones), can recover the encryption key from the RAM. The attacker first installs the tool in the recovery partition of the phone, and boots the phone from the recovery partition. Frost can then recover the encryption keys from the RAM and decrypt the user's partition. It can also recover PINs with a brute force attack.

2.1.3 Cache Timing Attack

Brumley and Boneh [14] demonstrated that the RSA private key can be extracted from an OpenSSL-based web server. Their attack used the timing difference of Montgomery reduction [64] and multiplication routines. Due to the nature of Montgomery reduction and Karatsuba [51], the required time for decryption changes depending on the ciphertext C. At first, the attack guesses C and decrypts possible combinations of the 2nd and 3rd most significant bits. The decryption time shows two spikes, which are respectively Q and P in OpenSSL. This information is used to refine the guess of C and iterate the same steps until $|Q|/2$ (where Q is of $|Q|$ bits in length) most significant bits of Q are recovered. Following that, $|Q|/2$ least significant bits of Q can be factorised by Coppersmith's algorithm [21].

It has been shown that a private key for the GnuPG RSA decryption can be recovered with a cache timing attack [97]. Their attack, called Flush + Reload attack, targets the L3 cache where the attack program and the victim programs share pages. The attack program can ensure that the specific memory line is evicted from the cache and monitor the access of the victim program to the memory line. The attack consist on three steps:

1. The attack program "flushes" the cache,
2. The attack program waits for the victim program accesses the memory line,
3. The attack program "reloads" the memory line while measuring the time required to execute it.

When the attack program reloads the memory line in the third step, the required time will be different depending on whether the victim program accessed that particular memory line or not. This time difference can be used to recover the private key for RSA. The same attack can be applied to recover a secret nonce for EC-DSA [96].

More recently security issues were found in speculative executions of modern CPUs called Spectre (referred as CVE-2017-5753 and CVE-2017-5715) and Meltdown (referred as CVE-2017-5754). Spectre exploits the branch prediction and Meltdown exploits out of order execution.

2.1.4 `ptrace` System Call

The `ptrace()` system call enables one process (the "tracer") to observe, control the execution of another process (the "tracee"), examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing. Table 2.1 summarises options of `ptrace`. Other than `ptrace()` system call, there are several other system calls that may help the adversary to monitor the process and its access to memory, such as `ltrace()` for monitoring dynamic libraries and `strace()` for system calls.

There is a mode called `PTRACE_SINGLESTEP` in `ptrace`, which can load a program or can be attached to an existing program. The `PTRACE_SINGLESTEP` mode allows the attacker to execute the program step-by-step. We can also acquire the values stored in the CPU registers in each step such as program addresses, register values of operand and values in RAM pointed to by the registers by using a `PTRACE_PEEKDATA` mode.

Table 2.1: ptrace

executing step-by-step	ptrace(PTRACE_SINGLESTEP, pid, 0, NULL);
Retrieving values from register	struct user_regs_struct regs; ptrace(PTRACE_GETREGS, pid, NULL, &regs);
Dumping RAM	ptrace(PTRACE_PEEKDATA, pid, NULL, &regs);

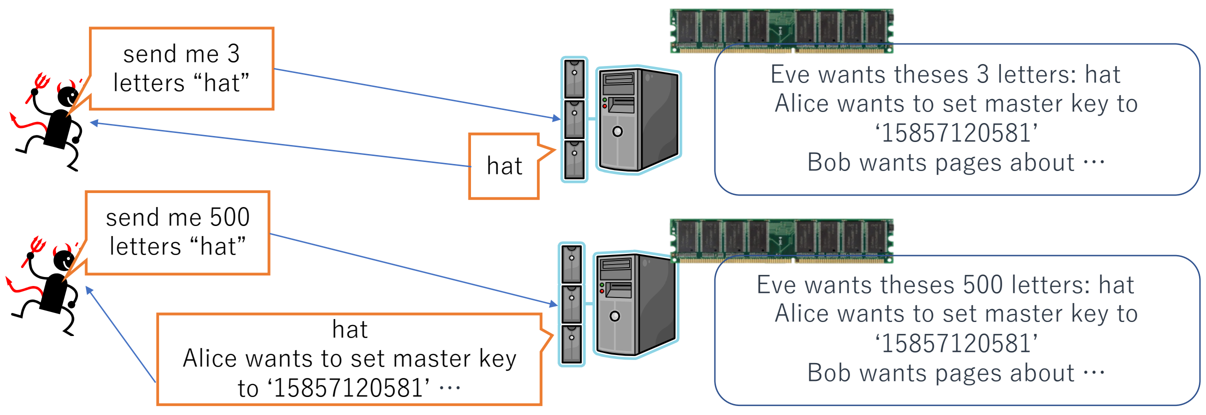

send me 3 letters "hat"
hat
Eve wants theses 3 letters: hat Alice wants to set master key to '15857120581' Bob wants pages about ⋯
send me 500 letters "hat"
hat Alice wants to set master key to '15857120581' ⋯
Eve wants theses 500 letters: hat Alice wants to set master key to '15857120581' Bob wants pages about ⋯

Figure 2.1: Heartbleed Description

2.1.5 Vulnerabilities

Vulnerabilities in software are often critical to the security of the system. In 2014, a critical bug called heartbleed , referred to as CVE-2014-0160, has been found in OpenSSL TLS Heartbeat extension [81], which makes for the attacker to recover 64 kilobytes of memory at a time [20]. If cryptographic keys are contained in this region, the adversary can retrieve these keys. Figure 2.1 describes the attack. In the attack, the adversary can send a request to repeat a word of his choice to check if the server is up. The request sent to the server also contains the length of the word, which the adversary can set arbitrary. In the example of Fig. 2.1, the requested word is 'hat' whose length is three, however, the adversary can disguise its length is 500. Then, the adversary will receive 'hat' and 497 letters which are stored next to 'hat'.

2.2 Related Works

This section summarise researches on ORAM and its related areas.

2.2.1 Oblivious RAM

Protecting sensitive information inside software has been a major concern for all software developers. Many software protection mechanisms have been proposed such as obfuscation mechanisms and combination of secure hardware. Many of those mechanisms are, however, ad-hoc and not based on theoretical foundations.

Pippenger and Fische [74] showed that a Turing Machine can be made oblivious. Here an oblivious machine was defined as the movements of the head was independent of the input to the machine and a fixed function of time. Goldreich [39], later extended by Goldreich and Ostrovsky [40], considered oblivious in RAM model and proposed Oblivious RAM (ORAM) for software protection. By using ORAM schemes, one can execute a program in a way that a polynomial-time adversary can only know how long it takes to finish the program even if the adversary can observe memory contents and accesses during execution. A brief overview of ORAM is given in Fig. 2.2. When a ORAM client, *i.e.* CPU, wants a data block a which is stored in the address x, the client request to perform read(x) to ORAM. Then ORAM translates the operation into a set of operations, in this example these are get(i), get(j) and get(k), and performs them to ORAM server *i.e.* RAM in order to retrieve the blocks a, b and c. Finally the block a is sent to the CPU. Because of the translation made by ORAM, the adversary who can closely observe physical accesses to RAM cannot distinguish if the given accesses have a certain pattern or not. The application of ORAM to realise a private access to remote servers has also been considered, for example [92, 55]. Goldreich and Ostrovsky proposed two main constructions, called *square root solution* and *hierarchical solution*.

After the first proposal of ORAM, several improvements have been proposed and many applications using ORAM schemes have been proposed, for example [1, 92, 73, 12, 25, 82, 46, 53, 55, 85, 87, 93, 88]. Improvements typically arise from the use of

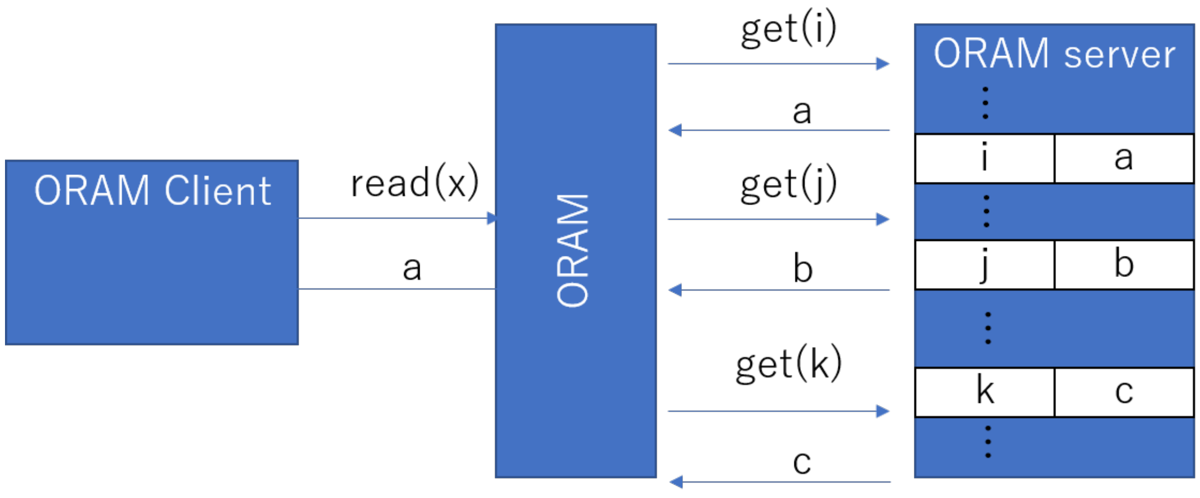


Figure 2.2: Overview of ORAM

different data structures and hash function schemes, more efficient sorting algorithms (for the oblivious shuffling), and the use of secure local (client) memory. Currently, the best amortised overhead is $O(\log^2 N/\log\log N)$ presented in the work of Kushilevitz *et al.* [53] whose security is based on the one-way function.

Batcher's sorting network [8] can be used for oblivious re-shuffling. Given a permutation π, Batcher's sorting network moves data in the address i to the address $\pi(i)$ in a way that all operations are independent of data and the permutation. During the sorting, two blocks are first read for the comparison, then they are swapped, if they are not sorted, before rewritten. Sorting N blocks requires $N\lceil\log N\rceil^2$ comparisons, which leads to the computational cost of $O(N\log^2 N)$ for re-shuffling.

One can show that the combination of the scanning method described above, and frequent oblivious re-shuffling can provide a high level of memory access pattern privacy protection. In particular, the re-shuffling following level overflow ensures that a data item is not visited twice in the same level (for $2 \le i \le \log N$) using the same hash function h_i. We note however that it was shown in the work of Kushilevitz *et al.* [53] that the choice of hash function may still leak information to adversaries.

Despite much recent progress, where the asymptotic efficiency as well as the constant terms of ORAM solutions have been improved (making it particularly attractive for remote storage access pattern protection), current solutions remain inefficient for preventing leakage of relatively limited-in-size memory access pattern. In these cases, the constant terms involved in the computational complexity make the overhead unacceptably high. This motivates the proposal of other methods, which while not achieving the same level of protection as ORAM, offer low computation (and storage) overheads, and may therefore achieve both security and performance levels acceptable in practice.

Suppose that we are accessing the data 'a' several times in the ORAM. The $\mathsf{Read}(a)$ process searches for 'a' in the set of $(A_t \cup P_t)$ where A_t indicates all element in the top level and P_t indicates the path which is determined by the hash function $h(a)$ and $h(dummy)$. At the next access to 'a', the read operation searches for 'a' in the set of $(A_{t+1} \cup P_{t+1})$. Here, we have to consider two cases: 1) re-shuffle has been done or 2) re-shuffle has not been done.

1. Since the re-shuffle has not been done, we are using the same hash functions. But, 'a' must be in the top level, and search dummy location for the lower levels. Hence P_t and P_{t+1} have no correlation.
2. Since the re-shuffle has been done, new hash functions are chosen. Therefore P_t and P_{t+1} have no correlation.

As shown, P_t and P_{t+1} are indistinguishable. And A_t and A_{t+1} are also indistinguishable, since the all data always is accessed in the top level. Therefore the adversary cannot tell if 'a' is being accessed more than once. Re-shuffling the elements is critical for the security of ORAM, but it is also critical for the cost of ORAM.

Comparison of computation overhead, storage overhead and security is summarised in Table 2.2. As shown in Table 2.2, the security of SDSFRY [88] is based on the randomised encryption. It is an encryption algorithm which uses randomness on the encryption and resulting ciphertexts correspond to the same plaintext are indistinguishable from one another. In the following sections, we describe some of ORAM schemes more in detail.

2.2.1.1 Square Root ORAM

The square root solution attaches area called *shelter*, which can contain $\sqrt{N}$ blocks, to main memory and adds $\sqrt{N}$ dummy blocks. Then original N data blocks and $\sqrt{N}$ dummy blocks are permuted by the secret permutation π. When the client accesses to data block x, one first scans entire shelter to check if x is inside the shelter. If the block is already inside the shelter, the dummy block will be copied to the shelter. Otherwise, the block x will be copied from $\pi(x)$. As proposed by Zahur *et al.* [98], the need of dummy blocks can be eliminated by performing the copy of real blocks from random addresses instead of the copy of dummy blocks.

As one block is copied to the shelter at each access, the shelter gets full after $\sqrt{N}$ accesses. Once the shelter gets full, new permutation π' is chosen and all blocks are re-shuffled according to π', which requires $O(N \log^2 N)$ computation. On each access, $O(\sqrt{N})$ physical accesses are required. Therefore, the amortised overhead is given by $O(\sqrt{N} \log^2 N)$. The amortised overhead can be optimised by setting the shelter size to $\Theta(\sqrt{N} \log N)$. The security of this scheme is based on an one-way function. The one-way function is a function which can be easily computed in one-way, that is computing $y = f(x)$ for given x, however, computing the other way, that is $f^{-1}(y)$ for given y, is computationally infeasible.

2.2.1.2 Hierarchical ORAM

In the hierarchical solution for RAM with N items, the data structure is organised in $n = O(\log N)$ levels consisting of hash-tables with 2^i buckets $(1 \leq i \leq n)$, each bucket containing $O(\log N)$ items. The storage requirement is therefore $O(N \log N)$. Data is mapped into the levels using different secret hash functions h_i, known by the client only.

An element r is read with the `read`(r) operation as follows:

1. Scan the entire first level.

2. If the element is not found, then for each level i $(2 \leq i \leq n)$, compute $j = h_i(r)$ and read the j-th bucket in the level i buffer, until the requested element is found.

3. Once the element is found (including during the scan of the first level), continue with the procedure in step 2. by reading a dummy location in each level i given by $h_i(\text{dummy} \circ t)$, where t is a counter.

4. The entire first level is scanned again, and the element r is written to the first level.

When we wish to update an element r with the `write`(r, x) operation, we perform the `read`(r) procedure as above, but insert the new value x into the first level at the step 4. It follows that the number of access operations for a data item request (either `read` or `write`) is $O(\log^2 N)$.

Because of the writing in step 4, buffer levels eventually overflow with data. Indeed, after 2^i requests, the buffer at level i will be full, and its full contents are moved down to level $i + 1$. Every time content is moved to a lower level, all data in both levels are permuted and a new random hash function is chosen. This process requires the re-shuffling of data, which needs to be done *obliviously.* This is the most complex component of the construction, and is the main factor in its (amortised) complexity overhead.

Based on the constructions above, the Goldreich's scheme [40] requires $O(N \log N)$ in storage, and has amortised computation overhead of $O(\log^3 N)$ per query using the AKS algorithm for the oblivious sorting [2]. Due to the very large constants, the complexity is considered $O(\log^4 N)$ in practice (by using Batcher's sorting network [8]).

2.2.1.3 Path ORAM

Recently the third construction called the tree construction is proposed by Shi *et al.* [82]. It has an N-element database in a binary tree of depth $\log N$. Each node in the tree has a bucket which can store k data items. Their scheme uses $O(N \log N)$ storage at the server. The client needs $O(1)$ memory and computation complexity is $O(\log^3 N)$. The scheme is proven secure given the access to a random oracle. Later, the tree construction is optimised by Gentry *et al.* [35]. The optimisations can reduce the storage overhead by an $O(k)$ factor and computation complexity by an $O(\log k)$ factor, where k is a security parameter.

Path ORAM is one of the most efficient scheme [88]. Data in Path ORAM is also maintained as a binary tree of the height L and it has 2^L leafs. Each node is called a *bucket* and always stores Z blocks. When the number of the data blocks is less than Z, dummy blocks will be padded. It also has a secure area in which *stash* and *position map* are stored. The stash acts like a shelter in square root ORAM and it can hold up to C blocks. The position map is a table relating the leafs and blocks inside each leaf. The data structure of Path ORAM is shown in Fig. 2.3.

When a data block a is accessed, Path ORAM retrieves that leaf ℓ has the data block a from the position map. Then it accesses the path to leaf ℓ and copies all data blocks into the stash. The data block a is now in the stash, it accesses a. After the access, choose new leaf ℓ' and update the position map. Also all data blocks in the stash are evicted into the buckets on path ℓ'. Dummy blocks are filled an empty space in the buckets on path ℓ'. When the block a is retrieved from the server, it is always written back to a new random location. Therefore, the sequence of accesses, from the server's view point, is indistinguishable from the random sequence.

Stefanov *et al.* also proposed a recursive Path ORAM for reducing the size of the position map which has to be stored in the secure area. The recursive Path ORAM has X ORAMs; ORAM_0, ORAM_1, …, ORAM_X where ORAM_0 contains the data blocks, and ORAM_{i+1} contains the position map for ORAM_i. Each recursion compress the size of the position map by a constant factor, and it can be compressed to $O(1)$ after a logarithmic number of recursion.

2.2.1.4 GORAM

Maffei *et al.* [59] presented a framework for privacy preserving cloud storage called GORAM. In GORAM, the user can store his data to cloud storage, and share data with other users with selectively allowing read and write accesses. Moreover these access patterns are oblivious from the server. In the original ORAM scheme and most of the following works only consider the single and honest client. On the other hand, Maffei *et*

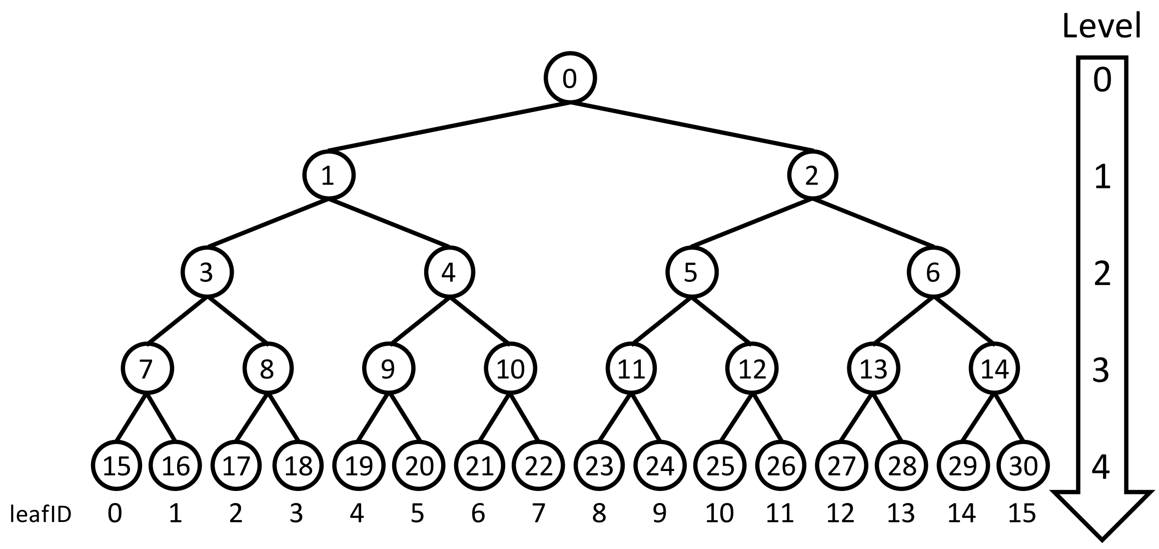


Figure 2.3: Data Structure of Path ORAM

al. considered multiple and possibly malicious clients and provided security proofs.

2.2.1.5 Circuit ORAM

When ORAM is used as a building block to realise a secure multi-party computation, its circuit size is critical to the performance of the resulting MPC. Though Path ORAM is one of the most efficient scheme, its eviction is complex and is not as efficient in terms of its circuit size as expected. Circuit ORAM is also a tree-based scheme, like Path ORAM, but with simpler eviction and can be implemented as a small circuit [90]. The eviction of Circuit ORAM, the client first chooses one block and keeps it. Then it can be evicted to somewhere along the path. The eviction algorithm is initiated from the bottom of the tree (i.e. leaves). It first searches a new path where the block can be contained, and move the block from the current path to the new path. If there is not such block, the eviction algorithm chooses different path. Each logical access to a block is translated to the access to a random path, retrieving part is proved to be indistinguishable from the random sequence. The eviction algorithms are also proven to be indistinguishable from the random sequence. Circuit ORAM is later extended to Circuit OPRAM to support parallelism [15].

2.2.1.6 Ring ORAM

Ren *et al.* [75] proposed Ring ORAM which achieves online bandwidth of $O(1)$ and the amortized overall bandwidth is independent of the bucket size. Ring ORAM is also a tree-based ORAM and shares its characteristics with Path ORAM [88] and SSS ORAM [87]. Server storage of Ring ORAM is organised as a binary tree and each bucket in the tree can hold up to $Z + S$ blocks where Z is the number of real blocks and S is the number of dummy blocks. The client storage is consisted of a position map and a stash.

The main difference between Path ORAM and Ring ORAM in accessing ORAM server is that Ring ORAM only reads one block from each bucket while Path ORAM reads all blocks. Hence Ring ORAM achieves smaller overhead. In order to fulfil the security requirement even when only one block is accessed in the bucket, each bucket will be shuffled after it runs out of un-accessed dummy blocks. Another difference is in an eviction. In Path ORAM, the eviction is performed for every access while Ring ORAM is every A accesses where A is a public parameter. As the result, Ring ORAM outperforms Path ORAM by 2.7×.

2.2.1.7 PRO-ORAM

Tople *et al.* [89] proposed PRO-ORAM which achieves constant latency to hide a pattern of read accesses. While it only protects read access, it achieves constant latencies per access. The key idea of PRO-ORAM is to decompose read access into 'access' and 'shuffle' and to prepare two separate copies of data. One copy, called active array, is used to perform access operation. The other copy, called next array, is used to perform shuffle operation, and resulting state after $\sqrt{N}$ accesses can be active array for the next epoch. They also utilise trusted hardware on the server side to delegate access and shuffle operation. Their scheme can be useful when clients repeat read-only accesses to data on the clouds after the initial upload. There are several services which fit in this model, for example photo sharing services, document sharing services and music/video streaming services, where the content owners upload their own contents to the cloud and other users consumes them.

2.2.1.8 PIR-MCORAM

Despite many proposals of ORAM schemes, majority of these schemes focused on a single client scenario. There is only a few schemes which support multi honest clients scenario. Maffei *et al.* [60] proposed PIR-MCORAM, one of the schemes that supports multi client. PIR-MCORAM also considers malicious clients.

2.2.2 Oblivious Data Structures

Despite the latest progress in the research of efficient ORAM schemes, even the most efficient scheme incurs relatively large overhead and it is not efficient enough for a practical use. The overhead is due to protecting arbitrary access patterns and it can be reduced if one considers only the specific pattern. Wang *et al.* [91] considered access patterns with a certain degree of predictability and proposed two techniques to make data structures and algorithms oblivious. Their proposal is asymptotically and empirically more efficient compared to ORAM. In their proposal, sequence of accesses is considered as a graph, called the access pattern graph, in which each node corresponds to memory cell and edges correspond to accesses. When the access pattern graph is a rooted tree with bounded degree, a parent node has pointers to its children. Then the location of children can be retrieved by accessing the parent node, hence no need to lookup the position map to locate the children. This is called the pointer-based technique. Another one is called locality-based technique. This is for the access pattern graph of a low doubling dimension. In this graph, nodes are partitioned into clusters and clusters keep pointers to their neighbouring clusters. When accessing a data block in the cluster, data blocks in the neighbouring clusters will be pre-fetched. This makes sure that for the following accesses, required data blocks are already in the client's storage.

2.2.3 Oblivious Parallel RAM

One of the limitations of ORAM is that the protected program cannot be paralleled even if the original program can enjoy large improvement in terms of performance. Boyle *et al.* [13] proposed Oblivious Parallel RAM (OPRAM) to overcome this limitation. When transforming ORAM into OPRAM, multiple CPUs may perform accesses to the same address, which leaks information that these CPUs are requesting the same data. In order to avoid the collision of accessing addresses, Boyle *et al.* made only the representative CPU perform the real access and distribute the data block to other CPUs. In tree-based ORAM, the data block will be re-inserted to the root node after the access. If the same policy is applied to the OPRAM, the root node can be easily overflown. Instead of re-inserting to the root node, their scheme re-inserts to nodes in level $\log m$. After inserting m blocks, m flush operations will be applied to push inserted blocks down towards leaves.

Inspired by the work of Boyle *et al.* [13], Chen *et al.* [16] presented an OPRAM scheme based on Path ORAM. First they proposed Subtree-ORAM, which only support single client, but does support multiple access at a time. Then extend it to Subtree-OPRAM. In Subtree-OPRAM, all clients emulate the Subtree-ORAM and interact each other. They also presented a generic transformation turning any (single-client) ORAM scheme into an OPRAM scheme.

2.2.4 Private Information Retrieval

Emerging of cloud storage services enables users to store and access their data very easily. Moreover, users can store huge data which is too large to store locally. However, it also raises a new security challenge, which is how to protect user's privacy. When the user request data to the server, the server will notice which data the user wants. The server might be curious about the user's request and try to extract user's private information. Encryption can provide confidentiality of data, however, sometimes it is not sufficient. For instance, if the particular file is accessed very often, it implies that file is more important for the user than the ones accessed less often. An adversary may try to delete those files.

Private Information Retrieval is a technology that allows clients to query a database in a way that even the database cannot learn anything about the clients' queries. A trivial solution is to download everything every query. This solution, however, is impractical due to a high communication overhead and a high storage overhead at the client side.

There are two PIR schemes: computational PIR (CPIR) [17, 54] and information theoretic PIR (ITPIR) [19]. In the CPIR, the client queries the database an encrypted query and the server returns the encrypted result to the client in order to prevent the computationally limited server from learning anything about the query. For example, Chor and Gilboa's scheme [17] and Kushilevitz and Ostrovsky's scheme [54] require $O(N)$ server storage for storing data of size N, the same as evaluated in ORAM schemes, and communication overhead is $O(N^{\epsilon})$. We evaluate the efficiency of PIR schemes in terms of the communication cost, not the computational cost, as the efficiency is generally measured by the communication cost. The security of Chor and Gilboa's scheme is based on an one-way function and Kushilevitz and Ostrovsky's one is based on the hardness of deciding quadratic residuosity. An integer q is called quadratic residue (QR) if there exists an integer x such that $x^2 = q \mod n$, and q is called quadratic non-residue (QNR) otherwise. It is considered hard to predicate if q is QR or QNR when n is a product of two distinct prime numbers of equal length. On the other hand, the ITPIR can offer perfect security, that is, the server cannot acquire any information about the client's query even if the server has unlimited computational resources and unlimited time. In order to achieve the ITPIR, usually multiple servers are required and these servers are assumed not to collude. The t-private ℓ-server PIR can information theoretically guarantee the privacy of the query even if up to t out of ℓ servers collude. Beimel and Stahl [9] introduced a notation called t-private k-out-of-ℓ PIR in which k out of ℓ servers need to respond and up to t servers may collude without compromising the security. In addition they examined a situation that v out of k servers can return incorrect answers, due to a malicious servers or database failure, which is termed as t-private v-Byzantine-robust k-out-of-ℓ

PIR. Chor *et al.* [18, 19] have proposed several schemes. Their proposals enable the client to retrieve one bit with $O(\sqrt{N})$ communication cost in a simple ℓ-server scheme, $O(N^{1/\ell})$ communication in a general ℓ-server scheme and $\frac{1}{3}(1+o(1)) \cdot \log^2 N \cdot \log\log(2N)$ communication cost in $\frac{1}{3}\log N + 1$-server scheme. Their scheme is information theoretic secure.

After the first proposal of PIR [18], several improvements in terms of communication cost have been shown [83, 84, 4, 9, 38, 92, 45, 26]. The scheme proposed by Goldberg [38] requires $O(N)$ server storage and $O(1)$ communication overhead, and is information theoretic secure. This scheme was implemented as an open-source project on SourceForge [37]. Smith *et al.* [83, 84] proposed a scheme with a tamper-proof device. The client sends the secure coprocessor (SC) with encrypted query. The SC receives the query and decrypt it. Then the SC reads the entire database and get the requested data item. When returning the data item to the client, the SC encrypts the item and send it back to the client. Williams and Sion [92] proposed a single-server PIR with an ORAM and a secure coprocessor. The communication and computational complexities of their scheme is $O(\log^2 N)$ and $O(\sqrt{N})$ client storage is required. The security of their scheme is proven when it has the access to a random oracle. Devet *et al.* [26] improved the scheme presented by Goldberg [38] and evaluated the performance of the scheme. They showed that their implementation was several thousands times faster than Goldberg's scheme [38].

Bao *et al.* [6] and Schnorr *et al.* [80] independently proposed similar PIR schemes using homomorphic encryption. Their schemes requires only $O(1)$ computation to be done on-line. The idea of the schemes is to do as many operations as possible off-line to achieve practical on-line overhead. However, because of heavy off-line computation, the client has to wait until the server is ready to be queried, and this latency may make the schemes less practical. The schemes work as follows: When the client wants to obtain a data item, which is encrypted with the servers secret key and publicly available, first the client download the item and encrypts it with the client's secret key. After the encryption at the client, the client sends the item to the server and asks to remove the server's encryption. Finally, the client obtain the data item by decrypting one's own encryption.

Asonov and Freytag's scheme [4] also assumes the SC inside the server and SC first shuffles the entire database according to a random permutation π. When the client request the data item i_1, the SC fetches the item from $\pi(i_1)$. This only requires $O(1)$ of computation and communication. For the second query of requesting the item i_2, the SC first has to read $\pi(i_1)$ and then read $\pi(i_2)$. When $i_1 = i_2$ (the second and the first query request the same data item), the SC reads $\pi(i_1)$ and a random item in order to hide the fact that the client is reading the same data item twice. Therefore, for the n-th query, the SC has to read all previously read items before reads $\pi(i_n)$. At a certain point, the SC has to pick a new permutation π' and shuffle the database with π'.

The PIR schemes can protect user's privacy from an honest-but-curious servers. However, PIR schemes can not offer a protection from a dishonest user. Gertner *et al.* [36] proposed a symmetric private information retrieval (SPIR) that prevents the user from learning additional information. Henry *et al.* [49] considered an application of SPIR for e-commerce and proposed a protocol that extended the PIR scheme [38] to a priced symmetric private information retrieval (PSPIR). Their PSPIR scheme maintain user's anonymity and does not leak any information about the record of user's purchases.

Recently implementations of CPIR and evaluating performances on real environments are attracting more attentions. Melchor and Gaborit [63, 62] proposed a lattice-based new scheme with a reasonable communication cost and with computational complexity being improved by a factor of one hundred. Later, Olumofin and Goldberg [70] implemented the lattice-based scheme and evaluated the performance. They demonstrated that the overhead of Olumofin and Goldberg's scheme was 10 to 1000 times smaller than the trivial scheme (i.e. downloading entire database). Ding *et al.* [28] proposed a PIR scheme, which does not require full-shuffle of the database. By shuffling only a portion of the database, their scheme achieves $O(\log N)$ communication cost, $O(1)$ runtime computation cost, and $O(\sqrt{N \log N/k})$ overall amortized computation cost per query, where k is the trusted cache size.

Recent schemes can outsource the database to untrusted servers and yet can protect both the privacy of the database owner and clients. Huang and Goldberg [50] proposed a scheme for outsourcing Private Information Retrieval. Their scheme requires $O(\sqrt{N})$ computational overhead and the server stores $O(\sqrt{N})$ data. The security of the scheme is proven when it has the access to a random oracle. They also implemented their scheme and evaluated the performance. When the client updates 1MB record in the 1 TB database, an amortised end-to-end latency is smaller than 300 ms.

The comparison of PIR schemes is summarised in Table 2.3. In Table 2.3, we compared communication overhead of schemes since they are generally evaluated by communication overhead[1], while ORAM schemes are evaluated by computational overhead.

2.2.5 Difference between ORAM and PIR

The target of both ORAM and PIR is the same: how to efficiently protect the pattern of accesses. The main difference is that (full functional) ORAM supports both the `read` and `write` operations to RAM (or server) while PIR usually consider only the `read` operation. As the functionality of PIR is limited compared to ORAM, PIR tends to be more practical, in terms of communication and storage cost, than ORAM. PIR works very well when one server or cloud operator provides a large database and many clients want to download part of the database in a privacy preserving manner. Video and audio streaming services are good applications of PIR, where providers upload their video and/or audio files to the cloud and subscribers enjoy the contents while hiding their privacy. However, as PIR does not support `write` operation, it does not work well in some services for example file hosting services where clients upload their files and they often update part of their files. Neither does PIR work well when a software developer wants to protect his/her software from reverse engineering. ORAM is usually less practical than PIR in terms of performance, however, ORAM supports both `read` and `write` operations. Hence ORAM is suitable for protecting access pattern in a database

[1]All literatures referred in Table 2.3 evaluate their own scheme in terms of the communication overhead, except Huang and Goldberg's scheme [50]

which is often updated and a software protection.

PIRs usually offer less functionality (i.e. protection only for read accesses) than full functional ORAMs, hence they are lighter than ORAMs and oppose smaller overhead. Because of their lightweightness, PIRs are more attractive for users who wish to hide only their reading pattern. Some ORAM schemes are practical and their applications to realise a private access to remote servers have also been considered. Our scheme is as light as one of the most efficient PIRs, our scheme can be used instead of PIRs without sacrificing performance.

2.2.6 Hardware-assisted Control Flow Obfuscation

ORAM constructions remain too expensive to be implemented on embedded processors. Zhuang *et al.* [99] proposed a practical, hardware-assisted scheme for embedded processors, with low computational overhead. Their *control flow obfuscation* scheme for embedded processors employs a small secure hardware obfuscator (called *shuffle buffer*) to hide program recurrence. We give a brief description of the scheme below; for more details, refer to [99].

Let n be the size (in blocks) of memory, and $m \ll n$ be the size of the shuffle buffer. The shuffle buffer is within the CPU trusted boundary, i.e. it is considered secure local storage (cache), and an adversary is not able to observe access pattern in the shuffle buffer. As in other parts of this paper, we assume that data is stored encrypted and access operations consist of sequential `read` and `write` operations. A random permutation is initially applied to data before loading it to RAM. The scheme then works as follows.

1. The first m blocks in memory are moved to the shuffle buffer.
2. When making a request for a data item, if the block is found in the shuffle buffer, access the block.
3. If the block is not found in the shuffle buffer, pick a random block in the shuffle buffer and swap it with the requested data block in memory (the accessed data item is now in the shuffle buffer).

4. When the program finishes its entire process, the full contents of shuffle buffer are written back into memory.

Note that item 3. implies that the permutation used to map data in memory is *dynamically* modified as the program runs. Although the dynamic secret permutation helps to protect the privacy of individual items being accessed (or being repeatedly accessed), it also means that the scheme needs to make use of a block address table to map data items into memory (describing the permutation at time t). As a result there is the storage requirement of size $O(n)$ within the trusted environment to represent this mapping (albeit with constant < 1).

The scheme trades security for low overhead. Besides the costs of having a secure on-chip buffer, the scheme trivially leaks information about access of data items during execution of program. In fact, the lack of memory access in step 2 indicates that when step 3 is executed, one knows the exact block being accessed (the one in memory, being brought into the buffer). Thus a step 2 followed by a step 3 indicates that the data items being accessed are definitely distinct (i.e. there is no 2-recurrence at this particular stage). Likewise, a step 3 followed by a step 2 indicates a 2-recurrence with probability $1/m$. Furthermore, an access in memory to a data item which was previously swapped out from the buffer indicates with high probability the existence of repeated access to a particular data. These could be confirmed by running the program several times.

Despite the limitations of the proposal, it adds a very low overhead to the program execution (besides the read/write and encryption/decryption overhead, only an extra read/write operation due to cache misses). We will adapt some of the ideas from this scheme in our proposal.

Table 2.2: Comparison of ORAM and related schemes. r is a small constant of $r > 1$. B is a size of data block ($B = \chi \log N$). ℓ_m and ℓ_h are respectively the size of buffer and history table. k is the security parameter whose typical setting may be $k \in [50, 80]$.

	Computational Overhead	Server Storage	Client Storage	Security
GO [40]	$O(\log^3 N)$	$O(N \log N)$	$O(1)$	random oracle
PR [73]	$O(\log^2 N)$	$O(N)$	$O(1)$	random oracle
BMP [12]	$O(\sqrt{N})$	$O(N)$	$O(\sqrt{N})$	random oracle
DMN [25]	$O(\log^3 N)$	$O(N)$	$O(1)$	information theoretic
GM [43]	$O(\log N)$	$O(N)$	$O(N^{1/r})$	one-way function
GMOT [44, 46]	$O(\log N)$	$O(N)$	$O(N^{1/r})$	random oracle[†]
SCSL [82]	$O(\log^3 N)$	$O(N \log N)$	$O(1)$	random oracle
SO [56]	$O(\log N)$	$O(N)$	$O(1)$	one-way function
KLO [53]	$O(\frac{\log^2 N}{\log \log N})$	$O(N)$	$O(1)$	one-way function
SSS [87]	$O(\log^2 N)$	$O(N)$	$O(\sqrt{N})$	random oracle
WS [93]	$O(\log^2 N \log \log N)$	$O(N)$	$O(\log N)$	one-way function
SDSFRY [88]	$O(\frac{\log^2 N}{\log \chi})$	$O(N)$	$O(\frac{\log^2 N}{\log \chi})) \cdot \omega(1)$	randomised encryption
GGHJRW [35]	$O(\frac{k \log^2 N}{\log k})$	$O(N)$	$O(1)$	random oracle
ZZLP [99]	2	$O(N)$	$O(1)$	probabilistic
Ours 1 [66]	$4 + 2\ell_h$	$O(N)$	$O(1)$	probabilistic
Ours 2 [66]	$2(\ell_m + \ell_h + 1)$	$O(N)$	$O(1)$	probabilistic

[†] This can be realised without a random oracle by using the work of Damgård *et al.* [25]

Table 2.3: Comparison of PIR. ℓ is the total number of servers and k is the minimal number of servers which are available at the time of retrieval. c is a constant of $c \geq 2$ and ϵ is a small constant of $\epsilon \geq 0$.

	Communication Overhead	Server Storage	Client Storage	Security
CG [17]	$O(N^{\epsilon})$	$O(N)$	$O(1)$	one-way function
KO [54]	$O(N^{\epsilon})$	$O(N)$	$O(1)$	quadratic residuosity problem
CKGS [18, 19]	$O(\log^2 N \log\log N)$	$O(N \log N)$	$O(1)$	information theoretic
AF[4]	$O(1)$	$O(N)$	$O(1)$	secure coprocessor
G [38]	$O(1)$	$O(N)$	$O(1)$	information theoretic
BS [9]	$O(N^{1/k})$	$O(N)$	$O(1)$	information theoretic
WS [92]	$O(\log^2 N)$	$O(N \log N)$	$O(\sqrt{N})$	random oracle
GMOT [45]	$O(1)$	$O(N)$	$O(N^{1/c})$	one-way function
HG [50]	$O(\sqrt{N})^{\ddagger}$	$O(N)$	$O(1)$	random oracle

‡ They evaluated the scheme in terms of computational overhead.

Chapter 3

Key Extraction Attack

3.1 Introduction

The growth of various services on the Internet has given rise to a dramatic increase in the information that is exchanged over Internet protocols. Sensitive information in private e-mails, confidential documents, e-commerce and other financial transactions need to be guarded against eavesdropping. In order to protect the communication between two network hosts, the Secure Sockets Layer [33] and Transport Layer Security [27] (SSL/TLS) are commonly used. OpenSSL[1] is the one of most commonly used open source libraries for the SSL and TLS protocols. The core library offers implementations for various cryptographic algorithms such as Blowfish [79], Camellia [3], AES [67], SHA-1 [68], SHA-2 [68], RSA [78], DSA [69] and Elliptic Curve [52] and other utility functions. The library is implemented in the C programming language and available for Windows as well as all Unix-based operating systems such as Linux, Mac OS X and Android. Recently, a critical bug, referred to as CVE-2014-0160, has been found in OpenSSL TLS Heartbeat extension [81], which makes for the attacker to recover cryptographic keys by reading 64 kilobytes of memory at a time [20].

Any unauthorised access to cryptographic keys constitutes a security breach. Tamper-proof devices [22, 100, 99] and obfuscation [7, 57, 41] can be used when a program is not

[1]See: `https://www.openssl.org/`.

running but not during program execution. It is well-known that the cryptographic keys are present in the random access memory (RAM) during the execution of a program; a knowledge that an adversary can use to extract the keys from the RAM [48, 58, 65]. Protecting the keys or any other valuable information from unauthorised access during program execution is an important area of on-going research. Oblivious RAM schemes and related works, such as [39, 40] can protect the RAM access patterns of programs from unauthorised access. However, these schemes require trustworthy and secure CPUs for the protection, and cannot prevent attacks where the attacker can access the CPU and extract information such as operations, access to memory addresses of operations and values stored in those addresses.

In this chapter, we present a new attack method that can extract a private key for RSA and a secret key for AES from dumped memory image. Then we implement a tool called the Process Peeping Tool (PPT) to demonstrate the attacks against RSA and AES. The PPT enables us to analyse the structure and behaviour of the target program by observing its memory use. The PPT can also statistically analyse the data that is acquired from the RAM, that is, when a function inside the target program makes request to a certain data blocks, the PPT can record which function made the request to which data block and how many times the access is made. Thus, it enables the attacker to determine cryptographic keys by observing memory accesses made by cryptographic functions. We use the PPT to extract cryptographic keys (both for RSA and for AES) from sample programs, which use the OpenSSL library. In the key recovery attack against RSA, we iterate through cycles of encryption and decryption and observe the memory values, which the decryption function accesses. Assuming that the private key remains fixed during the execution of the program, the number of accesses to the key grows as we iterate through encryptions and decryptions. The key can be extracted by observing the accesses to memory. In the case of AES, we encrypt a random number with one fixed secret key and observe the memory values accessed by the encryption process. We demonstrate that the secret key can be extracted by observing the memory access patterns. The address of any shared libraries has to be public for the PPT to be able to analyse the program. Using a static library and deleting symbol information makes it harder for the attacker to obtain the keys. However, the keys can still be extracted once the attacker determines the functions that need to be observed. Although, adding dummy data and/or dummy operations also make the attack harder, the keys can be extracted with the help of additional information. Even when dummy data and dummy accesses are added, these can be distinguished from the actual data since the dummy data and accesses do not affect the output of the program.

3.2 Background: RSA and AES

In this section, we briefly describe the basics of the two encryption algorithms: the RSA [78] asymmetric cipher and the AES [67] block cipher.

3.2.1 RSA Description

RSA was invented by Rivest, Shamir and Adleman [78] as one of the first practical public key cryptography. The security of RSA is based on the difficulty of factoring the product of two large prime numbers. Now, it is the most widely used public key cryptosystem and also used as the standard public key cryptography for SSL/TLS. RSA consists on three algorithms: key generation, encryption and decryption. A pair of keys consisting of the public key and the private key can be generated in the key generation algorithm. In the key generation, one first finds two prime numbers p and q, and let $N = pq$ and $\phi(N) = (p-1)(q-1)$. Then one chooses e which satisfies $E < \phi(N)$ and $\gcd(E, \phi(N)) = 1$. The public keys are D and N, and the private key can be given by $D = E^{-1} \mod \phi(N)$. With the public key, one can encrypt a plaintext M with a public exponent E and modulus N as ciphertext $C \leftarrow M^E \mod N$. The ciphertext is decrypted as $M \leftarrow C^D \mod N$, where D is a private exponent.

The Chinese Remainder Theorem (CRT) is often used to perform these exponentiation operations and it is also used in OpenSSL library. The exponentiation is computed in two steps:

1. Evaluate $M_1 = C^{D_1} \mod P$ and $M_2 = C^{D_2} \mod Q$ (here $D_1 = D \mod \phi(P)$ and $D_2 = D \mod \phi(Q)$ are precomputed from D), and then

2. combine M_1 and M_2 using CRT to yield M.

3.2.2 AES Description

AES is based on Rijndael [24] and has been standardised by NIST as FIPS197. It is now one of the most widely used symmetric key encryption algorithms, and OpenSSL also uses AES. It makes use of a substitution-permutation network. It supports three different key lengths: 128-bits, 192-bits and 256-bits. The key length decides the number of rounds N_r: 10 for 128-bits, 12 for 192-bits and 14 for 256-bits. The encryption can be described as Algorithm 1.

Algorithm 1 AES [67]

```
1: KeyExpansion {expands the initial key to round keys.}
2: AddRoundKey {adds a round key to the state.}
3: for i=1 to $N_r - 1$ do
4:   AddRoundKey
5:   SubBytes {applies 8-to-8-bit substitution to the state.}
6:   ShiftRows {shifts the $i$-th row of the state to $i$ positions to the left.}
7:   MixColumn {mixes the column $C$ by multiplying a $4 \times 4$ maximum distance
     separable matrix $M$ as $C \leftarrow M \times C$.}
8: end for
9: SubBytes
10: ShiftRows
11: AddRoundKey
```

Here we describe **KeyExpansion** for AES-128 in detail. It has two functions, which are `SubWord()` and `RotWord()`. `SW()` takes a four-byte input and applies the same 8-to-8-bit substitution as `SubBytes`. `RW()` performs a cyclic shift to the 32-bit word $[a_0, a_1, a_2, a_3]$ and returns $[a_1, a_2, a_3, a_0]$. **KeyExpansion** also uses the array `Rcon`(i)

which contatins $[x^i \bmod x^8+x^4+x^3+x+1, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}]$ where $x = \texttt{0x02}$. The round key $w[i]$ $(0 \leq i \leq 43)$ can be generated from the initial key $[key[0], key[1], key[2], key[3]]$ as following:

$$w[i] = \begin{cases} key[i] & (0 \leq i \leq 3) \\ \texttt{SW}(\texttt{RW}(w[i-1])) \oplus \texttt{Rcon}(i/4) \oplus w[i-4] & (4 \mid i) \\ w[i-1] \oplus w[i-4] & (4 \nmid i, i > 4) \end{cases}$$

3.3 Attack Scenarios and Key Extraction Attack

We first consider three attack scenarios, then propose attacks against RSA and AES.

3.3.1 Attack Scenarios

The private key of RSA and the secret key of AES are assumed to be of fixed values, or at least fixed during the execution of the program. Such a key can be hidden inside the program with some protection, for instance encrypted or obfuscated. It is also possible that the key is stored in the tamper-proof device and loaded in the RAM when the program is initiated. Thus, we assume that the key is securely protected and it is impossible for the attacker to extract the key by any means when the program is not executed. However, when the program is initiated, the key must be loaded in the RAM in plaintext. If the attacker can dump memory when the key is loaded in the RAM, it is possible to extract the secret key from the dumped image. We assume that the adversary has the access to the target program and memory dump data.

A service provider provides various services such as hosting, web application and file sharing. The clients connect to the server through secure channels established with SSL/TLS or any equivalent protocols. The secret key can be stored in tamper-proof devices when they are not communicating. However, the secret keys have to be stored in the RAM so long as the clients are connected to the service provider's service. While the connection is active, the malicious operator can attach his attacking process on the server-side to the target and dump the related area in the memory without being detected by the clients. Following this, the adversary can attempt to extract the secret key or any valuable information from the dump of the memory image. Even when there is no malicious operator, there is a chance that the server is attacked by malware (e.g., Trojan horse) which acquires the root privilege and mounts similar attacks on the processes handling secure client connections.

Another possible scenario is that several users share the same physical server (i.e., public cloud) on which they operate their own virtual machines (VMs). Usually each VM is separated, and they cannot communicate. Once the attacker can login as the administrative user, it is possible to attach the attacking process to the victims' VM processes. Ristenpart *et al.* [77] showed that it is possible to extract information from a target VM. The authors identified where in memory a particular target VM was likely to reside, and then started new VMs until one was placed co-resident with the target.

Last but not the least is the user-as-the-attacker scenario. The user can have full access to the target program and have full control of his own system. The aim of the user is to extract the secret keys or other valuable information from the target program by attaching the attacking process to the target process.

3.3.2 Attack against RSA

The exponentiation of RSA decryption involves variable length arithmetic, it is expected that decryption process uses shift operations. Also the Chinese Remainder Theorem (CRT), which involves division operations, is often used for exponentiation operations of encryption and decryption. Therefore, shift and division operations deal the private key and we can recover the key by dumping and analysing memory region used by these operations.

However, the private key is not the only value which these processes deal and another technique for identifying the key is required. Assume both the public and private keys are fixed while the plaintext and the ciphertext keep changing. Then every time the encryption or decryption is executed, the program's accesses to the key retrieve the fixed value, while its accesses to the plaintext or the ciphertext retrieve different values. Therefore, as the number of execution increases, the accesses to keys can be distinguishable from other accesses by counting the number of accesses to each value. The procedure can be summarised as follows:

1. Iterate encryption and decryption of random numbers with the fixed key,
2. Dump values which are accessed by shift and division operations,
3. Terminate the encryption program,
4. Count how many times each value is accessed,
5. Output key candidate values which are accessed considerably more than other values.

If the multiplication of two primes PQ matches the modulo N, the key is correctly recovered.

3.3.3 Attack against AES

As described in Algorithm 1, the round keys are XORed to the state in AddRoundKey. Hence, we can recover the round keys, also the secret key, by dumping and analysing memory region accessed by AddRoundKey. As AddRoundKey takes 2 inputs, the round key and the state, the values accessed by AddRoundKey are not always the round key. The similar technique to the RSA can be applied to AES in order to extract the secret key. As one of the input to AddRoundKey is the fixed key and the other one is the variable internal state, the key can be distinguished from the internal state by counting the number of accesses during the iteration of encryption. As AES is symmetric, we only need to operate either encryption or decryption in order to extract the secret key. The procedure can be summarised as follows:

1. Iterate the encryption of random numbers with the fixed key,
2. Dump values which are accessed by AddRoundKey operations,

3. Terminate the encryption program,
4. Count how many times each value is accessed,
5. Output key candidate values which are accessed considerably more than other values.

If the round keys derived from the recovered secret key matches the recovered round keys, the secret key is correctly recovered.

3.4 Process Peeping Tool (PPT)

The PPT can analyse the structure of the target program, including which shared libraries it uses and which functions are used in each library. It can also analyse memory addresses that the target process accesses and values which the target process uses. These addresses and values can be recorded and statistically analysed. Dumped data can be efficiently used for the key extraction attack as, unlike existing memory dump tools, we can specify libraries and functions of interest.

One can attach the PPT process to the target program by specifying the target's process ID (PID). Once successfully attached to the target PID, the evaluator can browse inside the target as if the target and the analysing tool were respectively a file system directory structure and the shell. This is enabled by using `ltrace` system call. When `ltrace` is called to analyse the target program, it first searches the absolute path of the target. Then it reads dependent libraries with `elfutils`, and obtains symbol information and a list of function's addresses from a table called Procedure Linkage Table (PLT). In the next step, the child process is executed with `PTRACE_TRACEME`. When the child process executes a function, the parent process receives `SIGTRAP` and pauses the child process. The parent process replaces, keeping a copy of the original values, the function's addresses with breakpoints in PLT. Then the original operation is restored to execute a single step of the child process with `PTRACE_SINGLESTEP`, and the parent process obtains information on which libraries the child process accessed. After the single step, the parent process again takes over the control and continues the operation until it encounters the next breakpoint.

The evaluator, therefore, can find out which libraries are used and which functions inside these libraries are called. Figure 3.1 shows an example of the result. As shown in Figure 3.1, `rsao0s_so` is the executable file and it uses the shared libraries, for instance `ld-2.17.so`, `libc-2.17.so` and `libcrypto.so.0.9.8`. Inside of `libcrypto.so.0.9.8`, many functions are called, for instance `RSA_generate_key`, `RSA_public_encrypt` and `RSA_private_decrypt`.

The evaluator can control how each function can be executed by setting its status. The available statuses are:

watch: execute the function step-by-step recording its data;
watchdeeply: in addition to *watch*, this status enables recording the behaviour of other functions called inside the target function;
through: execute the function step-by-step without recording data; and
skip: execute the function as usual.

When the function is under surveillance with “watch” status, the function is executed with a `PTRACE_SINGLESTEP`. When the single step of the child process is executed, the addresses and the values from the child task can be read by `PTRACE_PEEKDATA`. The addresses and values are recorded by PPT and used for static analysis.

Any function with a “watch” status is skipped if that function is called by one with a “skip” status. In order for the “watch” status to work with a function, it should be ensured that its caller function has a status set to “through”.

3.5 Key Extraction from Memory Dump

In this section we demonstrate how the Process Peeping Tool can help extract the private key and the secret key of RSA and AES respectively from two separate target

```
TYPE               |OBJECT
-------------------|--------------------
Executable         |rsao0s_so
SharedLibrary      |ld-2.17.so
SharedLibrary      |libc-2.17.so
SharedLibrary      |libcrypto.so.0.9.8
SharedLibrary      |libdl-2.17.so
SharedLibrary      |libgcc_s.so.1
SharedLibrary      |libm-2.17.so
SharedLibrary      |libssl.so.0.9.8
SharedLibrary      |libstdc++.so.6.0.18
SharedLibrary      |libz.so.1.2.8
SharedLibrary      |[vdso]
SharedLibrary      |[vsyscall]
```

Figure 3.1: List of libraries used by RSA sample program. The boxed library is for cryptographic operations.

sample programs. The attack procedure is summarised as follows:

1. analyse the structure of the target program including libraries and functions,
2. specify which library or function to monitor,
3. execute the target program while specified libraries and functions are executed step-by-step,
4. record addresses and values,
5. recover the key by statically analysing data acquired in step 3.

The experimental environment is summarised in Table 3.1.

In case of RSA, PPT retrieves the private key and a lot of random numbers from RAM. Hence we require the statistical analysis to distinguish random numbers and

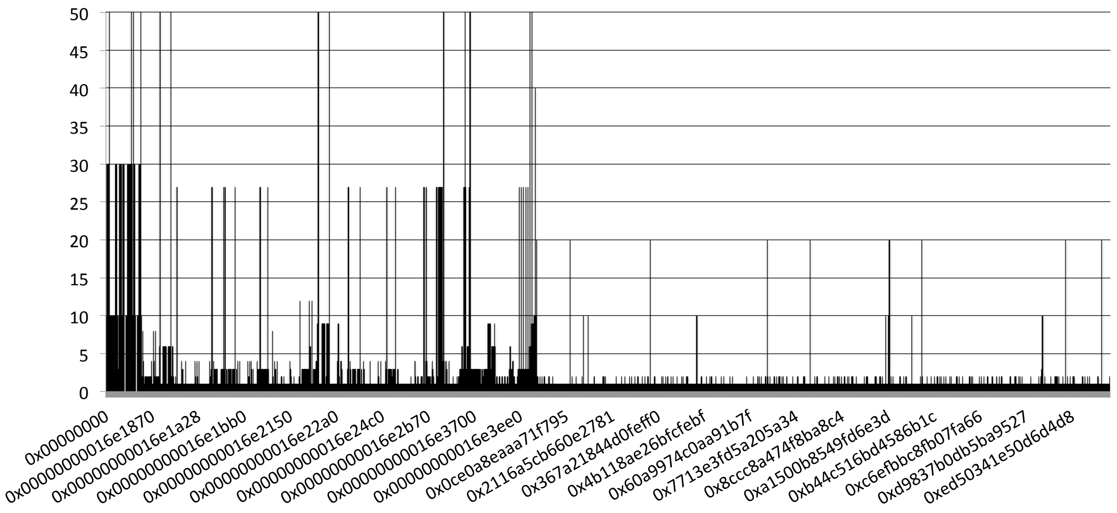


Figure 3.2: Values referred from RAM when we iterate RSA encryption and decryption ten times

the private key. On the other hand, the key extraction of AES is simple and we do not need any additional analysis in order to separate the secret key from other values. Maartmann-Moe *et al.*'s attack [58] uses the facts that round keys are derived from the initial key and the round keys are stored on RAM right after the initial key. Therefore, their attack cannot be applied when the initial key and round keys are stored on the separate locations on RAM. On the other hand, our attack can recover the key even when the initial key and round keys are stored on the separate locations as our attack observes the values which are accessed by the encryption function.

Table 3.1: Experimental environment

CPU	Intel Core i7 4930K
RAM	24GB
OS	Ubuntu 13.10 64-bit
Library	OpenSSL 0.9.8

Table 3.2: The watch list of functions for RSA decryption and AES encryption

RSA		
Library	Function	Status
`rsao0s_so`	`RSA_private_decrypt`	through
`libcrypto.so.0.9.8`	`BN_div`	watch
`libcrypto.so.0.9.8`	`BN_lshift`	watch
`libcrypto.so.0.9.8`	`BN_rshift`	watch

AES		
Library	Function	Status
`aesopenssl`	`AES_encrypt`	watchdeeply

3.5.1 RSA

3.5.1.1 Sample Program

We implemented a simple RSA encryption and decryption program using the OpenSSL library. This sample program repeatedly encrypts random numbers and decrypts the generated ciphertexts. Both the public and the private keys remain unchanged during the experiment.

3.5.1.2 Which functions to watch

In the `libcrypto.so.0.9.8` library, there are two shift operations which are `BN_lshift` and `BN_rshift`. We set the statuses of these two functions as “watch”. Another function that we should watch is `BN_div`. In order to *watch* the functions we are interested in, in this case `BN_lshift`, `BN_rshift` and `BN_div`, we have to specify that these functions not to be skipped. This can be realised by set the status of the function in the upper layer as “through”. Therefore, we also set the status of `rsao0s_so` as “through” in order to “watch” three functions of interest. Table 3.2 summarises the list of the methods to be observed.

3.5.1.3 Key Recovery Phase

We initiate the sample program and start the encryption and decryption operations. Then, we initiate PPT and attach its process to the sample program. PPT can show the structure of the program as shown in Fig. 3.1, when it is successfully attached to the target. By executing the program while watching the specified functions in Table 3.2, we record the values and their frequencies in which they are referred to in the RAM. Figure 3.2 shows relations between the values and their frequencies. The x-axis shows the values and the y-axis shows the number of referred times. As it is unlikely that the private key is a sparse value, we can eliminate the sparse candidates, for instance `0x0000000000000001`. These sparse values are mostly used for controlling the operations such as counters.

For the remaining candidates, we can use number of referred times as a clue. In this example, we iterate encryption and decryption 10 times. Thus, the private key has to be used at least 10 times. Even when we do not know how many times encryption and decryption is iterated, we can still extract key when they are iterated long enough to separate random numbers and the private key. The numbers which are not sparse and are referred more than 10 times are shown in Fig. 3.3, which match with the private key shown in Table 3.3.

Table 3.3: Private key for the sample program

p	`0xF97EDAC39DD1895CF22132485C484099CA88F457825CA818D2`
	`2C4DFF547902960AFE653B3A9F44CA0F5B3440702AA78587E067`
	`AB435443291A0C2A42299EBCE1`

q	`0xCC95FEE773FE65D8F8A2744972C68704580560E858D4A33B30`
	`31F267B70E59B992C76BD41829D499B7A1E027B04D5B54811A01`
	`9267102E85CBFB9B96317D03F7`

```
|-----address------|------value-------|
| 0000000000090b20 | 0afe653b3a9f44ca |
| 0000000000090b20 | 0f5b3440702aa785 |
| 0000000000090b20 | 1a0c2a42299ebce1 |
| 0000000000090b20 | 3031f267b70e59b9 |
| 0000000000090b20 | 580560e858d4a33b |
| 0000000000090b20 | 811a019267102e85 |
| 0000000000090b20 | 87e067ab43544329 |
| 0000000000090b20 | 92c76bd41829d499 |
| 0000000000090b20 | b7a1e027b04d5b54 |
| 0000000000090b20 | ca88f457825ca818 |
| 0000000000090b20 | cbfb9b96317d03f7 |
| 0000000000090b20 | cc95fee773fe65d8 |
| 0000000000090b20 | d22c4dff54790296 |
| 0000000000090b20 | f22132485c484099 |
| 0000000000090b20 | f8a2744972c68704 |
| 0000000000090b20 | f97edac39dd1895c |
```

Figure 3.3: Recovered value of the RSA key. The 16 digit hex values of the lefthand side show the address of the operation and righthand side values show the ones accessed by the operation. Boxed values are part of p and non-boxed values are part of q.

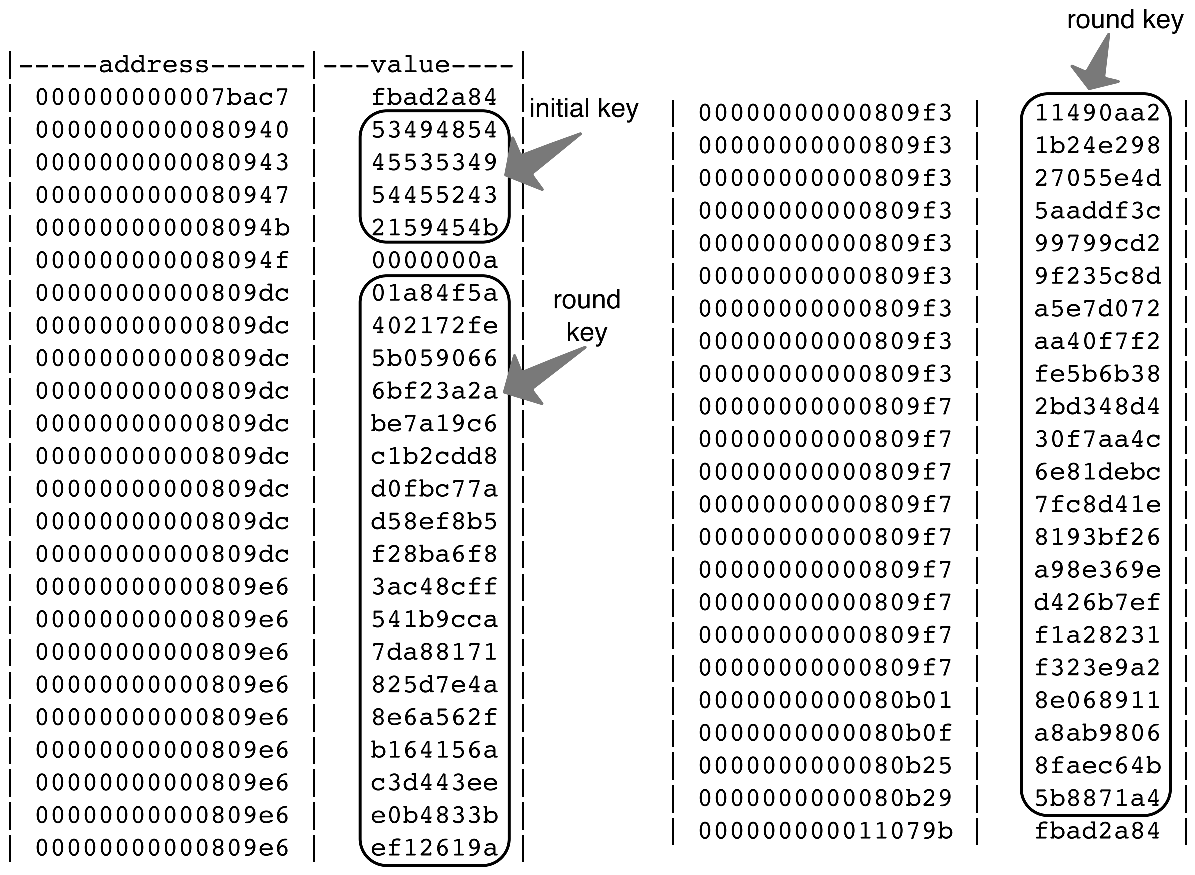
3.5.2 AES

3.5.2.1 Sample Program

We also implemented a simple AES encryption program, named `aesopenssl`, using the OpenSSL library. This sample program continuously encrypts random numbers with a fixed secret key.

3.5.2.2 Which Function to watch

Table 3.2 shows the function to be observed. As described in Sect. 3.2.2, the round keys are XORed to the state in AddRoundKey. Hence, we can recover the secret key and the round keys by observing the values accessed by AddRoundKey. AddRoundKey



```
|-----address------|---value----|
| 000000000007bac7 |   fbad2a84 |
| 0000000000080940 |   53494854 |
| 0000000000080943 |   45535349 |
| 0000000000080947 |   54455243 |
| 000000000008094b |   2159454b |
| 000000000008094f |   0000000a |
| 00000000000809dc |   01a84f5a |
| 00000000000809dc |   402172fe |
| 00000000000809dc |   5b059066 |
| 00000000000809dc |   6bf23a2a |
| 00000000000809dc |   be7a19c6 |
| 00000000000809dc |   c1b2cdd8 |
| 00000000000809dc |   d0fbc77a |
| 00000000000809dc |   d58ef8b5 |
| 00000000000809dc |   f28ba6f8 |
| 00000000000809e6 |   3ac48cff |
| 00000000000809e6 |   541b9cca |
| 00000000000809e6 |   7da88171 |
| 00000000000809e6 |   825d7e4a |
| 00000000000809e6 |   8e6a562f |
| 00000000000809e6 |   b164156a |
| 00000000000809e6 |   c3d443ee |
| 00000000000809e6 |   e0b4833b |
| 00000000000809e6 |   ef12619a |

| 00000000000809f3 |   11490aa2 |
| 00000000000809f3 |   1b24e298 |
| 00000000000809f3 |   27055e4d |
| 00000000000809f3 |   5aaddf3c |
| 00000000000809f3 |   99799cd2 |
| 00000000000809f3 |   9f235c8d |
| 00000000000809f3 |   a5e7d072 |
| 00000000000809f3 |   aa40f7f2 |
| 00000000000809f3 |   fe5b6b38 |
| 00000000000809f7 |   2bd348d4 |
| 00000000000809f7 |   30f7aa4c |
| 00000000000809f7 |   6e81debc |
| 00000000000809f7 |   7fc8d41e |
| 00000000000809f7 |   8193bf26 |
| 00000000000809f7 |   a98e369e |
| 00000000000809f7 |   d426b7ef |
| 00000000000809f7 |   f1a28231 |
| 00000000000809f7 |   f323e9a2 |
| 0000000000080b01 |   8e068911 |
| 0000000000080b0f |   a8ab9806 |
| 0000000000080b25 |   8faec64b |
| 0000000000080b29 |   5b8871a4 |
| 000000000011079b |   fbad2a84 |
```

Figure 3.4: Recovered value of the AES secret key

is called inside `AES_encrypt`, which is `aesopenssl` executable file. We set the status of `AES_encrypt` as "watchdeeply". We can specify the functions in more detail than setting `AES_encrypt` as "watchdeeply" as we did in RSA for more efficient analysis. However, compared to RSA, AES is simpler and we can recover the secret key efficiently enough by setting the status of `AES_encrypt` only to "watchdeeply".

3.5.2.3 Key Recovery Phase

We execute the program while observing the `aesopenssl` function, and apply the method similar to what we did for RSA to eliminate the non-key values. The result is shown in Fig. 3.4. The secret key we used for the sample program is "THISISSECRETKEY!", which is `0x54,0x48, 0x49, 0x53, 0x49, 0x53, 0x53, 0x45, 0x43,` `0x52, 0x45, 0x54, 0x4b, 0x45, 0x59, 0x21` in ASCII code. After the secret key, PPT recovered `0x0000000a`, which is the number of rounds in AES-128, followed by the round key.

We can also deduce that listed values are part of the secret key from the address. In the first line of the Fig. 3.4, the value `0xfbad2a84` is referred from the address of `0x000000000007bac7`. Then the program is observed to jump to the address of `0x0000000000080940` and access the value `0x53494854`. After this operation, the program continues to access successive addresses until it jumped to `0x000000000011079b`. Hence, we can deduce that the encryption process starts at `0x0000000000080940` and ends at `0x0000000000080b29`.

3.6 Conclusion of This Chapter

In this chapter, we introduced a statistical key extraction attack on cryptographic keys using memory dump data, and confined the effectiveness of the attack by utilising our Process Peeping Tool (PPT). The tool can be attached to the target process and can trace the target's memory usage. We used RSA and AES as example cryptosystems in the target programs, which made use of the OpenSSL library implementations of those cryptosystems. Thus, it is possible to apply the same approach to other applications using OpenSSL library or similar cryptographic libraries. Although we only applied PPT to RSA and AES implemented in the OpenSSL library, it is possible to apply the same extraction mechanism to other, including non-cryptographic, algorithms or libraries. Although we execute the PPT with the root privilege, we can still apply our method obtaining memory dump data without the root privilege.

In the next chapter, we propose a countermeasure against the threat caused by PPT. The attack using PPT is only possible because some data is accessed more often the others. Hence the attack can be prevented by making the distributions of accesses equally, or less concentrated.

Chapter 4

Lightweight Access Pattern Protection

4.1 Introduction

The problem of preventing leakage of information arising from both running software on untrusted systems, as well as storing data in remote untrusted servers has attracted much attention. While encryption can be used to protect data confidentiality, the problem of protecting access pattern with manageable and practical overhead is harder to address.

In the software execution environment, one can identify two main motivations for protecting memory access pattern privacy: the traditional application is to protect Intellectual Property, and prevent software piracy. More recently, it has been shown how access pattern leakage can be used to attack certain implementations of cryptographic algorithms (in the so-called *cache attacks* [71]). In the remote storage environment, one would like to protect access pattern from a curious but not malicious server, which may benefit from gaining information about the client's pattern of access of stored data.

The traditional solution for memory access pattern protection is known as *Oblivious RAM(ORAM)*. It was first proposed by Goldreich [39], and later extended by Goldreich and Ostrovsky [40]. The main construction is based on the *hierarchical solution*, in which the data structure is organised in levels consisting of hash-tables (using secret hash functions known by the client only), and requires periodic expensive *oblivious* re-shuffling of data.

In the past few years, many improvements have been proposed, for example [1, 73, 25, 44, 82, 53]. Several schemes consider improvements for the application to cloud computing [18, 92, 42, 43, 45, 46, 55, 94]. Improvements typically arise from the use of different data structures and hash function schemes, more efficient sorting algorithms (for the oblivious shuffling step), and the use of secure local (client) memory. Besides the hierarchical solution, Goldreich and Ostrovsky [40] also proposed the *square root* construction, which uses secret random permutations when storing data. Boneh *et al.* [12] have recently presented a hybrid algorithm between the square-root and hierarchical algorithms, and proposed a new notion of *oblivious storage*.

Despite much recent progress, where both the asymptotic efficiency as well as the constant terms of ORAM solutions have been improved (making it particularly attractive for remote storage access pattern protection), current solutions remain inefficient for *software execution protection*, i.e. to prevent leakage of relatively limited-in-size memory access pattern. In these cases, the constant terms involved in the computational complexity make the overhead unacceptably high. Yet an efficient and secure mechanism for access pattern protection is a particularly desirable feature in this environment. For instance, an emerging issue is the rapid increase of malicious software targeting smartphones. Most existing protection schemes, originally designed for PCs, are not suitable for smartphones due to several limitations, such as the computational power and available storage size. In these environments, solutions range from use of obfuscation to hardware-based access pattern protection mechanisms. For instance, Zhuang *et al.* [99] proposed a practical, hardware-assisted scheme for embedded processors, with low computational overhead. Their *control flow obfuscation* scheme for embedded processors employs a small secure hardware obfuscator to hide program recurrence. Their proposal however trades security for low overhead (and the cost of the trusted hardware buffer), and in some situations an adversary with access to the device can retrieve information about memory access.

As the storage size glows, the overhead incurred by existing ORAM schemes, which is dependent on the storage size, can be unacceptably high. In this chapter, we propose a practice-oriented scheme for protecting RAM access pattern, which leverages the history of memory access to help hiding the pattern. We first consider an instance which, similar to the proposal by Zhuang *et al.*, also relies on the use of a secure (trusted) hardware buffer. However it achieves higher security by adapting ideas from Goldreich and Ostrovsky's square root solution, yet without the expensive (re-)shuffling of buffers. By applying this scheme, we can construct a secure platform that is suitable for executing software that deals with user private information. A potential application is to secure program execution in smartphones: these devices typically contain sensitive user information and are increasingly under severe threat from malware. Most smartphones have a SIM card, which is generally considered as a secure area, and deployment and security of our scheme could rely on the SIM card. Another instance requires no special hardware, but as a result leads to a higher, yet practical overhead. This scheme can offer the same level of security without any special hardware. Many applications have their security relying on IC cards or other trusted hardware. One of the roles of trusted hardware is offering a secure computation, which can be realised with our proposal. Another role is a secure storage for a secret information, which is yet to be realised.

The main feature of our proposal is to maintain the *history* of memory access, which together with the access of dummy data, helps one to hide data access pattern. The security of the schemes depends on the size of the buffer (as cache or in RAM) and how the history is used. We claim that under reasonable assumptions and by selecting appropriate parameters, the schemes achieve both security and performance levels acceptable in practice. One of the advantages of our proposal is its lightweightness. Another advantage is that the overhead is independent of the storage size. Unlike ORAM schemes, which hide addresses of accesses, our proposal achieves access pattern protection by hiding the access timing. This helps to realise the constant overhead required to hide the pattern. Therefore, it is suited for protecting large storage where most of ORAM schemes become impractical. We note that although the proposal is particularly focused on the software execution protection environment (i.e. to prevent RAM access pattern leakage), its

Table 4.1: Summary of Difference between ORAM and Our Proposal

	Security	Computational Cost	Adversary
ORAM	Hide accessing addresses	Dependent on N	Observe memory access
Ours	Hide accessing timing	Constant	Observe memory access

security may well be appropriate for most uses in the remote storage environment, to prevent access pattern leakage of cloud storage with much lower performance overhead than existing solutions. Then we consider practical implementation of the scheme and propose three techniques for the higher performance.

Table 4.1 summarises the difference between ORAM and our proposal. ORAM hides addresses of blocks which are accessed, and ensures given two access patterns are indistinguishable. While its computational cost is dependent on the storage size. Our proposal only hides when the block is accessed and cannot offer the same level of security as ORAM. However it achieves constant overhead independent on the storage size.

4.2 Access Pattern Protection Problem

We can model the problem of access pattern protection as follows. We consider a *client*, with potentially small secure memory, and a *server* providing large insecure storage. This storage consists of several data blocks (for simplicity, all of the same size). In the software execution environment, we can think of the CPU as the client, and RAM as the server, where a malicious entity (e.g. malware) has access to RAM and the memory bus used in the communication between the CPU and memory.

The client accesses data by making requests to the server to either retrieve the contents of a particular data block at location i or by writing x into a data block at location j. We denote these operations by `read`(i) and `write`(j, x), respectively. We consider the security goals as: to protect the confidentiality[1] of data, as well as hide the client's ac-

[1] In environments where there is a requirement, we will also want to protect data integrity; this can be done by adding a MAC or by using an authenticated encryption algorithm.

cess pattern to the stored data blocks from a computationally bounded adversary, which can access the data storage and the communication channel between the client and the server.

We can address the first goal by using a semantically secure probabilistic encryption scheme, such that two encryptions of the same data block will look indistinguishable to a computationally bounded adversary. The use of encryption adds a constant overhead to the system. For the remaining of this work, we will assume that the data is stored encrypted; it is decrypted whenever it is read from memory, and re-encrypted whenever it is written into memory.

For the second goal, we would like that a particular sequence of operations in the stored data does not substantially leak any information, other than how many data blocks were accessed. To formalise it, we use the definition in [73] for a *secure oblivious RAM* system.

Definition 1 ([73]). *The input y of the client is a sequence of data blocks, denoted by $((v_1, x_1), \dots, (v_n, x_n))$ and a corresponding sequence of operations, denoted by $(op_1, \dots, op_m)$, where each operation is either a read or a write operation. The read operation, denoted `read`(v), retrieves the data of the block indexed by v. The write operation, denoted `write`(v, x), sets the value of block v to be equal to x.*

The access pattern $A(y)$ is the sequence of accesses to the remote storage system. It contains both the indices accessed in the system and the data blocks read or written. An oblivious RAM system is considered secure if for any two inputs y and y' of the client, of equal length, the access patterns $A(y)$ and $A(y')$ are computationally indistinguishable for anyone but the client.

The typical, straightforward method for preventing an adversary from distinguishing between the `read` and `write` operations is to always perform both operations in every access. As a result of using this method, an access pattern $A(y)$ (for the purpose of indistinguishability) can be thought as simply as a sequence of indices i (corresponding to the data blocks accessed). The trivial solution to the access pattern protection problem consists of accessing all data blocks on the memory for each query. Another trivial solution is to use a secure client hardware. Both schemes are however too costly in practice.

In addition, data blocks are typically organised in memory based on a secret permutation or hash function in an oblivious way. This is the most expensive component of the schemes, and is the main responsible for the (amortised) computational overhead. Our proposal does not employ oblivious re-shuffling of memory; while this will affect the security provided by the scheme, we claim that under reasonable assumptions, the proposals achieve both security and performance levels acceptable in practice.

4.3 Our Scheme for Memory Access Pattern Protection

While oblivious RAM constructions can completely hide memory access pattern, they are too expensive to be implemented on resource-constrained devices. In this section, we describe our proposal for a practice-oriented memory access pattern protection scheme. One of its main features is the inclusion of an extra buffer used to maintain the *history* of memory access, to help hiding the access pattern. We present below two instances: the first one uses a secure (trusted) hardware buffer, as in the work by Zhuang *et al.* [99]; however it achieves higher security by adapting ideas from oblivious RAM mechanisms, yet without the expensive (re-)shuffling of buffers. The second instance requires no special hardware, but as a result leads to a higher, yet practical overhead.

4.3.1 Assumptions

We consider the problem of hiding the access pattern of memory with n data blocks. Our scheme will assume that the *client* has access to an efficient pseudo-random number generator (to make random choices of addresses), and a semantically secure probabilistic encryption scheme. Data is always stored encrypted: it is decrypted whenever it is read from memory, and re-encrypted whenever it is written into memory. Furthermore, either a `read` or `write` operation will always perform the two operations in every access, the difference is the value being written in the `write` step. As a result of using this method, an access pattern (for the purpose of indistinguishability) can be thought as simply as a sequence of indices i (corresponding to the data blocks accessed).

Our scheme will also require a way to generate a pseudo-random permutation, to map memory addresses. This can be achieved by using a deterministic encryption algorithm $\mathcal{E}$. This mapping will be described either by the function $\mathcal{E}$, with the output computed in each call, or explicitly described as a table look-up (with input-output pairs). We note that the latter requires $O(n)$ memory within the client's trusted boundary (as in the work by Zhuang *et al.* [99]), which in some scenarios may be impractical.

Perhaps the main challenge of access pattern protection schemes is to hide the repeated access of data blocks. In general, when not deploying expensive ORAM solutions, access pattern can typically be distinguishable by observing a long series of accesses. Therefore, we define a relaxed, still practical, security definition of access pattern protection, which we call *δ-length security.* Assume that during a certain sequence of data access, a particular block 'a' is accessed twice by a program at t-th access and $(t+\delta)$-th access; we call this a *δ-distance access* of 'a'. Informally, an access pattern protection scheme is δ-length secure if the probability that an adversary identifies repeated accesses in $A(y)$ at distance at most δ is small.

Definition 2. *We say that an access pattern protection scheme is δ-length ϵ-secure if the probability that an adversary identifies any d-distance access in $A(y)$ is at most ϵ for every $d \leq \delta$.*

4.3.2 Set-up

On loading n data blocks to memory, our scheme will use $\mathcal{E}$ to permute the corresponding addresses. Dummy data will be typically added to the original data, so that at initialisation we have kn data blocks being loaded to memory, with $k \geq 1$ a small constant. The constant k will be selected based on the typical *epoch* length of the program and the availability of memory within the client's trusted boundary to describe the random permutation $\mathcal{E}$. We discuss the use of dummy data in more detail in Section 4.4.3.

Our scheme will partition memory into two regions: a (secure) buffer $\mathcal{M}$ and an unsecured memory $\mathcal{L}$, of sizes m and ℓ, respectively, with $m \ll \ell$. It follows that $m + \ell = kn$. Furthermore, we will require a secure table $\mathcal{H}$, called the *history* table. The table $\mathcal{H}$ stores addresses of data blocks that have been moved from the secure buffer $\mathcal{M}$ to the unsecured memory $\mathcal{L}$, and has size ℓ_h. We denote the address of a data block 'a' as i_a. Typically, we may have $\mathcal{H}$ implemented as part of $\mathcal{M}$. In the case where we are able to store the permutation mapping table explicitly, $\mathcal{H}$ can be implemented by adding a *set* bit into the table. Despite these choices, in our discussions below we consider $\mathcal{H}$ as a separate table.

4.3.3 Instance 1: Construction with Secure Memory

The first instance of our scheme considers the buffer $\mathcal{M}$ being implemented within the client's trusted boundary (as in the work by Zhuang *et al.* [99]). The table $\mathcal{H}$ is also stored within this boundary. After loading data into memory, m data blocks are copied into $\mathcal{M}$; the table $\mathcal{H}$ starts empty[2]. On the first access to a data block 'a' (either a $\texttt{read}(a)$ or $\texttt{write}(a, x)$), we search for 'a' in $\mathcal{M}$ and access $\mathcal{L}$ twice: if 'a' is in $\mathcal{M}$, we replace two random elements (not 'a') from $\mathcal{M}$ with two random *dummy* elements from $\mathcal{L}$ and we access 'a'; if 'a' is not in $\mathcal{M}$, two random elements from $\mathcal{M}$ are replaced by 'a' and one random *dummy* element from $\mathcal{L}$ (and 'a' is accessed). In both cases, the corresponding addresses of blocks being kicked out from $\mathcal{M}$ are written in $\mathcal{H}$. In subsequent calls, we proceed as follows:

1. if 'a' is in $\mathcal{M}$, we replace two random elements (not 'a') from $\mathcal{M}$ by a random element from $\mathcal{L}$ and a random element from $\mathcal{L}$ which had already been accessed before (as recorded in the *history* buffer), and we access 'a'.

2. if 'a' is not in $\mathcal{M}$, and its address is in the history table, we replace two random elements from $\mathcal{M}$ by a random element from $\mathcal{L}$ and 'a' (as recorded in the *history*

[2]although, before the program starts to run, the scheme can operate by accessing a number of dummy data blocks to populate the history buffer.

Algorithm 2 Pseudocode of access pattern protection scheme

```
1: scan $\mathcal{M}$ for ‘$a$’
2: if ‘$a$’ $\in \mathcal{M}$ then
3:   replace two random elements (not ‘$a$’) in $\mathcal{M}$ with two random blocks in $\mathcal{L}$, one of
     them is chosen from the history $\mathcal{H}$ and the other is randomly chosen from $\mathcal{L}$
4: else
5:   scan $\mathcal{H}$ for ‘$i_a$’
6:   if ‘$i_a$’ $\in \mathcal{H}$ then
7:     replace two random blocks in $\mathcal{M}$ with a random block in $\mathcal{L}$ and ‘$a$’
8:   else
9:     replace two random blocks in $\mathcal{M}$ with ‘$a$’ and a random block whose address is
       registered in $\mathcal{H}$
10:  end if
11: end if
12: choose $\ell_h$ elements from $\ell_h + 2$ to update history table $\mathcal{H}$
13: access ‘$a$’
```

buffer). Note that $\mathcal{H}$ holds only addresses and data itself is stored in $\mathcal{L}$.

3. if ‘a’ is not in $\mathcal{M}$, and its address is not in the history table either, we replace two random elements from $\mathcal{M}$ by ‘a’ and a random element from $\mathcal{L}$ which had already been accessed before (as recorded in the *history* buffer).

Every time the data blocks are kicked out from $\mathcal{M}$ to $\mathcal{L}$, data blocks are written in $\mathcal{L}$ taking their original position (as described by $\mathcal{E}$), and addresses of those blocks are registered in the history table $\mathcal{H}$. As the program continues to access data blocks, the table may eventually get full. When this is the case, at each access we select at random ℓ_h elements among the $\ell_h + 2$ elements (the current history elements and the two new ones). We have a pseudocode of our scheme in Algorithm 2 and show an example in the appendix.

4.3.3.1 The Objective of the History Table

The goal of our scheme is to efficiently protect the privacy of data access pattern. It is clear that if during a run of the program, data blocks are only accessed once, then the use of a random permutation alone will suffice to hide the access pattern. The case of relevance is thus when a data block is accessed more than once. When it is accessed for the first time, it is copied into $\mathcal{M}$. If accessed again, and it is still in $\mathcal{M}$, then access is oblivious from an adversary; still we would like to hide the fact that the data accessed was found in $\mathcal{M}$. If it is no longer in $\mathcal{M}$, then we would like to hide the fact we are accessing it again from an adversary. Thus, if we consider the case that a program keeps reading 'a' at different intervals, we have the following three types of access to consider:

1. 'a'$\in \mathcal{M}$;

2. 'a'$\notin \mathcal{M}$ and 'i_a' $\in \mathcal{H}$;

3. 'a'$\notin \mathcal{M}$ and 'i_a' $\notin \mathcal{H}$.

Note that case 3 is likely to occur when 'a' is accessed for the first time. As discussed, the cases that require most attention are 2 and 3 when 'a' is accessed again: since we do not perform an oblivious re-shuffling, the adversary would notice that 'a' is being accessed again.

To address this, in our scheme, we first search 'a' in $\mathcal{M}$ and then, depending on the cases described above, the access of memory is done as follows:

1'. access (r, p),

2'. access (r, a),

3'. access (a, p),

where r is a random location in $\mathcal{L}$ and p is a location recorded in the table $\mathcal{H}$.

Note that we need to keep the contents in the history table secret: although the adversary can record all blocks previously accessed, in practice we may not be able to keep the addresses of all accessed blocks in the history table (since ℓ_h may be small). If we had the addresses in cleartext, one may note that in case 3' above, although 'a' had being accessed before, its address was no longer in the history table (for lack of space).

4.3.4 Instance 2: Construction without Secure Memory

We consider a second instance of our proposal, which does not require a secure buffer. The buffer $\mathcal{M}$ and history table $\mathcal{H}$ are kept in the unsecured memory area, as with $\mathcal{L}$. In this case, access to $\mathcal{M}$ (and $\mathcal{H}$) is made by reading and writing every data block in the buffers (which requires decryption and encryption of data). Thus, to find out whether 'a' is in $\mathcal{M}$, we read/write all values; when replacing data blocks in $\mathcal{M}$, we again read/write all values. Except for this, the access is made as described in Section 4.3.3. The security provided is the same, but the computation overhead is obviously increased, and it is dependent of the sizes of buffers $\mathcal{M}$ and $\mathcal{H}$.

4.3.5 Applying to Other Structures

We consider applying the idea of access pattern hiding using the history of accesses to other structures, which are hierarchical and tree.

In hierarchical ORAM, the requested blocks are moved to the top level. After several accesses, the top level buffer will be full. Then all blocks stored in the buffer will be pushed downwards with a new permutation, i.e. re-shuffling. If we apply our idea to hierarchical ORAM, the block will be written back to the same position as it has been before the access, hence there is no need to pushing the blocks downwards. This means it is essentially same as applying the idea to the square root ORAM. In hierarchical ORAM, one block in each level is touched, we can modify this operation to touching previously accessed blocks to hide the pattern.

In the tree structured ORAM (like Path ORAM), all blocks along the path to the leaf node will be touched, and blocks are written back to a new path. When considering to apply our idea to tree structured ORAM, it also becomes similar to doing to square root because all blocks will be back to the same path. We can add touching another path which has been accessed in the past in order to hide which path contains the block currently required.

4.4 Security Analysis

We discuss the security of our scheme.

4.4.1 Access Pattern Hiding

Regarding recurrence, recall that in our scheme, we first search 'a' in $\mathcal{M}$ and then, depending on the cases described in Section 4.3, the access of memory is as follows:

1. 'a'$\in \mathcal{M}$, and 'a' is accessed: access (r, p) in $\mathcal{L}$;
2. 'a'$\notin \mathcal{M}$, $i_a \in \mathcal{H}$, and 'a' is accessed: access (r, a) in $\mathcal{L}$;
3. 'a'$\notin \mathcal{M}$, $i_a \notin \mathcal{H}$, and 'a' is accessed: access (a, p) in $\mathcal{L}$.

Assume that the program accesses 'a' at time t; we denote it by $X_t = a$. We have the following lemma.

Lemma 1. *Assume that $X_t = a$, and let m and ℓ_h denote the sizes of the $\mathcal{M}$ and $\mathcal{H}$, respectively. Then after δ steps we have the following:*

$$p_M = \Pr[a \in \mathcal{M}] \geq \left(\frac{m-2}{m}\right)^{\delta},$$
$$p_H = \Pr[i_a \in \mathcal{H}] \geq \left(\frac{\ell_h}{\ell_h+2}\right)^{\delta}.$$

Proof. To compute $\Pr[a \in \mathcal{M}]$, note that the right-hand side of the expression corresponds to the probability that an element remains in a set of size m after δ replacements of 2 elements at time, which is how the scheme manages the buffer $\mathcal{M}$. The inequality comes from the fact that even if removed after $d < \delta$ steps, 'a' may be re-inserted during the normal operation of the scheme. Showing the second probability is similar (noting however that in the history table, the scheme draws 2 elements among $\ell_h + 2$ elements). $\square$

For the sake of simplicity, we will in the remaining of this paper assume equality in the two expressions above. Furthermore, we will also assume that the two events are independent (obviously the probability that the index of block a is in $\mathcal{H}$ after δ steps will be influenced by whether/when block a leaves the buffer $\mathcal{M}$; however we believe this assumption is reasonable for typical values of m, ℓ_h and δ – and substantially simplifies our computations). The simple lemma below then follows.

Lemma 2. *Consider the three cases for memory access discussed above, and assume that $X_t = X_{t+\delta} = a$. Then the probability that we have case 1 is p_M, the probability that we have case 2 is $p_H(1-p_M)$, and the probability that we have case 3 is $(1-p_H)(1-p_M)$.*

Now assume that an adversary observes the scheme at time $t+\delta$ in case 1, i.e. we have $a \in \mathcal{M}$ and access to (r,p) from $\mathcal{L}$. Then following a conservative estimate, we have $\Pr[X_{t+\delta} = a \mid \text{ case 1 }] \leq \frac{1}{m-2}$. For case 2, we have a similar upper bound: $\Pr[X_{t+\delta} = a \mid \text{ case 2 }] \leq \frac{1}{m-2}$. Case 3 is perhaps the one in which an adversary can extract more information (since it is very likely that, unlike the other two cases, the pair of elements (a,p) drawn from $\mathcal{L}$ have already been observed by the adversary). We will again adopt a conservative approach, and have the upper-bound $\Pr[X_{t+\delta} = a \mid \text{ case 3 }] \leq 1/2$.

Theorem 1. *The proposed scheme is δ-length ϵ-secure access pattern protection scheme, where*

$$\epsilon \leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.$$

Proof. Let $\mathcal{A}$ be adversary who is able to observe an access sequence $X_i = a_i$, for $i = 1, \ldots, N$. Let us assume that $X_t = X_{t+\delta} = a$. We wish to compute the probability $\Pr[X_t = a, X_{t+\delta} = a]$. We assume that the first access to a is at time t (the proof and figures can be slightly modified when this is not the case), and that the accesses at time t and $t+\delta$ are independent events (i.e. we assume no knowledge of statistics of the original program being protected). Then we have $\Pr[X_t = a] \leq 1/(m-2)$, and by lemmas and discussion above.

$$\begin{aligned}
\Pr[X_t = a, X_{t+\delta} = a] &= \Pr[X_t = a] \cdot \Pr[X_{t+\delta} = a] \\
&\leq \frac{1}{m-2}(\Pr[X_{t+\delta} = a \mid \text{case } 1\ ] \cdot \Pr[\text{ case } 1\ ] \\
&+ \Pr[X_{t+\delta} = a \mid \text{case } 2\ ] \cdot \Pr[\text{ case } 2\ ] \\
&+ \Pr[X_{t+\delta} = a \mid \text{case } 3\ ] \cdot \Pr[\text{ case } 3\ ]) \\
&\leq \frac{p_M}{(m-2)^2} + \frac{(1-p_M)p_H}{(m-2)^2} + \frac{(1-p_M)(1-p_H)}{2(m-2)}.
\end{aligned}$$

□

4.4.2 Parameters: Size of Secure Memory and History Table

The choice for sizes of the secure memory $\mathcal{M}$ and history table $\mathcal{H}$ have an obvious influence in the security and efficiency/cost of the scheme: it follows from Theorem 1 that large values for m and ℓ_h significantly decreases the chances that an adversary can identify a repeated access to a particular memory block. However large $\mathcal{M}$ and $\mathcal{H}$ negatively affect the performance of the scheme (as well as increase its costs in the hardware-assisted version).

For instance, if we take the size of the shuffle buffer as 128 16-byte blocks (as in the work by Zhuang *et al.* [99]), and the same size for $\mathcal{H}$ (that is, 512 32-bit addresses), then for $\delta = 20$, we have $p_M \approx 0.73$ and $p_H \approx 0.92$, and as a result $\epsilon \leq 1.4 \times 10^{-4}$. The value increases to 4.4×10^{-4} if $\delta = 50$, and to 1.06×10^{-3} if $\delta = 100$ In general, SIM cards have a capacity of 64KB, which means we can allocate much more space, say 1024 blocks and 4×1024 history addresses. In this case, we have $p_M \approx 0.82$ and $p_H \approx 0.95$, and as a result $\epsilon \leq 0.5 \times 10^{-5}$ for $\delta = 100$.

As discussed before, we note however that our scheme does not achieve strong indistinguishability: As we use a static permutation $\mathcal{E}$, addresses are fixed (after the permutation), and this implies some leakage of information (an adversary will, for instance, know when the sets of potentially accessed blocks are disjoint, implying no recurrence). Overall, we believe that, a choice of parameters can be made to achieve both security and performance levels acceptable in practice.

4.4.3 How much dummy data should we use?

Oblivious RAMs provide security by making sure that a data block is only accessed once while it remains at the same address. When it is accessed again, it will be allocated to a completely different address and the adversary will have no information about the contents and whether/when it was accessed before. Our scheme, on the other hand, tries to ensure the security in the circumstance that the data block is accessed multiple times while in the same address.

If there are several dummy data blocks, that can be used as random elements that the protection scheme can access. They will also prevent the adversary from determining which blocks are actually accessed by the program. We can also make dummy accesses to the actual data blocks when it is not accessed by the program. When the program accesses a data block, the access pattern protection scheme will access 2 blocks as the scheme always swaps 2 blocks between $\mathcal{M}$ and $\mathcal{L}$. If we add n dummy data blocks, the number of accesses to the actual data and dummy data will be roughly the same. Guessing the access pattern correctly becomes harder as n increases, and less dummy blocks will be required. While we still need to confirm it experimentally, we expect that the required number of dummy blocks will be at most $2n$ and the constant k will be $1 \leq k \leq 3$.

4.4.4 Access Timing Hiding

ORAM can be said that it hides the accessing addresses while our proposal hides the timing of accesses. When a certain data block is being accessed multiple times, ORAM ensures the physical access will be made to different addresses. Our proposal eliminates the re-shuffling data blocks, whenever the block is kicked back to the ORAM storage, they are allocated to the same addresses as before they are brought into the buffer. Hence, the adversary will notice when the block of his interest moves to the buffer and comes back to the storage. However, the adversary cannot distinguish if these physical accesses are made due to the logical accesses or dummy accesses. When accessing a block, the block is first moved to the buffer, and stays in the buffer until it is kicked back to the memory. When the block is moved to the memory, its address is recorded to the history table. Then the history is used to perform the accesses to previously accessed blocks. The accesses to previously accessed blocks can be dummy ones or necessary ones depending on the blocks in the buffer. When it is the necessary one and the adversary can detect it, the access pattern will be leaked. When it is dummy one (i.e. the required data is in the buffer), the pattern will not be leaked.

Unlike ORAMs, which hide the addresses to be accessed in order to hide the pattern, our scheme hides the timing of the access and cannot completely hide the access pattern. Yet the weaker security can be suffice in some scenarios. One example is the counter-measure to the cache timing attack. Both ORAM and the proposal perform dummy accesses, which means the block accessed by the dummy access will be temporary stored in the CPU cache and makes the cache timing attack difficult.

4.4.5 Copying More Than Two Blocks

We consider security when more than two block are swapped between the two region. In the following discussion, we assume the block 'a' is accessed multiple times and three blocks are copied into the buffer, instead of two as in the original scheme. The choice of blocks can be two blocks from history and the third one not from history. After the first access to the block 'a', the access can be simulated as follows:

1. 'a'$\in \mathcal{M}$, and 'a' is accessed: access (r, p_1, p_2) in $\mathcal{L}$;
2. 'a'$\notin \mathcal{M}$, $i_a \in \mathcal{H}$, and 'a' is accessed: access (r, a, p_1) in $\mathcal{L}$;
3. 'a'$\notin \mathcal{M}$, $i_a \notin \mathcal{H}$, and 'a' is accessed: access (a, p_1, p_2) in $\mathcal{L}$.

The probability of the block 'a' in the buffer after δ accesses is given by $p_{M3} \leq (\frac{m-3}{m})^\delta$, and that of the address of block 'a' in the history is given by $p_{H3} \leq (\frac{\ell_h}{\ell_h+3})^\delta$. Then the probability of the adversary detecting the first and second accesses to the block 'a' can be given by:

$$\begin{aligned}\epsilon &\leq \Pr[X_t = a] \cdot \Pr[X_{t+\delta} = a] \\ &\leq \frac{1}{m-3}(\Pr[X_{t+\delta} = a \mid \text{case } 1\ ] \cdot \Pr[\text{ case } 1\ ] \\ &\quad + \Pr[X_{t+\delta} = a \mid \text{case } 2\ ] \cdot \Pr[\text{ case } 2\ ] \\ &\quad + \Pr[X_{t+\delta} = a \mid \text{case } 3\ ] \cdot \Pr[\text{case } 3\ ]) \\ &\leq \frac{p_{M3}}{(m-3)^2} + \frac{(1-p_{M3})p_{H3}}{(m-3)^2} + \frac{(1-p_{M3})(1-p_{H3})}{3(m-3)}.\end{aligned}$$

The similar argument can be applied for the case of k blocks are copied on every accesses, and the probability can be given as follows:

$$\epsilon \leq \frac{p_{Mk}}{(m-k)^2} + \frac{(1-p_{Mk})p_{Hk}}{(m-k)^2} + \frac{(1-p_{Mk})(1-p_{Hk})}{k(m-k)},$$

where $p_{Mk} \leq (\frac{m-k}{m})^{\delta}$ and $p_{Hk} \leq (\frac{\ell_h}{\ell_h+k})^{\delta}$.

When two blocks are copied on every access, $\epsilon \leq 1.4 \times 10^{-4}$ when $m = 128$, $\ell_h = 512$ and $\delta = 20$. When three blocks are copied, $\epsilon \leq 1.7 \times 10^{-4}$ under the same condition. Figure 4.1 shows the relation between the number of blocks ($2 \leq m \leq 127$, x-axis) and ϵ (y-axis). As shown in the figure, ϵ takes minimum when the number of blocks equals to two, then stays low until the number of blocks equals 120, then rapidly rises.

4.5 Comparison

Our scheme requires secure memory for storing m data blocks. It also requires storage for the history table of size ℓ_h. When we read a data block, we first access the secure memory to check whether the data block sought is in $\mathcal{M}$. We then move two data blocks into $\mathcal{M}$ and read the data block. We also need to access and update the history table. Thus the total cost is 4 operations, plus the cost associated with reading and writing the history table (which can be only two extra accesses if $\mathcal{H}$ is within the secured boundary). Thus we have the overhead of 7 operations. We discuss how to improve the overhead later this section. The history can be potentially stored (encrypted) in the memory area $\mathcal{L}$. When $\mathcal{H}$ is not within the secured boundary, we have to read and update all blocks

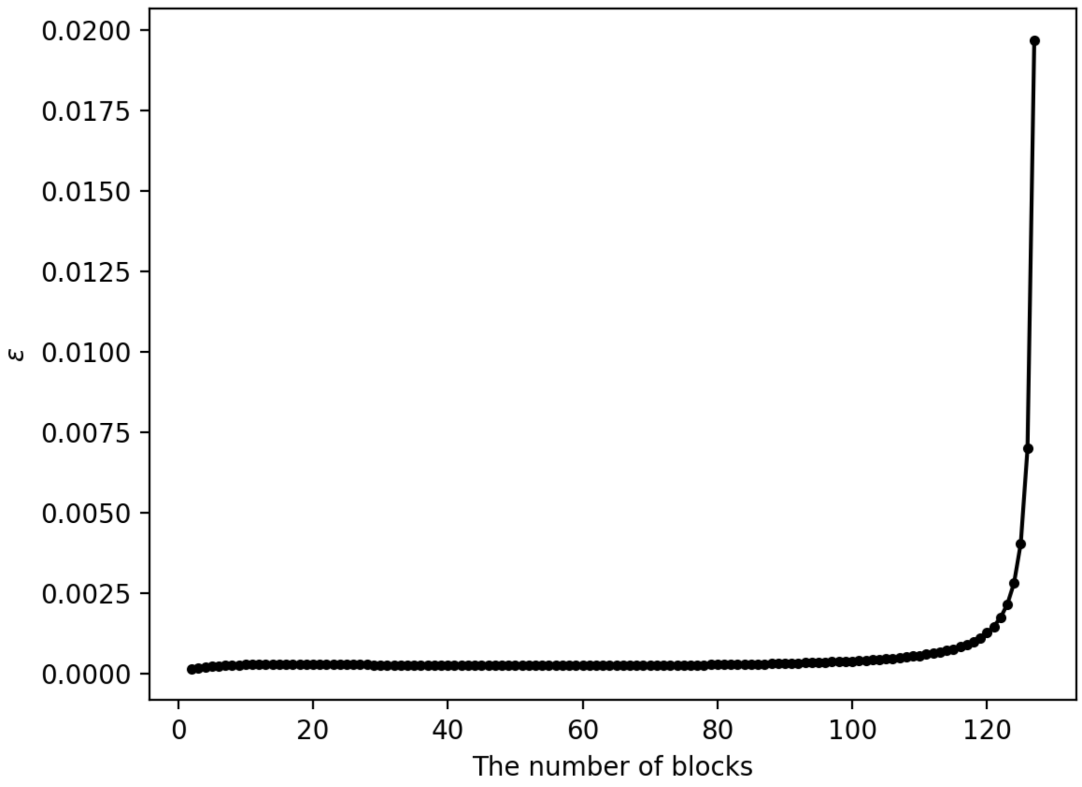


Figure 4.1: The number of blocks (x-axis) and ϵ (y-axis) when $m = 128$, $\ell_h = 512$ and $\delta = 20$

in the table. Thus we have the overhead of $4 + 2\ell_h$. Finally, if using *dummy* data, we also have data storage overhead in $\mathcal{L}$ depending on the constant k.

Our scheme can also be implemented without any special hardware. The difference in this case is that we need to scan the entire buffer $\mathcal{M}$ twice, plus the access to the history table and two swaps. We have therefore the cost of $2m + 2\ell_h + 2$ operations. As we assume m is much smaller than n, our scheme without a hardware is still more efficient than most of oblivious RAMs. According to the work by Stefanov *et al.* [86], practical overhead of oblivious RAMs are, in general, on the range of thousands to hundred of thousands. When we choose $m = \ell_h = 128$, the overhead is 514, which is less than half of that of oblivious RAMs. The scheme proposed in the work of Stefanov *et al.* [86]

achieved the overhead of $(20 - 35)\times$, which is faster than our second scheme, still our scheme has the advantage in terms of storage size, that is, ours requires less than $3n$ storage for n original data while the scheme in the work of Stefanov *et al.* [86] requires $4n + o(n)$.

Overall, the parameters m, ℓ_h and k can be set to suitable values, to offer the appropriate trade-off between security and implementation/computational costs. While oblivious RAMs are too expensive especially for resource constrained devices and Zhuang *et al.*'s scheme is not suitable scheme for this purpose, our scheme can offer reasonable security for the access pattern protection problem with constant computational overhead and practical storage.

Unlike the ORAMs, the proposal hides the timing of the logical access. By adding the physical access based on the history (i.e. bringing the block which has been kicked out from the buffer into the buffer again), the timing can be hidden. As there is no need to hide the addresses, re-shuffling can be eliminated, and this realises the constant overhead.

Improvement of Performance. When we update the history table, addresses of two data blocks are registered into the history table independently in the original scheme. Assume that two addresses 'i_a' and 'i_b' need to be registered, then two random locations are selected and those locations are updated with the new blocks. Therefore, the update requires two operations in each cycle. The update can be done with only one operation by modifying the entry as a bigger one $i_a||i_b$, where $||$ is concatenation. When choosing one data block from the history table, one can first choose the concatenated block and then can choose higher half or lower half of the block. As a result, we can reduce the cost for updating the history table from 2 to 1 and the overall overhead is improved to 6.

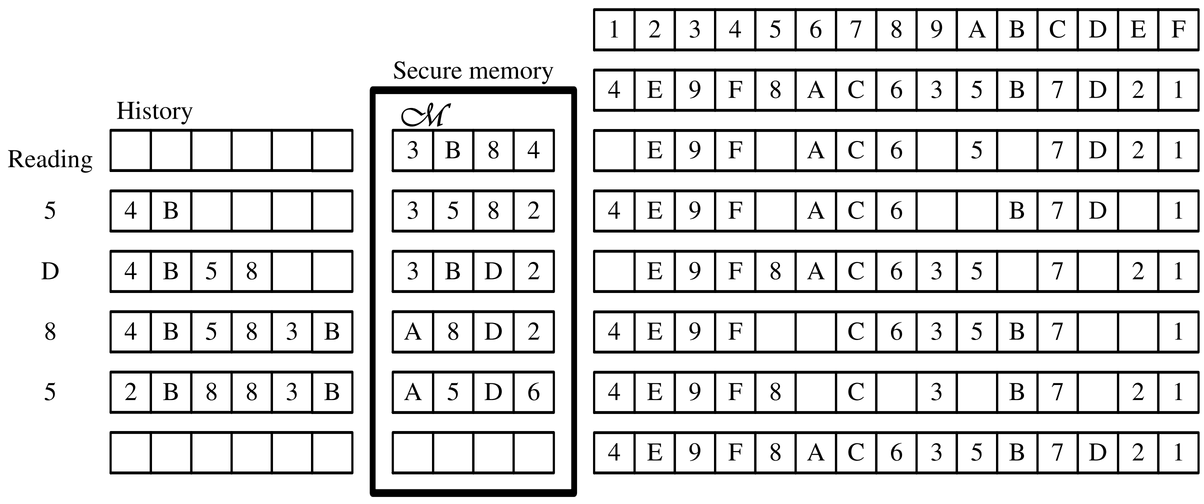


Figure 4.2: Our Scheme

4.6 Small Example of Proposed Scheme

We present a small example of the scheme in Figure 4.2. In this example, we have $m = 4$ and $\ell_h = 6$, then the program reads data in the order $5 \to D \to 8 \to 5$. For simplicity, we will not be using dummy data (i.e. $k = 1$) and we have a fixed permutation.

1. All data in the memory $\mathcal{L}$ is randomly permuted and $\mathcal{M}$ is filled with the 4 random blocks.

2. When the program tries to access 5, since it is not in $\mathcal{M}$, blocks 5 and 2 are brought into $\mathcal{M}$. Since we do not have any history yet, block 5 and a random block is chosen. Two blocks (4 and B) in $\mathcal{M}$ are written back to $\mathcal{L}$, and their addresses are entered in the history.

3. When the program accesses D, the block D is brought into $\mathcal{M}$. The block B is chosen from the history and also brought into $\mathcal{M}$. Blocks 5 and 8 are instead written back to $\mathcal{L}$, and their addresses are entered in the history.

4. When the program accesses 8, since 8 is in the history, block A is randomly chosen. Blocks 3 and B are written back to $\mathcal{L}$, and their addresses are entered in the history.

5. When the program accesses 5, since the address of 5 is in the history, the block 6 is randomly chosen to be brought into the $\mathcal{M}$. Blocks 2 and 8 are written back to $\mathcal{L}$. Now the history table is full, two addresses of random blocks (in this example, 4 and 5) are replaced with those of 2 and 8.

6. All blocks in $\mathcal{M}$ are written back into $\mathcal{L}$.

4.7 Implementation Issues

Considering the practical implementation of ORAM, there were several questions to be addressed and improvements to be achieved. We propose practical solutions for these issues. The issues are

- Management of data blocks in the buffer
- Construction of a secure area
- Size of each block.

In Sec. 4.7.1, we propose a better management of data blocks. The construction of secure area based only on software is discussed in Sec. 4.7.2. Finally we discuss better use of each block for saving storage in Sec. 4.7.3.

4.7.1 Managing Data in Buffer Using Flags

The square-root based solutions first scan all data blocks in order to know if the accessing block is in the shelter or not at the beginning of the process. If the accessing block is in the shelter, the block from a dummy location will be fetched into the shelter. When the dummy block is being fetched, there is a chance that the block which is already in the shelter is chosen again. If this block is the dummy one, it is not a problem. However, if the block is the real one, which is the one actually accessed by the program, we cannot distinguish which block is newer. As the result, the program may misbehave in the software protection scenario and the database may be ruined in the

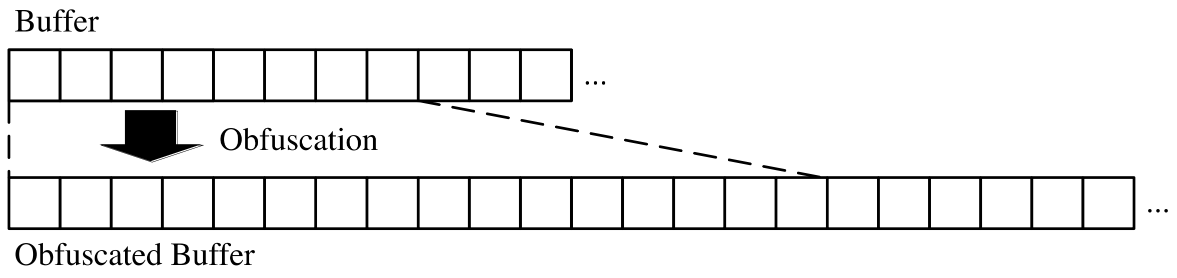


Figure 4.3: Constructing Secure Area Using Obfuscation

database access protection scenario. Therefore, we must ensure that all blocks in the shelter are the latest and there is no duplication.

The trivial solution to ensure that is to scan the entire shelter before fetching the data block, which impose large overhead. We can omit this scan by using n 1-bit flags F_i indicating whether the data block is in the buffer or not. We set $F_i = 1$ when data stored in address i is stored in the buffer and $F_i = 0$ otherwise. When bringing two blocks into the buffer, we check flag of each block. If it is 0, we bring the block into the block. If it is 1, we pick different block and check the flag again.

The same situation can happen in the lightweight scheme proposed in the work of Nakano *et al.* [66]. Their scheme always fetches two blocks in to the shelter, the chance of the block which is already in the shelter being chosen is higher than the square-root solution. We have confirmed that this solution can improve the performance by roughly 3 times in many parameter settings with a prototype implementation. When the hierarchical solution is applied, there is no need for the management of newer data as newer data blocks are always upper level. The scheme proposed in the work of Zhuang *et al.* [99] does not either require this management as their scheme swaps two blocks between the shelter and main area.

4.7.2 Constructing Secure Region

ORAM of square-root solution has a shelter of size $\sqrt{N}$ and the scheme has to access all data blocks in the shelter at least twice per one access. As the size of the program or database grows, the size of the shelter also grows, which increase the computational overhead of the implementation. ORAM of hierarchical solution also has the same problem as it also has to access all data blocks in the top level buffer. Though it may not be serious as square-root solution, as the size of top level buffer is fixed and usually smaller than $\sqrt{N}$.

The cost of accessing shelter or top level buffer can be reduced by using secure hardware where only the client can access as done in the works of Nakano *et al.* [66] and Zhuang *et al.* [99]. Though this is looks promising, requiring secure hardware may compromise the practicality. The third option is to construct a secure region with obfuscation. Let $access(K)$ be the complexity of accessing K blocks and $obf(V)$ be the complexity of obfuscating variables V. Also let X be the variables after the obfuscation. Then the obfuscation can offer higher performance than accessing all data blocks in the buffer when the following inequation holds;

$$access(K) \leq obf(V) + access(X).$$

In the square root construction, the size of the shelter tends to large as its size is $\sqrt{N}$. Therefore, it is likely that we can improve the performance of ORAM scheme with this method. Depending on the size of the top level buffer for hierarchical solution and the complexity of obfuscation $obf(V)$, this method is also applicable to the hierarchical solutions. It is also applicable for the scheme proposed by Nakano *et al.* [66].

We use the scheme proposed by Fukushima *et al.* [34] for obfuscation. By using obfuscation with the memory protection scheme, we only need to protect the shelter (square-root [40]), top level buffer (hierarchical [40]) or buffer and history table (Nakano *et al.*'s scheme [66]), which are much smaller than N. As described in the following, we divide the buffer and history table into smaller ones and apply the obfuscation repeatedly. Hence we can further decrease the overhead due to obfuscation.

By obfuscating data blocks inside the secure region, the accesses to memory is also "obfuscated", that is, the access to a certain data block is transformed into the access(es) to obfuscated block(s). When the protection scheme needs to access one of data blocks, it accesses obfuscated blocks and unlock the obfuscation in order to obtain the real value. As the correspondence between the original access and the “obfuscated” access(es) is secret, an adversary cannot understand which block is actually accessed. The obfuscation does not affect any operation done by the protection scheme as it only encodes the data blocks. The obfuscation also realise less access overhead than that of accessing all blocks in the buffer by appropriately choosing the parameters, which we discuss later this section. Thus, by using obfuscation for data blocks, we can construct a secure area without hardware, and less overhead than accessing all blocks in the buffer.

Each block which requires to be secret is implemented as variables, and these variables are encoded into obfuscated variables. The following is a small example of the obfuscation from a set of variables $\{v_1, v_2, v_3\}$ into a set of obfuscated variables $\{x_1, x_2, x_3, x_4, x_5\}$;

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ d \end{pmatrix} \oplus \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix}, \tag{4.1}$$

where the matrix is chosen as its rank to be the same as the number of elements in the original variables (in this example, it is 4). Each element of the matrix is randomly chosen except the rightmost column, which is set to all 1s. A random variable d is generated by a pseudo-random number generator and it is attached to the original variables. Finally a secret vector $\{y_1, y_2, y_3, y_4, y_5\}$ is exclusive-ORed, this operation acts like random masking. We repeatedly apply the same obfuscation for the multiple sets of the original values until we obtain enough size of the secure region. In this example, we obtain the secure region of size 4 per one iteration.

When one of v_is, say v_1, is requested, we first determine which set of obfuscated variables to access depending on the index of v_1, then access all x_is to decode v_1. Suppose v_1 is updated to v_1' after the operation, then we apply the same obfuscation as given in Eq. (4.1) with newly generated random number d'. We use different random numbers every time the program accesses to the secure region. As the corresponding column to d

is set to all 1s, the change of d will cause the changes of all variables of $\{x_i\}$ even if none of variables $\{v_i\}$ are changed (i.e. `read` operation). Moreover, as the encoding applies the secret matrix, the adversary cannot detect the correspondence between $\{v_i\}$ and $\{x_i\}$. Though the adversary may be able to detect which one of $\{x_i\}$ is being accessed, one cannot understand which one of $\{v_i\}$ is being accessed. Thus, we can construct a secure region using obfuscation.

There is, however, a limitation to use obfuscation for constructing secure area, which is the size and number of all variables must be pre-determined. In order to overcome this limitation, we divide the secure region into smaller units. Figure 4.3 shows an example when 8 blocks to be obfuscated into 16 blocks with one random number. We can of course use different matrices and vectors for each obfuscation unit, which makes obfuscation more difficult to be analysed, though this will increase the code size. We also note that we can of course choose arbitrary size for the unit of obfuscation.

4.7.3 Reducing Storage Overhead

In Zhuang *et al.*'s scheme [99], the uniform unit of data was defined as a block of size 128-bit. However, some programs uses a byte block or smaller as the smallest unit. When one byte data is allocated to 16 byte block, it wastes remaining 15 bytes. For the better management of storage, we modify the scheme so that one block can store multiple smaller blocks.

This modification not only saves storage, but also improves the performance for operations such as data copy and move. When a program copies for example four integer-type blocks, the original scheme requires four copy operations. On the other hand, our proposal only requires one copy operation when these four blocks are in the same 128-bit block. By carefully choosing smaller blocks on storing them into a large block, we can expect speed-up and saving storage at the same time.

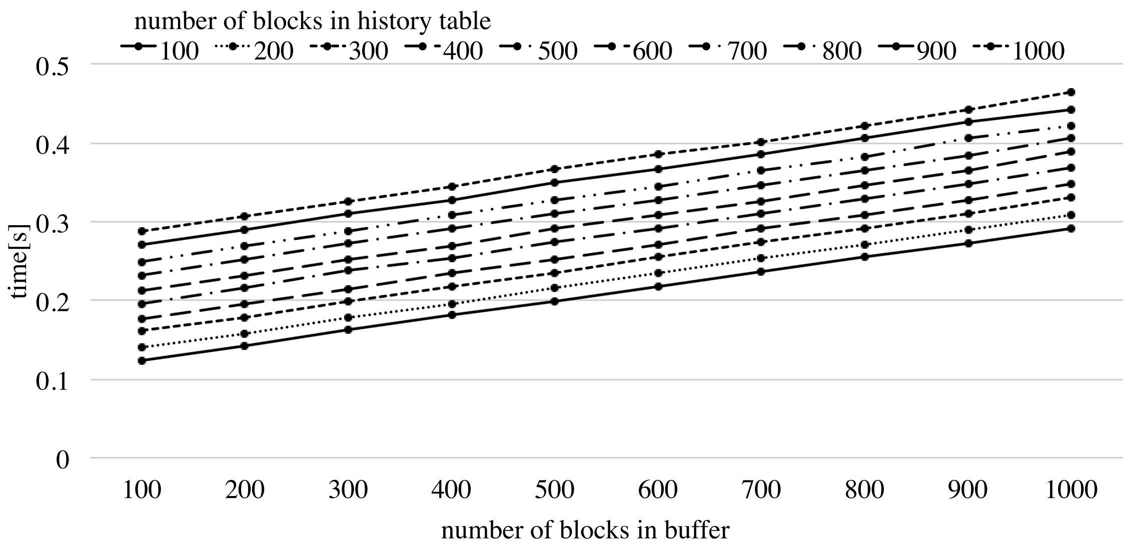


Figure 4.4: Performance Evaluation in Various Settings

4.7.4 Implementation Result

We implemented our scheme to which three modifications are applied. Then we measured the required time to write 1MB data on RAM using the protection scheme. The evaluation was done on the PC of CPU: Intel(R) Core(TM) i7 3930K and RAM: 8 GB which runs Ubuntu 13.04 x64 and gcc 4.7.3. As our scheme has two parameters (e.g. number of blocks in the buffer and history table) which affects the performance, we measured the required time while we change the number of blocks in the buffer and the history table from 100 to 1000. Figure 4.4 shows the relation between parameters and performance of our efficient implementation. The x-axis denotes the number of blocks (from 100 to 1,000 for every 100) in the secure buffer, the y-axis denotes required time to load 1 MB data to the protection scheme and each curve corresponds to each size of the history table. In the fastest setting, our scheme only requires 0.125[s] to write 1 MB data. As shown in the figure, the overhead grows linear to the size of buffer and history table. This is because more time required to scan the entire buffer and history table as their size grow. However, growth of overhead is very slow compared to that of size of buffer and history table thanks to the efficient implementation of scanning process.

Table 4.2: Performance of Path ORAM

depth	No. of blocks	time [s]
4	124	0.023174
5	252	0.026869
6	508	0.031161
7	1020	0.036253
8	2044	0.040308
9	4092	0.045745
10	8188	0.050515
11	16380	0.055507
12	32764	0.061350
13	65532	0.067854
14	131068	0.070075
15	262140	0.074905

4.8 Performance Comparison with Path ORAM

We compare the performance of our method and Path ORAM in several settings when both are applied to protect the secret key of AES-128. We perform the encryption of 1024-bit data with AES-128 five times and compare the average time required to complete the encryption. The evaluation was done on the PC of CPU: Intel(R) Core(TM) i7 7700K and RAM: 32 GB which runs Ubuntu 18.04.1 LTS x64 and gcc 7.3.0. The results are summarised in Tables. 4.2 and 4.3. These results are updated ones because the implementation used for the original comparison is no longer available and we used different implementation for this comparison. The Path ORAM implementation is available at `https://github.com/AmarSaggu/ORAM`.

Each node of Path ORAM has 4 blocks hence, over all number of blocks are 4 times the number of nodes. In case the depth of the tree is 4, the number of blocks will be $4 \times 31 = 124$. For our proposed scheme, we choose the size of buffer and history (i.e. the

Table 4.3: Performance of Proposed Scheme

ORAM Storage [byte]	time [s]
1296	0.014449
1808	0.014576
2832	0.015511
4880	0.013627
8976	0.015189
17168	0.013736
33552	0.014095
66320	0.014743
131856	0.016971
262928	0.014286
525072	0.013470
1049360	0.013698

number of blocks that can be stored in buffer or history table) to be 16. For the size of ORAM storage, which is equivalent to the number of blocks in Path ORAM, we set the storage size to roughly matches to that of Path ORAM, note that our scheme always preserve temporary storage of 800 bytes and each block in Path ORAM is 4 bytes.

As shown in Tables. 4.2 and 4.3, our scheme outperfoms Path ORAM by maximum 5 times. Path ORAM tends to slow down as the level increases, while our scheme can achieve the constant performance. Our scheme has the larger advantage when the size of ORAM storage increases.

4.9 Conclusion of This Chapter

In this chapter, we proposed two new schemes for protecting memory access patterns. The distinctive character of our scheme is that we do not re-shuffle the order of the data blocks in memory. To protect the access pattern without re-shuffling, we used a *history* of the accesses. We first considered an instance which is similar to the proposal in the work of Nakano *et al.* [99], and also relies on the use of a secure (trusted) hardware buffer; however it achieves higher security by adapting ideas from Oblivious RAM (ORAM) mechanisms, without the expensive (re-)shuffling of buffers. Another instance requires no special hardware, but as a result leads to a higher, yet practical overhead. We defined a new security notion called δ-security and proved that the proposed two schemes are δ-secure. We also discussed the size of parameters, which are the size of secure memory, history table and dummy data and compared the performance with existing schemes. We claim that under reasonable assumptions, the schemes can achieve both security and performance levels acceptable in practice.

We then discussed three general implementation issues of our proposed scheme, namely management of data blocks in the buffer, construction of a secure area and size of each block, and their solutions. We applied these methods to our proposed scheme and compared the performance with Path ORAM. Our scheme outperforms Path ORAM by 5 times. The solutions can be widely used for implementing ORAM and improving the performance.

Chapter 5

Active Attack against ORAM

5.1 Introduction

The access pattern to RAM contains valuable information of the program, for example, the location of the secret key can be easily identified by observing the access pattern of a cipher program. Another example is digital right management (DRM) program. When the protected digital content is consumed, the player application first accesses to RAM to obtain the secret key. Then the content can be decrypted and played. When the user is malicious, he can analyse the access pattern of the player application in order to efficiently identify the location of the secret key. Once he identifies the location, he can use this key to decrypt the content and obtain unprotected version. *Oblivious RAM (ORAM)* is an effective countermeasure against such threat [40]. ORAM sits in between the program and RAM, and hides the access pattern of the program from the adversary who is observing read/write access to RAM. ORAM can be also effective when the program has vulnerability which can be exploited to extract data stored on RAM. The Heartbleed vulnerability is one of such examples, which the adversary can obtain data from RAM on OpenSSL servers. Another application of ORAM is called private information retrieval (PIR), where the client's access to cloud storage can be protected from curious cloud operators [92, 45, 46, 55, 93, 95, 47]. Applying ORAM for secure computations is also considered for example in the works of Faber *et al.* [30] and Doerner and Shelat [29].

The *square root solution* and *hierarchical solution* are the basic constructions of ORAM. In the square root solution, a region called 'shelter', where accessed data blocks are stored temporary, is attached to RAM. Before the program starts to access data, all data blocks must be permuted according to the secret permutation. The hierarchical solution has layered structure and access is done from top level to th e bottom. The square root solution is simpler but imposes higher overhead than the hierarchical solution. After the first proposal of ORAM, many researches have been done to improve the performance of ORAM [1, 73, 25, 44, 82, 53, 10, 98]. One of the most efficient schemes is called Path ORAM [88].

The security of ORAM is often considered under passive adversaries who can observe memory access but cannot alter data stored on RAM. However, adversaries with the ability to observe memory access, in general, also have the ability to modify data on RAM using a debugger. Ren *et al.* [76] considered security of Path ORAM under the active adversary and showed that naïve recursive Path ORAMs leak information on memory access patterns.

In this chapter, we further consider the security of several ORAM schemes against the active adversaries and propose two active attacks. These two attacks assume more powerful adversary than one considered by Ren *et al.* [76] as the adversary can re-execute the ORAM in the same setting. Yet, the proposed attack is more critical since the adversary can identify the pattern of accesses. The first attack we propose is identifying addresses where dummy blocks are stored in ORAM. The second one is to try to identify in which step of the program a certain data block will be accessed. The basic idea behind our attacks is that dummy blocks do not affect the behaviour (i.e. internal state of the program) and the output, while *real* data blocks do. The adversary can use a debugger and record all internal states in each step then modifies blocks and observes if any difference will be incurred by the modification.

5.2 Preliminaries

A program access RAM to obtain data block to complete its functionality, that is, data blocks are the minimum set of blocks required by the program. Oblivious RAM can add dummy blocks to hide the access pattern of the given program. Therefore, dummy blocks do not affect the functionality of the program. We say block when it can be either data or dummy.

5.2.1 Side Channel Attack against ORAM

As ORAM can hide the access patterns made by a program, the program can be securely executed even if the program is on malicious system with a aid of secure processors. However, timing information of ORAM accesses can leak information. ORAM accesses are only made when the necessary data misses processor's cache. Fletcher *et al.* [31] evaluated the upper bound on timing channel leakage and proposed a countermeasure which only impose reasonable overheads.

Bao and Srivastava [5] also investigated the security of Path ORAM against timing attacks and showed the adversary can obtain secret information though timing. Even though the security of Path ORAM is proven, its proof is under the assumption that CPU is trusted. This assumption, however, sometimes can not be met when ORAM is executed along with malicious program. Bao and Srivastava assumed the malicious program which makes continuance requests to ORAM and can measure the time between the request of a data block and its completion. If the victim program includes data dependent operations and these operations involve ORAM access, it affects the request completion time of the malicious program, which can reveal secret data of the victim program.

5.2.2 Ren *et al.*'s Active Attack

Ren *et al.* [76] considered the security of Path ORAM under active adversaries. Their attack enables the active adversary to distinguish two access patterns: (a) repeated accesses to the same data block, and (b) accesses to different data blocks. Assume the adversary tries to distinguish access patterns of 2-level recursive ORAM (ORAM$_0$ and ORAM$_1$), one first initialise two labels ℓ_0^*, ℓ_1^* and $\mathcal{P} \leftarrow \perp$. Then their attack has following four steps:

1. first perform access(ORAM$_1$, ℓ_1) and if $\ell_1^* = \ell_1$ go to step 4, if not go to step 2;
2. set $\mathcal{P} \leftarrow \ell_1$ and $\ell_1^* = \ell_1$, and perform access(ORAM$_1$, ℓ_1);
3. perform access(ORAM$_0$, ℓ_0) and set $\ell_0^* = \ell_0$, and go to step 1;
4. perform access(ORAM$_0$, ℓ_0'), if $\ell_0' = \ell_0$, then guess the pattern is repeated accesses to the same block, otherwise accesses to different block.

They also proposed a countermeasure to the active adversary, which ensures *authenticity* and *freshness* of retrieved data block. They added another tree, called authentication tree, to that of Path ORAM and each node of authentication tree contains hash value of corresponding bucket in ORAM tree. More precisely i-th node of authentication tree contains the hash value of $hash_i = HASH(B_i||hash_{2i+1}||hash_{2i+2})$, where $HASH$ is a secure hash function, B_i is i-th bucket of ORAM tree and $hash_{2i+1}||hash_{2i+2}$ is the concatenation of hash values of two children of ith node of hash tree. All hash values $hash_i(i > 1)$ can be stored unsecure memory except the root hash $hash_0$. This countermeasure, however, requires calculating hash values for all buckets of ORAM tree upon the initialisation of the ORAM. They also proposed a countermeasure which does not require the initialisation of authentication tree.

In the following sections, we consider the active attack further, that is, try to identify the access pattern of the program.

5.3 Identifying Dummy Blocks

Both Square root ORAM and our scheme attach a secure region to ORAM's main memory and copying blocks from main memory to the secure region. They also share

Table 5.1: Probabilities that the adversary can detect dummy blocks

Scheme	Probability
square root	$\frac{\sqrt{N}p_S}{N+\sqrt{N}}$
ours	$\frac{1+p_M}{2} \cdot \frac{d}{N+d}$

Algorithm 3 Identifying Dummy Blocks

```
Require: ORAM data set $\mathbf{D} = (D_1, D_2, \ldots D_N)$
Ensure: Identified dummy blocks
    $\mathbf{I} = ((D_1, \mathsf{data/dummy}), \ldots, (D_N, \mathsf{data/dummy}))$
 1: $\mathsf{Init} \leftarrow$ initial state
 2: $\mathsf{Out} \leftarrow$ output of the program
 3: for $1 \leq m \leq N$ do
 4:   revert to $\mathsf{Init}$
 5:   $D_m \leftarrow$ random number et
 6:   $\mathsf{Out}' \leftarrow$ output of the program
 7:   if $\mathsf{Out} = \mathsf{Out}'$ then
 8:     $I_m \leftarrow (D_m, \mathsf{dummy})$
 9:   else
10:     $I_m \leftarrow (D_m, \mathsf{data})$
11:   end if
12: end for
```

the idea that dummy blocks will be added to the original blocks. In order to hide the access pattern, one block (two blocks in our scheme) is always copied to the secure region. Therefore, by observing if these copied blocks affect the behaviour of the program, the adversary has chance to identify all dummy blocks efficiently. Algorithm 3 gives the attack description. The Table 5.1 summaries the probabilities.

5.3.1 Identifying Dummy Blocks in Square Root ORAM

In square root ORAM, the data block copied into the shelter will be determined if it is already inside the shelter. When the block a is already in the shelter, a block will be randomly chosen. Otherwise the block a will be chosen. We consider the probability that the adversary can identify the dummy block in two cases.

(1) $a \in S$

When the block a is inside the shelter S, a block is randomly chosen and copied into the shelter. Let p_S be the probability which the data block a is inside the shelter, the probability that the block copied to the shelter is dummy is given by the following:

$$\frac{\sqrt{N}p_S}{N+\sqrt{N}}.$$

(2) $a \notin S$

When block a is not inside the shelter, it is copied into the shelter, hence it cannot be dummy block.

From discussions of (1) $a \in S$ and (2) $a \notin S$, the probability which the adversary can distinguish dummy block is following:

$$\frac{\sqrt{N}p_S}{N+\sqrt{N}}.$$

From $0 \leq p_S \leq 1$, the following inequation also holds, which imply that the adversary does not gain any advantage in detecting dummy block:

$$\frac{\sqrt{N}p_S}{N+\sqrt{N}} \leq \frac{\sqrt{N}}{N+\sqrt{N}}.$$

5.3.2 Identifying Dummy Blocks in Our Proposal

In our scheme, at least one of two blocks copied into the buffer $\mathcal{M}$ is randomly chosen. Their scheme has three possible cases depending on (1) the block is inside the buffer, (2) the block is not inside the buffer but inside the history and (3) the block is not inside the buffer and the history. In the following we consider the probability which the adversary can identify the dummy block.

(1) $a \in \mathcal{M}$

Since the required data is in the buffer, both two blocks are randomly chosen. However, the adversary does not know which block is being accessed and if that block is in the buffer, we assume he marks one of the two blocks as randomly chosen block. Let p_M be the probability of $a \in \mathcal{M}$, the adversary can detect randomly chosen block with the probability of p_M.

(2) $a \notin \mathcal{M}$, $i_a \in \mathcal{H}$

In this case, the block chosen from history is the required data and another block is randomly chosen. Therefore, the adversary can detect the randomly chosen block with the probability of 1/2, if he chooses one of the two blocks. Let p_H be the probability of $i_a \in \mathcal{H}$, the probability of $a \notin \mathcal{M}$ and $i_a \in \mathcal{H}$ will be given by $(1 - p_M)p_H$. Hence, the probability which the adversary can detect the randomly chosen block will be

$$\frac{(1 - p_M)p_H}{2}.$$

(3) $a \notin \mathcal{M}$, $i_a \notin \mathcal{H}$

One block is randomly chosen from history and a will be copied to the buffer. Therefore, the adversary can detect the randomly chosen block with the probability of 1/2, if he chooses one of the two blocks. The probability that $a \notin \mathcal{M}$, $i_a \notin \mathcal{H}$ happens will be given by $(1 - p_M)(1 - p_H)$. Hence, the probability which the adversary can detect the randomly chosen block will be

$$\frac{(1 - p_M)(1 - p_H)}{2}.$$

As the probability that a randomly chosen block is dummy can be denoted as $d/(N+d)$, the discussions of (1) $a \in \mathcal{M}$, (2) $a \notin \mathcal{M}$, $i_a \in \mathcal{H}$ and (3) $a \notin \mathcal{M}$, $i_a \notin \mathcal{H}$ give the adversary can detect dummy block with the following probability:

$$\frac{1 + p_M}{2} \cdot \frac{d}{N + d}.$$

Similar to the conclusion in section 5.3.1, the adversary cannot gain any advantage in detecting dummy blocks when he modifies the copied block.

5.4 Identifying Access Pattern

We consider another attack where an active adversary modifies the block in the initial state of a program. The adversary we assume in this section has abilities to repeat the execution of the program step-by-step with the same setting and have full access to the internal states. When data block is modified by the adversary, it should affect the output of the program, while modifying dummy should not. By exploiting this fact, the active adversary can identify the access pattern of the given program.

First, the adversary saves the initial state of the program and ORAM. Then he runs the program while recording all observable accesses, internal states of the program and the output. After the initial run of the program he reverts the states of the program and ORAM into the initial state. Next he adds a modification to one of the blocks and re-run the program step-by-step, while comparing the observable access and the internal states of the program. Once he notices any difference to the internal state of the program, he marks the modified block with in which step the difference occurred. If any difference to the internal state or the output of the program, he marks the chosen block as dummy. The adversary can repeat this process until all blocks are marked. We have a pseudocode of our attack in Algorithm 4. Here, Algorithm 4 takes ORAM data set $\mathbf{D} = (D_1, D_2, \dots D_N)$ as input and it outputs $\mathbf{I} = ((D_1, i_1), \dots, (D_N, i_N))$, which indicates block D_i being accessed in step i.

When the block is accessed many times, the active adversary can detect the first access to it by modifying the data and monitor the behaviour of the program. However, the second and onward accesses cannot be detected as the modification to the initial state will cause unexpected behaviour to the program. To detect the repeated accesses to the same block, the adversary also needs modify the block while executing the program. First he modifies the block in the initial state and observes the effect to the behaviour of the program. Also he needs to observe a new location of the modified block after the first access. Then, he goes back to the initial state and reverts the modified block to the original one and executes the program right after the first access to it. After the first access, this block will be moved to the new location, then the adversary adds another modification to it. If this block is accessed second time, he can detect this access by observing its effect to the behaviour of the program. He can repeat the same process to detect the third and onwards accesses.

In the following, we consider applying the attack to three ORAM schemes.

5.4.1 Identifying Access Pattern in Square Root ORAM

There are two cases where the block is copied to the shelter: 1) because the block is requested by the program or 2) because the block is chosen at random. In the case of 1), the modified block is not copied into the shelter until it is required by the program, and when it is accessed, it affects the behaviour of the program. The adversary will notice the effect of his modification by monitoring the update of the internal state. In the case of 2), the modified block is copied to the shelter as a randomly chosen block. The adversary might notice that the modified block has been copied, however, his modification will not affect the behaviour as it will not be accessed in this step by the program. After several steps, the program might access it and then the behaviour will be affected. Even though the separation between the step in which the modified block being copied to the shelter and the actual access by the program, the adversary can still identify in which step the modified block is accessed by monitoring its effect to the internal state.

In square root ORAM, all blocks are periodically re-shuffled according to a new permutation with re-encryption. As the permutation is secret to the adversary, and the encryption is probabilistic, a passive adversary cannot link the blocks before and after the re-shuffle. An active adversary, however, can track a block before and after the re-shuffle. The adversary first observes the re-shuffle. Then he modifies a block and let the ORAM to perform the same re-shuffle again while observing. Since he only modified one block there should be only one block which is different from the first re-shuffle. By iterating the process N times, the adversary can identify where each block is allocated after the re-shuffle. When the total number of accesses done by the program is given by X, the re-shuffle will be performed X/N. Therefore the cost for the attack against re-shuffle will be X.

5.4.2 Identifying Access Pattern in Hierarchical ORAM

Similar discussion to square root ORAM can by applied to hierarchical ORAM. The main difference is that in square root ORAM only one block is copied to the shelter in each access while in hierarchical ORAM multiple blocks can be copied to the shelter.

In hierarchical scheme, one block will be chosen from each level. When making access to a block a, block stored in $HASH_i(a)$ will be retrieved from each level i until the block a is found. After the block is found, block stored in a dummy location $HASH_i(dummy)$ will be retrieved from each level. As multiple blocks are copied to the shelter on each access, there is slightly higher chance of the modified block being copied to the shelter when the program is requesting different block. Still, the modified block will not affect the behaviour of the program until it is being requested by the program.

When level i gets full, all blocks in level i are pushed one level down (to level $i+1$) with a new permutation and re-encrypting all blocks in level. Since we assume the adversary modifies only one block at a time, the modified block is traceable even when the block is moved between levels. Therefore, when the adversary modifies one block at the initial state, he can trace when that block is moved into which location also by comparing the behaviour of the program the adversary can guess correctly when the block is accessed by the program.

5.4.3 Identifying Access Pattern in Our Proposal

Our scheme always picks two blocks, one looks randomly chosen and the other looks chosen from the past accesses, and moves them to the buffer. The modified block can be moved to the buffer even when it is not being accessed by the program. In this case, the modified block will stay inside the buffer until it is kicked back to the unsecure region. While it is inside the buffer, the block can be accessed by the program. Then the ORAM will move two blocks into the buffer and both of them are actually not the block required by the program. However, the adversary knows which block he modified and since the only one block has been modified, if the behaviour of the program has been affected, it must be by the block which he modified. Therefore, the adversary can identify the access to a block.

5.4.4 Identifying Access Pattern in Path ORAM

We consider an example where the adversary chooses to modify a data block in bucket 4 as described in Fig 5.1. After modifying the block, he executes program step-by-step. In a certain step, this modified block will be read by the program to update its internal state. The behaviour of the program will be affected by the modification, and it will be observable to the adversary as a difference of the internal state between the original bucket 4 and modified bucket 4. This difference can be detected by the adversary using a debugger without accessing protected region of ORAM. Sometimes, the modification does not affect the transition of the states and the final output of the program. Then the adversary can conclude that the chosen block is dummy.

When any of buckets 4, 9, 10, 19, 20, 21 or 22 is being accessed, bucket 4 will be copied to the stash before the program make read/write access to data. Then data in stash will be evicted to buckets on a new path. As we make modification to only a block, the adversary can locate the bucket in which the target block is stored.

During the eviction to a new path, empty space will be filled with dummy blocks. From the assumption that the adversary can repeat the behaviour of ORAM, the same dummy blocks are generated and stored in the same bucket every time one repeat the ORAM. Therefore, only the modified block has a difference and the adversary can locate the updated path.

5.5 Experiment

We applied the proposed attack to AES with Path ORAM. We use the implementation of Path ORAM available at `https://github.com/AmarSaggu/ORAM`.

AES has been standardised by NIST as FIPS197 [67]. It is now one of the most widely

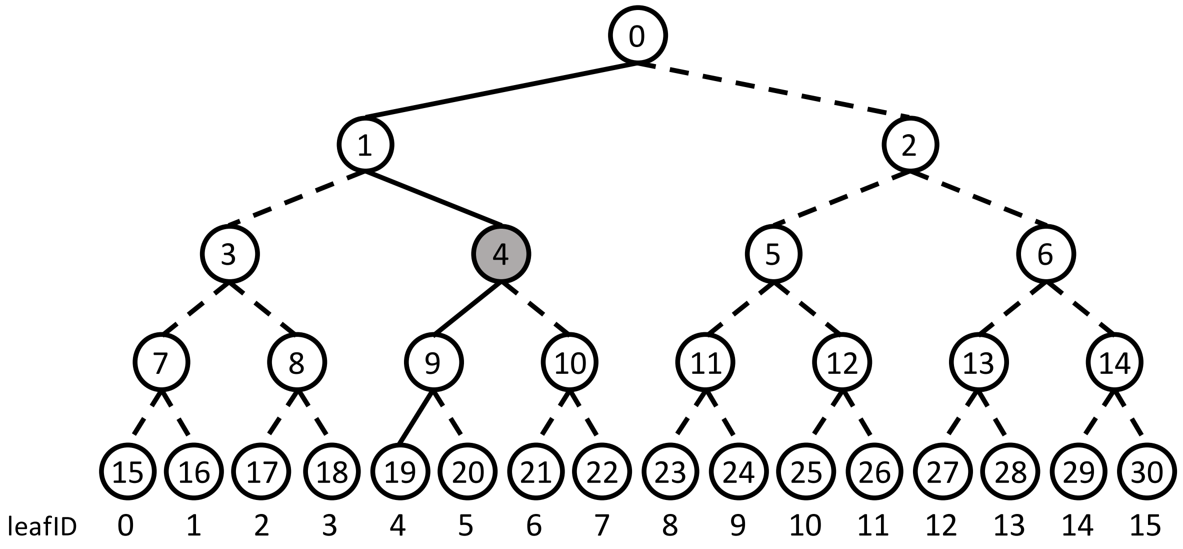


Figure 5.1: Modifying bucket 4

used symmetric key encryption algorithms. It makes use of a substitution-permutation network. It supports three different key lengths: 128-bits, 192-bits and 256-bits. We use AES-128, which has 10 rounds, in our experiment. In each round, following four functions are applied to the state (**AddRoundKey** is applied before the first round and **MixColumn** is omitted in the final round):

- **SubBytes** applies 8-to-8-bit substitution to the state;
- **ShiftRows** shifts the i-th row of the state to i positions to the left;
- **MixColumn** mixes the column C by multiplying a 4×4 maximum distance separable matrix M as $C \leftarrow M \times C$;
- **AddRoundKey** adds a round key to the state.

A key expansion is applied in the initialisation, which takes 128-bit initial key to generate ten 128-bit round keys.

In the implementation, each node contains an IV block and three blocks in the following format: $IV||block1||block2||block3$. Blocks 1 and 2 have two data blocks each, which can be represented as: $bid1||data1||bid2||data2$ and block 3 is the pudding block, therefore each bucket has 4 blocks. Here *bid* and *data* are size of 4bytes, each *block* contains 16 bytes. Path ORAM has four levels and 30 nodes in total.

During the experiment, we first modify 4 bytes *data* after the initialisation of AES to `0xffffffff`. Then we execute the encryption and detect in which round the modification will affect the internal state. We use `00112233445566778899aabbccddeeff` as a plaintext and `000102030405060708090a0b0c0d0e0f` as a key. Hence the ciphertext should be `69c4e0d86a7b0430d8cdb78070b4c55a`. Table. 5.2 and Table. 5.3 are the lists of all modifications which cause differences in ciphertexts and in which round the modification affects the internal state. The column 'node ID' shows the node ID where the block is modified and the column 'byte' shows the specific modified bytes in the node, for example, 04-07 means from fifth to eighth bytes are modified. The column 'affected round' shows the number of round in which internal state of AES has been affected. Table. 5.2 shows the result with ORAM's storage encryption enabled while Table. 5.3 shows the result with the storage encryption disabled. In both cases, round keys are stored in the same nodes. When the ORAM's encryption is enabled, we can only apply modifications to encrypted blocks.

Due to the encryption applied to entire blocks of the node, modification to 4 bytes of data will also incur difference to other bytes when it is decrypted. As a result, we observe the internal state after round 0 is affected by 12 bytes, while the internal state after round 6 is affected by only 4 bytes. For example, bytes 4 to 7 and 12 to 15 of node 1, where 10th round key is stored, are destroyed by modifying bytes 20 to 23. Bytes 20 to 23 are misidentified as 10th round key. Bytes 20 to 23 of node 2 are misidentified as 8th round key, which is actually 10th round key, because modification to bytes 20 to 23 destroys bytes 12 to 15, which is 8th round key, and incurs difference in 8th round.

Table 5.2: Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is Encrypted

node ID	byte	affected round (w/ enc)	node ID	byte	affected round (w/ enc)
1	04–07	10	17	04–07	0
1	12–15	10	17	20–23	-
1	20–23	-	17	28–31	-
1	28–31	-	20	04–07	0
2	04–07	10	20	12–15	1
2	12–15	8	20	20–23	7
2	20–23	10	20	28–31	-
2	28–31	-	22	04–07	5
3	04–07	8	22	12–15	8
3	20–23	-	22	20–23	6
3	28–31	-	22	28–31	4
4	04–07	9	23	04–07	4
4	20–23	-	23	20–23	-
4	28–31	-	23	28–31	-
8	04–07	1	24	04–07	2
8	20–23	-	24	12–15	5
8	28–31	-	24	20–23	-
9	04–07	7	24	28–31	-
9	12–15	8	25	04–07	0
9	20–23	-	25	12–15	3
9	28–31	-	25	20–23	1
10	04–07	9	25	28–31	-
10	20–23	-	27	04–07	0
10	28–31	-	27	20–23	-

Table continues to the next page

Table 5.2: Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is Encrypted

node ID	byte	affected round	node ID	byte	affected round
12	04–07	6	27	28–31	-
12	20–23	-	28	04–07	3
12	28–31	-	28	12–15	2
14	04–07	9	28	20–23	-
14	12–15	9	28	28–31	-
14	20–23	1	29	04–07	4
14	28–31	-	29	20–23	-
15	04–07	5	29	28–31	-
15	12–15	6	30	04–07	5
15	20–23	6	30	12–15	7
15	28–31	2	30	20–23	3
16	04–07	7	30	28–31	4
16	12–15	3			
16	20–23	2			
16	28–31	-			

Table 5.3: Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is NOT Encrypted

node ID	byte	affected round (w/ enc)	node ID	byte	affected round (w/ enc)
1	04–07	10	17	04–07	0
1	12–15	10	17	20–23	0
1	20–23	10	17	28–31	0
1	28–31	10	20	04–07	0

Table continues to the next page

Table 5.3: Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is NOT Encrypted

node ID	byte	affected round (w/ enc)	node ID	byte	affected round (w/ enc)
2	04–07	10	20	12–15	1
2	12–15	8	20	20–23	0
2	20–23	8	20	28–31	0
2	28–31	8	22	04–07	5
3	04–07	8	22	12–15	8
3	20–23	8	22	20–23	5
3	28–31	8	22	28–31	4
4	04–07	9	23	04–07	4
4	20–23	9	23	20–23	4
4	28–31	9	23	28–31	4
8	04–07	1	24	04–07	2
8	20–23	1	24	12–15	5
8	28–31	1	24	20–23	2
9	04–07	7	24	28–31	2
9	12–15	8	25	04–07	0
9	20–23	7	25	12–15	3
9	28–31	7	25	20–23	0
10	04–07	9	25	28–31	0
10	20–23	9	27	04–07	0
10	28–31	9	27	20–23	0
12	04–07	6	27	28–31	0
12	20–23	6	28	04–07	3
12	28–31	6	28	12–15	2
14	04–07	9	28	20–23	2

Table continues to the next page

Table 5.3: Modified Bytes and Affected Rounds in AES Encryption when ORAM Storage is NOT Encrypted

node ID	byte	affected round (w/ enc)	node ID	byte	affected round (w/ enc)
14	12–15	9	28	28–31	2
14	20–23	1	29	04–07	4
14	28–31	9	29	20–23	4
15	04–07	5	29	28–31	4
15	12–15	6	30	04–07	5
15	20–23	5	30	12–15	7
15	28–31	2	30	20–23	3
16	04–07	7	30	28–31	4
16	12–15	3			
16	20–23	2			
16	28–31	3			

5.6 Countermeasures

In this section, we consider countermeasures for active attacks. The adversary uses effect of modified block to the intermediate state or the output of the program in order to detect dummy or the access pattern, we can prevent the attack by detecting the modification by the adversary and/or making the modification incur unexpected behaviour to the program.

5.6.1 Parallelism

Multiple ORAMs can be implemented in parallel and checks if the all blocks matches. If one of the blocks is different from others, it is likely that the adversary modified one of the blocks and the program should be terminated. In case more than two ORAMs are implemented, the modification can be reverted with majority voting. However, this countermeasure has a flow that the adversary can modify the blocks without being detected if the adversary modifies all corresponding blocks of ORAMs implemented in parallel. This countermeasure also impose even higher overhead than applying single ORAM.

5.6.2 Integrity Checking

Similar to Ren *et al.*'s [76] proposal, we consider adding checksums to all blocks to realise any modification made by the adversary can be detected. However, the adversary can identify the location where the checksum is stored and modify the checksum so that it matches to the modified block. In order to provide better security, we can use a keyed hash as follows:

$$H(data||index||key),$$

where key must be stored in a secure region. The checksum can be located along with the blocks. When ORAM is accessing a block, the checksum is also copied into the secure region and ORAM compares two checksums.

5.7 Conclusion of This Chapter

We considered two active attacks against ORAM, one is identifying dummy blocks another is identifying access pattern. As we showed the adversary with ability to repeat ORAM with the same setting can identify the access pattern by comparing the behaviour of the program with and without modification to blocks. We applied our scheme to AES implemented with Path ORAM and showed the location of round keys can be obtained. Therefore, the countermeasure to active attack should be applied to ORAM schemes.

Algorithm 4 Attack Description for Detecting Access Pattern

```
Require: ORAM data set $\mathbf{D} = (D_1, D_2, \dots D_N)$
Ensure: Identified pattern $\mathbf{I} = ((D_1, T_1), \dots, (D_N, T_N))$
 1: $\mathsf{Init} \leftarrow$ initial state
 2: for $0 \le i \le Len$ do
 3:   execute step $i$ of the program
 4:   $\mathsf{Int}_i \leftarrow$ internal state
 5: end for
 6: $\mathsf{Out} \leftarrow$ output of the program
 7: for $1 \le m \le N$ do
 8:   revert to $\mathsf{Init}$
 9:   modify block $m$
10:   for $0 \le i \le Len$ do
11:     execute step $i$ of the program
12:     $\mathsf{Int'}_i \leftarrow$ internal state
13:     if $\mathsf{Int}_i \neq \mathsf{Int}'_i$ then
14:       $I_m \leftarrow (D_m, i)$
15:       break
16:     end if
17:   end for
18:   $\mathsf{Out}' \leftarrow$ output of the program
19:   if $\mathsf{Out} = \mathsf{Out}'$ then
20:     $I_m \leftarrow (D_m, \perp)$
21:   end if
22: end for
```

Chapter 6

Application of ORAM

6.1 Volume Encryption

In order to protect sensitive data stored on a hard disk, entire volume can be encrypted and only a legitimate user who knows a password to decrypt the volume can access data. When the encrypted volume is visible to an adversary, the owner of the device might be forced to enter the correct password. Therefore, there is a need for a solution which hides the existence of encrypted volume. TrueCrypt, for example, provides this functionality. This functionality of TrueCrypt is called 'hidden volume'. When this is enabled, the user can create two encrypted volumes, instead of one, and the first encrypted volume contains public data while the second one contains private data. When the adversary asks the user to give the password, the user reveals the one for the first volume and keeps the other. Then the adversary can not tell the existence of the second encrypted volume. However, TrueCrypt fails to hide the hidden volume when the adversary has an ability to take multiple snapshot of the hard disk [23].

Blass *et al.* [11] presented HIVE which is resistant to attacks using multiple snapshots. The key point of their solution is using Oblivious RAM to hide the access pattern. The reason TrueCrypt fails to hide the hidden volume is that the accesses to first volume and second volume are distinguishable as the second volume (or hidden volume) is stored separately from the first volume. By using ORAM, the accesses to first volume and second one will be indistinguishable and the adversary, even with ability to take multiple snapshots, cannot tell the existence of the hidden volume. When applying ORAM to disk encryption solution, the overhead imposed by ORAM can be huge. Therefore, Blass *et al.* proposed a more efficient ORAM which only hides the write accesses, while conventional ORAM schemes hide both read and write accesses. Based upon the efficient write only ORAM, they built HIVE.

Paterson and Strefler [72] evaluated the security of HIVE. Even though Blass *et al.* analysed the security of HIVE and gave security proof, the implementation of HIVE had a flaw which was biases of RC4 keystreams [32, 61]. The HIVE implementation uses two encryption algorithms, RC4 and AES CBC mode. RC4 is used to fill free space with pseudorandom data and AES is used to encrypt actual blocks. Since RC4 and AES are used for different purposes and keystream of RC4 is biased, the adversary can tell if the disk space of his interest is encrypted either RC4 or AES and the disk has hidden volumes. Note that the attack is only possible due to the biases of RC4 and can be prevented by using other secure algorithms.

Chapter 7

Conclusion

In this paper, we first studied a key extraction attack and a memory access pattern protection scheme.

In Chapter 2, we analysed security concerns in software protection and showed following five issues:

1. memory dump,
2. cold boot attack,
3. cache timing attack
4. debugging tool,
5. bug.

We also introduced related works in Chapter 2.

Chapter 3 showed one of the critical threats against cryptographic programs and demonstrated that sensitive information used by them can be easily recovered if any protection mechanism is not applied.

ORAM can mitigate these threat, however, even the most efficient ORAM scheme imposes impractical overhead to the performance. In Chapter 4, we proposed very lightweight scheme to overcome large overheads of ORAM schemes. The empirical performance of the proposed scheme was further improved by applying the following:

1. Efficient management of data blocks in the buffer,
2. Construction of a secure area,
3. Efficient use of blocks.

In Chapter 5, we evaluated the security of ORAM under more powerful adversary. In general, security of ORAM is analysed under the assumption that the adversary is 'honest-but-curious', that is, he tries to learn only by observing the access pattern. In some scenarios, however, the adversary can be more powerful, that is, he can not only observe but modify data blocks stored inside ORAM server and observe how the modification affects the program. We applied this attack to AES implemented with Path ORAM and showed the adversary can identify in which node the secret key is stored.

We discussed applications of ORAM to hard disk encryption in Chapter 6.

By using ORAM or similar scheme, the security of software can be improved. The drawback of applying countermeasures are performance overhead. After the first introduction of ORAM by Goldreich in 1987, many improvements have been proposed and computational cost of ORAM is getting smaller. Yet it may not be realistic to protect entire software as its size increases, so does the performance overhead. In order to achieve both security and performance levels acceptable in practice, there are two options. One is to limit the coverage of protection. There could be a lot of operations performed inside software but only the part of them are critical for its security. By limiting the operations to be protected by ORAM, the computational overhead caused by ORAM can be smaller and acceptable. As the software becomes more and more complicated, it also becomes harder and harder to determine which operations are critical to security and need to be protected. It may be required to develop a systematic tool to analyse the entire software to determine which operations are critical and require ORAM protection. Another approach is the one we took in this thesis, that is, considering a new scheme much lighter than ORAM. The new scheme may not be as secure as ORAM, but the lower security level could be more than enough considering the actual threat. It will be required to analyse the threats and consider how the protection should be.

Other than improving the performance, considering a new application is also important. By using ORAM, the access pattern from ORAM client to ORAM server can be hidden. This characteristic can be utilise in order to improve user's privacy in many applications.

Acknowledgements

First of all, I would like to express my sincere gratitude to Professor Kouichi Sakurai, in Department of Informatics at Kyushu University, who has been my supervisor since the beginning of my study. He provided me with many helpful suggestions, important advices, and constant encouragement during this work. He also gave me many opportunities and advantages for my research activities. I wish to express my sincere appreciation to Professor Koji Inoue in Department of I&E Visionaries and Associate Professor Masaya Yasuda in Institute of Mathematics for Industry at Kyushu University who gave me many helpful suggestions and important advice. I also wish to express my sincere appreciation to Professor Kazuo Ohta and Associate Professor Mitsugu Iwamoto in Department of Informatics at The University of Electro-Communications who made many valuable suggestions and gave constructive advices.

I wish to present my deep gratitude to Dr. Toshiaki Tanaka, the executive vice president at KDDI Research, Inc., for his continuous support, helpful comments. I could challenge anything new under his kind supervising and help. I wish to present my heartfelt appreciation to Dr. Shinsaku Kiyomoto, the senior manager of Information Security Laboratory at KDDI Research, Inc. for his constant encouragement, kind leading, and constructive comments on my research, and giving me a chance to study in external Ph.D. course. His comments provided me with many insightful ideas. I wish to present my heartfelt appreciation to Dr. Yutaka Miyake, the senior manager of Smart Security Laboratory at KDDI Research, Inc. for his constant encouragement, kind leading, and constructive comments on my research. His comments provided me with many insightful ideas. My keen appreciation goes to the members of Security department at

KDDI Research Inc., for kind support and useful comments. Discussions with them were remarkably helpful and fruitful to carry out my research.

Last but by no means least, I would like to thank my family, especially my wife and children, for their supports during my study and writing this thesis.

References

[1] Miklós Ajtai. Oblivious RAMs without cryptographic assumptions. In Leonard J. Schulman, editor, *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing, STOC 2010*, pages 181–190. ACM, 2010.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ Sorting Network. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, STOC 1983*, pages 1–9. ACM, 1983.

[3] Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima, and Toshio Tokita. Camellia: A 128-bit block cipher suitable for multiple platforms - design and analysis. In Douglas R. Stinson and Stafford E. Tavares, editors, *Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000*, volume 2012 of *LNCS*, pages 39–56. Springer, 2001.

[4] Dmitri Asonov and Johann Christoph Freytag. Almost Optimal Private Information Retrieval. In Roger Dingledine and Paul F. Syverson, editors, *Privacy Enhancing Technologies*, volume 2482 of *LNCS*, pages 209–223. Springer, 2002.

[5] Chongxi Bao and Ankur Srivastava. Exploring timing side-channel attacks on path-orams. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST*, pages 68–73. IEEE Computer Society, 2017.

[6] Feng Bao, Robert H. Deng, and Peirong Feng. An Efficient and Practical Scheme

for Privacy Protection in the E-Commerce of Digital Goods. In Dongho Won, editor, *ICISC*, volume 2015 of *LNCS*, pages 162–170. Springer, 2000.

[7] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *LNCS*, pages 1–18. Springer, 2001.

[8] Kenneth E. Batcher. Sorting Networks and Their Applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[9] Amos Beimel and Yoav Stahl. Robust Information-Theoretic Private Information Retrieval. *J. Cryptology*, 20(3):295–321, 2007.

[10] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing Oblivious Access on Cloud Storage: the Gap, the Fallacy, and the New Way Forward. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[11] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious RAM. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[12] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote Oblivious Storage: Making Oblivious RAM Practical. Technical Report MIT-CSAIL-TR-2011-018, Massachusetts Institute of Technology, 2011.

[13] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 175–204. Springer, 2016.

[14] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5):701–716, 2005.

[15] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying Statistically and Computationally Secure ORAMs and OPRAMs. In Yael Kalai and Leonid Reyzin, editors, *TCC*, volume 10678 of *LNCS*, pages 72–107. Springer, 2017.

[16] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious Parallel RAM: Improved Efficiency and Generic Constructions. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 205–234. Springer, 2016.

[17] Benny Chor and Niv Gilboa. Computationally Private Information Retrieval (Extended Abstract). In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, STOC 1997*, pages 304–313. ACM, 1997.

[18] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *FOCS*, pages 41–50. IEEE Computer Society, 1995.

[19] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *J. ACM*, 45(6):965–981, 1998.

[20] Codenomicon Ltd. The Heartbleed Bug. `http://heartbleed.com`, 2014.

[21] Don Coppersmith. Small solutions to polynomial equations, and low exponent rsa vulnerabilities. *J. Cryptology*, 10(4):233–260, 1997.

[22] Dallas Semiconductor Corporation. DS1955 Java-Powered Cryptographic iButton. `http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp111.pdf`, 2000.

[23] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling OS and applications. In Niels Provos,

editor, *3rd USENIX Workshop on Hot Topics in Security, HotSec'08*. USENIX Association, 2008.

[24] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *CARDIS*, volume 1820 of *LNCS*, pages 277–284. Springer, 1998.

[25] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In Yuval Ishai, editor, *TCC*, volume 6597 of *LNCS*, pages 144–163. Springer, 2011.

[26] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, Berkeley, CA, USA, 2012. USENIX Association.

[27] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, 2008.

[28] Xuhua Ding, Yanjiang Yang, and Robert H. Deng. Database access pattern protection without full-shuffles. *IEEE Trans. Information Forensics and Security*, 6(1):189–201, 2011.

[29] Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS*, pages 523–535. ACM, 2017.

[30] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT*, volume 9452 of *LNCS*, pages 360–385. Springer, 2015.

[31] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the Oblivious RAM Timing Channel While Making Information Leakage and Program Efficiency Trade-offs. In *20th IEEE*

International Symposium on High Performance Computer Architecture, HPCA 2014, pages 213–224. IEEE Computer Society, 2014.

[32] Scott R. Fluhrer and David A. McGrew. Statistical analysis of the alleged RC4 keystream generator. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE*, volume 1978 of *LNCS*, pages 19–30. Springer, 2000.

[33] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101, 2011.

[34] Kazuhide Fukushima, Shinsaku Kiyomoto, Toshiaki Tanaka, and Kouichi Sakurai. Analysis of Program Obfuscation Schemes with Variable Encoding Technique. *IEICE Transactions*, 91-A(1):316–329, 2008.

[35] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, volume 7981 of *LNCS*, pages 1–18. Springer, 2013.

[36] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, STOC 1998*, pages 151–160. ACM, 1998.

[37] Ian Goldberg. Percy++ project. SourceForge, http://percy.sourceforge.net/.

[38] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy*, pages 131–148. IEEE Computer Society, 2007.

[39] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 1987*, pages 182–194. ACM, 1987.

[40] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[41] Shafi Goldwasser and Guy N. Rothblum. On Best-Possible Obfuscation. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *LNCS*, pages 194–213. Springer, 2007.

[42] Michael T. Goodrich. Data-Oblivious External-Memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA*, pages 379–388. ACM, 2011.

[43] Michael T. Goodrich and Michael Mitzenmacher. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP (2)*, volume 6756 of *LNCS*, pages 576–587. Springer, 2011.

[44] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with Efficient Worst-Case Access Overhead. In Christian Cachin and Thomas Ristenpart, editors, *CCSW*, pages 95–100. ACM, 2011.

[45] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical Oblivious Storage. In Elisa Bertino and Ravi S. Sandhu, editors, *CODASPY*, pages 13–24. ACM, 2012.

[46] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. In Yuval Rabani, editor, *SODA*, pages 157–167. SIAM, 2012.

[47] Steven Gordon, Xinyi Huang, Atsuko Miyaji, Chunhua Su, Karin Sumongkayothin, and Komwut Wipusitwarakun. Recursive matrix oblivious RAM: an ORAM construction for constrained storage devices. *IEEE Trans. Information Forensics and Security*, 12(12):3024–3038, 2017.

[48] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W.

Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, 2009.

[49] Ryan Henry, Femi G. Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 677–690. ACM, 2011.

[50] Yizhou Huang and Ian Goldberg. Outsourced Private Information Retrieval. In Ahmad-Reza Sadeghi and Sara Foresti, editors, *WPES*, pages 119–130. ACM, 2013.

[51] Anatoly A. Karatsuba and Y. Ofman. Multiplication of multidigit numbes on automata. *Soviet Physics Doklady*, 7:595–596, 1963.

[52] Neal Kobliz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[53] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme. In Dana Randall, editor, *SODA*, pages 143–156. SIAM, 2012.

[54] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *FOCS*, pages 364–373. IEEE Computer Society, 1997.

[55] Jacob R. Lorch, James W. Mickens, Bryan Parno, Mariana Raykova, and Joshua Schiffman. Toward Practical Private Access to Data Centers via Parallel ORAM. *IACR ePrint*, 2012:133, 2012.

[56] Steve Lu and Rafail Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. *IACR ePrint*, 2011:384, 2011.

[57] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive Results and Techniques for Obfuscation. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *LNCS*, pages 20–39. Springer, 2004.

[58] Carsten Maartmann-Moe, Steffen E. Thorkildsen, and André íRnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digit. Investig.*, 6:S132–S140, September 2009.

[59] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy, SP 2015*, pages 341–358. IEEE Computer Society, 2015.

[60] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously Secure Multi-Client ORAM. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017*, volume 10355 of *LNCS*, pages 645–664. Springer, 2017.

[61] Itsik Mantin. Predicting and distinguishing attacks on RC4 keystream generator. In Ronald Cramer, editor, *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 3494 of *LNCS*, pages 491–506. Springer, 2005.

[62] Carlos Aguilar Melchor, Guilhem Castagnos, and Philippe Gaborit. Lattice-based homomorphic encryption of vector spaces. In Frank R. Kschischang and En-Hui Yang, editors, *ISIT*, pages 1858–1862. IEEE, 2008.

[63] Carlos Aguilar Melchor and Philippe Gaborit. A fast private information retrieval protocol. In Frank R. Kschischang and En-Hui Yang, editors, *ISIT*, pages 1848–1852. IEEE, 2008.

[64] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44:519–521, 1985.

[65] Tilo Müller and Michael Spreitzenbarth. Frost - forensic recovery of scrambled telephones. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel,

and Reihaneh Safavi-Naini, editors, *ACNS*, volume 7954 of *LNCS*, pages 373–388. Springer, 2013.

[66] Yuto Nakano, Carlos Cid, Shinsaku Kiyomoto, and Yutaka Miyake. Memory Access Pattern Protection for Resource-Constrained Devices. In Stefan Mangard, editor, *CARDIS*, volume 7771 of *LNCS*, pages 188–202. Springer, 2012.

[67] NIST. Advanced Encryption Standard (AES). FIPS 197 `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf`, 2001.

[68] NIST. Secure hash standard. FIPS 180-3 `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`, 2008.

[69] NIST. Digital Signature Standard (DSS). FIPS 186-4 `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, 2013.

[70] Femi G. Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In George Danezis, editor, *Financial Cryptography*, volume 7035 of *LNCS*, pages 158–172. Springer, 2012.

[71] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In David Pointcheval, editor, *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[72] Kenneth G. Paterson and Mario Strefler. A practical attack against the use of RC4 in the HIVE hidden volume encryption system. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS*, pages 475–482. ACM, 2015.

[73] Benny Pinkas and Tzachy Reinman. Oblivious RAM Revisited. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *LNCS*, pages 502–519. Springer, 2010.

[74] Nicholas Pippenger and Michael J. Fischer. Relations Among Complexity Measures. *J. ACM*, 26(2):361–381, April 1979.

[75] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15*, pages 415–430. USENIX Association, 2015.

[76] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity Verification for Path Oblivious-RAM. In *HPEC*, pages 1–6. IEEE, 2013.

[77] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS*, pages 199–212. ACM, 2009.

[78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.

[79] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In Ross J. Anderson, editor, *Fast Software Encryption, Cambridge Security Workshop, Cambridge, UK, December 9-11, 1993, Proceedings*, volume 809 of *LNCS*, pages 191–204. Springer, 1993.

[80] Claus-Peter Schnorr and Markus Jakobsson. Security of Signed ElGamal Encryption. In Tatsuaki Okamoto, editor, *ASIACRYPT*, volume 1976 of *LNCS*, pages 73–89. Springer, 2000.

[81] R. Seggelmann, M. Tuexen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. `https://tools.ietf.org/html/rfc6520`, 2012.

[82] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM

with $O((\log N)^3)$ Worst-Case Cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *LNCS*, pages 197–214. Springer, 2011.

[83] S. W. Smith and D. Safford. Practical private information retrieval with secure coprocessors. Technical report, IBM Research Division, 2000.

[84] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM SYSTEMS JOURNAL*, 40(3):683–695, 2001.

[85] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy*, pages 253–267. IEEE Computer Society, 2013.

[86] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. *CoRR*, abs/1106.3652, 2011.

[87] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards Practical Oblivious RAM. In *NDSS*. The Internet Society, 2012.

[88] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS*, pages 299–310. ACM, 2013.

[89] Shruti Tople, Yaoqi Jia, and Prateek Saxena. PRO-ORAM: Constant Latency Read-Only Oblivious RAM. *IACR Cryptology ePrint Archive*, 2018:220, 2018.

[90] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.

[91] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In Gail-Joon Ahn, Moti

Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226. ACM, 2014.

[92] Peter Williams and Radu Sion. Usable PIR. In *NDSS*. The Internet Society, 2008.

[93] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 293–304. ACM, 2012.

[94] Peter Williams and Radu Sion. SR-ORAM: Single Round-trip Oblivious RAM. In *ACNS*, industrial track, pages 19–33, 2012.

[95] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: a parallel oblivious file system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS*, pages 977–988. ACM, 2012.

[96] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack. Cryptology ePrint Archive: Report 2014/140, 2014.

[97] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive: Report 2013/448, 2013.

[98] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *IEEE S&P*, pages 218–234. IEEE Computer Society, 2016.

[99] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In Mary Jane Irwin, Wei Zhao, Luciano Lavagno, and Scott A. Mahlke, editors, *CASES*, pages 292–302. ACM, 2004.

[100] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In Shubu Mukherjee and Kathryn S. McKinley, editors, *ASPLOS*, pages 72–84. ACM, 2004.

Index

δ-length ϵ-security, 32

active attack, 54
AES, 13

Chinese Remainder Theorem, 12
circuit ORAM, 20
cold boot attack, 10
computational PIR, 1

dd, 9

GORAM, 20

Hardware-assisted Control Flow Obfuscation, 5
heartbleed, 13
hidden volume, 71
Hierarchical ORAM, 17
History Table, 34
HIVE, 71

Identifying Access Pattern, 59
Identifying Dummy Blocks, 55
information theoretic PIR, 1
Integrity Checking, 69

Lightweight Access Pattern Protection, 26
ltrace, 17

Memory dump, 9

Obfuscation, 45
Oblivious Data Structures, 0
Oblivious Parallel RAM, 0
Open SSL, 13

Parallelism, 68
Path ORAM, 18
PIR-MCORAM, 21
Private Information Retrieval, 1
PRO-ORAM, 21
Process Peeping Tool, 17
ptrace, 9, 12

quadratic non-residue, 2
quadratic residue, 2

Ring ORAM, 20
RSA, 12

Secure Sockets Layer, 10
Square Root ORAM, 16

Transport Layer Security, 10
TrueCrypt, 71

Volume Encryption, 71

List of Publications

Related Papers

Journal Papers

1. Yuto Nakano, Shinsaku Kiyomoto, Yutaka Miyake, and Kouichi Sakurai. Comparison of access pattern protection schemes and proposals for efficient implementation. *IEICE Transactions*, 97-D(10):2576–2585, 2014.

International Conference Papers

1. Yuto Nakano, Carlos Cid, Shinsaku Kiyomoto, and Yutaka Miyake. Memory Access Pattern Protection for Resource-Constrained Devices. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2012.

2. Yuto Nakano, Anirban Basu, Shinsaku Kiyomoto, and Yutaka Miyake. Key extraction attack using statistical analysis of memory dump data. In Javier López, Indrajit Ray, and Bruno Crispo, editors, *Risks and Security of Internet and Systems - 9th International Conference, CRiSIS 2014, Trento, Italy, August 27-29, 2014, Revised Selected Papers*, volume 8924 of *Lecture Notes in Computer Science*, pages 239–246. Springer, 2015.

3. Yuto Nakano, Seira Hidano, Shinsaku Kiyomoto, and Kouichi Sakurai Active

Attack Against Oblivious RAM. In Leonard Barolli, Makoto Takizawa, Tomoya Enokido, Marek R. Ogiela, Lidia Ogiela, and Nadeem Javaid, editors, *32nd IEEE International Conference on Advanced Information Networking and Applications, AINA 2018, Krakow, Poland, May 16-18, 2018*, pages 744–751. IEEE Computer Society, 2018.

Domestic Conference Papers

1. Yuto Nakano, Shinsaku Kiyomoto and Yutaka Miyake. Implementation of Memory Access Pattern Protection [in Japanese]. The 31st Symposium on Cryptography and Information Security, 2014.

2. Yuto Nakano, Shinsaku Kiyomoto and Yutaka Miyake. Access pattern protection considering biased accesses [in Japanese]. IEICE Society Conference, 2014

3. Yuto Nakano, Seira Hidano, Shinsaku Kiyomoto and Koichi Sakurai. Security Evaluation of Memory Access Pattern Protection against Active Adversaries [in Japanese]. Computer Security Symposium, 2017.

Other Papers

Journal Papers

1. Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto, Tsukasa Ishiguro, Yutaka Miyake, Toshiaki Tanaka, and Kouichi Sakurai. Fast implementation of kcipher-2 for software and hardware. *IEICE Transactions*, 97-D(1):43–52, 2014.

2. Takafumi Hibiki, Naofumi Homma, Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto, Yutaka Miyake, and Takafumi Aoki. Chosen-iv correlation power analysis on kcipher-2 hardware and a masking-based countermeasure. *IEICE Transactions*, 97-A(1):157–166, 2014.

International Conference Papers

1. Ryu Watanabe, Yuto Nakano, and Toshiaki Tanaka. Single sign-on techniques with pki-based authentication for mobile phones. In Hamid R. Arabnia, Kevin Daimi, Michael R. Grimaila, George Markowsky, Selim Aissi, Victor A. Clincy, Leonidas Deligiannidis, Donara Gabrielyan, Gevorg Margarov, Ashu M. G. Solo, Craig Valli, and Patricia A. H. Williams, editors, *Proceedings of the 2010 International Conference on Security & Management, SAM 2010, July 12-15, 2010, Las Vegas Nevada, USA, 2 Volumes*, pages 152–156. CSREA Press, 2010.

2. Yuto Nakano, Jun Kurihara, Shinsaku Kiyomoto, and Toshiaki Tanaka. On a construction of stream-cipher-based hash functions. In Sokratis K. Katsikas and Pierangela Samarati, editors, *SECRYPT 2010 - Proceedings of the International Conference on Security and Cryptography, Athens, Greece, July 26-28, 2010, SECRYPT is part of ICETE - The International Joint Conference on e-Business and Telecommunications*, pages 334–343. SciTePress, 2010.

3. Yuto Nakano, Carlos Cid, Kazuhide Fukushima, and Shinsaku Kiyomoto. Analysis of message injection in stream cipher-based hash functions. In Javier López and Gene Tsudik, editors, *Applied Cryptography and Network Security - 9th International Conference, ACNS 2011, Nerja, Spain, June 7-10, 2011. Proceedings*, volume 6715 of *Lecture Notes in Computer Science*, pages 498–513, 2011.

4. Shinsaku Kiyomoto, Matthew Henricksen, Wun-She Yap, Yuto Nakano, and Kazuhide Fukushima. MASHA - low cost authentication with a new stream cipher. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2011.

5. Yuto Nakano, Jun Kurihara, Shinsaku Kiyomoto, and Toshiaki Tanaka. Stream cipher-based hash function and its security. In Mohammad S. Obaidat, George A. Tsihrintzis, and Joaquim Filipe, editors, *e-Business and Telecommunications - 7th*

International Joint Conference, ICETE 2010, Athens, Greece, July 26-28, 2010, Revised Selected Papers, volume 222 of *Communications in Computer and Information Science*, pages 188–202. Springer, 2012.

6. Takafumi Hibiki, Naofumi Homma, Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto, Yutaka Miyake, and Takafumi Aoki. Chosen-iv correlation power analysis on kcipher-2 and a countermeasure. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design - 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers*, volume 7864 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2013.

7. Shinsaku Kiyomoto, Andre Rein, Yuto Nakano, Carsten Rudolph, and Yutaka Miyake. LMM - A common component for software license management on cloud. In Pierangela Samarati, editor, *SECRYPT 2013 - Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29-31 July, 2013*, pages 284–295. SciTePress, 2013.

8. Yuto Nakano, Youssef Souissi, Robert Nguyen, Laurent Sauvage, Jean-Luc Danger, Sylvain Guilley, Shinsaku Kiyomoto, and Yutaka Miyake. A pre-processing composition for secret key recovery on android smartphone. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things - 8th IFIP WG 11.2 International Workshop, WISTP 2014, Heraklion, Crete, Greece, June 30 - July 2, 2014. Proceedings*, volume 8501 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2014.

9. Juan Camilo Corena, Anirban Basu, Yuto Nakano, Shinsaku Kiyomoto, and Yutaka Miyake. A multiple-server efficient reusable proof of data possesion from private information retrieval techniques. In Mohammad S. Obaidat, Andreas Holzinger, and Pierangela Samarati, editors, *SECRYPT 2014 - Proceedings of the 11th International Conference on Security and Cryptography, Vienna, Austria, 28-30 August, 2014*, pages 307–314. SciTePress, 2014.

10. Juan Camilo Corena, Anirban Basu, Yuto Nakano, Shinsaku Kiyomoto, and Yutaka Miyake. Data storage on the cloud under user control. In *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*, pages 739–742. IEEE Computer Society, 2014.

11. Kazuhide Fukushima, Youssef Souissi, Seira Hidano, Robert Nguyen, Jean-Luc Danger, Sylvain Guilley, Yuto Nakano, Shinsaku Kiyomoto, and Laurent Sauvage. Delay PUF assessment method based on side-channel and modeling analyzes: The final piece of all-in-one assessment methodology. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 201–207. IEEE, 2016.

12. Seira Hidano, Yuto Nakano, and Shinsaku Kiyomoto. An evaluation framework for fastest oblivious RAM. In Víctor Méndez Muñoz, Gary Wills, Robert John Walters, Farshad Firouzi, and Victor Chang, editors, *Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security, IoTBDS 2018, Funchal, Madeira, Portugal, March 19-21, 2018.*, pages 114–122. SciTePress, 2018.

Domestic Conference

1. Yuto Naknao, Hidenori Kuwakado and Masakatu Morii. Inspection of sufficient conditions of MD5 [in Japanese]. IEICE Tech. Rep., vol. 106, no. 175, ISEC2006-27, pp. 133-139, July 2006.

2. Yuto Naknao, Hidenori Kuwakado and Masakatu Morii. Reexamination of Collision Conditions of MD5 [in Japanese]. The 29th Symposium on Information Theory and its Applications (SITA2006), pp.109–112, 2006

3. Yuto Nakano, Hidenori Kuwakado, and Masakatu Morii. Redundancy of the wang-yu sufficient conditions. *IACR Cryptology ePrint Archive*, 2006:406, 2006.

4. Yuto Nakano, Jun Kurihara, Shinsaku Kiyomoto and Toshiaki Tanaka. A Study on Stream Cipher Based Hash Functions [in Japanese]. IEICE Tech. Rep., vol. 109, no. 113, ISEC2009-29, pp. 153-159, July 2009.

5. Yuto Nakano, Jun Kurihara, Shinsaku Kiyomoto and Toshiaki Tanaka. A Construction of Stream-cipher-based Hash Function. The 27th Symposium on Cryptography and Information Security, 2010.

6. Yuto Nakano, Jun Kurihara, Shinsaku Kiyomoto and Toshiaki Tanaka. A Message Injection in SCH [in Japanese]. IEICE General Conference, 2010.

7. Matt Henricksen, Wun-she Yap, Chee-hoo Yian, Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. Side-channel Analysis and Countermeasures of K2 [in Japanese]. IEICE Tech. Rep., vol. 110, no. 44, ISEC2010-2, pp. 5–10, May 2010.

8. Yuto Nakano, Shinsaku Kiyomoto and Toshiaki Tanaka. Comparison of Message Injection Functions for SCH [in Japanese]. IEICE Society Conference, 2010.

9. Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto and Toshiaki Tanaka. On the Hardware Implementation of K2 Stream Cipher [in Japanese]. IEICE General Conference, 2011

10. Takafumi Hibiki, Kazuya Saito, Naofumi Homma, Takafumi Aoki, Yuto Nakano, Kazuhide Fukushima, Shinsaku Kiyomoto and Yutaka Miyake. Correlation Power Analysis on KCipher-2 and Its Countermeasures [in Japanese]. The 29th Symposium on Cryptography and Information Security, 2012

11. Takafumi Hibiki. A countermeasure against power analysis attacks on KCipher-2 and its evaluation [in Japanese]. Computer Security Symposium, 2012.

12. Hajime Uno, Sho Endo, Naofumi Homma, Takafumi Aoki, Yuto Nakano, Shinsaku Kiyomoto and Yutaka Miyake. A Study on Correlation Electromagnetic Analysis against KCipher-2 Implemented on a Microcontroller for ZigBee Device [in Japanese]. IEICE Tech. Rep., vol. 113, no. 484, ISEC2013-88, pp. 35–40, March 2014.

13. Hajime Uno, Sho Endo, Naofumi Homma, Takafumi Aoki, Yuto Nakano, Shinsaku Kiyomoto and Yutaka Miyake. Implementation and Evaluation of KCipher-2 Software for Smart Cards [in Japanese]. Computer Security Symposium, 2014.

14. Yuto Nakano, Shinsaku Kiyomoto and Yutaka Miyake. A Key Protection Mechanism for Cryptographic Applications [in Japanese]. IEICE General Conference, 2015.