M.Sc. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Cache Timing Attacks on Public Key Encryption

Mohamed Heikal (s155309)

Kongens Lyngby 2017

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Matematiktorvet
Building 303B
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

The rise of cloud computing has made it a lot easier for attackers to be able to run code on the same processors as their target. This has made many attacks more viable. This thesis discusses a cache timing attack targeting the LibTomMath library. LibTomMath is a mathematical library for computations using large integers. The library is used in some cryptographic libraries such the commercial solution WolfCrypt.

The attack mainly focuses on the modular exponentiation function of LibTomMath which is a major part of RSA implementations. The aim of the attack is to use cache timing in order to extract the long term private key used by the server for encrypting communications. Recovering the private key, gives the attacker access to past and future communications secured using this key, which usually has a lifespan of at least one year. The attack only requires that it shares a processor with the victim and works even if the attack process and the victim process are running on different Virtual Machines.

The thesis includes a description of the RSA cipher as well as the various optimizations that are used in a lot of cryptographic libraries. Next, it describes how to use cache timing to exploit some of those optimizations in order to gain information about the secret exponent based on the memory access patterns of the target code.

Finally, it discusses the limitations of the attack as well as how cloud services providers, cryptographic library developers as well as processor manufacturers may be able to mitigate this class of attacks.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a M.Sc. degree in Computer Science and Engineering.

Kongens Lyngby, June 22, 2017

Mohamed Heikal (s155309)

Contents

CHAPTER 1

Introduction

1.1 Infrastructure as a Service

Infrastructure as a service (IaaS), is one of the most popular ways small to mid-range companies handle their computing needs. The main technique that IAAS providers rely on is virtualization. Virtualization allows sharing physical resources such as processing power, memory and storage, among multiple customers and thus optimizing the use of those resources. IaaS providers offer cheaper and more easily scalable solutions to companies and they use virtualization to maximize the usage of their physical machines.

The largest companies in the field, like Amazon with AWS (Amazon Web Services) and Google with GCP (Google Cloud Platform) are all offering either full virtualization solutions or similar services such as containers. Those solutions create an illusion of isolation, where each inhabitant/user of physical resources, is in their own sandbox unable to access or directly interact with other inhabitants on the same physical machine.

This is true to some extent, for example, without an exploit to allow you to break out of the virtualized sandbox, you are unable to directly access shared resources that are currently allocated to other inhabitants. However, complex usable abstractions always leak more information than they intend. The main leaks in the abstraction, that this thesis focuses on is the memory and cache.

The speed at which code can load and execute instructions as well as data from memory/cache has a very direct relation to the performance. Even if the code and processor are fast, if it takes a long time to store/load data to and from memory then the overall execution would be slow. This makes it so there is very little virtualization on the memory/cache front. Therefore, since memory/cache accesses are direct, the time it takes to fetch data from memory is affected by the contents of the real memory (that is shared by all inhabitants). This fact allows one inhabitant to infer memory utilization patterns by the other inhabitants across the sandbox by timing memory accesses, even within the sandbox.

1.2 Cryptography

Cryptography is the method by which unencrypted data (labeled as plaintext) is converted to encrypted data (labeled as ciphertext) for transport across insecure mediums (eg: the internet) and then finally converted back to the original plaintext once more. The main aim of cryptography is, even if the ciphertext is intercepted, it cannot be decrypted back to plaintext by the interceptor.

$$C = E_k(P)$$

$$P = D_{k'}(C)$$

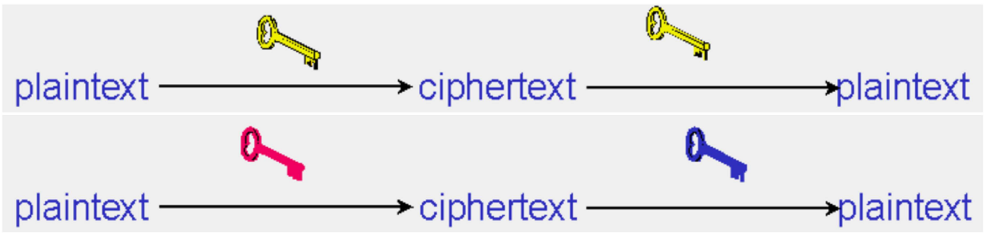
C and P are the ciphertext and plaintext respectively. E and D are the encryption and decryption functions respectively. The keys used in the functions k and k' can be the same key or can be different, depending on the type of encryption used.

Encryption has two main classifications, Secret Key Encryption (AKA Symmetric Encryption) and Public Key Encryption (AKA Asymmetric Encryption). Symmetric encryption revolves around a secret key that is shared by all those who are authorized. It is used during both encryption and decryption and it requires the recipient to know the key to be able to decrypt the message.

Asymmetric Encryption, on the other hand, uses two keys a public and a private key. If one key is used to encrypt the message then only the other key can be used to decrypt it. The public key is called so, because it is usually publicly available. The public key is used to encrypt private messages that the owner of the private key can then decrypt and be sure of its confidentiality.

Asymmetric encryption can also be used for signatures. This is where the private key is used to encrypt a message which can then be decrypted by the public key. It is used to authenticate that the message came from the owner of the private key rather than for confidentiality of the message. This is an major aspect of SSL/TLS (Secure Socket Layer / Transport Layer Security), which is what used to encrypt internet traffic. Asymmetric encryption is used in the SSL/TLS handshake to verify the public key belonging to the website/webserver as well as to send the first few messages of the handshake confidentially to the web server.

Figure 1.1: Top: Symmetric encryption using the same key, Bottom: Asymmetric encryption using different keys [Gar10].



1.3 Issues with cryptographic implementations

Encryption standards are usually defined in mathematical notation. They deal with the problem of encryption and security in the abstract sense. Mathematical equations showing the difficulty of solving the decryption problem without the key is usually how encryption algorithms are shown to work. The problem statement usually assumes that the encryption process is done on a private black box where no information goes out unless stated by the algorithm.

The problem arises in the next step of the process which is implementing the defined algorithm in a real environment, i.e. with an Operating System (OS) with a certain Instruction Set Architecture (ISA) on an certain processor. The actual implementation step can expose certain side channels not taken into account by the design of the algorithm. For example, the time it takes to do certain steps in the algorithm may depend on some secret information, therefore, being able to time the algorithm may allow an attacker to gain secret information.

The most promising attacks on modern encryption algorithms are the ones based on side channels. The reason is, it is usually very hard to reason about the information leaked by running the algorithm on a certain hardware. The aim of optimizing the implementation usually goes against leaking as little information as possible.

The main focus point of the thesis is on cache side channel. If an algorithm implementation accesses certain locations in memory based on secret key for example. Then by monitoring the cache for those locations you are able to figure out if and when they are accessed and then be able to infer information about the key or reduce the key space enough to be able to bruteforce the rest of it within a reasonable time-frame.

1.4 Encryption in the real world

Encryption is an important facet in the internet as it is today. During the rise of internet commerce and internet banking, encrypting sensitive information such as logins and credit card details as they traveled over the open internet was very important. Unfortunately encryption is expensive on the server side with having to perform expensive mathematical operations every new request being a major drawback to having full encryption for all traffic. So, for most none extremely sensitive traffic, basically any traffic other than logins and payment data, was left unencrypted.

With the rise of open hotspots allowing nefarious attackers to more easily orchestrate Man in the Middle (MitM) Attacks [But10], it is becoming easier for private data communicated over the internet to be intercepted by malicious entities. This lead to more proponents advocating for full encryption for websites [BI14] with new tools available to deploy HTTPS (Hyper Text Transfer Protocol Secure) encryption more easily [Aas14].

HTTPS/TLS (Transport Layer Security) used in encrypting internet traffic relies on both asymmetric (RSA) and symmetric cryptography (AES). RSA and other

asymmetric cryptography can allow both parties to exchange encrypted information without having a preshared key. This is important because a new user of a website does not need to have a key to encrypt the communication. However, asymmetric ciphers are expensive and thus cannot be used for the entire communication. So at first during the TLS handshake, asymmetric ciphers are used to exchange a preshared key at the start of the session then afterwards symmetric ciphers like AES are then used to encrypt the rest of the communication.

1.5 Prior work

There have been previous attacks on encryption implementations in the past. Some targeted the browser implementation of the TLS protocol [Jee13]. Others targeted the implementation on the server side. This thesis focuses on server side implementations and how to extract private keys from running servers.

Some previous attacks targeted the hardware directly by measuring (using lab equipment) emanations such as power consumption [KJJ99; Koc+11], electromagnetic radiation[QS01] or sound [GST14]. Other attacks focused on software methods of key extraction not requiring external hardware. Those either timed the cryptographic processes themselves [BT11; BB05] or timed the cache.

Of those that used cache timing, some targeted the L3 cache (see section 4.4 for cache levels) [Liu+15; Inc+16; IES15a]. This allowed the attacks to be usable across VMs. The L3 cache is too large to probe enirely with enough frequency thus making attacks on the L3 cache more complex because they need to reduce the address space first. However, attacks targetting the L3 cache are able to work across VMs. There are some attacks that target the L1 cache [YGH16], which can even bypass some of the protections that are provided by some libraries to prevent cache timing attacks.

1.6 Threat model

The threat model targeted by this paper is based on shared physical CPU and cache by both the attacker and the victim. The attacker and victim can be assumed to be in separate virtual sandboxes. Those can be using virtual machines or through modern alternatives such as containers. Any virtualization solution is applicable as long as the physical cache is shared and not emulated. Most IaaS providers share the physical hardware across customers unless the customer purchases the entire hardware server.

The virtual machine is assumed to be running on memory allocated using huge pages (see section 4.2). This is not a difficult condition to satisfy because this is the most popular configuration for VMs provided by IaaS providers. Using huge pages improves performance by more efficiently using limited resources.

The attacker and the victim are not assumed to be running on the same core. However the attacker is assumed to have root permissions within their own sandbox.

1.7 Our contributions

The main focus of this thesis is to implement a cache timing attack on the LibTomMath library. The goal is to show that even with the isolation provided by virtualization, cache leakages can still reveal security critical confidential data. LibTomMath provides functions that work with huge numbers, the type used in some cryptographic algorithms including RSA. LibTomMath is used by a few cryptographic libraries like WolfSSL which is a commercial cryptographic library.

The target of the attack is the modular exponentiation implementation provided by LibTomMath. Modular exponentiation is a major part of the cipher RSA. RSA is used most commonly in SSL/TLS and thus RSA encryptions are expected to be performed repeatedly on websites that are secured via SSL/TLS. The attack monitors the cache during the encryptions and uses the timing data in order to extract the secret key used.

Chapter 2 describes the RSA cipher in more detail, chapter 3 discusses how operations on the large integers used in RSA are implemented and optimized. Chapter 4 describes the memory and cache architecture, then chapter 5 describes how to implement a generic cache timing attack. Finally chapters 6, 7 discuss the attack on LibTomMath and its applicability on other libraries and platforms as well as some mitigation techniques.

CHAPTER 2

RSA

RSA is a popular asymmetric key cipher that allows two parties to communicate securely without requiring a preshared key [RSA78]. RSA plays a major role during the handshake of TLS sessions before agreeing on a shared key for the rest of the session. As an asymmetric cipher, RSA has a private key held by one of the parties and a public key that is distributed to other parties.

2.1 Using asymmetric cryptography

Asymmetric ciphers including RSA have two keys, a private and a public key. Given a plaintext message M, encrypting M using the public key (K) gives a ciphertext.

$$C = E_K(M) \tag{2.1}$$

To get the plaintext message back from the ciphertext C the data is decrypted using the private key (K^{-1}).

$$M = D_{K^{-1}}(E_K(M)) \tag{2.2}$$

The main premise of asymmetric cryptography is that releasing the public key K does not compromise the security of the encryption and there is no easy way to compute function D' using the public key K such that:

$$M = D'_K(C) \tag{2.3}$$

2.2 Basics of RSA

RSA is based on modular exponentiation. In RSA the public key consists of two integers $< N, e >$. N is the modulus and e is the public exponent. To create the public key component N two large random prime numbers P and Q are generated, then $N = PQ$. The exponent e does not have many restrictions for the cipher to be secure. It just has to fulfill the following requirement:

$$gcd(e, (P-1)(Q-1)) = 1 \tag{2.4}$$

It is usually chosen as 3 or 65537 since they are small numbers and thus make encryption faster. To encrypt a message m in to the cipher text c, you use the following formula:

$$c = m^e \bmod N \tag{2.5}$$

The private key is made up of three other integers $< P, Q, d >$. P and Q are mentioned above and d is the private exponent. d is chosen so that it satisfies the following equation:

$$e \cdot d \equiv 1 \bmod (P-1)(Q-1) \tag{2.6}$$

To decrypt the ciphertext back into plaintext, the ciphertext c is exponentiated to the power of the private exponent d modulo N.

$$m = c^d \bmod N \tag{2.7}$$

Usually the public key is used to encrypt the data and the private key decrypts ensuring that only the owner of the private key can read the message, which is how it was used above. However another way of using RSA is to create signatures. If the private key is used to sign the message, the public key can then be used to check the signature as so:

$$c' = \bar{m}^d \bmod N \tag{2.8}$$

$$\bar{m} = c'^e \bmod N \tag{2.9}$$

Using the private exponent first then the public one would still get back the original plaintext. However, the intermediate product after "encrypting" with the private key can be read by anyone with the public key. This order is not used to protect the secrecy of the data but rather to ascertain the identity of the sender of the data as well as its authenticity. Usually, when using signatures to verify authenticity, the hash of the message ($\bar{m}$) is used rather than the actual message (m).

2.3 Correctness of RSA

The reason exponentiation with the private exponent cancels out the exponentiation by the public exponent and viceversa starts with this identity by Euler and Fermat [HW79]:

$$m^{\phi(n)} \equiv 1 \bmod n \tag{2.10}$$

The function ϕ is defined on an integer N as the number of integers less than N that are coprime with N. If N is a product of two prime numbers P and O as is the N used in RSA then $\phi(N) = (P-1)(Q-1)$.

Rewriting equation (2.6) to use ϕ:

$$e \cdot d \equiv 1 \bmod \phi(N) \tag{2.11}$$

This means that:

$$ed = 1 + k\phi(N) \tag{2.12}$$

Let us consider the case of a message encrypted by the public key then decrypted by the private key. This means that it is raised to the power of ed:

$$\begin{aligned} m &= (m^e)^d = m^{ed} \bmod N \\ &= m^{1+k\phi(N)} \bmod N \\ &= m \cdot m^{k\phi(N)} \bmod N \\ &= m \cdot (m^{\phi(N)})^k \bmod N \end{aligned} \tag{2.13}$$

Using equation (2.10) this can be reduced to:

$$m \cdot (m^{\phi(N)})^k = m \cdot 1^k = m \bmod N \tag{2.14}$$

Equation (2.14) shows that encryption and decryption are inverses of each other and doing one after the other results in the original message.

2.4 Security of RSA

The security of RSA is based on the difficulty of factoring large products of primes. Both P and Q are secret and are not really needed after the calculation of the private exponent d. Calculating $\phi(N)$ is trivial with access to the prime factors P and Q, however it is not so easy with just the access to N.

There is no formal proof showing that RSA is difficult, however, the original RSA paper [RSA78] shows that the private exponent d can be used to factor N efficiently. This means that breaking RSA is at least as hard as factoring large products of primes. The best algorithm currently known is the "number field sieve factoring algorithm" [Lan01] which still has a sub-exponential time complexity to guarantee security for RSA.

In order for RSA to be secure, the primes P and Q have to be large. Currently for secure RSA, it should be implemented with 2048 bit moduli at least. This leaves a margin of error, since the 768 bit RSA modulus has already been broken [Kle+10], potentially leaving the 1024 bit moduli in jeopardy.

2.5 RSA in TLS

RSA is sometimes used during the handshake part of TLS (Transport Layer Security). The end result of TLS is to have a preshared secret key between the server and the

user's browser for use in encrypting communication. This starts by a handshake where the server and the browser exchange credentials as well as work together to create the shared key.

The TLS handshake (shown in fig 2.1) starts with the client/browser sending the "client hello" message. The message usually lists the capabilities of the browser, with regards to the cryptographic algorithms it supports. The server then responds with a choice of one of the supported algorithms as well as the server certificate.

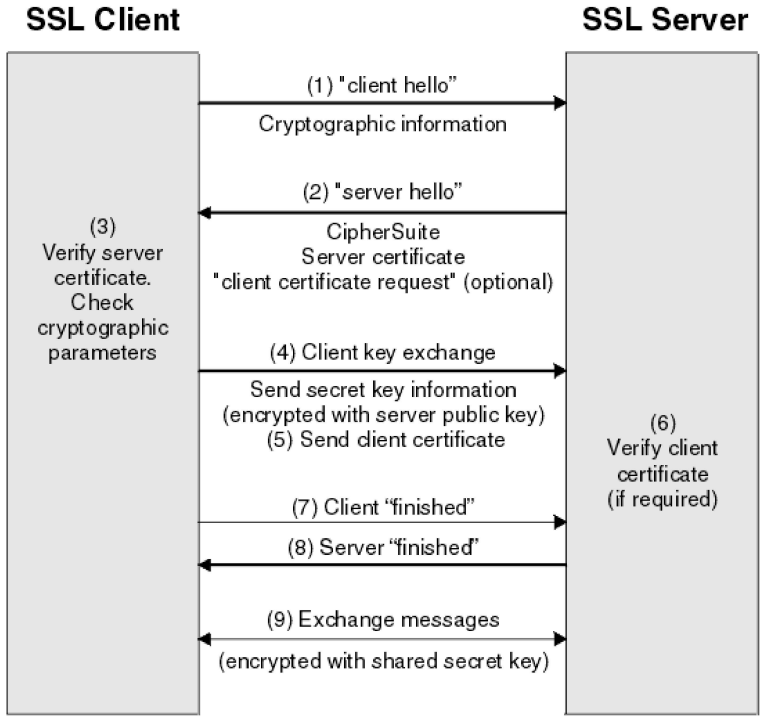
The server may also request a client certificate to authenticate the user, however that is optional and rarely used. The client replies with its portion of the shared key material encrypted using the server's public key. Finally they exchange "finished" messages signed using the shared key they came up with (to make sure that both arrived at the same key). This ends the handshake portion and the rest of the communication is encrypted using the newly created shared key.

Certificates contain an RSA public key as well as the domain name of the website. Trusted third parties, ie. Certificate Authorities (CA), sign the certificate by hashing the certificate contents and encrypting the hash using their private key to arrive at a signature. The client/browser can verify the signature by calculating the hash of the certificate contents and decrypting the signature comparing it to the calculated hash. If they match then the certificate is authentic.

Since the signature is created by encrypting the hash with the private key of the CA, it can be verified using the public key of the CA (which is assumed to be preloaded in the browser/OS of the user).

The server certificate is static, ie. it does not change per request. This means that every request uses the same certificate with the same private/public key pair. Thus, many encryptions with the same key are performed every day/hour. Furthermore, discovering said private key can lead to an attacker being able to unencrypt secure connections.

Figure 2.1: Overview of the SSL or TLS handshake [Cor14].



CHAPTER 3

Implementing RSA

The security of RSA is based on the fact that the numbers are very large and thus factorization is expensive. The problem with having such large numbers is that computations using those numbers are more expensive. RSA deals with numbers more than a 1000 bits long. These numbers will not fit on even the largest registers on a CPU. This means that certain optimizations and implementation decisions have to be considered when implementing RSA in actual code. This chapter discusses what those optimizations are as well as how these optimizations may lead to vulnerabilities that are the subject of this thesis.

3.1 Square and multiply

To naively implement modular exponentiation, you would raise to the power of the exponent first then reduce modulo the modulus. However the exponentiation part would become very expensive and if the exponent is big, which it is in RSA, you will end up with extremely huge numbers, many orders of magnitude larger than the modulus and the required result.

The simplest way of solving this problem is to solve the modular exponentiation problem sequentially and reduce regularly keeping the intermediate steps within an order of magnitude of the modulus.

The square and multiply method starts with a running result value 1. It then goes over each bit of the exponent from right to left and squares the current result. If the current bit of the exponent is a 1 then it multiplies the base into the result before the squaring. Algorithm 1 shows this in detail.

3.2 Multiple precision integers

As is stated, the numbers used in RSA are very large and they can not fit on a single register. In order to compute with and store such large numbers, they need to be split off into multiple digits. This complicates the computation since you can no longer just depend on the hardware to perform the multiplication or squaring but all those operations have to be implemented in software. This is done using multiple precision

Algorithm 1 Square multiply exponentiation

Input: base b, modulus m, N-bit exponent e represented as N bits e_i $\{e_N, e_{N-1}, ..., e_1\}$.
Output: $b^e \bmod m$

```
1: r ← 1
2: b ← b mod m
3: for i = 1 to N do
4:     if e_i = 1 then
5:         r ← r * b mod m
6:     end if
7:     r ← r * r mod m
8: end for
9: return r
```

integers that are split into multiple digits with a base b. The base used is usually a multiple of 256 since each digit is stored in one or more bytes.

Multiplication is one of the required operations used in the modular exponentiation since it can also be used instead of squaring. Algorithm 2 shows how to do multiplication between multiple precision integers. The algorithm is the same as the pen and paper multiplication method taught in grade school.

Algorithm 2 Multiple precision integer multiplication

Input: integers x $(x_n x_{n-1}...x_0)_b$, y $(y_t y_{t-1}...y_0)_b$ having $n+1$ and $t+1$ digits, base b, respectively.
Output: the product $x \cdot y = (w_{n+t+1}...w_1 w_0)_b$

```
1: for i = 0 to n + t + 1 do
2:     w_i ← 0
3: end for
4: for i = 0 to t do
5:     c ← 0
6:     for j = 0 to n do
7:         (uv)_b ← w_{i+j} + x_i · y_j + c
8:         w_{i+j} ← v
9:         c ← u
10:    end for
11:    w_{i+n+1} ← u
12: end for
13: return (w_{n+t+1}...w_1 w_0)
```

3.3 Sliding window exponentiation

In RSA, modular exponentiation is the main operation used in encryption and decryption. Thus, this is the main target for optimization from the algorithmic perspective. One way of optimizing the modular exponentiation is using a sliding window representation of the exponent.

The sliding window representation of an exponent e is a sequence of windows w_i each having length $L(w_i)$. Each window is either all 0s or starts with a 1 and ends with a 1. With a window size S each non-zero window satisfies $1 \leq L(w_i) \leq S$, thus has an odd value between 1 and $2^S - 1$.

As shown in Algorithm 3, the sliding window exponentiation technique first calculates the multipliers for each possible window and stores them in array g then uses the multipliers in g to calculate the actual result of the exponentiation.

Algorithm 3 Sliding window exponentiation

Input: Window size S, base b, modulo m, N-bit exponent e represented as n windows w_i of length $L(w_i)$ each.
Output: $b^e \bmod m$

1: //Precomputation
2: $g[0] \leftarrow b \bmod m$
3: $s \leftarrow \text{MULT}(g[0], g[0]) \bmod m$
4: **for** $j = 1$ to 2^{S-1} **do**
5: $\quad g[j] \leftarrow \text{MULT}\ (g[j-1], s) \bmod m$
6: **end for**//Exponentiation
7: $r \leftarrow 1$
8: **for** $i = n$ downto 1 **do**
9: $\quad$ **for** $j = 1$ to $L(w_i)$ **do**
10: $\quad\quad r \leftarrow \text{MULT}(r, r) \bmod m$
11: $\quad$ **end for**
12: $\quad$ **if** $w_i \neq 0$ **then**
13: $\quad\quad r \leftarrow \text{MULT}(r, g[(w_i - 1)/2]) \bmod m$
14: $\quad$ **end if**
15: **end for**
16: **return** r

For example let us take an exponent $e = 11749 = (10110111100101)_2$ and window size 3. Spliting it into windows would generate $(101, 101, 111, 00, 101)$. We then only need to multiply 4 times corresponding to the none-zero windows.

3.4 Montgomery modular exponentiation

Montgomery reduction is a method for modular multiplication that does not require the expensive classical modular reduction step. $TR^{-1} \bmod m$ is called a Montgomery

reduction of T modulo m with respect to R. If are R is chosen correctly, Montgomery reduction can be computed efficiently.

For Montgomery reduction to work, R and T have to be integers such that $R > m$, $gcd(R, m) = 1$ and $0 \leq T < mR$. For Montgomery reduction to be computed efficiently R is chosen to be b^n where b is the base of digits of the modulus m and n is the number of digits of m. This guarantees that $R > m$, and for practical purposes (i.e. RSA implementations), b is a power of 2 and m is odd thus $gcd(R, m) = 1$.

Let x and y be integers such that $0 \leq x, y < m$. Let $\tilde{x} = xR \bmod m$ and $\tilde{y} = yR \bmod m$, called the Montgomery forms of x and y. The Montgomery reduction of $\tilde{x}\tilde{y}R^{-1} \bmod m = xyR \bmod m$.

If R is chosen as b^n where b is a power of 2 then multiplication by R is a shift left and division is a shift right. If multiple modular multiplications with the same modulus are to be expected, as is the case in modular exponentiation, the entire algorithm can be changed to use Montgomery reduction by converting the inputs of the multiplication to Montgomery form, doing the multiplication and then using Montgomery reduction. [Men+96] (14.29) implies that if R is chosen as b^n then Montgomery reduction can be computed more efficiently than classical modular reduction.

The sliding window exponentiation in Algorithm 3, in practical implementations, is usually improved by converting the multipliers to Montgomery form, then using Montgomery multiplication instead of the `MULT` function and performing Montgomery reduction before returning the result.

3.5 Attacking optimized implementations

Optimizations such as the one shown in section 3.3 improve the run time of the cryptographic function, however, they may indirectly create new attack surfaces. As shown in the computation part, the access patterns of the multipliers depend on the values of the bits in the secret RSA exponent thus leak information to an observer.

If an attack process monitors the relevant cache lines, it can find the access times of the multipliers relative to each other and to the start of the exponentiation. This sequence of multipliers can then be used to get the secret exponent since each multiplier has an index that is the value of a window inside the exponent.

The index of a certain multiplier can be found by looking at the access patterns of the multiplier at the start of the exponentiation. The multipliers are initialized in order, which points to their index and the value of the window they represent. Even if in the implementation of the library, they are not initialized in ascending order but in a more complicated one to obfuscate their index, the code of the library will still show the actual order of initialization and thus an attacker can still get the index from monitoring cache accesses.

CHAPTER 4

Memory Architecture and Cache Timing Techniques

4.1 Memory paging and virtual addresses

Memory is a contested resource when multiple programs are running at the same time. The OS tries to alleviate some of this contention by running each process in its own virtual address space. To any running program, the entirety of the virtual address space is free for its own use and it does not have to worry about interfering with other running programs.

The virtual address space is divided into pages and the physical address space is divided into frames that fit those pages. Pages are blocks of address ranges of a specific size specified by the OS. For each program, the OS maps the used pages inside the virtual address space of the program to frames of the same size in the physical address space. Virtual memory is transparent to a normal program and any memory accesses by the program are translated on the fly to their real physical address space counter part before reads and writes are passed to the main memory.

The virtual to physical mapping, maps entire pages from the virtual address space to the physical one. This means that within a virtual page the offset of a certain line is the same as its offset within the counterpart physical frame. So for an N-bit address and a page size S, the least significant $log_2(S)$ bits are the offset within the page while the most significant $N - log_2(S)$ bits constitute the page address/frame number that is replaced when the translation occurs. The most common page size is 4KiB. This means that the least significant $log_2(4096) = 12$ bits are used as an offset within the page.

4.2 Hugepages

Memory mappings of virtual pages to frame numbers have to be stored and looked up frequently. Basically every time a memory or cache access occurs (outside the L1 cache which uses virtual addresses, see section 4.4) this translation has to be looked up. To speed this up, the most recent mappings are stored in a TLB, or a Translation Lookaside Buffer. The TLB allows fast translations for the most recently used pages. However, the TLB has a limited size. If there is a large number of pages, then many of them may not be stored in the TLB at any given time.

If there is a TLB miss, the memory mappings have to be read from main memory. The page table in the main memory has a layered hierarchy. In x86-64 there are 4 layers of page tables [AMD]. A virtual address in a standard 4KiB page has 12 bits page offset. The remainder of the 48-bit virtual address is divided into 4 9-bit indices used to select entries from the 4 levels of the page table.

Each entry in one level of the page table points to the next level of the page table or to the actual physical page in case of the last layer of the page table. This process to retrieve the memory mapping in case of a TLB miss is called the page walk. The page walk is expensive since it requires multiple consecutive accesses to the main memory.

The TLB is a scarce and precious resource. With the rise of applications requiring huge amounts of memory, optimizing the use of the TLB is important. This leads to newer kernels offering the option of larger page sizes. Instead of the standard 4KiB page, kernels with huge pages support offer page sizes of 2/4 MiB and even 1GiB. Larger pages use less of the TLB for the same amount of physical memory mapped. This makes huge pages very beneficial in virtualized environments. Since virtual machines require large memory allocations, by using huge pages, the TLB is able to fit more of those mappings and reduce the latencies that occur in case of TLB misses.

4.3 Cache overview

The processor is currently the fastest part of pipeline in modern computing. However, the real world performance is usually bounded by the speed of the memory, which provides the instructions and the data that the processor needs. With this being the case, speeding up the processor without an equivalent speedup to the memory does very little to the overall performance of the machine and applications that run on it.

While high speed memory does exist, it is a lot more expensive if all available memory was high speed. That would make the computer very expensive. The way the speed issue is solved while keeping computer prices low, is by using high speed caches above the main memory which is slower but more reasonably priced.

Whenever the processor requests access to a certain memory location, the cache is checked first, if it exists (a cache hit) then the cache responds appropriately otherwise (a cache miss) the main memory responds and the data is loaded into the cache for

next time. This is usually transparent to the software layer, except sometimes data loads faster on a cache hit.

4.4 Multi-level cache

In modern processors, the cache is usually divided into multiple levels (usually 3) above the main memory, each being smaller and faster than the next with the main memory being the slowest. The fastest and smallest cache level is L1, usually divided into instruction vs data caches. These are exclusive to each processor core and are not shared outside. The Last Level Cache (LLC), is the largest and slowest of the cache levels and is shared among all the processor cores.

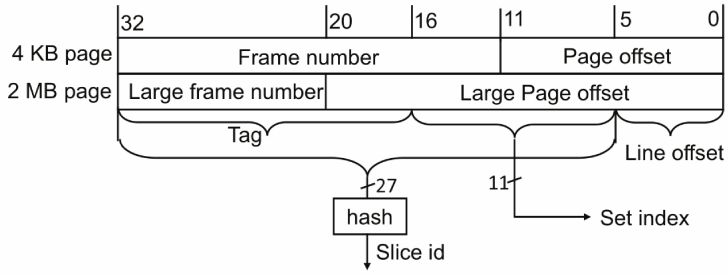
The cache hit/miss check is done for each cache level consecutively and only if all of them miss, is the main memory read directly. Each cache level is 2-3x faster than the next level while the LLC can be as much as 10x faster than main memory.

In modern Intel machines, cache levels are inclusive. This means all the memory locations in L1 are present in L2 and L3. Som if anything is not in the LLC, then, it cannot be present in any higher cache level.

4.5 Cache structure and associativity

The smallest unit of data that is loaded into or ejected out of the cache is a cache line, having a size B bytes. Most caches are divided into an S number of sets with each set being able to fit a W number of lines. W is called the ways of associativity of a cache, signifying how many lines of the same set index can the cache contain. In modern Intel processors there is a further classification of cache slices above cache sets, i.e. there are a number of cache slices and each cache slice has a number of cache sets.

Figure 4.1: How address bits are used to index the cache. The address space is 8GiB and each slice has 2048 sets with 64B lines [Liu+15].



The way cache addressing works is that for an address of a byte in memory the lowest order $log_2(B)$ bits of the address are the offset within its cache line for that byte. The next $log_2(S)$ bits are then used as a set index for the cache line. The rest of the high order address bits are used as a tag to differentiate between the lines within a cache set.

If the cache architecture uses cache slices then a hash function of the high order tag bits is used to determine the cache slice index for that cache line. This hash function is not usually released. Figure 4.1 shows how cache addressing would work for a 64B cache line in an 8GB address space with 4KiB and a 2MiB page size.

4.6 Cache eviction

Since there are more than W lines that share their set index bits in the main memory address space, not all of them can fit in the cache at the same time. This means that if a cache set is full, some of the other cache lines in that set have to be evicted for a new cache line to be loaded in.

There are a number of strategies that are used to decide which cache line to evict according to recent access patterns. One is called LRU (Least Recently Used) which evicts the cache line which has not been been accessed for the longest time.

The other cache eviction strategy is the MRU (Most Recently Used). This evicts the cache line that has been accessed most recently. It acts almost completely opposite to LRU. However, both of these strategies are better in different situations depending on the accesses patterns of the code currently running on the processor.

The hardware either has to pick one strategy and stick with it for all programs or has to somehow pick the better strategy based on the running program. The way Intel handles this problem is by using an adaptive eviction strategy. It basically runs each eviction strategy on a small sample of cache sets and measures the number of cache misses. Whichever strategy produces the least cache misses is applied to more cache sets. This way the cache eviction strategy adapts to the running program.

4.7 Inclusive vs exclusive caches

As was described in section 4.4, there are multiple layers of cache. Each layer is larger than the previous with the LLC being the largest. There is an important property in Intel caches which is inclusivity. Inclusivity means that each layer of cache stores all the data inside the previous layers. So if an entry exists in the L1 cache it is guaranteed to exist in the L3 cache as well.

The other side of the coin is exclusive caches. In exclusive caches, a cache line can be present in only one of the cache levels. Exclusive caches allow for more efficient use of the cache space since cache lines do not need to be duplicated across the cache levels. Inclusive caches on the other hand, while they do store redundant data, they are much simpler to implement (new lines are just propagated up the cache hierarchy).

A main point to consider with inclusive caches, that is very relevant to this thesis, is how eviction works with inclusive caches. If the cache is inclusive and a line is evicted from the LLC then it is also evicted from lower cache levels as well. This allows an attack process to only monitor the LLC but still have a complete picture of the memory accesses performed by a victim process. The attacks described in chapter 5 and 6 make use of this property of Intel caches to bypass the requirement of running on the same core to monitor the L1 cache.

4.8 Cache latencies vs main memory

While all cache levels are faster than memory, different cache levels have different latencies. For skylake based processors the L1 cache which is closest to the core has a latency of 4 cycles [Coo09]. L2 has a latency of 12 clock cycles and L3 has a latency of 44 clock cycles. A miss on the LLC can have latencies higher than 150 cycles.

While possible, it is not that easy to differentiate between the cache hit vs miss of lower level caches. However, an LLC hit/miss is very visible to an attack process monitoring a cache set.

4.9 Exploiting cache vs memory latencies

There are many ways of using the cache side channel in order to gain information about the memory access patterns of an operation inside another process, for example an encryption with a private key within a server process. In the subsections below, two of the techniques, the Evict+Time and the Prime+Probe techniques, are described.

4.9.1 Evict+Time technique

This technique assumes that the start time and end time of any encryption operation is known and that the triggering of the encryption operation can be performed by the monitoring process.

The way this technique works is, first, trigger an encryption operation twice and measure the time it takes for the second run. The first run fills the cache with the required data while the second run is used as a control measurement. Then before the next run of the encryption operation, a cache set is filled up so that all lines previously inside are evicted. This cache set should be chosen in such a fashion that access to a line mapped to it is dependant on the secret key used in the encryption operation.

The encryption is then triggered in the server process and the time it takes for the operation to complete is measured and compared to the control time previously measured.

If the encryption tried to access a cache line mapped to the evicted cache set then a cache miss would occur and that would cause a time penalty while the data is

fetched from the main memory. However if no access to a cache line mapped to that cache set is required by the encryption operation then the time it takes will not be affected by the cache eviction.

This allows the monitoring process to find out if a certain cache set has been accessed during the encryption operation and with knowledge of how the encryption code works and strategic choices for the cache set(s) being evicted, the monitoring process can gain information about the secret key used by the encryption.

4.9.2 Prime+Probe technique

This is the approach being used in this thesis. This allows for high resolution monitoring of a cache set throughout the encryption operation. By reading the code of the encryption operation, you can find certain memory locations that are accessed based on the secret key or more probably a set of memory locations whose access order and pattern depends on the secret key. To monitor a certain cache set, an eviction set is created for that cache set. An eviction set is a set of W memory lines that all share the same set index thus fill up a cache set.

The monitoring process then primes the cache by accessing all the elements in the eviction set. Then it probes the set by accessing them all again and recording the time taken for the probe operation. Whenever the encryption operation accesses a line that maps to that cache set, it loads the line into the cache and evicts one of the lines from the eviction set of the monitoring process.

When the monitoring process tries to probe the eviction set again it would take longer for the probe to finish due to the eviction caused by the monitored process. By doing this process repeatedly during the encryption operation, the monitoring process is able to figure out the memory access patterns of the encryption with regards to the monitored cache sets.

CHAPTER 5

Practical Cache Timing Attacks

In order to abuse the cache side channel to get memory access patterns from a victim, the attack process needs to first get an eviction set for a certain cache set. Then it has to use said eviction set to monitor the cache set and collect timing data. Finally some post processing on the collected timing data is needed to get information about the memory access patterns of the victim process.

This chapter discusses the practical aspects of creating an attack process that can monitor memory access patterns of another victim process. The attack was implemented on an Intel Core i7-4600 CPU, because of access to the cache selection hash function. The attack can be implemented on most processors as long as the cache slice selection function is known.

5.1 Controlling the set index

The first step is to get an eviction set for a specific cache set. Figure 4.1 shows how an address is used to determine where in the cache it is stored. As discussed in section 4.5 the least significant $log_2(B)$ bits are used for cache line offset, where B is the cache line size. The next $log_2(S)$ bits are used as a cache set index where S is the number of sets in the cache/slice.

The figure shows how the address is divided for cache indexing. In an 8GiB address space with 2048 sets in a slice and 64B per cache line, the set index is $log_2(2048) = 11$ bits long and starts at bit 6 through 16. If the cache under consideration has W ways of associativity for each set, then, to create an eviction set for a certain set index we need W bytes mapping to different cache lines all having an address with the same set index bits (6 through 16). For caches other than L1 which is not shared between cores, these addresses are physical addresses rather than virtual ones. This is because address translation happens before reading and writing to caches (other than L1).

In virtual address space, only the bits in the page offset are visible to the running process. This is because during translation to physical address space the higher address bits forming the page number are changed to the associated frame number in physical address space.

In 4KiB pages the page offset is only 12 bits long. The set index is then split, with some of the bits inside the page offset can be seen and controlled by the running process and the rest inside the frame number are not controlled or seen by the running process. In huge pages of size 2MiB, however, the page offset bits are 21 bits long. This is more than enough to cover all the set index bits.

So to create the eviction set, a large buffer (at least twice the size of the monitored cache), is allocated on huge pages. The eviction set can be chosen as a set of lines from this buffer that all share the same set index bits in their address.

5.2 Controlling the cache slice

As discussed in section 4.5 the higher index bits of the physical address are used to determine the cache slice. The exact cache slice selection function differs between each processor. The Intel Core i7-4600 processor is used here as an example. The Intel Core i7-4600 uses this cache slice selection function:

$$C_i = p_{17} \oplus p_{18} \oplus p_{20} \oplus p_{22} \oplus p_{24} \oplus p_{25} \oplus p_{26} \oplus p_{27} \oplus p_{28} \oplus p_{30} \oplus p_{32} \tag{5.1}$$

C_i refers to the cache slice index that would be chosen based on the result of the hash function and p_i refers to the i-th bit of the physical address (the least significant bit is the 0-th bit).

The Intel Core i7-4600 has only two cache slices. To choose which cache slice a data line ends in, the relevant bits according to the above equation are XORed and the resultant 1 or 0 is the cache slice used. The bits that affect the cache slice selection all start from bit 17. This is the very first bit after the most significant bit in the set index. To monitor a specific cache set, the eviction set must be in the same cache slice.

Using equation (5.1) the attack process is able to find the cache slice for the monitored cache line. The next step is to augment the process described in section 5.1 so that not only do all lines in the eviction set share the same set index bits but also they all reside on the same cache slice.

It is important to note that there are 4 bits being used in the cache slice selection function that are within the page offset bits for huge pages. We can use this to our advantage. The Intel Core i7-4600 L3 cache is 16 way set associative, thus to monitor a specific cache line, an eviction set of 16 lines that all map to the same cache slice and cache set as the monitored line is required.

Given an address to monitor, the set index and cache slice can be extracted by looking at the relevant bits. Allocating a 2MiB huge page gives us 21 bits of address space preceded by a prefix of a frame number. We can choose addresses from this page to add to the eviction set. We can get the physical address of the allocated page and then XOR the relevant bits of the address, according to equation (5.1). This results in the the cache slice that this address will be mapped to (if we don't change bits 17, 18 and 20 that are under our control).

We can compare this cache slice to the cache slice of the cache line we want to monitor. If they match then for all our choices for the eviction set within the page, bits 17, 18 and 20 should XOR to 0 (so that the cache slice does not change). If the cache slice of our allocated page and the monitored cache line do not match, then we should choose addresses such that bits 17, 18 and 20 should XOR to 1.

Of the 21 bits that can be chosen freely, bits 0 through 5 are useless since addresses with those bits different would still map to the same cache line, then bits 6 through 16 are the set index for the cache line that is monitored. Based on the restrictions from the previous paragraph, bits 17 through 20 allow for only 8 different addresses that map to a specific cache set and cache slice. So to create a 16 address eviction set at least 2 huge pages must be allocated with 8 addresses extracted from each.

5.3 Measuring time

Section 5.2 talked about getting an eviction set, the next step is to time the probe. X86 processors have a Time Stamp Counter (TSC) which is a 64bit register that stores the number of cycles since the last processor reset. It can be used to measure the relative time of events [Coo].

The "RDTSC" instruction copies the TSC into the registers EAX and EDX. The main idea of timing is to call "RDTSC" to start the timer and store EAX and EDX in a 64 bit register. After what you want to measure is complete, to stop the timer, call "RDTSC" again for the new value of the TSC. Finally subtracting the old value from the new value results in the elapsed CPU cycles between the start and the stop of the timer.

There are a few issues with the naive approach described above. One of which is that "RDTSC" is not a serializing instruction. The reason why this is a problem is instruction reordering. Instruction reordering allows the CPU to execute instructions in an order different than they are read in to improve the speed at which program runs or to better utilize the CPU resources. For example a memory read can be started early in case it is a cache miss while the rest of the code completes or a write can be pushed further down if it will not affect the execution.

Since "RDTSC" is not serializing, then the scheduling of the instruction is up to the CPU at runtime and it may not wait for the previous instructions to complete before it runs and instructions following it may be run before it does. To fix this, Intel recommends it should be preceded by a serializing instruction such as "CPUID" or "LFENCE".

On newer processors that support it, the instruction "RDTSCP" may be used instead. "RDTSCP" also reads the TSC however waits until previous instructions have been completed before the read is performed [Coo]. There are other issues related to optimization that affect the timing code and its effectiveness, section 5.6 talks about that in more details.

5.4 Implementing Prime+Probe

For this thesis, the prime+probe technique [TOS10] described in section 4.9.2 was implemented to monitor the memory access patterns of a victim process. The method described in section 5.2 was implemented to get an eviction set for a specific data line.

The eviction set is then placed in a singly linked list, where each entry points to the next address on the list. The prime function accesses the eviction set to add them them to the relevant cache set and evict all other entries. Then the attack process loops, probing the cache set by accessing the eviction set and measuring the time it takes. The entries from this stage were then stored in file for later post-processing.

One thing to consider with this method is thrashing. Probing the eviction set implicitly primes it for the next probe. However, using consecutive probes can cause problems. This is due to the LRU cache replacement policy. Basically, if the victim process evicts a line from the cache set being monitored, it will evict the oldest line (which is the first entry in the linked list). On the next probe, loading the first entry would then evict the second entry from the cache. This will go on for the entire probe. A way to solve this is by using a doubly linked list so that the prime stage loops over the eviction set in the opposite order than the probe stage.

5.5 Using cache side channel for communication

To test the setup that was described in this chapter, we tried to implement a covert communication channel between willing parties using cache-timing [WXW12]. Since both the sender and receiver are not synchronized a Return-To-Zero (RZ) self clocking scheme is used [Liu+15; GG].

The algorithm starts by the sender picking two lines that map to different cache sets and sharing them with the receiver. The receiver then uses the previously mentioned methods to create an eviction set for each line. The receiver then starts monitoring both lines one at a time and keeping track of the probe time.

The sender on the other hand uses the preshared lines to send the message. One line is for a 0 bit and the other line for the 1 bit. To send a bit, the sender reads the relevant line repeatedly for a period of time T_{mark}. After each bit there is a period of waiting T_{pause} to make sure that no bit is missed by the receiver. Algorithms 4 and 5 shows this methodology in detail for both sides.

Figures 5.1 and 5.2 shows the probe time detected by the receiver during the message sending. ”Line 0” refers to the memory address that is used for sending a ”0” bit and ”Line 1” is used for sending a ”1” bit. The figure shows alternating peaks in the lines representing the 0 and and the 1 bits. T_{mark} and T_{pause} were set to $1,000,000$ clock cycles each (based on the TSC). The figure shows that there were a lot of redundant probes happening for each bit so we could reduce the T_{mark} and T_{pause} times to increase throughput of the channel, however, this will lead to a reduction in the signal to noise ratio.

Figure 5.1: Probe time detected by receiver. "Line 0" is used to send a "0" bit and "Line 1" is used to send a "1" bit..

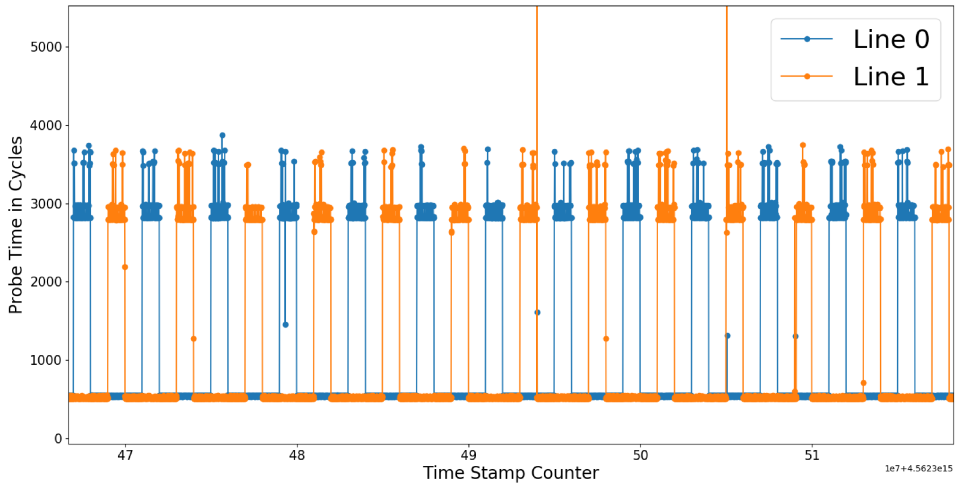
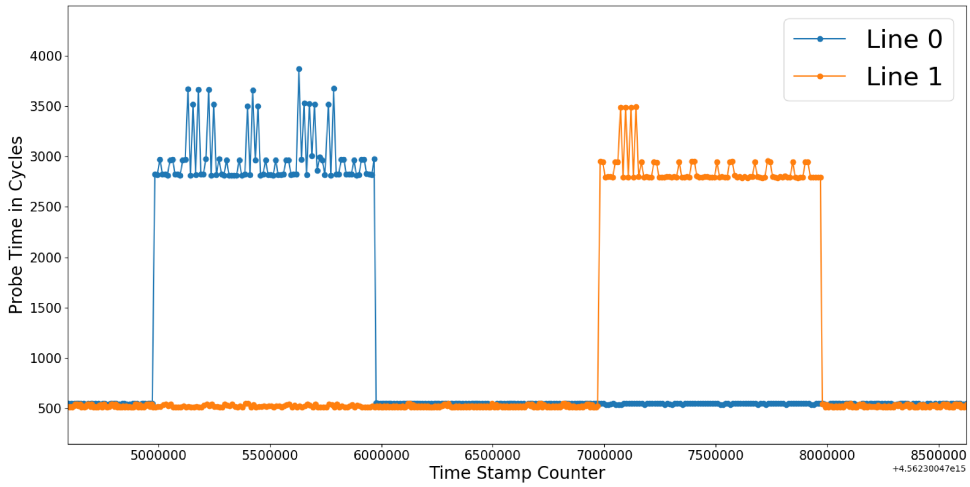


Figure 5.2: Probe time detected by receiver (zoomed in to just two bits).



Algorithm 4 Cache timing communication protocol (Sender Side)[Liu+15]

Input: $D[N]$ the N-bit message to send.

```
for i = 0 to N − 1 do
    if D[i] = 1 then
        for an amount of time T_mark do
            access line1
        end for
    else
        for an amount of time T_mark do
            access line0
        end for
    end if
end for
```

Algorithm 5 Cache timing communication protocol (Receiver Side)[Liu+15]

```
for an amount of time T_monitor do
    Prime set 0
    Prime set 1
    Probe set 0
    Probe set 1
end for
```

5.6 Fighting "optimization"

Normally, optimization is a desirable result, especially when it is applied automatically by the compiler or by the hardware at runtime. However, timing code is very sensitive to optimization. Instruction reordering or data prefetching may mess up the timing to the level of uselessness. This section discusses the different "optimizations" that may happen to the timing code and how they were overcome. It will also discuss how certain strategies for timing could not be implemented because optimizations ruined the end result.

5.6.1 Compiler optimization

Nowadays, compilers are very good at removing inefficiencies from the code, changing code order or ignoring useless statements to try and speed up the final result. Normally this is a good thing, you want your program to run as fast as possible without having to optimize everything manually and maybe make your code less readable.

However, timing code that deals with assembly and makes a lot of assumptions about code order is very vulnerable to compiler "Optimizations". A compiler may look at a useless variable access and remove it (during the prime or probe stage when you are accessing locations just to put them in the cache). It can reorder reads vs

the timing code, and also it can choose to ignore statements that lead to undefined behavior.

All of these are reasonable behaviors in real world code, where rarely if ever you require the final machine code to do exactly as you want it to do, in a fixed order. In a cache timing attack, however, you can run into many of those situations unwittingly because sometimes the aim of the code is just to fill the cache rather than anything particularly useful.

To play around that, the best thing to do is to disable optimizations from the compiler, for example using the flag "-O0" with GCC. It is also recommended to enable all warnings to make sure that the compiler does not complain about your code so as to make sure nothing gets "optimized" away.

5.6.2 Prefetching

Section 4.8 describes how main memory is much slower than cache by sometimes even 2 orders of magnitude or more. A cache miss is a big setback for the performance of the CPU in terms of instructions executed per second. Processor manufacturers see this as a problem and thus try to reduce the number of cache misses as much as possible.

One way the CPU tries to improve the cache hit rate is by using cache prefetching. Prefetchers work by predicting a likely address that might be requested in the next couple of instructions that will lead to a cache miss. The prefetcher can then load those lines from memory into a separate buffer earlier than required before an instruction is even encountered that would lead to the aforementioned cache miss.

If an instruction requests a data line not in cache but has been predicted by the prefetcher, the prefetched data from the buffer is promoted to the cache and is sent back to the execution unit of the CPU faster than it would have if the cache miss lead to reading the data directly from main memory.

When measuring timing for a cache timing attack, a correct prediction by the prefetcher can cause the code to believe a cache hit rather than the cache miss that would have actually happened if not for the prefetcher. This can lead the attack process to make wrong predictions about the memory access patterns of the victim process and increase the noise in the collected data.

One thing that prefetchers are vulnerable to is pointer dereferencing. If the next requested address is based upon the information stored in the currently requested address it would usually defeat the prefetcher predictions and allow cache hits and cache misses to happen naturally.

To try and increase the probability that the prefetcher mispredicts, the addresses in the eviction set should be farther from each other which prevents spacial based prefetchers to work. Also the strides between the lines in the eviction set should be irregular so that a stride prefetcher does not predict cache misses either.

It is difficult to defeat the prefetcher 100% for all processors since most manufacturers keep the prefetcher designs they use private and because every new processor

may bring a more complicated prefetcher. However following the tips above can reduce the effectiveness of most practical prefetchers that currently exist [Mit16].

5.7 Avoiding overhead

The way the probe and timing works, it detects if the cache set was used since the last prime. If the time between the prime and the probe is too large then multiple accesses can be detected as only one. If the time between the probe end and then next prime is too large, a memory access between them may not be detected. Therefore, it is important for the time between probes to be as little as possible so as to allow for the highest resolution.

Since compiler optimizations are bad because of the reasons listed in section 5.6.1, some practices have to be observed when writing the code, specifically the code that is used around and within the timed probe. For example, any functions called should be marked `inline` so as to reduce the function call overhead. Unnecessary functions should not be called, only the minimum of the prime and probe function should be used during the monitoring phase.

Another thing is using complex structures such as vectors, maps, pairs and other c++ STL constructs is a bad idea. Although they are easier to work with in terms of code readability, on the other hand, without any code optimizations by the compiler they are very expensive to use. It is best to store the timings returned by the probe function in C arrays so that the code is as efficient as possible. Finally there should be minimal branching and loops should be unrolled as much as possible to reduce those overheads.

Postprocessing can then be done after the monitoring stage is complete to move the data into better containers for further processing or to write them out to files to allow for offline processing.

CHAPTER 6

Attacking LibTomMath

The main contribution of this thesis is the development of a cache timing attack on the LibTomMath [Lib15] modular exponentiation implementation. LibTomMath is used in a some cryptographic libraries including LibTomCrypt as well as WolfCrypt part of the WolfSSL commercial cryptographic library [Inc17]. For simplicity, we first attack a modified variant of the library that allows us some assumptions such as knowing the addresses of the multipliers and knowing the start and end of the exponentiation process. The modifications are outlined in more detail in section 6.2. In section 6.5, we then discuss the required modifications for the attack to work on a vanilla LibTomMath implementation.

6.1 LibTomMath implementation details

6.1.1 Multi-precision integers

The main idea of LibTomMath (LTM) is to provide functions that work with very large numbers that dont fit in a single register. The kind of numbers required to make things like RSA secure. So they provide an implementation for Multi-precision integers as well as functions that can perform mathematical operations on said implementation, i.e. addition, multiplication, exponentiation, etc.

Multi-precision integers are defined as a C struct and contain a reference to an allocated array for the bits of the integer. All the functions then pass around references to these structs for their inputs and outputs.

6.1.2 Multi-precision modular exponentiation

The main focus of the attack is on the modular exponentiation function of the LibTomMath. The modular exponentiation function, in the LibTomMath library, is implemented based on sliding window exponentiation using montgomery reduction. The algorithm is explained in section 3.4 in more detail.

The function starts with allocating an array on the stack to store the structs for the multipliers. It then initializes the multipliers based on the window size. The exponent is divided into windows such that all non-zero windows start with the 1 bit. This means that only multipliers in the upper half of the table (the ones that

start with the 1 bit) will be used during the actual computation phase. This fact is exploited by the precomputation part which only computes the upper half of the multiplier table.

The next part of the function performs the actual computation. The windows are extracted. Whenever a 1 is encountered it starts a window which continues on for the maximum window size. The current result is then squared as many times as there are bits in the window and then multiplied with the multiplier that has an index equal to the value inside the current window.

Zeros encountered outside a window, i.e. the previous window is complete and another 1 bit has not yet been encountered to start a new window, are not added to any window but the result is squared for each one.

After the entirety of the exponent is processed, the memory allocated for the multipliers is freed and the result is returned.

6.2 Attacking a simplified implementation

The original code of the modular exponentiation function was modified to enable a simpler version of the attack. The attack theoretically could be performed without the modifications, section 6.5 goes into more details on how to enhance the attack described below to work on the vanilla implementation of LibTomMath.

There are two main changes to the original code. The first is meant to allow the monitoring process to find the start of the modular exponentiation. This was done by adding code to the exponentiation function that accesses a certain pre-shared memory line repeatedly for a certain period of time before starting the actual exponentiation. The activity over this line then acts as a clock for the start of the exponentiation. The attack process would then monitor that line, using the Prime+Probe technique described in section 5.4. At the start of each modular exponentiation, the cache set associated with that line would show high activity for an extended period allowing the attack process to synchronize with the exponentiation.

The second major change to the code is the fixed location of the multipliers. In the original code the function would store an array of structs on the stack. Each of those structs would contain a reference to the location containing the actual multiplier. The multiplier data would reside on dynamically allocated memory on the heap. We made the change to allocate the multipliers statically so as to know their location in memory from the attacking process much more easily without the need for clever preprocessing as shown in section 6.5.

Finally the last change was needed to output the relevant addresses for the above changes. So a printf statement was written to output the clocking line from the first change as well as the starting address of the list of multipliers from the second change. And since the physical addresses were required, because virtual addresses are process scoped, the LibTomMath code was run as root and a function was written in to convert the virtual addresses to physical ones before printing them to the console.

6.3 Implementing the attack

The attack uses prime+probe to monitor specific lines in the victim process. The attack process takes as input the synchronization line as well as the first line from the multiplier table (both outputed by the modifications in section 6.2). It starts by creating an eviction set for each line in the multiplier table as well as the synchronization line.

Then starts the probing phase. The main issue is how to arrange the primes and probes of the table lines. If there is too much time between the prime and the next probe then the resolution of the measurement is very low. If code primes then probes one line then moves on to the next, for the entire table, then by the time the code goes back to monitoring the first line in the table again, many memory accesses have potentially been missed.

So to fix this problem and to allow for a high resolution, each line from the multiplier table was monitored separately in conjunction with the synchronization line. By monitoring only one line from the table (plus the synchronization line) at a time, the resolution is high enough to not miss consecutive accesses to the same multiplier line. By using the synchronization line, it is possible to then synchronize the traces of all probe times of the different table lines since the memory access patterns are the same.

Each line is monitored for a total of 20000 prime+probe cycles and then the attack process switches to the next line in the table. It also keeps track of the probe time as well as the TSC value at each probe. After all the lines of the table have been monitored for the requisite number of probes, the data is written to a file for post-processing.

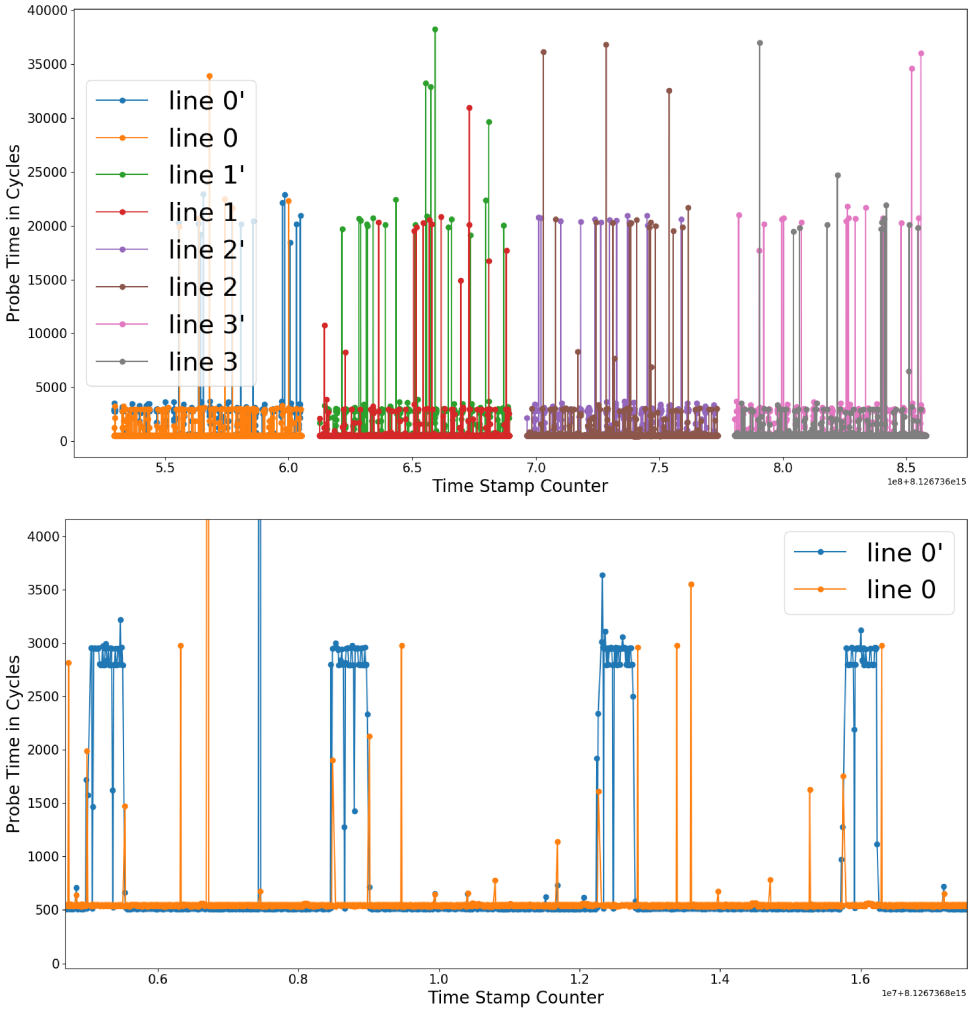
Figure 6.1 shows a graph of the captured data for 4 multipliers. Each multiplier is monitored separately along with the synchronization line.

6.4 Post-processing the results

To process the results, we start by converting the probe times to be either a 0 or a 1 based on whether a memory access has been detected or not. After experimentation, it was found that a probe time below 800 clock cycles means that there was no cache miss and no memory accesses occurred between the prime and the probe. Between 800 and 4000 clock cycles, means that a memory accesses occurred. Anything higher than 4000 is undefined. This is because really high probe times most likely correspond to context switches in the attack function rather than just a memory access in the victim process. Figure 6.2 shows how the final result of a monitored cache line looks.

The next step was to extract entire exponentiation phases to study them. Using the synchronization line, multiple consecutive accesses to the line was a synchronization signal for the start of an exponentiation. All probes between synchronization signals belonged to one exponentiation. This meant that we now have the access patterns for multiple exponentiations with the same exponent for each multiplier.

Figure 6.1: Top: traces of 4 multipliers each with the synchronization line (ending in '). Bottom: close up of one trace showing how the synchronization line is active at the start of every exponentiation..



We merge all the exponentiations together by monitoring the time of the access since the start of the exponentiation. The exponent is basically found at this point since it is the concatenation of the window values that correspond to the sequence of multipliers. The exponent represents the long term private key stored and used by the server for securing its communications. By recovering the private key, the attacker can decrypt past and future communications that were secured by this key, which has an average lifetime of around one year.

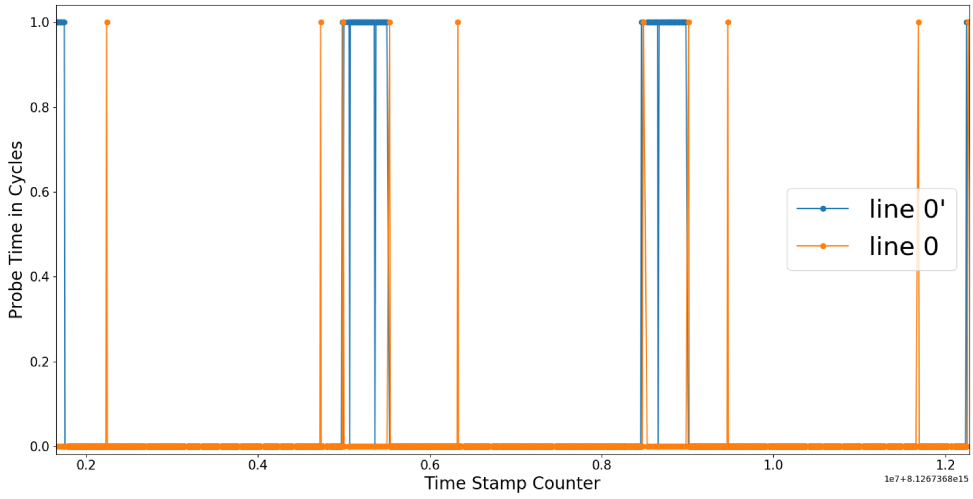
6.5 Attacking without library modifications

This implementation of the attack cannot yet be used on a vanilla LibTomMath implementation without modifications. The addresses for the multipliers are required as an input to the attacking process. However there are ways around that limitation.

The idea is, most cryptographic code, including LibTomMath is opensource. This is to allow for easier auditing by whitehats among other motivations. With access to the source code you are able to look at the expected access patterns for the multipliers during the initial phase of the modular exponentiation.

At the start of the function, the array of multipliers is first populated. The multipliers are populated in a certain order and each multiplier is used to calculate the next one as shown in the first section of Algorithm 3. For example, the first multiplier $g[0]$ is squared to calculate s then to calculate the value of each other multiplier, the previous multiplier is multiplied by s, and so on and so forth. This

Figure 6.2: Close up of one trace after it has been converted to binary and removing some noise..



means that if you have a set of multiplier cache traces, the index of the multiplier belonging to each cache trace can be found by looking at the access patterns during the initial phase of the multiplication.

For the rest of the exponentiation function, the access patterns of the multipliers can no longer be predicted so easily. However some statistical analysis can be used to differentiate between multiplier cache traces vs other cache traces. The average distance between non-zero windows in an exponent for window size S is $S+1$ [Men+96]. So for an N-bit exponent we can expect around $N/(S+1)$ non-zero windows. Since there are 2^{S-1} different values for a non-zero window, (window size S and all non-zero windows must start with the 1 bit), this means that each multiplier is expected to be used around $2^{1-S} \cdot N/(S+1)$ times during the computation phase [Liu+15].

Using the previous result we can filter out cache traces of lines that are not accessed very often or accessed too often to be a multiplier. The end result can have some false positives, however they can be more easily filtered manually by looking at the access patterns at the start of the exponentiation during the precomputation phase.

The other problem that required changes to the original LibTomMath code was the clocking line to figure out when the encryption starts. This can be done by monitoring the lines of code at the start of the function. Since the code location is static with respect to the offset of the library in the address space, if you figure out where in the physical address space the library is mapped, you can monitor accesses to specific lines of code. On linux, the attack process can check memory mappings for other processes if it is running as root.

The final change mentioned in section 6.2 is the outputting of the relevant line addresses as well as the conversion of said addresses from virtual to physical addresses is no longer required if the attack process already has the relevant addresses from the enhancement mentioned in this section.

CHAPTER 7

Applicability and Mitigation

7.1 Applicability

The attack described in this thesis does not depend on a specific processor or library implementation but is potentially applicable to many libraries that implement table based optimizations for modular exponentiation. It does require the processor to have inclusive caches as well as the attacker to have access to huge pages.

The attack requires that the attacking code and the victim code run on the same CPU. The attack also works in a virtual machine based environment, where the attacker and the victim are running on different VMs. The attack does not require any escalation of privileges outside of having root within its own VM.

Intel processors have inclusive caches thus the attack is applicable on a wide range of different processors. Furthermore, hugepages allow for a much better utilization of the TLB which is a scarce resource, therefore, this feature is enabled for most virtualization implementations offered by IaaS providers.

Finally, whether a library is affected or not depends on which optimizations they use as well as on how they store and access the multiplier table. This is discussed further in section 7.2.

7.2 Mitigation

One way to mitigate the attack depends on the method of accessing the multiplier. Some implementations, such as in LibGnuPG, always access all multipliers in the table each time but uses a bitwise ”AND” to only get the value of the correct multiplier. OpenSSL stores the multipliers vertically rather than horizontally. As in each line in memory consists of a single part of each multiplier. So the first line in memory contains the first 4 bits of each multiplier then the next line in memory contains the next 4 bits from each multiplier and so on. To get a single multiplier all memory lines must be accessed and thus making it harder for an attacker to find which multiplier is used.

Another way to mitigate the attack is to use a constant time implementation that does not have any secret dependant branching or memory references. Techniques for this have been explored and implemented [BLS12]. However, these implementations seem to be tricky to get right as is shown by attacks on the "constant time" OpenSSL implementation on ARM processors [Coc+14].

IaaS providers can protect their customers against these kinds of attacks by disabling hugepages for their virtualization solution. This however leads to a degradation in the quality of service due to more TLB misses. Another way is avoiding co-residency of Virtual Machines on the same processor package. This however goes against the motivation for cloud computing which is resource sharing.

Finally, another technique that can be used to limit the applicability of cache timing based attacks is page coloring. The technique works by grouping frames into different colours and ensuring that frames from different colors cannot have lines mapped to the same cache set. VMs from different customers can then have their pages mapped to frames of different colors thus limiting the applicability of cache timing based attacks [Shi+11; WL08].

CHAPTER 8

Conclusion

Our contribution is an implementation of a cache timing attack on the LibTomMath modular exponentiation implementation. The attack uses the prime+probe technique to use cache timing as a side channel in order to extract the secret exponent.

The attack targets the LLC, thus is possible cross-VM, which makes it applicable in situations where an attacker is able to run their code on another VM on the same processor package as the target. This scenario is not unlikely due to the rising popularity of using IaaS providers by smaller companies which run VMs belonging to different customers on the same machine.

The attack requires a few things to work. Hugepages support must be turned on by the service provider. This requirement is easily satisfied because IaaS providers usually have it turned on to prevent performance degradation due to TLB contention. The attack also requires the processor to have inclusive caches however almost all Intel CPU caches are inclusive.

The attack also requires the knowledge of the cache slice selection algorithm in advance in order to craft the eviction set. Those are not released by Intel however there are techniques available in order to reverse engineer those hash functions for any processor [IES15b].

Since the requirements of the attack are prevalent in most IaaS installations, it is up to library developers to protect themselves against this kind of attack. Changing the way the multiplier table is accessed so that cache timing attacks are not able to get secret based information appears to be the best strategy.

Bibliography

[Aas14] Josh Aas. "Let's Encrypt: Delivering SSL/TLS Everywhere". In: *Let's Encrypt* 18 (2014).

[AMD] AMD AMD. "Architecture Programmer's Manual: Volume 2: System Programming". In: *AMD Pub* 24593 ().

[BB05] David Brumley and Dan Boneh. "Remote timing attacks are practical". In: *Computer Networks* 48.5 (2005), pages 701–716.

[BI14] Zineb Ait Bahajji and Gary Illyes. *HTTPS as a ranking signal*. `https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html`. [Online; accessed 1-June-2017]. 2014.

[BLS12] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. "The security impact of a new cryptographic library". In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2012, pages 159–176.

[BT11] Billy Bob Brumley and Nicola Tuveri. "Remote timing attacks are still practical". In: *European Symposium on Research in Computer Security*. Springer. 2011, pages 355–371.

[But10] E Butler. "FireSheep: cookie snatching made simple". In: *ToorCon Conference. San Diego, CA*. 2010, page 8.

[Coc+14] David Cock et al. "The last mile: An empirical study of timing channels on sel4". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pages 570–581.

[Coo] Intel Coorporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3*.

[Coo09] Intel Coorporation. *Intel 64 and IA-32 architectures optimization reference manual*. 2009.

[Cor14] BM Corporation. *An overview of the SSL or TLS handshake*. `https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm`. [Online; accessed 1-June-2017]. 2014.

[Gar10] K Gary. "An overview of Cryptography". In: *an article available at `http://www.garykessler.net/library/crypto.html`* (2010).

[GG] IA Glover and PM Grant. “Prentice Hall, 2010”. In: *Digital Communications* ().

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. “RSA key extraction via low-bandwidth acoustic cryptanalysis”. In: *International Cryptology Conference*. Springer. 2014, pages 444–461.

[HW79] Godfrey Harold Hardy and Edward Maitland Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[IES15a] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing–and Its Application to AES”. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pages 591–604.

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. “Systematic reverse engineering of cache slice selection in Intel processors”. In: *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE. 2015, pages 629–636.

[Inc+16] Mehmet Sinan Inci et al. “Cache attacks enable bulk key recovery on the cloud”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, pages 368–388.

[Inc17] wolfSSL Inc. *WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud*. `https://www.wolfssl.com/wolfSSL/Home.html`. [Online; accessed 1-June-2017]. 2017.

[Jee13] Zubair Jeelani. “An insight of SSL security attacks”. In: *International Journal of Research in Engineering and Applied Sciences* 68 (2013).

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential power analysis”. In: *Advances in cryptology—CRYPTO’99*. Springer. 1999, pages 789–789.

[Kle+10] Thorsten Kleinjung et al. “Factorization of a 768-bit RSA modulus”. In: *Annual Cryptology Conference*. Springer. 2010, pages 333–350.

[Koc+11] Paul Kocher et al. “Introduction to differential power analysis”. In: *Journal of Cryptographic Engineering* 1.1 (2011), pages 5–27.

[Lan01] Eric Landquist. “The quadratic sieve factoring algorithm”. In: *Math* 448.2 (2001), page 6.

[Lib15] LibTom. *LibTomMath*. `http://www.libtom.net/LibTomMath/`. [Online; accessed 1-June-2017]. 2015.

[Liu+15] F. Liu et al. “Last-level cache side-channel attacks are practical”. In: *Security and Privacy (SP), 2015 IEEE Symposium* (2015), pages 605–622.

[Men+96] Menezes et al. *Handbook of Applied Cryptography*. CRC Press, 1996.

[Mit16] Sparsh Mittal. “A survey of recent prefetching techniques for processor caches”. In: *ACM Computing Surveys (CSUR)* 49.2 (2016), page 35.

[QS01] Jean-Jacques Quisquater and David Samyde. “Electromagnetic analysis (ema): Measures and counter-measures for smart cards”. In: *Smart Card Programming and Security* (2001), pages 200–210.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pages 120–126.

[Shi+11] Jicheng Shi et al. “Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring”. In: *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*. IEEE. 2011, pages 194–199.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. “Efficient cache attacks on AES, and countermeasures.” In: *Journal of Cryptology* 23.1 (2010), pages 37–71.

[WL08] Zhenghong Wang and Ruby B Lee. “A novel cache architecture with enhanced performance and security”. In: *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. IEEE. 2008, pages 83–93.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. “Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud.” In: *USENIX Security symposium*. 2012, pages 159–173.

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: A timing attack on OpenSSL constant time RSA”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2016, pages 346–367.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ