Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Cesar Pereida Garcia

Cache-Timing Techniques:
Exploiting the DSA Algorithm

Master's Thesis
Espoo, June 30, 2016

Supervisors:	Prof. N. Asokan, Aalto University
	Prof. Dominique Unruh, University of Tartu
Advisor:	Prof. Billy Bob Brumley, Tampere University of Technology

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Cesar Pereida Garcia		
Title: Cache-Timing Techniques: Exploiting the DSA Algorithm			
Date:	June 30, 2016	**Pages:**	viii + 70
Major:	Security and Mobile Computing	**Code:**	T-110
Supervisors:	Professor N. Asokan Professor Dominique Unruh		
Advisor:	Professor Billy Bob Brumley		

Side-channel information is any type of information leaked through unexpected channels due to physical features of a system dealing with data. The memory cache can be used as a side-channel, leakage and exploitation of side-channel information from the executing processes is possible, leading to the recovery of secret information. Cache-based side-channel attacks represent a serious threat to implementations of several cryptographic primitives, especially in shared libraries.

This work explains some of the cache-timing techniques commonly used to exploit vulnerable software. Using a particular combination of techniques and exploiting a vulnerability found in the implementation of the DSA signature scheme in the OpenSSL shared library, a cache-timing attack is performed against the DSA's *sliding window exponentiation* algorithm.

Moreover, the attack is expanded to show that it is possible to perform cache-timing attacks against protocols relying on the DSA signature scheme. SSH and TLS are attacked, leading to a key-recovery attack: 260 SSH-2 handshakes to extract a 1024/160-bit DSA hostkey from an OpenSSH server, and 580 TLS 1.2 handshakes to extract a 2048/256-bit DSA key from an stunnel server.

Keywords: applied cryptography; digital signatures; side-channel analysis; timing attacks; cache-timing attacks; DSA; OpenSSL

Language: English

Acknowledgements

This work was supported by TEKES as part of the Cyber Trust program of DIGILE (the Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

The present thesis work has been carried out during the spring semester of 2016 at the Security System Group of Aalto University, Finland as part of the Erasmus Mundus Master's Programme in Security and Mobile Computing (NordSecMob). The thesis is presented at Department of Computer Science, Aalto University Finland, and Institute of Computer Science at University of Tartu (UT) Estonia.

The thesis work was conducted by the author under the guidance and collaboration of Dr. Billy Bob Brumley (Tampere University of Technology), under the supervision of Dr. N. Asokan (Aalto University), Dr. Dominique Unruh (University of Tartu) and in collaboration with Dr. Yuval Yarom (University of Adelaide).

Firstly, I would like to extend my dearest gratitude to Billy and Asokan for their guidance, collaboration, patience, support and time during the thesis. I would also like to thank Yuval for providing his mathematical expertise and collaborating with me. And I would like to thank Dr. Dominique for the support to extend the submission deadline and the feedback provided allowing me to improve the quality of this work.

Secondly, I would like to thank all my dearest friends, colleagues, peers and my study coordinator Aino. All of them motivated me and supported me to complete this work.

A special thanks goes to my two families: my Mexican family for all the moral support, love and motivation through my whole life; my Estonian family for a place of rest, growth and reflection.

Espoo, June 30, 2016

Cesar Pereida Garcia

Abbreviations and Acronyms

AES	Advanced Encryption Standard
AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machines
ASLR	Address Space Layout Randomization
BDD	Basic Bounded Decoding
BIGNUM	Arbitrary-Precision data structure
CDH	Computational Diffie-Hellman
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVP	Closest Vector Problem
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DL	Discrete Logarithm
DDH	Decisional Diffie-Hellman
DH	Diffie-Hellman key exchange
DOS	Denial of Service
DSA	Digital Signature Algorithm
dcache	data cache
ECDSA	Elliptic Curve Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
EM	Electromagnetic
GF	Galois Field
GMR	Goldwasser, Micali and Rivest
HMM	Hidden Markov Model
HNP	Hidden Number Problem
IP	Internet Protocol
icache	instruction cache
KB	Kilobyte, 1024 bytes
KEX	Key Exchange

L1	First Level Cache
L2	Second Level Cache
LFU	Least Frequently Used
LLC	Last Level Cache
LRU	Least Recently Used
LSB	Least Significant Bit
MAC	Message Authentication Code
MRU	Most Recently Used
MSB	Most Significant Bit
NAF	Non-Adjacent Form
NP	Nearest Plane
RSA	Rivest, Shamir and Adleman public key cryptosystem
SHA-1	Secure Hash Algorithm 1
SHA-256	Secure Hash Algorithm 2, 256 bits variant
SHA-512	Secure Hash Algorithm 2, 512 bits variant
SM	Square and Multiply
SMT	Simultaneous Multi-Threading
SSH	Secure Shell
SWE	Sliding Window Exponentiation
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VQ	Vector Quantization
X11	X Window System

Contents

Chapter 1

Introduction

Often there is a big gap between theoretical and applied cryptography due to the problems of translating the theoretical security proofs to real world software and hardware implementations. On one hand, theorists define and prove in paper the strong security of cryptographic primitives and protocols. On the other hand, engineers have to implement cryptography that is strongly secure and efficient at the same time. Unfortunately, it is hard to find middle ground where strong security of actual implementations can be proved.

In order for cryptographic primitives to be standardized, they must prove strong theoretical security and resistance to known cryptanalytic techniques such as linear and differential cryptanalysis, but as it is in the world of cryptography, theoretical security is based on the assumptions about the power and abilities that an adversary has. These abilities are, at least in paper, powerful enough to cover a strong adversary. Unfortunately when the cryptographic primitives are implemented in software, the adversaries acquire new and unexpected abilities that were never considered from a theoretical perspective.

Cryptanalysis is one important branch in cryptology, which is the science of analyzing and breaking cryptographic primitives and protocols. Traditional cryptanalytic techniques try to break the security of primitives and protocols by studying them from a purely mathematical point of view, isolated from the system in which they are implemented and executed. However, it is far from being a complete approach since the systems where cryptography is executed plays an important role to determine its security.

Cryptographic primitives are built, most of the time, using well known components that are easy to implement or using predefined instructions tailored specifically for the microprocessors in which they are running. The fact that cryptographic primitives depend on multiple layers and components makes it hard to guarantee the security of the software implementations and often this leads to serious vulnerabilities. The analysis of the information leaked from the software and hardware components is called side-channel analysis.

Side-channel analysis is a cryptanalytic technique born from practice. Original side-channel timing analysis, and the most well known technique in this area, was first discussed by Kocher [30] back in 1996. This technique understands and derives private information from the side-channel information collected from physical power analysis. The main goal of side-channel attacks is to obtain complete or partial information of the internal state of a system by measuring and analyzing physical properties—e.g. power consumption, time, electromagnetic radiation, acoustic emanation and temperature variation.

A type of side-channel analysis focused on software implementations, is timing attacks. Timing attacks exploit timing information leaked from software implementations that do not run in constant time, thus by measuring the time it takes for an algorithm to complete, an attacker can deduce information that otherwise should be secret.

The main topic of this thesis is a specific type of timing attack that exploits the caching components commonly found in modern microprocessors. Cache-timing attacks exploit the availability of data in the cache, then the information is correlated to the running cryptographic algorithm to finally achieve secret key recovery.

1.1 Motivation

Digital signatures are essential for data authentication, thus attacking this cryptographic primitive allows malicious parties to authenticate any data they want. Digital signatures are a cryptographic building block and side-channel resistant algorithms must be used to ensure the security and authentication of the communications in the Internet.

The work presented here is motivated by the ongoing and young field of research trying to exploit leakage of side-channel information to mount key recovery attacks against hardware and software implementations of cryptographic primitives.

Side-channel information leakage is hard to detect by software developers as it requires a high level of expertise about the techniques and the system hardware underneath. Additionally, tests used to discover or detect side-channel information leakage are not part of the standard developer practices and therefore is hard to detect these types of vulnerabilities. Nevertheless, side-channel attacks represent a serious and immediate threat to the security of Internet.

Moreover, using well known open source and audited cryptographic libraries is a two-edged sword — as it is shown in this work. On one hand, vulnerabilities are reduced by having many people auditing the code. On the other hand if a vulnerability is found, it affects all the software depending on that library.

1.2 Goals

This thesis work focuses in detailing current cache-timing techniques used to exploit software implementations of cryptographic primitives.

Moreover, the work presented here serves the following purposes:

- To summarize the categorization of implementation attacks and define side-channel attacks.
- To define cache-timing attacks and explain current techniques used to exploit this type of attack.
- To show what small defects in software implementing cryptographic primitives can do to the security of communications.
- To demonstrate that cache-timing attacks are practical and they represent a real threat for the Internet users.

The ultimate goal is to demonstrate that software implementation of constant-time cryptographic primitives is not a trivial task to do and small software defects may introduce serious vulnerabilities that might go unnoticed for very long periods of time.

1.3 Contributions

The work presented in this document started as an overview and analysis of side-channel attacks and then evolved into a cache-timing attack implementation, exploiting a software defect in the OpenSSL implementation of the DSA algorithm. Several sections of Chapter 5 and Chapter 6 presented here are a collaborative work among researchers at Aalto University, Tampere University of Technology and University of Adelaide. Therefore, the wording in those chapters is switched to the plural form. The author of this work made several contributions to the collaborative work, including timing-measurement process, spy process, protocol client, software fix, responsible disclosure.

1.4 Structure of the Thesis

The thesis is structured as follows: Chapter 2 presents the background information fundamental to understand cache-timing attacks, the topics presented include cryptography and protocols relevant for this work, the memory hierarchy, covert channels and categorization of implementation attacks. Chapter 3 goes deeper into cache-timing attacks by explaining some of the techniques used to exploit and analyze cache side-channel information. Chapter 4 discusses related publications relevant to cache-timing attacks and relevant to this work in general. Chapter 5 demonstrates how a software defect introduced in OpenSSL's DSA code allowed to exploit both, the DSA algorithm and the protocols using it. Chapter 6 shows the results of the experiments performed during this work. Chapter 7 discuses challenges, mitigation techniques, disclosure of the vulnerability and future work. Chapter 8 examines the outcome and the lessons learned in this work. Appendix A contains the code provided to OpenSSL, LibreSSL and BoringSSL to fix the software defect.

Chapter 2

Background

This chapter summarizes the background information, concepts, algorithms and techniques used during the analysis, development and evaluation of the present work. Section 2.1 discusses signature schemes. Section 2.2 explains the *discrete logarithm* (DL) problem and the assumptions associated to the problem for digital signature schemes. Section 2.3 explains in detail the *Digital Signature Algorithm* (DSA) signature scheme — parameter generation, key generation, signing and verification. Section 2.4 explains the *Sliding Window Exponentiation* (SWE) algorithm used in DSA to compute exponentiations with modular arithmetic. Section 2.5 briefly explains the communication protocols exploited in this work, detailing the information relevant for the cache-timing attack. Section 2.6 gives an overview of the memory hierarchy and additionally, it mentions some features fundamental for understanding cache-timing attacks. Section 2.7 defines the concept of covert channels and explains the cache as a covert channel. Finally, Section 2.8 discusses the categorization of cryptographic attacks.

2.1 Digital Signature Schemes

Digital signature schemes are the digital analogue to handwritten signatures and their main goal is to provide authentication, data integrity and non-repudiation. In the case of digital communications, an important use of digital signatures is to provide authentication and integrity of messages between two communicating parties. Typically, this is done through the use of cryptographic algorithms. Such algorithms use randomness and public-private key pairs to sign and verify messages.

As in handwritten signatures, the digital signatures should be impossible to fake if private information of the signature is never revealed — such as position, grip and color.

A digital signature scheme consists of four algorithms:

- A *domain parameter generation* algorithm that generates the set D of required parameters for the scheme.
- A *key generation* algorithm that takes as input the parameters D and generates the public-private key pairs (y, α).
- A *signature generation* algorithm that takes as input the set D, private key α and message m. Outputs a signature Σ.
- A *signature verification* algorithm that takes as inputs the set D, public key y, message m and signature Σ and accepts or rejects the signature.

Assuming the set of parameters D and the public key y are valid, the signature verification algorithm always accepts if Σ is generated by the signature generation algorithm using the appropriate parameters.

Ideally, a digital signature scheme must be secure. Security is difficult to define since it is a function of the goals desired to achieve. Goldwasser, Micali and Rivest (GMR) [21] offer a strong security definition for digital signatures, such definition states that a signature scheme is said to be GMR-secure if it exists an adversary who can obtain digital signatures for messages of its choice from a signer and still it is unable to forge a signature for any new message for which it has not already requested previously.

2.1.1 Domain Parameters

Public domain parameters are a set of parameters (p, q, g) associated with a key-pair and these parameters are used in signature or encryption schemes. Domain parameters are called public because generally they are shared by many entities, and depending on the application and implementation, they may be specific to each entity. The parameters should be, and generally they are, chosen in a mathematical group where the discrete logarithm problem and the associated mathematical assumptions are hard to break.

In general, Hankerson et al. [25] [P. 9 Sec 1.2.2] define the parameters as follows:

- p is a prime integer.
- q is a prime divisor of $p - 1$.
- $g \in [1, p - 1]$ is a generator of order q, where q is the smallest positive integer satisfying $g^q = 1 \pmod{p}$.

Algorithm 1 shows the procedure to generate correct domain parameters.

Algorithm 1: DL domain parameter generation

```
Input: Security parameters $l, t$.
Result: DL domain parameters $(p, q, g)$
begin
    $p \leftarrow l$-bit prime;
    $q \leftarrow t$-bit prime such that $q$ divides $p - 1$
    $h \leftarrow [1, p - 1]$
    $g \leftarrow h^{(p-1)/q} \bmod p$
    while $g = 1$ do
        $h \leftarrow [1, p - 1]$
        $g \leftarrow h^{(p-1)/q} \bmod p$
    return$(p, q, g)$
```

2.2 Discrete Logarithm Problem and Related Problems

The security of all *discrete logarithm* (DL) schemes — such as ElGamal and DSA, is based in the DL problem. The hardness of this number-theoretical problem has proved to be essential for the security of public-key cryptographic schemes and therefore, allowing secure communications on the Internet.

The DL problem defined over multiplicative cyclic groups states that given a multiplicative cyclic group G, a generator g of the group and an element $h \in G$, find the integer $\alpha \in G$ such that $g^{\alpha} = h$. The integer α is the discrete logarithm of h to the base g, or more commonly defined as $\alpha = log_g\ h$.

The hardness of the DL problem provides some level of security for the cryptographic schemes but it is not sufficient, for that reason, additional assumptions related to the DL problem are used to prove the security of different cryptographic schemes.

2.2.1 Computational Diffie-Hellman Assumption

The *Computational Diffie-Hellman* (CDH) assumption states that for a cyclic group G of order q, given a triplet (g, g^a, g^b) for a randomly chosen generator g and random integers $a, b\ \in \{0, ..., q - 1\}$ it is computationally intractable to compute g^{ab}.

2.2.2 Decisional Diffie-Hellman problem.

The *Decisional Diffie-Hellman* (DDH) assumption states that given two elements g^a, g^b for independent uniformly chosen $a, b \in \mathbb{Z}_q$, the integer g^{ab} is computationally indistinguishable from g^c for a randomly and independently chosen $c \in \mathbb{Z}_q$.

Choosing the correct security parameters described in Section 2.1.1 prevents known attacks on cryptographic schemes based on the DL problem, CDH assumption and DDH assumption.

2.3 The Digital Signature Algorithm (DSA)

The Digital Signature Algorithm (DSA) is a variant of the ElGamal signature scheme [19], DSA was first proposed by the U.S. National Institute of Standards and Technology (NIST). DSA uses the multiplicative group of a finite field. This work uses the following notation for the DSA.

2.3.1 DSA Parameters

Primes p, q such that q divides $(p-1)$, a generator g of multiplicative order q in $GF(p)$ and an approved hash function h — e.g. SHA-1, SHA-256.

2.3.2 DSA Private-Public Key Pairs

The private key α is an integer uniformly chosen such that $0 < \alpha < q$ and the corresponding public key y is given by $y = g^{\alpha} \bmod p$. Calculating the private key given the public key requires solving the DL problem and for correctly chosen parameters, this is an intractable problem.

Algorithm 2 demonstrates the procedure for generating valid DSA key-pairs. Note that this procedure generates only a key-pair therefore it should be performed by each of the parties involved.

2.3.3 Signing

A given party, Alice, wants to send a signed message m to Bob — the message m is not necessarily encrypted. Using her public-private key pair $\{\alpha_A, y_A\}$, Alice performs the Algorithm 3 to sign the message m and attaches the signature (r, s) to the original message m. At the end Alice sends to Bob (m, r, s).

Algorithm 2: Key Generation for DSA

```
    Input: DSA domain parameters $(p, q, g)$.
    Result: DSA key pair $(\alpha, y)$
1   begin
2       $\alpha \in_R [1, q-1]$ ;
3       $y \leftarrow g^{\alpha} \bmod p$
4       return($\alpha, y$)
```

Algorithm 3: DSA Signature Generation

```
    Input: Message $m$, private key $\alpha_A$, domain parameters $(p, q, g)$,
           secure hash $H$.
    Result: DSA signature $(r, s)$
1   begin
2       $k \in_R [1, q-1]$ ;
3       $r \leftarrow (g^k \bmod p) \bmod q$ ;
4       if $r = 0$ then
5           goto 1
6       $h \leftarrow H(m)$;
7       $s \leftarrow k^{-1}(h + \alpha_A r) \bmod q$;
8       if $s = 0$ then
9           goto 1
10      return($m, r, s$)
```

2.3.4 Verifying

Bob wants to be sure the message m he received comes from Alice — a valid signature gives a strong evidence of authenticity. Bob performs the procedure in Algorithm 4 to verify Alice's signature.

2.3.5 DSA in Practice

Putting it mildly, there is no consensus on key sizes, and furthermore keys seen in the wild and used in ubiquitous protocols have varying sizes—sometimes dictated by existing and deployed standards. For example, NIST defines 1024-bit p with 160-bit q as "legacy-use" and 2048-bit p with 256-bit q as "acceptable" [6]. This two parameter sets are the focus on this work.

Algorithm 4: DSA Signature Verification

```
Input: Message $m$, public key $y_A$, domain parameters $(p, q, g)$, secure
       hash $H$.
Result: Accept or Reject DSA signature.
begin
    if $0 < r < q$ and $0 < s < q$ then
        $h \leftarrow H(m)$;
        $w \leftarrow s^{-1} \bmod q$;
        $u_1 \leftarrow hw \bmod q$;
        $u_2 \leftarrow rw \bmod q$;
        $r' \leftarrow (g^{u_1} y_A^{u_2} \bmod p) \bmod q$ ;
        if $r = r'$ then
            return Accept;
        else
            return Reject;
    else
        return Reject;
```

SSH's Transport Layer Protocol[1] lists DSA key type `ssh-dss` as "required" and defines r and s as 160-bit integers, implying 160-bit q. In fact, the OpenSSH tool `ssh-keygen` defaults to 160-bit q and 1024-bit p for these key types, not allowing the user to override that option, and using the same parameters to generate the server's host key. It is worth noting that recently as of version 7.0, OpenSSH disables host server DSA keys by a configurable default option[2], but of course this does not affect already deployed solutions.

As a countermeasure to previous timing attacks, OpenSSL's DSA implementation pads nonces by adding either q or $2q$ to k, the padding guarantees a fixed bit size exponent. The claim is the following

$$\begin{aligned} g^k = & \ g^{k+q} \\ = & \ g^{k+2q} \\ = & \ g^k g^q \\ = & \ g^k g^{2q} \end{aligned}$$

[1] https://tools.ietf.org/html/rfc4253

[2] http://www.openssh.com/legacy.html

The original signing algorithm computes $r = (g^k \bmod p) \bmod q$, recall

$$\begin{aligned} h = &\quad [1, p-1] \\ g = &\quad h^{(p-1)/q} \bmod p \\ g^q = &\quad h^{\frac{q(p-1)}{q}} \bmod p \\ = &\quad h^{(p-1)} \bmod p \end{aligned}$$

By Fermat's little theorem,

$$h^{(p-1)} \bmod p \quad = \quad 1 \bmod p$$

Therefore,

$$\begin{aligned} r = &\quad (g^k \bmod p) \bmod q \\ = &\quad (g^k g^q \bmod p) \bmod q \\ = &\quad (g^k g^{2q} \bmod p) \bmod q \end{aligned}$$

For the DSA signing algorithm, Step 2 is the performance bottleneck and the exponentiation algorithm used will prove to be of extreme importance to later collect side-channel information in Section 5.2.

Implementations that fail to produce random nonces or that reuse the nonce k, are vulnerable to the recovery of the secret key. Given two different signatures (r, s_A), (r, s_B) using the same secret nonce k, generated from two different messages — therefore $H(m_A) \neq H(m_B)$, it is possible to compute k as follows:

$$\begin{aligned} s_A &= k^{-1}(H(m_A) + \alpha r) \bmod q \\ s_B &= k^{-1}(H(m_B) + \alpha r) \bmod q \end{aligned}$$

Subtracting the two signatures,

$$\begin{aligned} s_A - s_B = &\quad k^{-1}(H(m_A) + \alpha r) - k^{-1}(H(m_B) + \alpha r) \bmod q \\ = &\quad k^{-1}(H(m_A) + \alpha r - H(m_B) - \alpha r) \bmod q \\ = &\quad k^{-1}(H(m_A) - H(m_B)) \bmod q \end{aligned}$$

Thus,

$$k = \frac{(H(m_A) - H(m_B))}{(s_A - s_B)} \bmod q$$

Successfully recovering the nonce k using only two signatures. Now, once the nonce k is known and given a signature (m, r, s), it is trivial to recover the secret key α using the following formula.

$$\alpha \quad = \quad r^{-1}(sk - H(m)) \bmod q$$

2.4 DSA's Sliding Window Exponentiation

Sliding window exponentiation (SWE) is a widely implemented software method to perform integer exponentiations — featured alongside other methods in the OpenSSL codebase. SWE is fairly popular due to its performance since it reduces the amount of pre-computation needed and, moreover, reduces the average amount of multiplications performed during the exponentiation.

An exponent e is represented and processed as a sequence of windows e_i, each of length $L(e_i)$ bits. Processing the exponent in windows reduces the amount of multiplications at the cost of increased memory utilization since a table of pre-computed values is used.

A window e_i can be a zero window represented as a string of "0"s or non-zero window represented as a string starting and ending with "1"s and such window is of width w — determined in OpenSSL by the size in bits of the exponent e. The length of non-zero windows satisfy $1 \leq L(e_i) \leq w$, thus the value of any given non-zero window is an odd number between 1 and $2^w - 1$.

As mentioned before, the algorithm pre-computes values and stores them in a table to be used later during multiplication operations. The multipliers computed are $b^v \bmod m$ for each odd value of v where $1 \leq v \leq 2^w - 1$ and these values are stored in table index $g[i]$ where $i = (v-1)/2$. For example, with the standard 160-bit q size, OpenSSL uses a window width $w = 4$ and with the 256-bit key size OpenSSL uses a window width $w = 5$. The algorithm pre-computes multipliers $b^1, b^3, b^5, ..., b^{15} \bmod m$ and stores them in $g[0], g[1], g[2], ..., g[7]$, respectively.

Using the sliding window representation of the exponent e, Algorithm 5 computes the corresponding exponentiation through a combination of squares and multiplications in a left-to-right approach. The algorithm scans every window e_i from the *most significant bit* (MSB) to the *least significant bit* (LSB).

For any window, a square operation is executed for each bit and additionally for a non-zero window, the algorithm executes an extra multiplication when it reaches the LSB of the window.

For novel reasons explained later in Section 5.1, the side-channel part of this work focuses on this algorithm. Specifically, in getting the sequence of squares and multiplies (SM) performed during its execution. Then partial information extracted from the SM sequence is later used in the lattice attack.

Algorithm 5: Sliding-window exponentiation.

```
Input: Window size $w$, base $b$, modulo $m$, N-bit exponent $e$
       represented as $n$ windows $e_i$, each of length $L(e_i)$.
Output: $b^e \bmod m$.
// Pre-computation
$g[0] \leftarrow b \bmod m$;
$s \leftarrow \mathrm{MULT}(g[0], g[0]) \bmod m$;
for $j \leftarrow 1$ to $2^{w-1}$ do
    $g[j] \leftarrow \mathrm{MULT}(g[j-1], s) \bmod m$;
// Exponentiation
$r \leftarrow 1$;
for $i \leftarrow n$ to $1$ do
    for $j \leftarrow 1$ to $L(e_i)$ do
        $r \leftarrow \mathrm{MULT}(r, r) \bmod m$;
    if $e_i \neq 0$ then $r \leftarrow \mathrm{MULT}(r, g[(e_i - 1)/2]) \bmod m$;
return $r$;
```

2.5 Protocols

The essence of Internet relies in the communication between two or several remote entities. Some classic examples of the use of Internet include remote system administration, file transfer, communication between clients and servers, banking services, etc. Nevertheless, the Internet does not provide any security guarantee or data protection for the information transmitted in it, and since it is naïve to rely in the good faith of the entities involved in each communication, security protocols were developed on top of the Internet to provide information security.

Nowadays, communication protocols are assumed to be safe and secure to use in the Internet but their security still depends on the security of their individual components, which includes software libraries and cryptographic libraries — e.g. OpenSSL, LibreSSL, BoringSSL.

Several communication protocols exist and are actively used on the Internet and new protocols are developed every year but two of the most widely used, widespread and relevant protocols for this work are TLS and SSH.

2.5.1 SSH

The SSH protocol[3] is a packet-based binary protocol working on top of the transport layer (e.g. TCP/IP) that provides secure login connections and secure file transfer over an untrusted network. SSH uses cryptographic algorithms to implement four basic security features: authentication, key exchange, encryption and integrity.

In SSH, at the beginning of every key exchange a random cookie is sent by both parties, additionally in every transmitted packet, random padding and integrity protection data are added at the end.

During the initial phase of communication both parties negotiate: the cipher algorithms used for data encryption, the *Message Authentication Code* (MAC) algorithms used for data integrity, the key exchange (KEX) methods used for one-time session key derivation, the public key algorithms for authentication and the compression algorithms for data compression.

In general, the SSH protocol performs the following phases to start a session:

1. The client opens a connection with the server.
2. The client and server negotiate cryptographic algorithms and additional parameters.
3. The server sends its public host key for authentication. The client accepts or rejects the public host key according to its own criteria and its own database of known host keys. Note that most of the time the client accepts the public host key and it adds an entry in its database of known host keys.
4. The client and the server derive the session keys.
5. The client is authenticated to the server, the supported methods include: password, public key, host-based and none.
6. The client requests features needed for its session – e.g. X11 forwarding, TCP/IP forwarding.
7. The interactive session starts.

In the SSH protocol, at any point during session establishment, the server can terminate the connection and notify the user if it detects an attempt of data tampering or if it receives a malformed packet.

[3] `http://www.ietf.org/rfc/rfc4253.txt`

Steps 1, 2 and 3 are relevant for this work since they represent the target of the attack described later on. The attack performed against OpenSSH is explained in detail in Section 5.3.2.

2.5.2 TLS

The TLS protocol[4] is a cryptographic protocol used to provide privacy and data integrity between two communicating parties. The TLS protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake protocol. Similar to SSH, TLS is built on top of a reliable transport protocol — e.g. TCP/IP.

The TLS Record Protocol provides data encryption and message integrity. The TLS Handshake Protocol allows the client and the server to authenticate each other and negotiate the cryptographic algorithms to use before any application data is transmitted. In short, three basic features are provided by the TLS Handshake Protocol to the communicating parties: authentication, secure key exchange and reliable negotiation.

The TLS Handshake Protocol involves the following steps:

1. Parties agree on algorithms by exchanging hello messages. Random nonces and session resumption happens at this stage too.

2. Parties exchange cryptographic parameters to agree on a pre-master secret.

3. Parties exchange certificates and cryptographic information to authenticate themselves.

4. Parties compute a master secret using the pre-master information and the random nonces.

5. Parties negotiate security parameters to the record layer.

6. Parties verify the handshake occurred without tampering and security parameters calculated are correct.

The TLS protocol is a complex protocol and therefore this work only deals with the TLS Handshake protocol, which is later exploited in Section 5.3.1.

[4] `https://tools.ietf.org/html/rfc5246`

2.6 Memory Hierarchy

Unfortunately, the amount of fast memory available in a system is relatively small compared to the total amount of memory available, therefore a memory hierarchy is needed to get the best trade-off between cost and performance. The *principle of locality* [26] says that a typical program spends 90% of its execution time in 10% of the code, therefore is possible to predict with reasonable accuracy the instructions the program will execute in the near future. Usually two different types of locality are observed in the memory hierarchy:

- *Temporal locality*: recently accessed items in memory are very likely to be accessed again in the near future.
- *Spatial locality*: contiguous memory addresses tend to be referenced together in time.

The principle of locality and the technology available at hardware level have led to complex memory hierarchies with different memory sizes and speeds.

Since fast memory is expensive and its efficient use is highly desirable, a memory hierarchy composed of several levels is used. Typically, each level is smaller, faster and more expensive than the next lower level. These smaller and faster memories are called *caches*.

Caches exploit the spatial and temporal locality, mentioned before, to access data and code instructions as fast as possible, thus allowing the processors to operate at extremely fast speeds, increasing the overall performance of a system.

2.6.1 Cache architecture

In modern processors the memory hierarchy is structured as follows: higher-level caches located closer to the processor core, are smaller and faster than low-level caches, which are located closer to main memory. Intel's architecture [17] has three levels of cache: L1, L2 and *Last-Level Cache* (LLC).

As depicted in Section 2.1, each core has two L1 caches, a data cache (dcache) and an instruction cache (icache), each small in size (32 KB) with a short access time (4 cycles). The L2 caches are, typically, core-private and they have an intermediate size (256 KB) with intermediate access times (11 cycles). The LLC is shared among all of the cores and it is a unified cache, containing both data and instructions. Typical LLC sizes are in the order of megabytes and access time is quite slow compared to the rest of caches (40 cycles).

The unit of memory and allocation in a cache is called *cache line*. Cache lines are of a fixed size B, typically, 64 bytes. The $\lg(B)$ low-order bits of a memory address, called *line offset*, are used to locate the datum in the cache line.

When a memory address is accessed, the processor checks the availability of the address line in the top-level L1 caches. If the data is found there then it is served to the processor, a situation referred to as a *cache hit*. In a *cache miss*, when the data is not found in the L1 caches, the processor repeats the search for the address line in the next cache level and continues through all the caches. Once the address line is found, the processor stores a copy in the cache for future use — temporal locality.

Most caches are *set-associative* and they are composed of S *cache sets*, each containing a fixed number of cache lines. The number of cache lines in a set is the cache *associativity* — i.e. a cache with W cache lines in each cache set is a W-way *set-associative* cache.

Since the main memory is orders of magnitude larger than the cache, more than W memory lines may map to the same cache set. If a cache miss occurs and all the cache lines in the matching cache set are in use, one of the cached lines is evicted, freeing a slot for a new line to be fetched from a lower-level memory. Several *cache replacement policies* exist to determine the cache line to evict when a cache miss occurs but the typical policy in use is an approximation to the *least-recently-used* (LRU).

Modern Intel processors maintain a well-defined relationship between levels of cache by using the *inclusive* property. This property ensures the L_{i+1} cache contains a superset of the contents of the L_i cache, therefore, flushing or evicting data from a lower-level cache also removes data from all other cache levels of the processor. In other words, when data is evicted from the LLC, it is also evicted from all of the other levels of cache in the processor.

The LLC of modern Intel processors, starting from the Sandy Bridge microarchitecture, uses a more complicated architecture to improve the performance. The LLC is shared among all the cores but it is divided into per-core *slices* connected by a ring bus. The slices are separate caches but the bus ensures that each core can access the full LLC.

To distribute the data uniformly in the LLC, Intel uses a hash function which maps the memory address (excluding the line offset bits) into the *sliced id*. In the LLC every cache set is uniquely identified by the slice id and the set index.

Intel architecture implements several cache optimizations. The spatial pre-fetcher pairs cache lines and attempts to fetch the pair of a missed

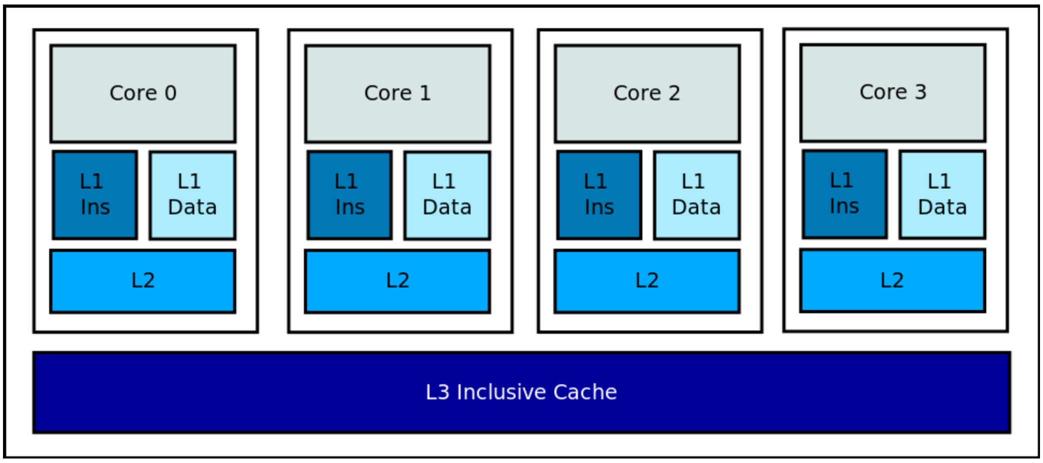

Figure 2.1: Intel Sandy Bridge Cache Architecture.

line [17]. Consecutive accesses to memory addresses are detected and prefetched when the processor anticipates they may be required [17]. Additionally, when the processor is presented with a conditional branch, *speculative execution* brings the data of both branches into the cache before the branch condition is evaluated [47].

In 2002, Page [39] noted that tracing the sequence of cache hits and misses of a software may leak information on the internal working of the software, including information that may lead to recovering cryptographic keys.

2.6.2 Cache Replacement Policies

In the event of a cache miss, old data must be evicted to make room for the new data. The logic algorithm that helps to determine which data to evict is called the *cache replacement policy*. The most common and frequently used cache replacement policy is an approximation to the *Least Recently Used* (LRU) policy. As the name implies, the cache controller keeps age fields for every cache line and when a cache line needs to be evicted, the cache line with the oldest age value (or least recently used) is evicted. An approximation to this LRU policy is used in common workstations and in the experiments performed in this work.

Additional cache replacement policies exist and they are used under specific scenarios, just to name a few different policies. *Most Recently Used* (MRU) evicts the cache line with the youngest age value. *Least Frequently Used* (LFU) evicts the cache line that is used least often. And finally, *Random* evicts a random cache line from the cache.

2.6.3 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a common technique used to protect the memory address space against buffer overflow and arbitrary code execution attacks [46]. ASLR aims to prevent an attacker from using the attack code to exploit the same flaw in multiple systems, this is usually done by adding a random offset to the base address.

Since the LLC is physically tagged, and it has no dependencies on the virtual address space, the present work and the example explained in Chapter 5 does not have to deal with virtual to physical address mapping, thus the attack is oblivious to this technique [52].

2.7 Covert Channels

According to Latham [32] in *The Orange Book*, covert channels are communication channels that can be exploited to transfer information, violating the system's security. Covert channels transfer information in unconventional ways, usually, exploiting system features that were not intended to be used as a mechanism to transfer data.

Covert channels can be exploited by users, programs or processes and most of the time covert channels are unintentional, a system user is unaware of their existence and a malicious party monitors the channel to retrieve information about the system user. On the other hand, covert channels can be used intentionally to communicate unregulated information through a system.

Covert channels are an old issue that became relevant due to the *confinement problem.* The confinement problem presented by Lampson [31] states that any program in confinement should be unable to leak any data except to its owner. In the same work, Lampson mentions that reducing to zero the information leaked requires "far-reaching" measures due to the impracticality of confined programs.

Covert channels are generally divided into two groups, storage channels and timing channels.

- Storage channels rely on system variables set by the party leaking the information. Such variables can be ordering, threshold and interlocks.
- Timing channels rely on a clock as a reference to measure the time it takes for an event to complete.

Timing channels are relevant for this work because they represent the foundations of side-channel attacks, including cache-timing attacks.

2.7.1 Memory Cache as a Covert Channel

Microprocessors using caches in the memory hierarchy introduce a covert timing channel. Since the caches are a resource shared among all the processes running in a system, they are able to communicate unintended information among each other.

A simple example is two processes, process A and process B, sharing a memory hierarchy. Process A reads enough information from main memory to fill up the cache. Then process B reads some "secret" information from main memory, process B will evict some of the content from the cache to include its information. Finally, process A re-reads its information, since some of its data was evicted from the cache, it will take a noticeable longer time to reload information from main memory than information from cache, thus introducing a timing channel.

As can be seen, this covert channel is introduced by the memory hierarchy at the hardware level and little can be done to prevent this covert channel since the goal of caches is to provide faster information retrieval compared to main memory.

2.8 Cryptographic Attacks

Typical methods of classical cryptanalysis are *linear* and *differential cryptanalysis* [9] but more recently *implementation attacks* started to take an important role in cryptanalysis as new attack vectors were discovered on the actual implementations.

2.8.1 Implementation Attacks

In contrast to classic cryptanalytic techniques where the attackers search for vulnerabilities in the mathematical properties and structure of the algorithms, *implementation attacks* target a physical or concrete implementation of a cryptographic primitive. Several implementation attacks are used depending on the implementation and the environment where cryptography is being used and according to Popp [42] the implementation attacks can be generally classified in two ways.

1. **Passive vs. Active**
   Passive attacks use information emitted by a cryptographic device under normal environment and standard operational specifications. On the other hand, in active attacks the environment is manipulated to

cause abnormal behavior of the cryptographic device and exploit this behavior.

2. **Non-invasive vs. Invasive**
   In non-invasive attacks only the emitted data is exploited, such as running time or power consumption. An invasive attack manipulates and accesses the components of the cryptographic device, for example, depackaging a chip, probing, rerouting wires, etc.

In most of the cases, passive and non-invasive attacks are easier and cheaper to conduct as the cryptographic devices are not permanently altered nor damaged during the attack.

2.8.2 Side-Channel Attacks

Side-channel attacks are passive, non-invasive, implementation attacks where the side channels are exploited to recover the secret key of the cryptographic algorithms. The most common and classic side channels exploited include execution time, power consumption and electromagnetic (EM) radiation, but are not limited to only those, sound and temperature are also considered. Almost any type of output from the cryptographic devices can be considered as a side channel even if such outputs are not hardware-based as mentioned before.

The work presented here deals with cache-based side-channel attacks [22] and in this type of attack the attacker passively monitors the CPU activity of a system. The threat model can be summarized as a user A which is a *spy* process running concurrently with user B, the *victim* process. Typically, A and B run in user space, where they have separate virtual memory space that cannot be accessed by other processes – this provides security by preventing manipulation from malicious processes. Nevertheless, the operating system shares resources such as libraries and cache memory between users to operate efficiently, allowing to perform side-channel attacks. An example scenario is the following, the *spy* process A monitors specific changes in a shared resource between the processes A and B – e.g. shared library, memory. While the *victim* B executes a cryptographic algorithm, the spy A records the changes in the shared resource and later, an attacker using the side-channel information collected by A correlates the recorded trace with the cryptographic algorithm executed by victim B to obtain secret information. This threat model assumes the attacker has access to the system where the victim process B is running to execute the spy process A, otherwise the attack is impossible to perform.

Chapter 3

Cache-Timing Attacks

Timing attacks are a specific type of side-channel attack where the main goal is to recover the private key of a cryptosystem by measuring the execution time, exploiting the implementations that do not run in constant time, mainly due to conditional branches in the algorithm. Slight differences in the input values make basic arithmetic operations and logic in cryptographic algorithms to run in non-constant time, revealing information about the algorithm and the state of the system.

Cache-timing attacks use the processor's cache memory as a side-channel to retrieve timing measurements of cryptographic algorithms which are key-dependent. Kocher [30] performed the first side-channel timing attack in a public key cryptosystem, his target was the right-to-left square-and-multiply implementation of the exponentiation algorithm in the RSA cryptosystem. For this example, the attacker's task is to time and trace the execution of the square operations and multiply operations that are key-dependent. Since the execution time for every iteration of the algorithm is different when the exponent bit is 1 (e.g. square-and-multiply) and when the exponent bit is 0 (e.g. square), the attacker is able to retrieve the exponent used in every step of the algorithm — which is the secret key for many public-key cryptosystems.

Cache-timing attacks can be generally categorized as time-driven, trace-driven and access-driven attacks. In time-driven attacks the attacker relies in the total execution time of the cryptographic computation — e.g encryption and signing. The timing information of the whole operation reveals the number of cache hits and cache misses during the execution of the algorithm. In trace-driven attacks, the attacker profiles the cache during a cryptographic operation to determine cache hits and cache misses. And in access-driven attacks, the attacker is able to determine which cache sets were accessed during an operation, useful for cryptosystems where lookup tables are used.

The three categories of cache-timing attacks are similar but used for different purposes, depending on the cryptographic primitive and the scenario a different approach may be used. Nevertheless, the three categories exploit the fact that accessing cached data is much faster compared to accessing data in the main memory.

3.1 Cache-Timing Techniques

Attacking a particular cryptographic primitive requires knowledge of the inner workings of the primitive and that knowledge leads to different approaches to achieve a successful cache-timing attack. The following sections present some of the techniques used to perform time-driven, access-driven and trace-driven attacks. The attack performed in this work and detailed in Chapter 5 is a trace-driven attack, therefore the focus is on Section 3.1.3 where the Flush+Reload technique is explained in detail.

3.1.1 The Evict+Time Technique

The Evict+Time technique was first discussed by Osvik et al. [38] as a cache-timing technique against AES. With the knowledge of the cache memory as a covert channel and the variations on the cache behavior, the authors introduced this time-driven cache-timing attack.

The Evict+Time technique is a side-channel cache-timing technique that requires the manipulation of the cache state before each encryption and then observes the execution time of the subsequent encryption. Additionally, this technique makes some reasonable assumptions, on one hand it assumes the ability to distinguish the beginning and end of an encryption, as well as the ability to trigger encryptions at will. On the other hand, since it is a cache-based attack, it assumes knowledge of relevant memory addresses used by the cryptographic primitive.

In a chosen-plaintext setting and using a plaintext p the Evict+Time technique works as follows:

1. *Trigger* an encryption of p in the target process.

2. *Evict* memory by accessing appropriate memory lines in the attacker's process.

3. *Trigger* a second encryption of p and time it.

The rationale behind this technique, and applied against AES, is the following: (1) in the first step all the AES lookup tables accessed by the target process during the encryption of p are cached, (2) then the attacker accesses memory blocks in its own memory space, such memory blocks happen to be mapped to the same cache sets as the target's memory blocks, therefore, completely replacing the prior content of the cache. (3) Finally, timing the encryption of the plaintext reveals information about AES tables thanks to the timing differences between cache hits and cache misses.

This attack can be extended to scenarios where the attacker triggers known but not chosen plaintexts, allowing it to narrow down the possible values of the key.

3.1.2 The Prime+Probe Technique

The Prime+Probe technique is an access-driven cache-timing technique proposed by Osvik et al. [38]. This technique resembles the Evict+Time technique and it is used in the same work by the authors to attack the same AES implementation. Prime+Probe differs in the sense that the attacker no longer uses the encryption time as a measurement score to determine the key but instead the state of the cache after encryption is examined.

Following a chosen-plaintext setting and using a plaintext p, the Prime+Probe technique works as follows:

1. *Prime* by filling up the cache.
2. *Trigger* an encryption of p.
3. *Probe* by reading memory addresses and measuring the reading time.

In step 1, the attacker fills the cache with its own data. The encryption in step 2 causes partial eviction of some memory lines and in step 3 the attacker probes each cache set to check if its own data is still present in the cache after the encryption. The cache sets used during encryption cause cache misses and the cache sets untouched during the encryption are cache hits, using this timing difference the attacker can recover the key.

The benefit of Prime+Probe over Evict+Time is the measurement resolution due to timing variance. In the Prime+Probe technique the attacker times operations performed by itself — e.g. reading the cache. While in the Evict+Time technique, the attacker relies on operations performed by the target — e.g. encryption and signing, which include overheads and might be noisy.

3.1.3 The Flush+Reload Technique

Often the main target of side-channel attacks is the L1 cache due to the close proximity to the cores. The initial techniques discussed previously (Evict+Time and Prime+Probe) are effective and possible due to the small sizes of L1 caches, but those techniques are limited. The possible attacks are limited because in order to perform a successful attack, the spy and the victim processes must run in the same execution core of the processor. To overcome this limitation, Yarom and Falkner [52] developed the Flush+Reload technique. This technique is a side-channel cache-based technique that targets the LLC in the Intel x86 and x86-64 processors.

The cache-timing attack discussed in this work is based in the Flush+Reload [24, 52] attack, which is a cache-based side-channel attack technique. Unlike the earlier Prime+Probe technique [38, 40] that detects activity in cache sets, the Flush+Reload technique identifies access to memory lines, giving it a high resolution, high accuracy and high signal-to-noise ratio.

Like Prime+Probe, Flush+Reload relies on cache sharing between processes. Additionally, it requires data sharing, which is typically achieved through the use of shared libraries or using page de-duplication [4, 48].

Algorithm 6 shows a round of the attack. A round of the attack consists of three phases:

Algorithm 6: Flush+Reload Technique

```
Input: Memory Address addr.
Result: True if the victim accessed the address.
begin
    flush(addr)
    Wait for the victim.
    time ← current_time()
    tmp ← read(addr)
    readTime ← current_time() - time
    return readTime < threshold
```

1. In the first phase, to identify victim access to a shared memory line, the attacker evicts the monitored memory line from the cache.

2. In the second phase, the attacker waits a period of time so the victim has time to access the memory line.

3. In the third phase, the attacker reloads the memory line and measures the time it takes to load.

If during the second phase the victim accesses the memory line, the line will be available in the cache and the reload operation in the third step will take a short time. If, on the other hand, the victim does not access the memory line then the third step takes a longer time as the memory line is loaded from main memory. Figure 3.1 (A) and (B) illustrate a round of the attack with and without victim access.

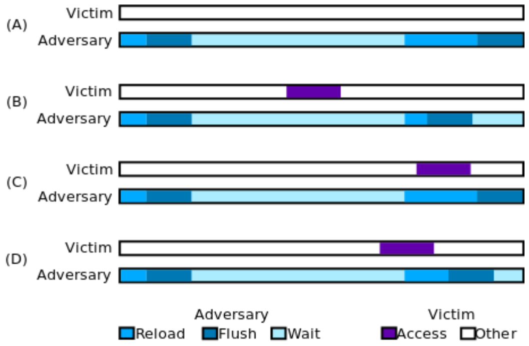

Figure 3.1: Timing of Flush+Reload. (A) No Victim Access (B) With Victim Access (C) With Victim Access Overlap (D) Partial Overlap.

The execution of the victim and the adversary processes are independent from each other, thus synchronization of probing is important and several factors need to be considered when processing the side-channel data. Some of those factors are the waiting period for the adversary between probes, memory lines to be probed, size of the side-channel trace and *cache-hit* threshold. Selecting the appropriate parameters make it possible to detect when the attacker and the victim partially overlap or completely overlap in a round of attack as depicted in Figure 3.1 (C) and (D).

One important goal for this attack is to achieve the best resolution possible while keeping the error rate low and one of the ways to achieve this is by targeting memory lines that occur frequently during execution, such as loop bodies. Several processor optimizations are in place during a typical process execution and an attacker must be aware of these optimizations to filter them during the analysis of the attack results.

A typical implementation of the Flush+Reload attack makes use of the `clflush` instruction of the x86 and x86-64 instruction sets. The `clflush` instruction evicts a specific memory line from *all* the cache hierarchy and since it is an unprivileged instruction, it can be used by any process.

The Flush+Reload attack is possible thanks to the *inclusion* property of the LLC. Whenever a memory line is evicted from the LLC, the processor also evicts the line from L1 and L2 caches. Yarom and Falkner [52] report that the Flush+Reload attack does not work on AMD processors due to their use of non-inclusive LLCs.

3.1.4 The Spy Process

The code implemented for this work is in Figure 3.2. The code is really simple, it only measures the time it takes to read data from a memory address and then evicts the content of the memory line from the cache.

```
 1  mfence
 2  lfence
 3  rdtsc
 4  lfence
 5  mov %rax, %r10
 6  mov (\addr), %rax
 7  lfence
 8  rdtsc
 9  sub %r10, %rax
10  movw %ax, \offset(%rdi, %r9, 2)
11  clflush (\addr)
```

Figure 3.2: Spy process.

The spy code starts with the `mfence` and `lfence` instructions, these instructions are used to serialize the instruction stream since usually the instructions are executed out of order or in parallel by the CPU. On one hand, the `lfence` instruction partially serialize by ensuring all the instructions preceding it have been completed before it is executed and no other instruction after it executes. On the other hand, `mfence` orders memory accesses, fence instructions and the `clflush` instruction.

After the initial fence instructions, the processor's time stamp counter is called with the `rdtsc` instruction. The `rdtsc` instruction reads the 64-bit counter, it returns the high 32-bits in the `%rdx` register and the low 32-bits in `%rax` register. Since the recorded timings are really small, in Line 5 the low 32-bits are copied to the `%r10` register. Line 6 reads the memory address `\addr` and then immediately in line 8 the counter is read again. Lines 9 and 10 compute the timing difference and store the result in the register `%rdi`. Finally in line 11 the memory address `\addr` is flushed from the cache.

Table 3.1 shows the cache parameters and access times for the Haswell Microarchitecture [17] used in this work. An important aspect to note is the access latency for the shared L3 cache, which is around 34 cycles.

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (cycles)	Access Throughput (cycles)
L1 Data	32 KB	8	64	3	1
L1 Instruction	32 KB	8	64	N/A	N/A
L2 Unified	256 KB	8	64	11	2
Shared L3	6 MB	12	64	34	2

Table 3.1: Cache Parameters of the Haswell Microarchitecture [17].

Knowing the real time (in CPU cycles) of moving data from one specific memory address to a register is extremely relevant to fine-tune the parameters used in the Flush+Reload technique, for that reason, the timings using real equipment are compared to the timings in Table 3.1. Later these timings are used to measure the timing difference of cache hits against cache misses.

To measure cache hits and cache misses, in a similar way as it would be during an actual attack, the spy process tracks a specific memory address containing the data of interest. In a cache-timing attack setup, the memory address belongs to a shared library which can be accessed by both the victim and the spy process.

A cache hit is achieved by calling the shared library, loading the memory address into the cache. Afterwards, the spy process measures the time required to reload the memory address available in cache without flushing the memory address. 90% of the memory loads from the cache took 52 cycles and the rest 10% was split between 42 and 43 cycles, as observed in Figure 3.1.4.

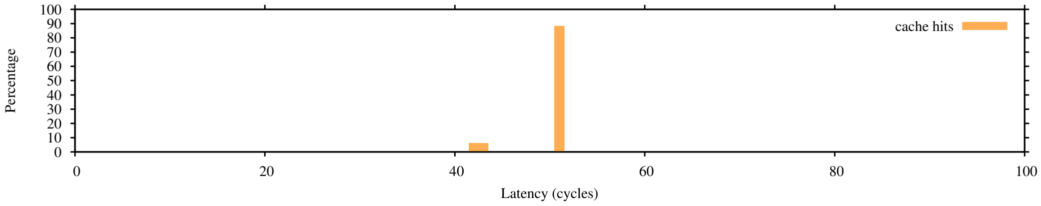

Figure 3.3: Probing time (cycles) for 8K cache hits.

Similarly, a cache hit is achieved by following the same procedure as before but this time after every measurement, the memory address is flushed from the cache using the `clflush` command. This forces the system to load the information from main memory in every iteration. Practically, 100% of the loads from main memory took at least 250 cycles, considerably more than 52 cycles.

Additionally, Intel [17] mentions that continuous cache eviction uses cache bandwidth and bus bandwidth which causes an overall degradation in cache response times, this last behavior is observed in Figure 3.1.4 where the timing results for cache misses are spread over times larger than 250 cycles and not as constant as in the case with cache hits.

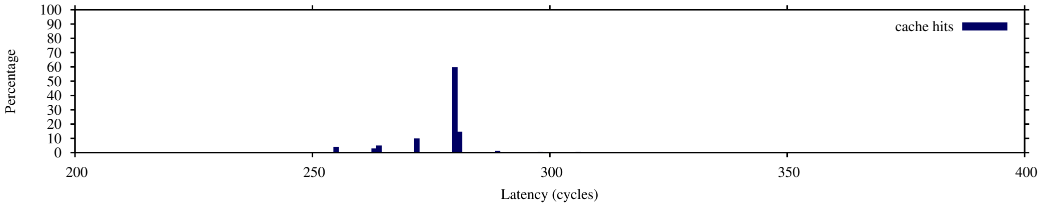

Figure 3.4: Probing time (cycles) for 8K cache misses.

Figure 3.1.4 shows the timing differences between cache hits and cache misses.

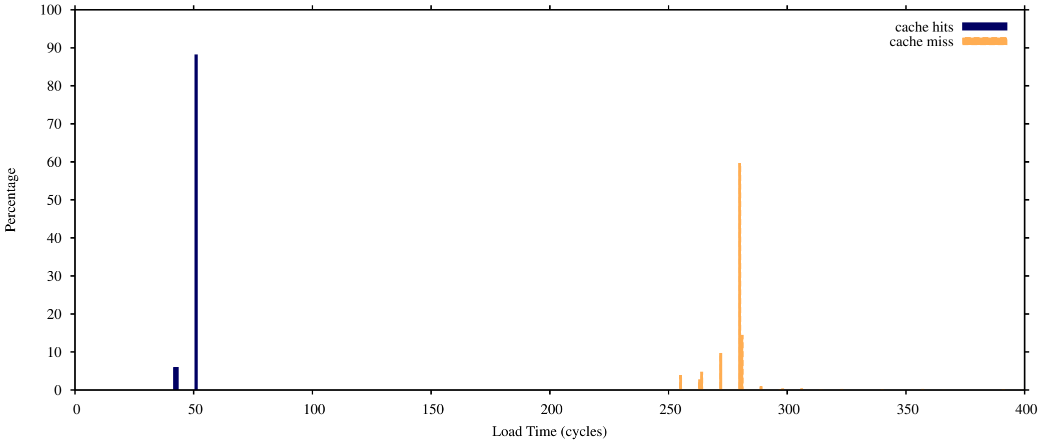

Figure 3.5: Probing time: cache hits vs. cache misses.

3.1.5 Performance Degradation Technique

To achieve maximum utilization on a single hardware platform multiple processes share the resources available to them. As a consequence the processes may interfere with each other, impacting the performance.

A malicious user can misuse the shared resources and exploit the interference to affect processes running on the same platform. These type of attacks are well known attacks and they are called *performance degradation attacks.*

Performance degradation attacks are mainly used as *denial of service* (DoS) attacks in which the attacker does not get any direct benefit, the only benefit is to damage and limit a process or service.

Allan et al. [2] showed that attackers can benefit from performance degradation attacks to amplify the side-channel attacks. By degrading the performance of a running process, an attacker can receive more side-channel information to improve the attack.

Following the idea of the FLUSH+RELOAD technique, the performance degradation attack used in this work targets pieces of code that are shared between the victim and the attacker. Frequently executed code is cached to improve execution performance, therefore access to the code is fast.

An attacker can degrade the performance by repeatedly evicting memory lines containing code that is executed frequently. By evicting such memory lines, the victim has to wait for the processor to load the code from main memory which is orders of magnitude slower than loading from cache memory.

A small number of memory lines are chosen as candidates. During execution, the chosen memory lines are repeatedly evicted from cache memory to slow down the execution of a specific program. An efficient attack is achieved by considering the amount of memory line candidates, the memory load time and the lines evicted. Memory lines that contain function calls are accessed twice during execution, one before the call and one on return.

Typically, performance degradation attacks target all of the processes running in a microprocessor. The performance degradation attack used in this work [2] only targets programs that use specific shared code segments, in this case, the OpenSSL library.

Similar to the FLUSH+RELOAD technique, the requirements for the degradation attack are a shared inclusive LLC and the ability to efficiently evict memory lines from cache.

3.1.6 The Degrading Process

The degrading process implemented for this work is shown in Figure 3.6. The degrading process is really simple and its main function is to evict memory lines from the cache.

The same `clflush` instruction used in the FLUSH+RELOAD technique is used for the performance degradation attack. The code iterates continuously

```
1:
clflush (\addr1)
clflush (\addr2)
clflush (\addr3)
jmp 1b
```

Figure 3.6: Degrading process.

through the `clflush` calls, evicting the content in the memory addresses of registers `addr1`, `addr2` and `addr3`.

As explained in Section 3.1.5 the memory addresses to evict are chosen according to the target process, for this work three addresses allow to degrade the victim process to a threshold that produces good results.

The degrading process runs in parallel with the spy and the victim processes, slowing down the performance of the victim process and allowing the spy process to get a high quality profile from the cache.

3.2 Partial key disclosure

Recall from Section 2.3.5 that in DSA the nonce k and the secret key α satisfy a linear congruence. The constants of the linear combination are specified by s, $h(m)$, and r, which, typically for a signed message, are all public. Hence, knowing the nonce k reveals the secret key α.

$$\alpha = r^{-1}(sk - H(m)).$$

Typically, side-channel leakage from SWE only recovers partial information about the nonce. The adversary, therefore, has to use that partial information to recover the key. The usual technique [1, 2, 7, 13, 14, 27, 36, 37, 41] for recovering the secret key from the partial information is to express the problem as a *hidden number problem* (NHP) [11] which is solved using a lattice technique.

3.2.1 The Hidden Number Problem

In the hidden number problem (HNP) the task is to find a hidden number given some of the MSBs of several modular linear combinations of the hidden number. More specifically, the problem is to find a secret number α given a number of triples (t_i, u_i, ℓ_i) such that for $v_i = |\alpha \cdot t_i - u_i|_q$ we have $|v_i| \leq q/2^{\ell_i+1}$, where $|\cdot|_q$ is the reduction modulo q into the range $(-q/2, \dots, q/2)$.

Boneh and Venkatesan [11] initially investigate HNP with a constant $\ell_i = \ell$. They show that for $\ell < \log^{1/2} q + \log\log q$ and random t_i, the hidden number α can be recovered given a number of triples linear in $\log q$.

Howgrave et al. [27] extend the work of Boneh and Venkatesan [11] showing how to construct an HNP instance from leaked LSBs and MSBs of DSA nonces. Nguyen and Shparlinski [36] prove that for a good enough hash function and for a linear number of randomly chosen nonces, knowing the ℓ LSBs of a certain number of nonces, the $\ell + 1$ MSBs or $2 \cdot \ell$ consecutive bits anywhere in the nonces is enough for recovering the long term key α. They further demonstrate that a DSA-160 key can be broken if only the 3 LSBs of a certain number of nonces are known. Nguyen and Shparlinski [37] extend the results to ECDSA, and Liu and Nguyen [34] demonstrate that only 2 LSBs are required for breaking a DSA-160 key. Benger et al. [7] extend the technique to use a different number of leaked LSBs for each signature.

3.2.2 Lattice attack

To find the hidden number from the triples we solve a lattice problem. The construction of the lattice problem presented here is due to Benger et al. [7], and is based on the constructions in earlier publications [11, 36].

Given d triples, a $d+1$-dimensional lattice is constructed using the rows of the matrix

$$B = \begin{pmatrix} 2^{\ell_1+1} \cdot q & & & \\ & \ddots & & \\ & & 2^{\ell_d+1} \cdot q & \\ 2^{\ell_1+1} \cdot t_1 & \dots & 2^{\ell_d+1} \cdot t_d & 1 \end{pmatrix}.$$

By the definition of v_i, there are integers λ_i such that $v_i = \lambda_i \cdot q + \alpha \cdot t_i - u_i$. Consequently, for the vectors $\mathbf{x} = (\lambda_1, \dots, \lambda_d, \alpha)$, $\mathbf{y} = (2^{\ell_1+1} \cdot v_1, \dots, 2^{\ell_d+1} \cdot v_d, \alpha)$ and $\mathbf{u} = (2^{\ell_1+1} \cdot u_1, \dots, 2^{\ell_d+1} \cdot u_d, 0)$ yield

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

The 2-norm of the vector $\mathbf{y}$ is about $\sqrt{d+1} \cdot q$ whereas the determinant of the lattice $L(B)$ is $2^{d+\sum l_i} \cdot q^d$. Hence $\mathbf{y}$ is a short vector in the lattice and the vector $\mathbf{u}$ is close to the lattice vector $\mathbf{x} \cdot B$. Solving the Closest Vector Problem (CVP) with inputs B and $\mathbf{u}$ finds $\mathbf{x}$, revealing the value of the hidden number α.

3.3 Cache-Timing Data Processing

Processing cache-timing data is not a trivial task and performing this task manually takes expertise, time and patience. Automating the analysis and processing of cache-timing data is highly desirable, since the amount of data can be huge depending on the target process. The following two techniques, Vector Quantization (VQ) and Hidden Markov Models (HMM) have been in use for several years to solve different data coding and modeling issues—e.g. data compression, pattern recognition, clustering.

Chari et al. [15] used VQ to perform power analysis, Karlof and Wagner [29] and Green et al. [23] analyzed side-channel information using HMMs. Side-channel analysis resembles data correction, where the closest approximation of the algorithm state sequence recovered from the trace to the real algorithm sequence is needed to successful recover information about the cryptographic operation.

3.3.1 Vector Quantization

Vector Quantization is a signal processing technique used to map a set of vectors from a big domain to a reduced set of vectors called the *codebook*. The *crux* of this technique is matching vectors to the closest representative in the *codebook* using the *Euclidean distance* as a metric.

Typically, VQ is used in side-channel processing due to the multidimensionality of cache-timing data. Depending on the cache-timing technique used and the target's hardware platform specifications, cache-timing data may have a big dimension—e.g. 32 or 64. As the dimension increases it is more complex to analyze and map the side-channel data to the *codebook*.

The *codebook* is produced during the profiling stage by the attacker where in a controlled setup, such as its own user space, the attacker spies the target process with known secret inputs. Using this approach the attacker is able to create a good *codebook* that yields a good guess at the target process.

VQ gives a good approximation to the sequence of states from a target, nevertheless, due to the noise and the errors of the spy process itself it is not sufficient to recover the original sequence of states which depends on the algorithm running in the target process. Therefore, the HMM technique is the next step to infer the algorithm state.

3.3.2 Hidden Markov Models

Hidden Markov Models (HMMs) are statistical models for modeling discrete-time stochastic processes and they are useful to model systems that behave like a probabilistic finite-state machine. Probabilistic finite-state systems have hidden states and only by observing the emissions of their states is possible to infer the system.

The previous definition adjusts perfectly to the goal of side-channel analysis, a cryptographic process is executed and it is known that the process has hidden states. By observing the timing emissions leaked through the side-channel is possible to infer the process.

HMMs offer a signal processing technique that enhance the result of the signal by eliminating noise in the signal and trying to guess the states of the algorithm used by the process.

3.3.3 Cache-Timing Data Analysis for DSA

Since the attack performed in this work is based on the FLUSH+RELOAD technique, the dimension of the cache-timing data is small enough to cluster it using the VQ technique, therefore this work skips that step. Knowing the sequence of algorithm states is crucial to perform a cache-timing attack and recover the DSA's secret key. Applying the HMM technique, the sequence of states for the SWE algorithm is recovered with high accuracy, allowing the extraction of partial information from every trace recorded by the spy process.

Constructing an HMM where the hidden part models the operations of the SWE algorithm – squares and multiplications, is an important step towards secret key recovery. Figure 3.7 illustrates the HMM transition model of the SWE algorithm exploited in this work.

Each label denotes the operation performed in each state and separate states are used to denote the system state before and after the execution of the algorithm. For the SWE algorithm implemented in OpenSSL's DSA, it is known that every window e_i starts and ends with "1"s and for a 160/1024-bit key pair, the window size $L(e_i) = 4$, therefore it is possible (although very unlikely) to have at most 4 sequences of square-multiply operations.

The most significant bit of the exponent is handled by the state s_2 and thanks to the performance degradation technique, no unknown states are observed.

The set of emissions for this HMM is $V = \{Sqr, Mul, Empty\}$ which is obtained from the possible states observed in the cache. The state set is defined by $S = s_1, ..., s_7$. States s_1 and s_7 are assumed to emit *Empty*, states s_2, s_3, s_5 and s_6 emit *Sqr* and finally, state s_4 only emits *Mul*.

Using rough estimates the initial model parameters are set and the model is trained using observations from the cache-timing information recorded

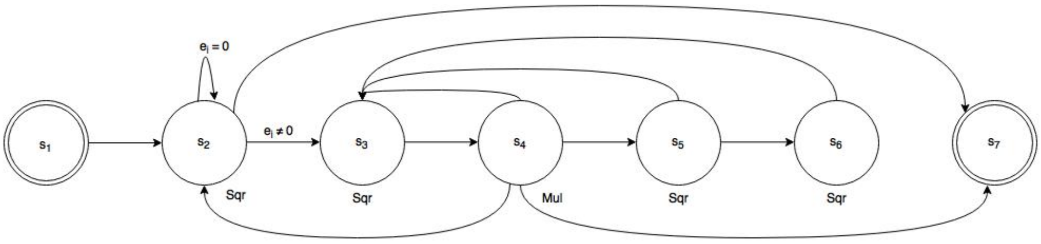

Figure 3.7: An HMM transition model for DSA's SWE algorithm.

during the profile phase mentioned in Section 3.1.3 where the secret and the algorithm operation is known.

The output sequence obtained in this step is then used in conjunction with the lattice-based attack to mount the key recovery attack. As mentioned earlier, due to limitations on the spy process, is not possible to get perfect traces and therefore is not possible to recover a perfect state sequence. Additionally, for DSA, knowing the perfect state sequence does not reveal the secret key directly although it would allow to use a considerable small amount of signatures during the key recovery attack.

Chapter 4

Related Work

Over the past few years, research work has focused in cache-timing attacks, and cryptographic algorithms are the standard choice to demonstrate the success of this type of attacks, mainly due to the nature of their relevance.

Several authors describe attacks on cryptographic systems that exploit partial nonce disclosure to recover long-term private keys.

Page [39] did the first formal studies of the security threats associated with the cache behavior and the information leaked through the cache. The author demonstrated the theoretical aspects of the attack and an implementation of his attack on DES encryption algorithm.

Percival [40] expanded the idea of cache attacks and performed an attack by observing the cache sets accessed during encryption of RSA's sliding window algorithm implemented in OpenSSL (0.9.7c). The access to cache sets revealed enough bits of the exponent to compute the secret exponent in polynomial time.

Brumley and Hakala [13] use an L1 data cache-timing attack to recover the LSBs of ECDSA nonces from the `dgst` command line tool in OpenSSL 0.9.8k. Combining VQ and HMMs explained in Section 3.3, they collect 2,600 signatures (8K with noise) and use the Howgrave et al. [27] attack to recover a 160-bit ECDSA private key. In a similar vein, Acıiçmez et al. [1] use an L1 instruction cache-timing attack to recover the LSBs of DSA nonces from the same tool in OpenSSL 0.9.8l, requiring 2,400 signatures (17K with noise) to recover a 160-bit DSA private key. Both attacks require HyperThreading architectures.

Gullasch et al. [24] expanded on the spy process by exploiting the *Completely Fair Scheduler* used in Linux systems and for the first time they proposed the use of the `clflush` instruction for evicting caches. The authors performed an *asynchronous attack* where they managed to profile the cache with a high granularity compared to previous works. They evicted and probed to create a trace of the cache which was later used to recover the cipher keys used on OpenSSL's AES (0.9.8n).

Brumley and Tuveri [14] mount a remote timing attack on the implementation of ECDSA with binary curves in OpenSSL 0.9.8o. They show that the timing leaks information on the MSBs of the nonce used and that after collecting that information over 8,000 TLS handshakes the private key can be recovered.

Benger et al. [7] recover the secret key of OpenSSL's ECDSA implementation for the curve *secp256k1* using less than 256 signatures. The authors make use of the Flush+Reload technique to target the LLC, they extend the lattice technique of Howgrave et al. [27] and after a considerable smaller effort they achieve success. They improved over previous lattice-based techniques by using all the leaked bits rather than limiting to a fixed number of bits.

Following the steps of Benger et al. [7], van de Pol et al. [41] exploit the structure of the modulus in some elliptic curves to use all of the information leaked in consecutive sequences of bits anywhere in the top half of the nonces, allowing them to recover the secret key after observing only a handful of signatures. Their target is the *secp256k1* curve of ECDSA on OpenSSL (1.0.1f).

Allan et al. [2] improve on these results by using a performance-degradation attack to amplify the side-channel. The amplification allows them to observe the sign bit in the wNAF representation used in OpenSSL 1.0.2a and to recover a 256-bit key after observing only 6 signatures.

Genkin et al. [20] perform electromagnetic and power analysis attacks on mobile phones. They show how to construct HNP triples when the signature uses the low s-value [50].

Finally, relevant to this work but in a different direction, Liu and Nguyen [34] offer a novel approach to solve the Basic Bounded Decoding (BDD) lattice problem used in cryptanalysis. The authors used BDD enumeration to perform a practical attack on DSA with partially known nonces.

Chapter 5

Implementation

5.1 A New Software Defect

Percival [40] demonstrated that the SWE implementation of modular exponentiation in OpenSSL version 0.9.7g is vulnerable to cache-timing attacks, applied to recover RSA private keys. Following the issue, the OpenSSL team committed two code changes relevant to this work. The first[1] adds a "constant-time" implementation of modular exponentiation, with a fixed-window implementation and using the scatter-gather method [12, 53] of masking table access to the multipliers.

The new implementation is slower than the original SWE implementation. To avoid using the slower new code when the exponent is not secret, OpenSSL added a flag (`BN_FLG_CONSTTIME`) to its representation of big integers. When the exponent should remain secret (e.g. in decryption and signing) the flag is set (e.g. in the case of DSA nonces, Figure 5.1, Line 252) at runtime and the exponentiation code takes the "constant-time" execution path (Figure 5.2, Line 413). Otherwise, the original SWE implementation is used.

The execution time of the "constant-time" implementation still depends on the bit length of the exponent, which in the case of DSA should be kept secret [11, 14, 36]. The second commit[2] aims to "make sure DSA signing exponentiations really are constant-time" by ensuring that the bit length of the exponent is fixed. This safe default behavior can be disabled by applications enabling the `DSA_FLAG_NO_EXP_CONSTTIME` flag at runtime within the DSA structure, although we are not aware of any such cases.

[1] -`https://github.com/openssl/openssl/commit/46a643763de6d8e39ecf6f76fa79b4d04885aa59`

[2] -`https://github.com/openssl/openssl/commit/0ebfcc8f92736c900bae4066040b67f6e5db8edb`

To get a fixed bit length, the DSA implementation adds γq to the randomly chosen nonce, where $\gamma \in \{1, 2\}$, such that the bit length of the sum is one more than the bit length of q. More precisely, the implementation creates a copy of the nonce k (Figure 5.1, Line 264), adds q to it (Line 274), checks if the bit length of the sum is one more than that of q (Line 276), otherwise it adds q again to the sum (Line 277). If q is n bits, then $k + q$ is either n or $n + 1$ bits. In the former case, indeed $k + 2q$ is $n + 1$ bits. As an aside, we note the code in question is not constant time and potentially leaks the value of γ. Such a leak would create a bias that can be exploited to mount the Bleichenbacher attack [3, 10, 18].

While the procedure in this commit ensures that the bit length of the sum kq is fixed, unfortunately it introduces a software defect. The function `BN_copy` is not designed to propagate flags from the source to the destination. In fact, OpenSSL exposes a distinct API `BN_with_flags` for that functionality—quoting the documentation:

> `BN_with_flags` creates a temporary shallow copy of `b` in `dest` . . . Any flags provided in `flags` will be set in `dest` in addition to any flags already set in `b`. For example this might commonly be used to create a temporary copy of a BIGNUM with the `BN_FLG_CONSTTIME` flag set for constant time operations.

In contrast, with `BN_copy` the `BN_FLG_CONSTTIME` flag does not propagate to kq. Consequently, the sum is not treated as secret, reverting the change made in the first commit— when the exponentiation wrapper subsequently gets called (Figure 5.1, Line 285), it fails the security-critical branch. Following a debug session in Figure 5.2, indeed the flag (explicit value `0x4`) is not set, and the execution skips the call to `BN_mod_exp_mont_consttime` and instead continues with the insecure SWE code path for DSA exponentiation.

In addition to testing our attack against OpenSSL (1.0.2h), we reviewed the code of two popular OpenSSL forks: LibreSSL[3] [44] and BoringSSL[4] [43]. Using builds with debugging symbols, we confirm both LibreSSL and BoringSSL share the same defect. It is worth noting that BoringSSL stripped out TLS DSA cipher suites in late 2014[5].

[3] -https://github.com/libressl-portable/openbsd/blob/873a225ece61e25b02dafea9676e0ba90519e764/src/lib/libssl/src/crypto/dsa/dsa_ossl.c

[4] -https://boringssl.googlesource.com/boringssl/+/master/crypto/dsa/dsa.c

[5] -https://boringssl.googlesource.com/boringssl/+/ef2116d33c3c1b38005eb59caa2aaa6300a9b450

```
246    /* Get random k */
247    do
248        if (!BN_rand_range(&k, dsa->q))
249            goto err;
250    while (BN_is_zero(&k)) ;
251    if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
252        BN_set_flags(&k, BN_FLG_CONSTTIME);
253    }
...
263    if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
264        if (!BN_copy(&kq, &k))
265            goto err;
266
267        /*
268         * We do not want timing information to leak the length of k, so we
269         * compute g^k using an equivalent exponent of fixed length. (This
270         * is a kludge that we need because the BN_mod_exp_mont() does not
271         * let us specify the desired timing behaviour.)
272         */
273
274        if (!BN_add(&kq, &kq, dsa->q))
275            goto err;
276        if (BN_num_bits(&kq) <= BN_num_bits(dsa->q)) {
277            if (!BN_add(&kq, &kq, dsa->q))
278                goto err;
279        }
280
281        K = &kq;
282    } else {
283        K = &k;
284    }
285    DSA_BN_MOD_EXP(goto err, dsa, r, dsa->g, K, dsa->p, ctx,
286                   dsa->method_mont_p);
```

Figure 5.1: Excerpt from OpenSSL's `dsa_sign_setup` in `crypto/dsa/dsa_ossl.c`. Line 252 sets the `BN_FLG_CONSTTIME` flag, yet `BN_copy` on Line 264 does not propagate it. The subsequent Line 285 exponentiation call will have pointer `K` with the flag clear.

```
   +--bn_exp.c------------------------------------------------------------------------+
   |402    int BN_mod_exp_mont(BIGNUM *rr, const BIGNUM *a, const BIGNUM *p,          |
   |403                        const BIGNUM *m, BN_CTX *ctx, BN_MONT_CTX *in_mont)    |
   |404    {                                                                           |
B+ |405        int i, j, bits, ret = 0, wstart, wend, window, wvalue;                  |
   |406        int start = 1;                                                          |
   |407        BIGNUM *d, *r;                                                          |
   |408        const BIGNUM *aa;                                                       |
   |409        /* Table of variables obtained from 'ctx' */                            |
   |410        BIGNUM *val[TABLE_SIZE];                                                |
   |411        BN_MONT_CTX *mont = NULL;                                               |
   |412                                                                                |
  >|413        if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0) {                           |
   |414            return BN_mod_exp_mont_consttime(rr, a, p, m, ctx, in_mont);        |
   |415        }                                                                       |
   |416                                                                                |
   |417        bn_check_top(a);                                                        |
   +--------------------------------------------------------------------------------+
   |0x7ffff779db3e <BN_mod_exp_mont+92>     mov    0x14(%rax),%eax                    |
   |0x7ffff779db41 <BN_mod_exp_mont+95>     and    $0x4,%eax                          |
   |0x7ffff779db44 <BN_mod_exp_mont+98>     test   %eax,%eax                          |
  >|0x7ffff779db46 <BN_mod_exp_mont+100>    je     0x7ffff779db85 <BN_mod_exp_mont+163> |
   |0x7ffff779db48 <BN_mod_exp_mont+102>    mov    -0x1b0(%rbp),%r8                   |
   +--------------------------------------------------------------------------------+
child process 29096 In: BN_mod_exp_mont                    Line: 413  PC: 0x7ffff779db46
(gdb) break BN_mod_exp_mont
Breakpoint 1 (BN_mod_exp_mont) pending.
(gdb) run dgst -dss1 -sign ~/dsa.pem -out ~/lsb-release.sig /etc/lsb-release
Starting program: /usr/local/ssl/bin/openssl \
dgst -dss1 -sign ~/dsa.pem -out ~/lsb-release.sig /etc/lsb-release
Breakpoint 1, BN_mod_exp_mont (...) at bn_exp.c:405
(gdb) backtrace
#0  BN_mod_exp_mont (...) at bn_exp.c:405
#1  0x00007ffff77eea62 in dsa_sign_setup (...) at dsa_ossl.c:285
#2  0x00007ffff77ee344 in DSA_sign_setup (...) at dsa_sign.c:87
#3  0x00007ffff77ee53d in dsa_do_sign (...) at dsa_ossl.c:159
#4  0x00007ffff77ee30c in DSA_do_sign (...) at dsa_sign.c:75
...
(gdb) stepi
(gdb) info register eax
eax            0x0      0
(gdb) print BN_get_flags(p, BN_FLG_CONSTTIME)
$1 = 0
(gdb) macro expand BN_get_flags(p, BN_FLG_CONSTTIME)
expands to: ((p)->flags&(0x04))
(gdb) print ((p)->flags&(0x04))
$2 = 0
(gdb)
```

Figure 5.2: Debugging OpenSSL DSA signing in `crypto/bn/bn_exp.c`. The Line 413 branch is not taken since `BN_FLG_CONSTTIME` is not set, as seen from the print command outputs. Hence `BN_mod_exp_mont_consttime` is not called— the control flow continues with classical SWE code.

5.2 Exploiting the Defect

In this section we describe how we use and combine the Flush+Reload technique with a performance degradation technique [2] to attack the OpenSSL implementation of DSA.

We tested our attack on an Intel Core i5-4570 Haswell Quad-Core 3.2GHz (22nm) with 16GB of memory running 64-bit Ubuntu 14.04 LTS "Trusty". Each core has an 8-way 32KB L1 data cache, an 8-way 32KB L1 instruction cache, an 8-way 256KB L2 unified cache, and all the cores share a 12-way 6MB unified LLC (all with 64B lines). It does not feature HyperThreading.

We used our own build of OpenSSL 1.0.2h which is the same default build of OpenSSL but with debugging symbols on the executable. Debugging symbols facilitate mapping source code to memory addresses but they are not loaded during run time, thus the victim's performance is not affected. Debugging symbols are, typically, not available to attackers but using reverse engineering techniques [16] is possible to map source code to memory addresses.

It is well known [34, 36], but not trivial, that knowing a few bits of sufficiently many signatures allows an attacker to recover the secret key. This is the goal of our attack, we trace and recover side-channel information of the SWE algorithm that reveals the sequence of *squares* and *multiplications*, from that sequence we recover a few bits that we use for the lattice attack described in Section 5.4.

As seen in Figure 5.2, every time OpenSSL performs a DSA signature, the exponentiation method `BN_mod_exp_mont` in `crypto/bn/bn_exp.c` gets called. There, the `BN_FLG_CONSTTIME` flag is checked but due to the software defect discussed in Section 5.1 the condition fails and the routine continues with the SWE pre-computation and then the actual exponentiation. For the finite field operations, `BN_mod_exp_mont` calls `BN_mod_mul_montgomery` in `crypto/bn/bn_mont.c` and from there, the multiply wrapper `bn_mul_mont` is called, where, by default for x64 targets, assembly code is executed to perform low level operations using BIGNUMs for square and multiplication. OpenSSL uses Montgomery representation for efficiency. Note that for other platforms and/or non-default build configurations, the actual code executed ranges from pure C implementation to entirely different assembly. The attacker can easily adapt to these different execution paths, but the discussion that follows is geared towards our target platform. It is worth mentioning, the spy process does not know when a signature starts and ends but this can be deduced by looking for a sequence of consecutive multiplications in each trace, this indicates the SWE's pre-computation phase.

The threshold set for the load time in the Flush+Reload technique (cache hit vs. cache miss) is system and software dependent. From our measurements we set this threshold accordingly since the load times from LLC and from memory were clearly defined. Figure 5.4 shows that loads from LLC take less that 100 cycles, while loads from main memory take more than 200 cycles.

As mentioned before, to get better resolution and granularity during the attack one effective strategy is to target body loops or routines that are invoked several times. For that reason we probe, using the Flush+Reload technique, inner routines used for square and multiply. Since squares can be computed more efficiently than multiplication, OpenSSL's multiply wrapper checks if the two pointer operands are the same and, if so, calls to assembly squaring code (`bn_sqr8x_mont`)—otherwise, to assembly multiply code (`bn_mul4x_mont`).

At the same time we run a performance degradation attack, flushing actively used memory addresses during these routines (e.g. assembly labels `Lsqr4x_inner` and `Linner4x`, respectively). We slow down the execution time to a safe, but not noticeable by the victim, threshold that ensures a good trace by our spy program. In our experiments, we observe slow down factors of roughly 16 and 26 for 1024-bit and 2048-bit DSA, respectively due to the degrade technique.

Using this strategy, our spy program collects data from two channels: one for square latencies and the other for multiply latencies. We then apply signal processing techniques to this raw channel data. A moving average filter on the data results in Figure 5.3 and Figure 5.4 for 1024-bit and 2048-bit DSA, respectively. There is a significant amount of information to extract from these signals on the SWE algorithm state transitions and hence exponent bit values. Generally, extracted multiplications yield a single bit of information and the squares yield the position for these bits. Some short examples follow.

Stepping through Figure 5.3, the initial low amplitude for the multiply signal is the multiplication for converting the base operand to Montgomery representation. The subsequent low amplitude for the square signal is the temporary square value used to build the odd powers for the SWE pre-computation table (i.e. s in Figure 5). The subsequent long period of low multiply amplitude is the successive multiplications to build the pre-computation table itself. Then begins the main loop of SWE. As an upward sloping multiply amplitude intersects a downward sloping square amplitude, this marks the transition from a multiplication operation to a square operation (and vice versa). This naturally occurs several times as the main exponentiation loop iterates. The end of this particular signal shows a final transition from multiply to a single square, indicating that the exponent is even and the two LSBs are 1 and 0.

Stepping through Figure 5.4 is similar, yet the end of this particular signal shows a final transition from square to multiply—indicating that the exponent is odd, i.e. the LSB is 1.

Even when employing the degrade technique, it is important to observe the vast granularity difference between these two cryptographic settings. On average, a 2048-bit signal is roughly ten times the length of a 1024-bit signal, even when the exponent is only 60% longer (i.e. 256-bit vs. 160-bit). This generally suggests we should be able to extract more accurate information from 2048-bit signals than 1024-bit—i.e., the higher security cryptographic parameters are more vulnerable to side-channel attack in this case. See [49, 51, 52] for similar examples of this phenomenon.

Granularity is vital to determining the number of squares interleaved between multiplications. Since, in our environment, there appears to be no reliable indicator in the signal for transitions from one square to the next, we estimate the number of adjacent squares by the horizontal distance between multiplications. Since the channel is latency data, we also have reference clock cycle counter values so another estimate is based on the counter differences at these points. Our experiments showed no significant advantage of one approach over the other.

Extracting the multiplications from the signal and interleaving them with a number of consecutive squares proportional to the width of the corresponding gap gives us the square and multiplication *SM* sequence, that the SWE algorithm passed through. Figure 5.7 shows an example of an SM sequence recorded by the spy program when OpenSSL signs using 2048-bit DSA.

Our spy program is able to capture most of the SM sequence accurately. It can miss or duplicate a few squares due to drift but is able to capture all of the multiplication operations. Closer to the LSBs, the information extracted from the SM sequence is more reliable since the bit position is lost if any square operation is missed during probing — on average three LSBs are recovered per trace.

In summary, we performed our attack in a system under normal workload but we focused only on the three processes of interest – i.e. victim, spy and degradation processes. The three processes executed in three different cores. The first core executed the victim process, in this case OpenSSL's DSA algorithm. The second core executed our spy process while the third core executed the performance degradation process. The knowledge of the message and the signature is assumed since this information is public and can be obtained legitimately – see Section 5.3.

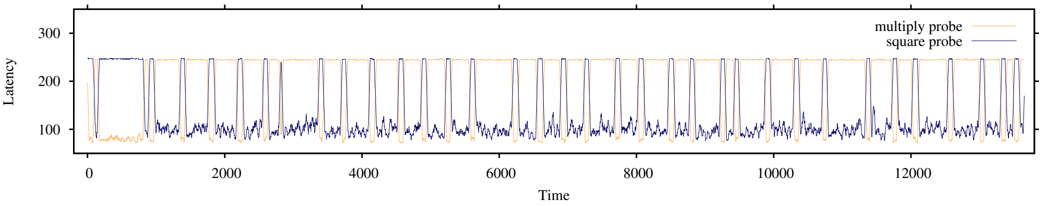

Figure 5.3: Complete filtered trace of a 1024-bit DSA sign operation during an OpenSSH SSH-2 handshake.

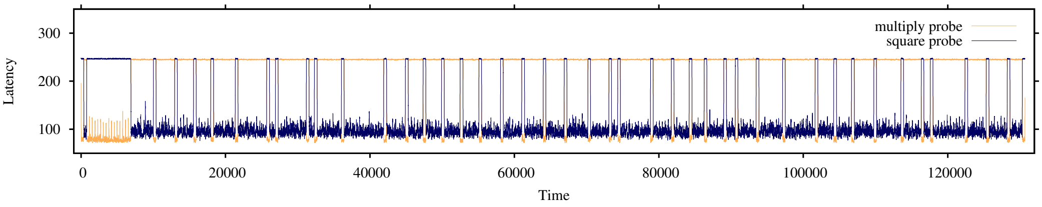

Figure 5.4: Complete filtered trace of a 2048-bit DSA sign operation during an stunnel TLS 1.2 handshake.

5.3 Victimizing Applications

The defect from the previous section is in a shared library. Potentially any application that links against OpenSSL for DSA functionality can be affected by this vulnerability. But to make our attack concrete, we focus on two ubiquitous protocols and applications: TLS within stunnel and SSH within OpenSSH.

As we discuss later in Section 5.4, the trace data alone is not enough for private key recovery—we also need the digital signatures themselves and (hashed) messages. To this end, the goal of this section is to describe the practical tooling we developed to exploit the defect within these applications, collecting both trace data and protocol messages.

5.3.1 Attacking TLS

To feature TLS support, one option for network applications that do not natively support TLS communication is to use stunnel[6], a popular portable open source software package that forwards network connections from one port to another and provides a TLS wrapper. A typical stunnel use case is listening on a public port to expose a TLS-enabled network service, then

[6] `https://www.stunnel.org`

```
Client                                     Server

ClientHello
[random,
CipherSuite]        ------->
                                       ServerHello
                                          [random,
                                     cipher_suite]
                                       Certificate
                                 ServerKeyExchange
                                          [params,
                                    HashAlgorithm,
                               SignatureAlgorithm,
                                    signed_params]
                    <-------       ServerHelloDone
ClientKeyExchange
(ChangeCipherSpec)
Finished            ------->
                                (ChangeCipherSpec)
                    <-------              Finished
Application Data    <------>      Application Data
```

Figure 5.5: Our custom client carries out TLS handshakes, collecting certain fields from the ClientHello, ServerHello, and SeverKeyExchange messages to construct the digest. It collects timing traces in parallel to the server's DSA sign operation, said digital signature being included in a SeverKeyExchange field and collected by our client.

connecting to a localhost port where a non-TLS network service is listening—stunnel provides a TLS layer between the two ports. It links against the OpenSSL shared library to provide this functionality. For our experiments, we used stunnel 5.32 compiled from stock source and linked against OpenSSL 1.0.2h. We generated a 2048-bit DSA certificate for the stunnel service and chose the DHE-DSS-AES128-SHA256 TLS 1.2 cipher suite.

We wrote a custom TLS client that connects to this stunnel service. It launches our spy to collect the timing signals, but its main purpose is to carry out the TLS handshake and collect the digital signatures and protocol messages. Figure 5.5 shows the TLS handshake. Relevant to this work, the initial ClientHello message contains a 32-byte random field, and similarly the server's ServerHello message. In practice, these are usually a 4-byte UNIX timestamp concatenated with a 28-byte nonce. The Certificate message contains the DSA certificate we generated for the stunnel service. The `ServerKeyExchange` message contains a number of critical fields for our attack: Diffie-Hellman key exchange parameters, the signature algorithm and hash function identifiers, and finally the digital signature itself in the `signed_params` field. Given our stunnel configuration and certificate, the 2048-bit DSA signature is over the concatenated string

```
ClientHello.random + ServerHello.random +
ServerKeyExchange.params
```

and the hash function is SHA-512, both dictated by the `SignatureAndHashAlgorithm` field (explicit values `0x6`, `0x2`). Our client saves the hash of this string and the DER-encoded digital signature sent from the server. All subsequent messages, including `ServerHelloDone` and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to build up a set of distinct trace, digital signature, and digest tuples. See Section 5.4 for our explicit attack parameters. Figure 5.4 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim stunnel service.

5.3.2 Attacking SSH

OpenSSH[7] is a suite of tools whose main goal is to provide secure communications over an insecure channel using the SSH network protocol.

OpenSSH is linked to the OpenSSL shared library to perform several cryptographic operations, including digital signatures. For our experiments we used the stock OpenSSH 6.6.1p1 binary package from the Ubuntu repository, and pointed the run-time shared library loader at OpenSSL 1.0.2h. The DSA key pair used by the server and targeted by our attack is the default 1024-bit key pair generated during installation of OpenSSH.

Similar to Section 5.3.1, we wrote a custom SSH client that launches our spy program, the spy program collects the timing signals during the handshake. At the same time it performs an SSH handshake where the protocol messages and the digital signature are collected for our attack.

Relevant to this work, the SSH protocol defines the Diffie-Hellman key exchange parameters in the `SSH_MSG_KEXINIT` message, along with the signature algorithm and the hash function identifiers. Additionally a 16-byte `random` nonce is sent for host authentication by the client and the server.

The `SSH_MSG_KEXDH_REPLY` message contains the server's public key (used to create and verify the signature), server's DH public key `f` (used to compute

[7] http://www.openssh.com

the shared secret K in combination with the client's DH public key e) and the signature itself. Figure 5.6 shows the SSH handshake with the critical parameters sent in every message relevant for the attack. To be more precise, the signature is over the SHA-1 hash of the concatenated string

```
ClientVersion + ServerVersion +
Client.SSH_MSG_KEXINIT + Server.SSH_MSG_KEXINIT +
Server.publicKey + minSize + prefSize + maxSize +
p + g + e + f + K
```

As the key exchange[8] and public key parameters, our SSH client was configured to use diffie-hellman-group-exchange-sha1 and ssh-dss respectively. Note that two different hashing functions may be used, one hash function for the Diffie-Hellman key exchange and another hash function for the signing algorithm, which for DSA is the SHA-1 hash function.

Similarly to the TLS case, our client saves the hash of the concatenated string and the digital signature raw bytes sent from the server. All subsequent messages, including SSH_MSG_NEWKEYS and any client responses, are not required by our attack. Our client therefore drops the connection at this stage, and repeats this process several hundred times to build up a set of distinct trace, digital signature, and digest tuples. See Section 5.4 for our explicit attack parameters. Figure 5.3 is a typical signal extracted by our spy program in parallel to the handshake between our client and the victim SSH server.

5.3.3 Observations

These two widely deployed protocols share many similarities in their handshakes regarding e.g. signaling, content of messages, and security context of messages. However, in the process of designing and implementing our attacker clients we observe a subtle difference in the threat model between the two. In TLS, all values that go into the hash function to compute the digital signature are public and can be observed (unencrypted) in various handshake messages. In SSH, *most* of the values are public—the exception is the last input to the hash function: the shared DH key. The consequence is side-channel attacks against TLS can be passive, listening to legitimate handshakes not initiated by the attacker yet collecting side-channel data as this occurs. In SSH, the attacker must be active and initiate its own handshakes—without knowing the shared DH key, a passive attacker cannot compute the corresponding digest needed later for the lattice stage of the attack. We find this

[8] https://tools.ietf.org/html/rfc4419

```
Client                                     Server

ClientVersion         ----->
                                    ServerVersion
                                  SSH_MSG_KEXINIT
                                 [Nonce, KEX_alg,
                      <-----       publicKey_alg]
SSH_MSG_KEXINIT
[Nonce, KEX_alg,
publicKey_alg]
SSH_KEXDH_GEX_REQUEST
[minSize, prefSize,
maxSize]              ----->
                              SSH_KEXDH_GEX_GROUP
                      <-----               [p, g]
SSH_KEXDH_GEX_INIT
[e]                   ----->
                              SSH_KEXDH_GEX_REPLY
                                   [publicKey, f,
                      <-----           Signature]
SSH_MSG_NEWKEYS
                      ----->
Application Data      <---->     Application Data
```

Figure 5.6: Our custom client carries out SSH handshakes, collecting parameters from all the messages to construct the digest. It collects timing traces in parallel to the server's DSA sign operation, said digital signature being included in a SSH_KEXDH_GEX_REPLY field and collected by our client.

innate protocol level side-channel property to be an intriguing feature, and a factor that should be carefully considered during protocol design.

5.4 Recovering the private key

In previous sections we showed how our attack can recover the sequence of square and multiply operations that the victim performs. We further showed how to get the signature information matching each sequence for both SSH and TLS. We now turn to recovering the private key from the information we collect.

The scheme we use is similar to past works. We first use the side-channel information we capture to collect information on the nonce used in each signature. We use the information to construct HNP instances and use a lattice technique to find the private key. Further details on each step are provided below.

5.4.1 Extracting the least significant bits

In Section 5.2 we showed how we collect the SM sequences of each exponentiation. From every SM sequence, we extract a few LSBs to be used later in the lattice attack. LSBs are extracted by observing the SM trailing sequence from each trace and then comparing it against known SM trailing sequences, always looking for the best trade-off between bits recovered from each trace and the number of signatures required for the lattice attack.

Table 5.1 contains our empirical accuracy statistics for various relevant patterns trailing the SM sequences, these trailing SM sequences were chosen according to multiplier usage patterns. Menezes et al. [35] mention that for a window size of S, the expected distance between non-zero windows is $S+1$ — i.e. number of squares between multiply operations, thus we focused in trailing patterns repeated with higher probability. Furthermore, not for the SWE in isolation but rather in the context of OpenSSL DSA executing in real world applications (TLS via stunnel, SSH via OpenSSH), as described above in Section 5.3. All of these patterns correspond to recovering $a = \bar{k} \bmod 2^{\ell}$ for an exponent $\bar{k}$.

From these figures, we note two trends. (1) The accuracy decreases as ℓ increases due to deviation in the square operation width. Yet weighed with the exponentially decreasing probability of the longer patterns, the practical impact diminishes. (2) As expected, we generally obtain more accurate results with 2048-bit vs. 1024-bit due to granularity. These numbers show

ℓ	a	Bit Pattern	SM Pattern	Accuracy (%)	Accuracy (%)
				1024-bit, SSH	2048-bit, TLS
1	1	`1`	`SSM`	99.9	99.9
2	2	`10`	`SMS`	99.9	99.7
2	3	`11`	`SMSM`	98.2	97.2
3	4	`100`	`SSMSS`	99.7	99.7
3	6	`110`	`SMSMS`	99.4	98.2
4	8	`1000`	`SSMSSS`	97.8	99.6
4	12	`1100`	`SMSMSS`	98.4	97.8
5	16	`10000`	`SSMSSSS`	96.7	99.1
5	24	`11000`	`SMSMSSS`	95.0	97.6
6	32	`100000`	`SSMSSSSS`	85.1	98.8
6	48	`110000`	`SMSMSSSS`	90.4	95.0
7	64	`1000000`	`SSMSSSSSS`	87.5	97.5
7	96	`1100000`	`SMSMSSSSS`	84.6	95.1
8	128	`10000000`	`SSMSSSSSSS`	67.7	98.7
8	192	`11000000`	`SMSMSSSSSS`	75.0	94.8

Table 5.1: Empirical results of recovering various LSBs from the spy program traces and their corresponding SM sequences.

that, exploiting our new software defect and leveraging the techniques in Section 5.2, we can recover a with extremely high probability.

5.4.2 Lattice attack implementation

Recall that to protect against timing attacks OpenSSL uses an exponent $\bar{k}$ equivalent to the randomly selected nonce k. $\bar{k}$ is calculated by adding the modulus q once or twice to k to ensure that $\bar{k}$ is of a fixed length. That is, $\bar{k} = k + \gamma q$ such that $2^n \leq \bar{k} < 2^n + q$ where $n = \lceil \lg(q) \rceil$ and $\gamma \in \{1, 2\}$.

```
SMMMMMMMMMMMMMMMMSSSMSSSSSSSSMSSSSMSSSSSSSMSSSSSSM
SSSSSSMSSSSSSSSMSSSMSSSSSSSSSSSMSSSSSSSSMSSSSSMSSS
SSMSSSSSSSSSMSSSSSMSSSSSSSMSSSSSMSSSSSSSMSSSSMSSSS
SSSMSSSSSSMSSSSMSSSSSSSSSMSSSSSSSSMSSSMSSSSSSSMSSS
SSMSSSSSMSSSSMSSSSSSMSSSSMSSSSSSSMSSSSSMSSMSSSSSSS
SSMSSSSMSSSSSSSSMSSSSSSSSSSMSMSSSSSSMSSSSSSSSMSSSS
SSSSSSMSSSSM
```

Figure 5.7: Example of an extracted SM sequence, where `S` and `M` are square and multiply, respectively.

The side-channel leaks information on bits of the exponent $\bar{k}$ rather than directly on the nonce. To create HNP instances from the leak we need to handle the unknown value of γ. In previous works, due to ECC parameters the modulus is close to a power of two hence the value of γ is virtually constant [7]. For DSA, the modulus is not close to a power of two and the value of γ varies between signatures. The challenge is, therefore, to construct an HNP instance without the knowledge of γ. We now show how to address this challenge.

Recall that $s = k^{-1}(h(m) + \alpha r) \bmod q$. Equivalently, $k = s^{-1}(h(m) + \alpha r) \bmod q$. The side-channel information recovers the ℓ LSBs of $\bar{k}$. We, therefore, have $\bar{k} = b2^{\ell} + a$ where $a = \bar{k} \bmod 2^{\ell}$ is known, and

$$2^{n-\ell} \leq b < 2^{n-\ell} + \lceil q/2^{\ell} \rceil . \tag{5.1}$$

Following previous works we use $\lfloor \cdot \rfloor_q$ to denote the reduction modulo q to the range $[0, q)$ and $| \cdot |_q$ for the reduction modulo q to the range $(-q/2, q/2)$. Within these expressions division operations are carried over the reals whereas all other operations are carried over $GF(q)$.

We now look at $\lfloor b - 2^{n-\ell} \rfloor_q$.

$$\begin{aligned}
&\lfloor b - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor (\bar{k} - a) \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor \bar{k} \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor k \cdot 2^{-\ell} + \gamma \cdot q \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor k \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor (s^{-1} \cdot (h(m) + \alpha \cdot r) \cdot 2^{-\ell} - a \cdot 2^{-\ell} - 2^{n-\ell} \rfloor_q \\
=\ & \lfloor \alpha \cdot s^{-1} \cdot r \cdot 2^{-\ell} - (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} \rfloor_q
\end{aligned}$$

Hence, we can set:

$$\begin{aligned}
t &= \lfloor s^{-1} \cdot r \cdot 2^{-\ell} \rfloor_q \\
u &= \left\lfloor (2^n + a - s^{-1} \cdot h(m)) \cdot 2^{-\ell} + \left\lceil \frac{q}{2^{\ell+1}} \right\rceil \right\rfloor_q \\
v &= |\alpha \cdot t - u|_q
\end{aligned}$$

and by (5.1) we have $|v| \leq \lceil q/2^{\ell+1} \rceil$.

Out of the HNP instances we generate, we select at random 49 for the SSH attack, 130 for the TLS attack and construct a lattice as described in Section 3.2.2. We solve the CVP problem with a Sage script, performing lattice reduction using BKZ [45], and enumerate the lattice points using Babai's Nearest Plane (NP) algorithm [5]. We apply two different techniques to extend NP to a larger search space. First, we take different rounding paths to explore 2^{10} different solutions in the tree paths [33, Sec. 4]. Second, we use a randomization technique [34, Sec. 3.5] and shuffle the rows of B between lattice reductions. We repeat with a different random selection of instances until we find the private key.

Chapter 6

Results

We implemented the attack and evaluated it against the two protocols, SSH with 1024/160-bit DSA and TLS with 2048/256-bit DSA. Table 6.1 contains the results. For both protocols we only utilize traces with $\ell \geq 3$. With this value we experimentally found that we require 49 such signatures for SSH and 130 for TLS in order to achieve a reasonable probability of solving the resulting CVP.

Because the nonces are chosen uniformly at random, only about one in every four signatures has an ℓ that we can utilize. To gather enough signatures and to compensate for possible trace errors, we collect 580 SM sequences from TLS handshakes and 260 from SSH.

On average, these collected sequences yield 70.8 (SSH) and 158.1 (TLS) traces with $\ell \geq 3$. Comparing the traces to the ground truth, we know that on average less than 3 have trace errors. However, because an adversary cannot check against the ground truth, we leave these erroneous traces in the set and use them in the attack. We note that due to the smaller key size

Victim	OpenSSH (SSH)	stunnel (TLS)
Key size	1024/160-bit	2048/256-bit
Handshakes	260	580
Lattice size	50	131
Set size	70.8	158.1
Errors	2.1	1.7
Iterations	13	22
CPU minutes	5.9	38.8
Success rate (%)	100.0	100.0

Table 6.1: Empirical lattice attack results over a thousand trials. Set size and errors are mean values. Iterations and CPU time are median values.

in SSH, trace errors are much more prevalent there.

We construct a lattice from a random selection of the collected traces and attempt to solve the resulting CVP. Due to the presence of the error traces there is a non-negligible probability that our selected set contains an error. Furthermore, even if all the chosen traces are correct, the algorithm may fail to find the target solution due to the heuristic nature of lattice techniques. In case of failure, we repeat the process with a new random selection from the same set. We need to execute a median value of 13 iterations for SSH and 22 for TLS until we find the target solution.

As seen from Table 6.1, repeating our experiment over a thousand trials on a cluster with hundreds of nodes, mixed between Intel X5660 and AMD Opteron 2435 cores, we find the private key in all cases requiring a median 5.9 CPU minutes for the SSH key and 38.8 CPU minutes for the TLS key. Although we executed each trial on a single core, in reality the iterations are independent of each other—the lattice attack is ridiculously parallel.

Chapter 7

Discussion

Finding and exploiting a cache-timing attack requires a lot of work but additional work needs to be done *a posteriori*. Section 7.1 explains some of the challenges abstracted from the experimentation that should be consider during a real life attack under a more chaotic setting where a system is running hundreds of processes at a given time. Section 7.2 explains some techniques recommended to mitigate cache-timing techniques and cache-timing attacks in general, hardware-based and software-based solutions are mentioned. And finally, Section 7.3 discusses some observations made during the responsible disclosure process.

7.1 Challenges

Performing a cache-timing attack is not a trivial task to do and it is a complex engineering problem as it is a complicated mathematical and cryptographic problem. Several challenges during the the experiments were faced and some of those challenges were abstracted for demonstration purposes. Some of those challenges are:

- Obtaining the memory addresses of the algorithm operations to be probed by an attacker—e.g. square and multiply for DSA's SWE algorithm, requires reverse engineering skills. Reverse engineering allows to extract the addresses from the shared library because, typically, debugging symbols are not loaded during execution time. A way to simplify this task for an attacker is to compute the offset for each memory address from the base memory address of the shared library. Using this approach has two purposes: (a) during the reverse engineering step, an attacker only has to find the base memory address and (b) even if memory randomization techniques are used the offset is the same, thus, an attacker only cares about the base memory address.

- Synchronization of the spy process and the victim process is easy during experimentation but during an actual attack is not always possible to synchronize both processes. Therefore, the spy process has to execute on the background and is the attacker's job to find the beginning and the end of every recorded cryptographic operation in the side-channel trace.
- Having access to the target machine is not a strong assumption, the SSH attack in Section 5.3.2 demonstrated this. Nowadays having legitimate access to target machines is easier thanks to cloud computing and virtualization.

7.2 Mitigation

The techniques presented in this work are some of the techniques currently in use for cache-timing attacks. The cache-timing attack described in Chapter 5 is a real threat against systems and users working with DSA as a digital signature algorithm for authentication within the TLS and the SSH protocols. Mitigating the attack in this work, cache-timing techniques and ultimately cache-timing attacks is extremely relevant to the security of Internet.

Fixing DSA is desirable and the fix is trivial once the software defect is detected, however, this does not address the general issue of preventing cache-timing attacks, specially cache-timing attacks based on the FLUSH+RELOAD technique. The FLUSH+RELOAD technique relies on four basic factors to successfully implement a cache-timing attack and some of these factors are shared with the performance degradation technique since it is based on the FLUSH+RELOAD technique. The four basic factors are: (1) sensitive data is recoverable from memory access patterns, (2) the spy process and the victim process share the LLC memory, (3) a high-accuracy clock is available to the spy process and (4) the lack of permission checks for using the `clflush` instruction. Preventing some of the factors block the techniques and therefore the cache-timing attacks depending on these techniques do not work anymore.

As mentioned in Section 3.1.3, `clflush` is an instruction that does not require elevated privileges to execute. A logical solution is to restrict the access of the `clflush` instruction to certain memory pages. Additionally, restrict the use of the `clflush` instruction to memory pages where the process using it has write access — e.g. the spy process. Yarom and Falkner [52] mention that AMD is not vulnerable to the Flush+Reload technique due to their non-inclusive caches. In AMD processors, evicting data from the LLC does not, necessarily, evicts the data from the L1 or L2 caches. Also, the ARM [8] architecture is not vulnerable to the Flush+Reload technique thanks to the restricted use of the command to evict memory lines.

Eliminating the use of the `rdtsc` command from the microprocessors is not possible but it is possible to introduce noise or reduce the accuracy of the clock using fuzzy time techniques [28]. However this approach does not completely solve the problem and it has an impact on benign applications that require access to a high-resolution timer. An attacker can start its own clock process in parallel or it can use the network as a clock.

Hardware based solutions require a considerable amount of money and time. Developing and adoption could take a long time, furthermore, these solutions do not protect existing hardware, thus, software-based solutions are required to fix the problem and impact as many affected devices as possible.

The easiest and fastest software-based solution to mitigate the attack proposed in this work is fixing OpenSSL's DSA. Fixing the software defect explained in Section 5.1 is an immediate countermeasure to this attack, since this fix allows the code to execute a constant-time implementation of the SWE algorithm. And as mentioned in Section 5.1, the constant-time implementation of the SWE algorithm uses a fixed-window and masks table access to the multipliers using the scatter-gather method [12, 53].

Intuitive software-based solutions include partially or fully disabling caching. Disabling caching completely prevents cache-timing attacks, nevertheless this approach was studied by Acıiçmez et al. [1] and proved to have an immense performance impact on both, the cryptographic process and on the system. A similar performance impact is observed if the cache is flushed periodically—similar to what the performance degradation technique does.

7.3 Disclosure of the Attack

Following good practices and responsible disclosure, the vulnerability found during this work was reported to the open source projects OpenSSL, LibreSSL and BoringSSL. OpenSSL is issuing a security advisory and they have assigned the `CVE-2016-2178`, the rest of the parties involved were very proactive to fix the vulnerability. Additionally, the vulnerability was reported to OpenSSH because the main goal of the SSH protocol is to provide remote legitimate access to a system and the attack presented in Section 5.3.2 represents a realistic scenario where an attacker does not require elevated privileges to exploit it.

At the same time that the security vulnerability was reported, two patches were proposed to fix it. The two versions of the patch were submitted to prevent future code changes that may re-introduce the possibility of a cache-timing attack. At the end OpenSSL chose one of the patches. The patch was applied on 6^{th} of June 2016 by OpenSSL[1] and LibreSSL[2]. DSA is deprecated in OpenSSH but it is still widely used. OpenSSH admitted that it is only possible for them to advise users not to use or re-enable deprecated features but ultimately the decision is up to the users.

[1] -https://github.com/openssl/openssl/commit/399944622df7bd81af62e67ea967c470534090e2

[2] -https://github.com/libressl-portable/openbsd/commit/075f24fcf49c54c48e55cd724e413880ebaffba6

Chapter 8

Conclusions

A simple software defect introduced in the DSA implementation in OpenSSL led to a critical security vulnerability exploited in this work. This vulnerability not only allowed to exploit OpenSSL but also to mount end-to-end attacks against fundamental Internet protocols: SSH (via OpenSSH) and TLS (via stunnel).

The contributions in this work are summarized as follows:

- Introduction to digital signatures, the DSA algorithm and the SWE algorithm used.

- Explanation of common cache-timing techniques used to perform cache-timing attacks and cache-timing data analysis against vulnerable algorithms running in non-constant time.

- Identification of a security weakness in OpenSSL which fails to use a side-channel safe implementation when performing DSA signatures.

- Description of how to use a combination of the Flush+Reload technique with a performance-degradation attack to leak information from the unsafe SWE algorithm.

- Presentation of the first key-recovery cache-timing attack on the TLS and SSH cryptographic protocols.

- Recovery of DSA's secret key by constructing and solving a lattice problem with the side-channel information, the digital signatures and messages.

As can be seen, applied cryptography requires extreme attention to detail and special care during design and implementation. Some technical advice regarding this vulnerability to the users:

- OpenSSH supports building without OpenSSL as a dependency. It is recommend that OpenSSH package maintainers switch to this option.

- OpenSSH administrators and users are recommended to migrate to `ssh-ed25519` key types, the implementation of which has many desirable side-channel resistance properties. Furthermore, ensure that `ssh-dss` is absent from the `HostKeyAlgorithms` configuration field, and any such `HostKey` entries removed.

- OpenSSH administrators and users are recommended to disable cipher suites that have DSA functionality as a pre-requisite.

8.1 Future work

The cache-timing attack on OpenSSL's DSA implementation and the results obtained from the experiments were positive and successful. The analysis presented here is far from being exhaustive. Although a fix was proposed and applied by the open source community, code verification is still required. DSA's constant-time SWE algorithm was implemented back in 2005 but it has not been tested against cache-timing attacks using current cache-timing techniques because the software defect explained in this work prevented the constant-time code from running. The constant-time implementation might provide enough side-channel information to mount a new cache-timing attack, recovering the secret key.

Furthermore, this work focused in one particular implementation of digital signatures — DSA, additional cryptographic primitives need to be verified to be safe against the techniques discussed here.

Bibliography

[1] Aciiçmez, O., Brumley, B. B., and Grabher, P. New results on instruction cache attacks. In *CHES* (Santa Barbara, CA, US, 2010).

[2] Allan, T., Brumley, B. B., Falkner, K., Van de Pol, J., and Yarom, Y. Amplifying side channels through performance degradation. IACR Cryptology ePrint Archive, Report 2015/1141, Nov 2015.

[3] Aranha, D. F., Fouque, P.-A., Gérard, B., Kammerer, J.-G., Tibouchi, M., and Zapalowicz, J.-C. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In *ASIACRYPT* (Kaohsiung, TW, Dec 2014), pp. 262–281.

[4] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using KSM. In *Linux symposium* (2009), pp. 19–28.

[5] Babai, L. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica 6*, 1 (Mar. 1986), 1–13.

[6] Barker, E., and Roginsky, A. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800-131A Revision 1, Nov 2015.

[7] Benger, N., Van de Pol, J., Smart, N. P., and Yarom, Y. "Ooh aah..., just a little bit": A small amount of side channel can go a long way. In *CHES* (Busan, KR, Sep 2014), pp. 75–92.

[8] Bernstein, D. J. Cache-timing attacks on AES, 2005. Preprint available at `http://cr.yp.to/papers.html#cachetiming`.

[9] Biham, E., and Shamir, A. Differential cryptanalysis of des-like cryptosystems. *Journal of CRYPTOLOGY 4*, 1 (1991), 3–72.

[10] Bleichenbacher, D. On the generation of one-time keys in DL signature schemes. Presentation at IEEE P1363 Working Group meeting, Nov 2000.

[11] Boneh, D., and Venkatesan, R. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In *CRYPTO'96* (Santa Barbara, CA, US, Aug 1996), pp. 129–142.

[12] Brickell, E., Graunke, G., and Seifert, J.-P. Mitigating cache/timing based side-channels in AES and RSA software implementations. RSA Conference 2006 session DEV-203, Feb 2006.

[13] Brumley, B. B., and Hakala, R. M. Cache-timing template attacks. In *15th ASIACRYPT* (Tokyo, JP, Dec 2009), pp. 667–684.

[14] Brumley, B. B., and Tuveri, N. Remote timing attacks are still practical. In *16th ESORICS* (Leuven, BE, 2011).

[15] Chari, S., Rao, J. R., and Rohatgi, P. Template attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002*. Springer, 2002, pp. 13–28.

[16] Cipresso, T., and Stamp, M. Software reverse engineering. In *Handbook of Information and Communication Security*. 2010, pp. 659–696.

[17] Corporation, I. Intel 64 and ia-32 architectures optimization reference manual, Jan 2016.

[18] De Mulder, E., Hutter, M., Marson, M. E., and Pearson, P. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In *CHES* (Santa Barabara, CA, US, Aug 2013), pp. 435–452.

[19] ElGamal, T. *Advances in Cryptology: Proceedings of CRYPTO 84*. Springer Berlin Heidelberg, 1985, ch. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms.

[20] Genkin, D., Pachmanov, L., Pipman, I., Tromer, E., and Yarom, Y. ECDSA key extraction from mobile devices via nonintrusive physical side channels. IACR Cryptology ePrint Archive, Report 2016/230, Mar 2016.

[21] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing 17*, 2 (1988), 281–308.

[22] Grabher, P., Großschädl, J., and Page, D. Cryptographic side-channels from low-power cache memory. In *Cryptography and Coding*. Springer, 2007, pp. 170–184.

[23] Green, P., Noad, R., and Smart, N. P. Further hidden markov model cryptanalysis. In *Cryptographic Hardware and Embedded Systems–CHES 2005*. Springer, 2005, pp. 61–74.

[24] Gullasch, D., Bangerter, E., and Krenn, S. Cache games – bringing access-based cache attacks on AES to practice. In *S&P* (May 2011), pp. 490–505.

[25] Hankerson, D., Menezes, A., and Vanstone, S. *Guide to Elliptic Curve Cryptography.* Springer New York, 2006.

[26] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[27] Howgrave-Graham, N., and Smart, N. P. Lattice attacks on digital signature schemes. *DCC 23*, 3 (Aug 2001), 283–290.

[28] Hu, W.-M. Reducing timing channels with fuzzy time. *Journal of computer security 1*, 3-4 (1992), 233–254.

[29] Karlof, C., and Wagner, D. *Hidden Markov model cryptanalysis.* Springer, 2003.

[30] Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology–CRYPTO 96* (1996), Springer, pp. 104–113.

[31] Lampson, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (1973), 613–615.

[32] Latham, D. C. Department of defense trusted computer system evaluation criteria.

[33] Lindner, R., and Peikert, C. Better key sizes (and attacks) for LWE-based encryption. In *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings* (2011), vol. 6558 of *Lecture Notes in Computer Science*, pp. 319–339.

[34] Liu, M., and Nguyen, P. Q. Solving BDD by enumeration: An update. In *Topics in Cryptology–CT-RSA 2013*. 2013, pp. 293–309.

[35] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. *Handbook of applied cryptography.* CRC press, 1996.

[36] Nguyen, P. Q., and Shparlinski, I. E. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology 15*, 2 (Jun 2002), 151–176.

[37] Nguyen, P. Q., and Shparlinski, I. E. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *DCC 30*, 2 (Sep 2003), 201–217.

[38] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In *2006 CT-RSA* (2006).

[39] Page, D. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive 2002* (2002), 169.

[40] Percival, C. Cache missing for fun and profit. In *BSDCan 2005* (Ottawa, CA, 2005).

[41] Van de Pol, J., Smart, N. P., and Yarom, Y. Just a little bit more. In *2015 CT-RSA* (San Francisco, CA, USA, Apr 2015), pp. 3–21.

[42] Popp, T. An introduction to implementation attacks and countermeasures. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign* (2009), pp. 108–115.

[43] Project, B. Boringssl.

[44] Project, L. Libressl.

[45] Schnorr, C. P., and Euchner, M. Lattic basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Prog. 66*, 1–3 (Aug 1994), 181–199.

[46] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (2004), CCS '04, pp. 298–307.

[47] Uht, A. K., Sindagi, V., and Hall, K. Disjoint eager execution: An optimal form of speculative execution. MICRO 28, pp. 313–325.

[48] Waldspurger, C. A. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.* (Dec 2002), 181–194.

[49] Walter, C. D. Longer keys may facilitate side channel attacks. In *SAC* (Waterloo, ON, Canda, Aug 2004), pp. 42–57.

[50] Wuille, P. Dealling with malleability. `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`, Mar. 2014.

[51] Yarom, Y., and Benger, N. Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, Feb 2014.

[52] Yarom, Y., and Falkner, K. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security* (San Diego, CA, US, 2014), pp. 719–732.

[53] Yarom, Y., Genkin, D., and Heninger, N. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES* (2016).

Appendix A

First appendix

Figures A.1, A.2 and A.3 show the patches submitted to OpenSSL, LibreSSL and BoringSSL respectively.

```
From 690706448bfecb1c73e8ff320d6289bd55370c04 Mon Sep 17 00:00:00 2001
From: Cesar Pereida <cesar.pereida@aalto.fi>
Date: Mon, 23 May 2016 12:45:25 +0300
Subject: [PATCH] Fix DSA, preserve BN_FLG_CONSTTIME

---
 crypto/dsa/dsa_ossl.c |    9 +++++----
 1 file changed, 5 insertions(+), 4 deletions(-)

diff --git a/crypto/dsa/dsa_ossl.c b/crypto/dsa/dsa_ossl.c
index ce1da1c..beb62b2 100644
--- a/crypto/dsa/dsa_ossl.c
+++ b/crypto/dsa/dsa_ossl.c
@@ -204,10 +204,6 @@ static int dsa_sign_setup(DSA *dsa, BN_CTX *ctx_in,
             goto err;
     } while (BN_is_zero(k));

-    if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
-        BN_set_flags(k, BN_FLG_CONSTTIME);
-    }
-
     if (dsa->flags & DSA_FLAG_CACHE_MONT_P) {
         if (!BN_MONT_CTX_set_locked(&dsa->method_mont_p,
                                     dsa->lock, dsa->p, ctx))
@@ -238,6 +234,11 @@ static int dsa_sign_setup(DSA *dsa, BN_CTX *ctx_in,
     } else {
         K = k;
     }
+
+    if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
+        BN_set_flags(K, BN_FLG_CONSTTIME);
+    }
+
     DSA_BN_MOD_EXP(goto err, dsa, r, dsa->g, K, dsa->p, ctx,
                    dsa->method_mont_p);
     if (!BN_mod(r, r, dsa->q, ctx))
--
1.7.9.5
```

Figure A.1: Patch submitted to OpenSSL fixing the software defect in `crypto/dsa/dsa_ossl.c`.

```
From a0014f43a682c11e0002ed12ae361da61f2270e0 Mon Sep 17 00:00:00 2001
From: Cesar Pereida <cesar.pereida@aalto.fi>
Date: Fri, 27 May 2016 10:27:42 +0300
Subject: [PATCH] Fix for CVE-2016-2178

---
 src/lib/libssl/src/crypto/dsa/dsa_ossl.c |   12 +++++++++---
 1 file changed, 9 insertions(+), 3 deletions(-)

diff --git a/src/lib/libssl/src/crypto/dsa/dsa_ossl.c b/src/lib/libssl/src/crypto/dsa/dsa_ossl.c
index b3eefcc..cb6be55 100644
--- a/src/lib/libssl/src/crypto/dsa/dsa_ossl.c
+++ b/src/lib/libssl/src/crypto/dsa/dsa_ossl.c
@@ -247,9 +247,6 @@ dsa_sign_setup(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp, BIGNUM **rp)
  if (!BN_rand_range(&k, dsa->q))
  goto err;
  } while (BN_is_zero(&k));
-if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
-BN_set_flags(&k, BN_FLG_CONSTTIME);
-}

  if (dsa->flags & DSA_FLAG_CACHE_MONT_P) {
  if (!BN_MONT_CTX_set_locked(&dsa->method_mont_p,
@@ -283,6 +280,15 @@ dsa_sign_setup(DSA *dsa, BN_CTX *ctx_in, BIGNUM **kinvp, BIGNUM **rp)
  } else {
  K = &k;
  }
+
+ /* Fix for CVE-2016-2178
+     * Setting the BN_FLG_CONSTTIME flag should occur immediately before
+     * the exponentiation code.
+     */
+ if ((dsa->flags & DSA_FLAG_NO_EXP_CONSTTIME) == 0) {
+ BN_set_flags(K, BN_FLG_CONSTTIME);
+ }
+
  DSA_BN_MOD_EXP(goto err, dsa, r, dsa->g, K, dsa->p, ctx,
      dsa->method_mont_p);
  if (!BN_mod(r,r,dsa->q,ctx))
--
1.7.9.5
```

Figure A.2: Patch submitted to LibreSSL fixing the software defect in `crypto/dsa/dsa_ossl.c`.

```
From 8593061ae8f4b6b2f4cd0fad03be3e5eabdda7e6 Mon Sep 17 00:00:00 2001
From: Cesar Pereida <cesar.pereida@aalto.fi>
Date: Fri, 27 May 2016 09:53:09 +0300
Subject: [PATCH] Fix for CVE-2016-2178

---
 crypto/dsa/dsa.c |    8 ++++++--
 1 file changed, 6 insertions(+), 2 deletions(-)

diff --git a/crypto/dsa/dsa.c b/crypto/dsa/dsa.c
index fe29aa0..6eca7c5 100644
--- a/crypto/dsa/dsa.c
+++ b/crypto/dsa/dsa.c
@@ -819,8 +819,6 @@ int DSA_sign_setup(const DSA *dsa, BN_CTX *ctx_in, BIGNUM **out_kinv,
     }
   } while (BN_is_zero(&k));

-  BN_set_flags(&k, BN_FLG_CONSTTIME);
-
   if (!BN_MONT_CTX_set_locked((BN_MONT_CTX **)&dsa->method_mont_p,
                               (CRYPTO_MUTEX *)&dsa->method_mont_p_lock, dsa->p,
                               ctx)) {
@@ -847,6 +845,12 @@ int DSA_sign_setup(const DSA *dsa, BN_CTX *ctx_in, BIGNUM **out_kinv,

   K = &kq;

+  /* Fix for CVE-2016-2178
+   * Setting the BN_FLG_CONSTTIME flag should occur immediately before
+   * the exponentiation code.
+   */
+  BN_set_flags(K, BN_FLG_CONSTTIME);
+
   if (!BN_mod_exp_mont(r, dsa->g, K, dsa->p, ctx, dsa->method_mont_p)) {
     goto err;
   }
--
1.7.9.5
```

Figure A.3: Patch submitted to BoringSSL fixing the software defect in `crypto/dsa/dsa.c`.