Cache Timing Attacks on DSA

Shourya Pandey (160050013) [1], **Animesh Bohara (160050040)** [1],
Eashan Gupta (160050045) [1] , **Ankit (160050046)** [1], **Anmol Singh (160050107)** [1]

[1]Department of Computer Science and Engineering,
Indian Institute of Science Bombay, Mumbai, India

Abstract. A timing attack is a security exploit in which the attacker attempts to compromise a cryptosystem by analysing the time taken to execute cryptographic algorithms. The memory cache can be used as a side-channel, and this could lead to the recovery of secret information. This report explains some of the common cache-timing techniques that is used to exploit vulnerable software.
It is known that the Sliding Window Exponentiation (SWE) implementation of the Digital Signature Algorithm (DSA) in OpenSSL 0.9.7g is vulnerable to cache-timing techniques. OpenSSL was then updated and in OpenSSL 1.0.2h to have an option for a "constant-time" implementation of modular exponentiation, with a fixed-window implementation, and was also updated to have an exponent of fixed length. However, this introduced a new software defect, in which the execution continues with the insecure SWE code path for DSA implementation.
We attempt to exploit this implementation of DSA in OpenSSL 1.0.2h using a FLUSH+RELOAD cache timing attack. The aim of our attack is to trace and recover side-channel information of the SWE algorithm that reveals the sequence of *squares* (S) and *multiplications* (M) performed, and from this sequence we recover a few bits that can be used for a lattice attack and its formulation to the Hidden Number Problem (HNP).

1. Introduction

Digital Signature Algorithm, DSA, is a widely accepted standard for digital signatures which is based on the algebraic properties of modular exponentiation and the discrete logarithm problem. The software implementation of this algorithm in OpenSSL 1.0.2h is known to be susceptible to a FLUSH+RELOAD attack. In this project we have tried to implement this attack and guess the key.

The aim of our project was to learn about various side channel cache timing attacks and implement them. We read about various cache timing attacks and tried to implement and experiment with them. We also read about the various victim algorithms such as the DSA and the SWE (which is a step of the DSA), and used various toolkits and open source codes for a better understanding and for gaining insights for our

codes. In some cases we also tweaked and ran these tool-kits and codes for better understanding of the implementations.

2. Background

Digital signature schemes are the digital analogue for handwritten signatures. A valid digital signature gives a recipient reason to believe that the message was created by some known sender (**authentication**), the sender cannot deny having sent the message (**non-repudiation**), the message was not altered in transition (**integrity**).
A digital signature scheme consists of four algorithms:

- A *domain parameter generation algorithm* that creates the set D of required parameters
- A *key generation algorithm* that generates the public and private key pairs (y, α)
- A *signature generation algorithm* that outputs a signature Σ
- A *signature verification algorithm* that accepts or rejects the signature Σ

2.1. Digital Signature Algorithm

In the DSA algorithm, the message is signed using a private key and then verified using a public key. It was first proposed by the U.S. National Institute of Standards and Technology (NIST) and is a variant of the ElGamal signature scheme. We have summarised the sub-algorithms used in DSA. We have also shown the algorithm of the Sliding Window Exponentiation implementation of the DSA Algorithm.

Algorithm 1: DSA domain parameter generation

Input: Security parameters L, N
Output: DSA domain parameters (p, q, g)
begin
  $p \leftarrow L$-bit prime
  $q \leftarrow N$-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**
    $h \leftarrow [1, p - 1]$
    $g \leftarrow h^{(p-1)/q} \bmod p$
  end
  return (p, q, g)
end

Algorithm 2: DSA Key and Signature Generation

Input: DSA domain parameters (p, q, g), message m, secure hash H
Output: DSA signature (r,s)
begin
  $\alpha \in_R [1, q-1]$
  $y = g^{\alpha} \bmod p$
  $k \in_R [1, q-1]$
  $r \leftarrow (g^k \bmod p) \bmod q$
  if $r = 0$ **then**
    goto 3
  end
  $h \leftarrow H(m)$
  $s \leftarrow k^{-1}(h + \alpha r) \bmod q$
  if $s = 0$ **then**
    goto 3
  end
  return (r, s)
end

Algorithm 3: DSA Signature Verification

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

2.1.1. Sliding Window Algorithm

Algorithm 4: SWE algorithm

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: $b^e \bmod m$
$g[0] \leftarrow b \bmod m$;
$s \leftarrow MULT(g[0], g[0]) \bmod m$;
for $j \leftarrow 1$ *to* 2^{w-1} **do**
  $g[j] \leftarrow MULT(g[j - 1], s) \bmod m$;
end
$r \leftarrow 1$
for $i \leftarrow n$ *to* 1 **do**
  for $j \leftarrow 1$ *to* $L(e_i)$ **do**
    $r \leftarrow SQUARE(r) \bmod m$
  end
  if $e_i \neq 0$ **then**
    $r \leftarrow MULT(r, g[(e_i - 1)/2]) \bmod m$;
  end
end
return r;

2.2. Cache Timing Attacks

Attacks on a system are defined as processes which have a single goal to obtain information about another process. These attacks are generally malicious attacks with the aim to compromise the security of the victim process. Timing attacks are a specific type of side-channel attacks that are implemented by measuring the time of execution of the victim and exploiting the difference in the executions of these processes.

Cache timing attacks are also timing attacks which exploit the fact that the memory access time from cache is less than that from main memory. These attacks reveal information about the victim process based on the time difference between cache hits and misses. They rely on cache sharing between processes which is achieved through the use of shared libraries. Once a library is loaded in memory, and another process loads the library, the same physical memory location is used instead of re-loading the library in physical memory, resulting in shared access of instructions in the cache by the attacker and the victim.

In our project, we studied mainly three types of cache timing attacks which can be used to attack various cryptographic algorithms.

2.2.1. Evict+Time Attack

Evict+Time technique manipulates the cache before and after running the cryptographic algorithm implementation. It measures the time difference of running this technique multiple times to guess the encryption key used by the algorithm.
For a chosen plain-text p to be encrypted, the technique is as follows:

- *Empty* or replace cache
- *Trigger* encryption of p and measure execution time
- *Manipulate* cache to replace some lines in the cache
- *Trigger* encryption of p again and measure execution time

This type of attack works in the case where the algorithm refers to pre-calculated data such as the AES algorithm. After the first encryption, the AES look-up tables are loaded in cache. When cache is manipulated, we specifically remove some part of the lines specifically using our own data. As memory locations of this data are known, we know which line in cache were replaced. Running the encryption for same plain-text and measuring the time difference due to cache hits and misses, we can determine some information regarding the AES key. Similarly, by running it again and again, we can guess the key of the algorithm.

2.2.2. Prime+Probe Attack

The Prime+Probe technique differs from the Evict+Time technique as it does not depend on the execution time for encryption but instead the various memory access times.
For a chosen plain-text p to be encrypted, the technique works as follows:

- *Fill* the cache with your own known data
- *Trigger* encryption for the plain-text p
- *Probe* the cache to access our own data again

The attack uses the time taken to access the memory which was filled earlier and depending on the memory access times, we can determine for cache hits and misses and thus conclude which lines the encryption algorithm replaced. This data reveals information regarding the encryption process when used properly.

2.2.3. Flush+Reload Attack

The main aim of our project was the Flush+Reload technique. This technique unlike others identifies access to memory lines, giving it high resolution, high accuracy and high signal-to-noise ratio. This technique works as follows:

- *Flush* the address
- *Wait* for the victim
- *Read* the address

- Note the time taken in reading the address

If victim accesses the memory line, the line will be available in cache and the reload operation will take less time. On the other hand, if the victim didn't access the memory line, the time taken will be more as in that case, it would be fetched from memory.
The main code used for this purpose is:

```
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)
```

Since modern processors execute instructions out of order, `lfence` instruction ensures that the instruction written before it are executed before its execution. 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.

3. Implementation

Attempt 1 was run on an Intel ® Core ™ i7-8550U quad-core processor
Attempts 2 and 3 were run on an Intel ® Core ™ i5-6300HQ quad-core processor

3.1. Attempt 1- MASTIK toolkit

Mastik v0.02 Aye Aye Cap'n [Yarom] is an open-source toolkit for experimenting with the cache timing attacks. It is a toolkit which provides various pre-defined functions which can be used to analyze memory access times, flush and monitor various memory locations.

The toolkit also provided some examples of code to analyze some algorithms. We tweaked some of the examples from the demos provided to run the Flush+Reload and observe the results. We also studied the code and function implementations of all the helper functions provide.

3.1.1. Experiment 1

We observed the code provided to analyze the GnuPG 1.4.13 implementation for modular exponentiation. This example is explained in detail in their documentation [Mastik:Documentation] also. We used this as a base to run our own implementation of modular exponentiation.

The code marks and monitors the line addresses of a binary file provided by us to be executed. The function `fr_trace` runs the binary file and continuously probes the mapped and marked addresses using the Flush+Reload technique to get the memory access times. The lines marked by us are respectively before and after the branch condition which depends on the bit of the exponent. By observing the access times of these two memory locations, we can guess the key or the exponent. Although this method is not very accurate, as the graph does not give the desired results every time, but it still gives reasonable results several times.
In Figure 1 we used a 30-bit exponent of the pattern 000011..110000011..11. The result can be observed as the line after branch takes less time when the step reaches a 1 bit. From the graph, we can guess the shape of exponent.

3.1.2. Experiment 2

We also wrote a code to probe a file continuously and based on the memory access time, we guessed if the file was accessed or not. Similarly we can observe multiple files concurrently and based on these we can collect a lot of information based on the order of access of these files. The function `fr_probe` gives the time taken to access the memory address which is used time and again for this.

3.2. Attempt 2 - Defuse

We started out by trying the basics of a side-channel attack. We checked out the repository by [Hornby], which implements an L3 cache side-channel attack, and is based on the paper by Yuval Yarom and Katrina Falkner [Yarom 2013]. Following their implementation, we implemented our own spy program which tries to find the branch taken based on the cache, using FLUSH + RELOAD . Our simple implementation worked with great success on a simple if-else code. We then proceeded to find the exponent in the modular exponentiation algorithm.

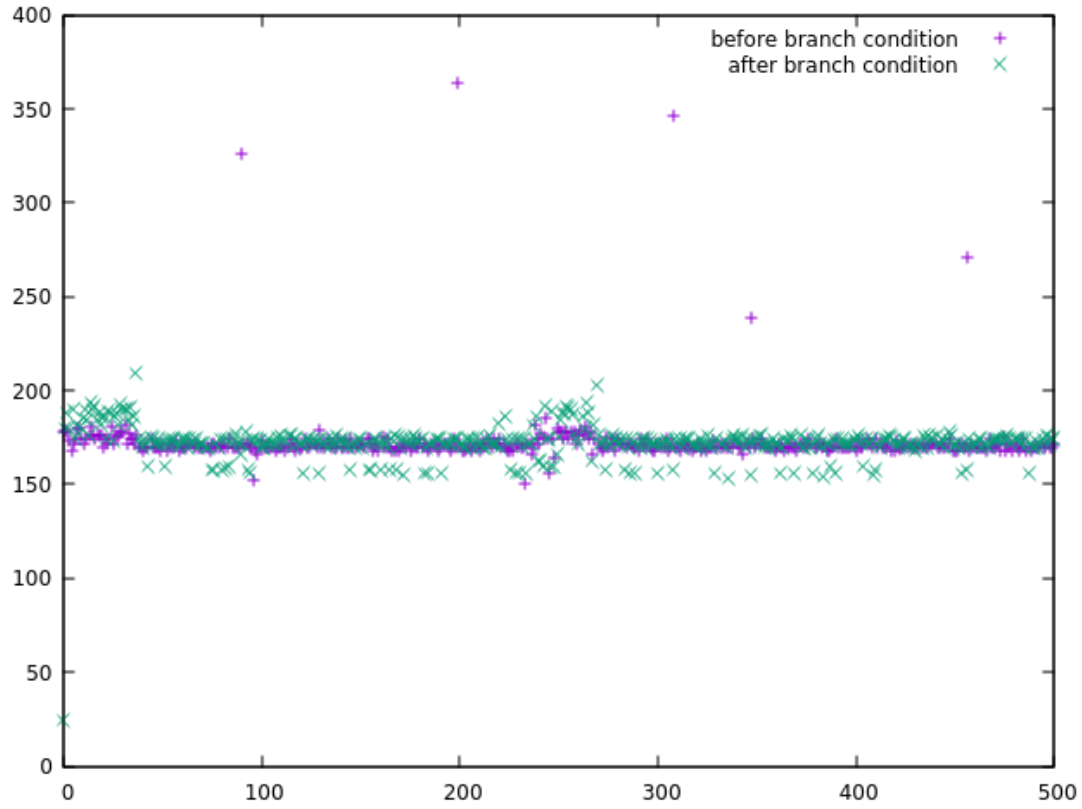

Figure 1. Memory access times for the two line addresses with 30 bit exponent

In our exponentiation algorithm, we made sure that the branches are sufficiently separated so as to ensure that the code in the branches are in different blocks in the cache. This should make sure that the cache side-channel attack works. This assumption is also reasonable because the branches in the implementation of modular exponentiation in OpenSSL are also considerably apart from each other.
Our attempt, however, was unsuccessful. The actual exponent and the exponent that our spy predicted were nowhere near close. Our guess is that the processor performs numerous optimizations, due to which the cache is not as straight-forward as we expected it to be.

3.3. Attempt 3

Following up with our previous attempt, we downloaded an old source code of **OpenSSL**, namely version 1.0.2h, which is known to be vulnerable to side channel attacks. We built this source code with debugging symbols enabled, which makes finding the virtual addresses of functions easier, helping with the attack.
We tweaked the spy program from the flush-reload-attacks repository by defuse

[Hornby] and analyzed the output of the spy with a Python script to plot the Probe time vs Time for the square and multiply operations in OpenSSL. This was done in conjunction with a **Performance Degradation attack** running simultaneously.

3.3.1. Spy

The spy takes as arguments

- The virtual addresses of the square and multiply functions respectively of the OpenSSL binary, found using gdb
- Slot size, which is the number of cycles in a slot, the time period of the FLUSH+RELOAD routine

It works as follows:

- Loads the binary using the *elf library* and then maps the binary into its own virtual address space using the `map` function
- Moves on to the `attackLoop` function which flushes the probe addresses and
- In every slot, time the access to all the probe addresses and flush them, using the assembly code given in Section 2.2.3

The output of the spy program is redirected to the python script, which plots the data for the square and multiply probes.

3.3.2. Performance Degradation

Observing the OpenSSL assembly code for multiplication and squaring, we found some labels such as *.L1st4x, .Louter4x, .Linner4x, .Lsqr4x inner, .Lsqr4x_shift_n_add, .L8x_reduce* which were frequently accessed during these operations. The memory addresses of these labels were found using gdb. The performance degradation process continuously flushes these memory addresses from the cache, so that these would have to be fetched from memory, slowing the encryption process considerably, thus ensuring a good trace by the spy program.

4. Results

In Figure 1, the Latency plot for Attempt 3 is shown for a single run of DSA encryption of a certificate using a *2048-bit* private key by the following command :

$ /usr/local/ssl/bin/openssl dgst -dss1 -sign ∼/dsa.pem -out ∼/lsb-release.sig /etc/lsb-release

while the spy was running in a different terminal, and the performance degrading attack

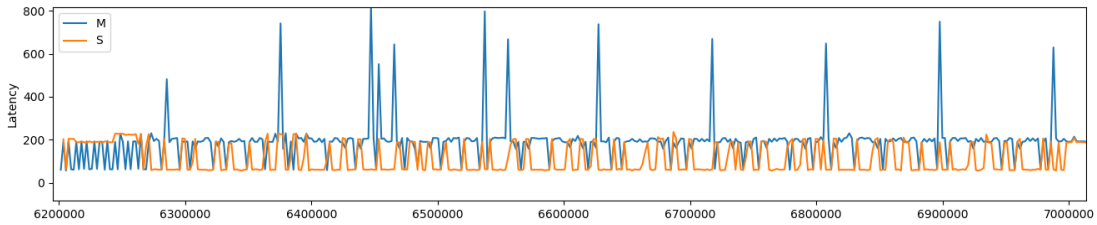

Figure 2. Latency plot for Attempt 3.
M stands for Multiplication operations, S stands for Square operations

was running in another terminal.
The first phase where the latency of Multiplication operations is small is the precomputation phase of the SWE algorithm and marks the start of the algorithm. The next phase denotes the main loop of the algorithm, which includes $L(e_i)$ square operations followed by a multiply operation for $i \in 1..n$, where $L(e_i)$ is the length of the i^{th} window, and n is the number of windows.
In the second phase, whenever the multiply signal is low, denotes that the victim accessed the multiply operation before that probe, implying that at least one bit in this window is non-zero.

```
animesh@animesh:~/iitb/sem5/cs305/Project/flush-reload-attacks/flush-reload/myversion$ gdb
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
(gdb) file /usr/local/ssl/bin/openssl
Reading symbols from /usr/local/ssl/bin/openssl...done.
(gdb)
(gdb) break bn_sqr8x_mont
Breakpoint 1 at 0x11f920: file x86_64-mont.s, line 645.
(gdb) break bn_mul4x_mont
Breakpoint 2 at 0x11f3c0: file x86_64-mont.s, line 233.
(gdb)
```

Figure 3. Getting the virtual address of functions for probing in the spy using gdb

5. Conclusion

Using the graph obtained, a few least significant bits can be estimated for the random number k used in DSA algorithm. Using these least significant bits over many different executions, the problem can be formulated to Hidden Number problem which can be solved using Closest Vector problem algorithm. The exact details can be found in this paper [Garcia 2016], [Ryan 2018].

6. Individual Contributions

Eashan and Anmol worked on the Mastik open-source toolkit. Shourya worked on the implementation of the spy program. Animesh and Ankit worked on analysing and using the spy code to understand and exploit the OpenSSL SWE Implementation of DSA. Everyone is aware of all work done in the project, and the contribution of each member is equally valuable.

References

Garcia, C. P. (2016). *Cache-Timing Techniques: Exploiting the DSA Algorithm*. Master's thesis, Aalto University, School of Science.

Hornby, T. Flush-reload-attacks:
`https://github.com/defuse/flush-reload-attacks`.

Mastik:Documentation. Mastik, a micro-architectural side-channel toolkit.

Ryan, K. (2018). *Return of the Hidden Number Problem*. PhD thesis, NCC Group.

Yarom, Y. Mastik version 0.02 aye aye cap'n:
`https://cs.adelaide.edu.au/~yval/Mastik/`.

Yarom, Y. (2013). Flush+reload: a high resolution, low noise, l3 cache side-channel attack.