Cache based side-channel attacks on DSA and ECDSA

Ritesh Goenka - 160050047
Abhishek Akkabathula - 160050074
Manoj Middepogu - 160050075
Sathvik Reddy Kollu - 160050077
Saiteja Nangunoori - 160050089

Abstract

Traditional cryptanalysis focuses on finding vulnerabilities in the mathematical properties and structure of the algorithms. Translating the cryptographic algorithms into real world software and hardware implementation presents new vulnerabilities. A class of such vulnerabilities arise when multiple processes share some common resources. Sharing memory pages between non-trusting processes is a common method of reducing the memory footprint of multi-tenanted systems. In this report we demonstrate that, due to a weakness in the Intel X86 processors, page sharing exposes processes to information leaks. We study various cache-timing attacks, with an emphasis on the FLUSH+RELOAD attack, which exploit this weakness to monitor access to memory lines in shared pages. We also study two cryptographic algorithms, namely DSA and ECDSA, along with their implementation in the openSSL library, as well as implement a cache-timing attack on DSA.

1 Background

In this section, we discuss basics of DSA and ECDSA, the Intel memory architecture and several cache-timing techniques which have been common used in the literature. Moreover, we discuss the steps of DSA and ECDSA whose implementation in openSSL is vulnerable to cache-timing attacks.

1.1 Digital Signature Schemes

Digital signature schemes are the digital analogue to real-life 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, which use public-private key pairs to sign and verify messages. The Digital Signature Schemes we are going to discuss below are DSA and its elliptic curve variant ECDSA.

1.1.1 The Digital Signature Algorithm (DSA)

The public domain parameters are a set of parameters (p, q, g), where p, q are prime numbers such that q divides $(p-1)$, and g is a generator of order q in the multiplicative group $GF(p)$. The private key α is an integer uniformly chosen such that $0 < \alpha < q$ and the corresponding public key y is given by $y = g^{\alpha} mod p$. A secure hash function h is used to compute the hash of the message. Calculating the private key given the public key requires solving the Discrete Logarithm (DL) problem, which for correctly chosen parameters is computationally intractable. Algorithm 2 demonstrates the procedure for generating valid private and public keys for DSA.

Algorithm 2: Key Generation for DSA

Input: DSA domain parameters (p, q, g).
Result: DSA key pair (α, y)

```
1 begin
2     α ∈_R [1, q − 1] ;
3     y ← g^α mod p
4     return(α, y)
```

A given party, A, wants to send a message m to B - the message m is not necessarily encrypted. Using his public-private key pair (α_A, y_A), A uses the procedure in Algorithm 3 to sign the message m and attaches the generated signature (r, s) to the original message m. B uses the procedure in Algorithm 4 to verify A's signature.
Line 3 in the signature generation algorithm uses modular exponentiation, which is performed using the sliding window exponentiation (SWE) algorithm in the openSSL library. Using the sliding window representation of the exponent e, Algorithm 5 computes the corresponding exponentiation through a combination of square and multiplication operations 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, the algorithm executes an extra multiplication when it reaches the LSB of a non-zero window.

Algorithm 3: DSA Signature Generation

Input: Message m, private key α_A, domain parameters (p, q, g), secure hash H.
Result: DSA signature (r, s)

```
1  begin
2      k ∈_R [1, q − 1] ;
3      r ← (g^k mod p) mod q ;
4      if r = 0 then
5          goto 1
6      h ← H(m);
7      s ← k^{-1}(h + α_A r) mod q;
8      if s = 0 then
9          goto 1
10     return(m, r, s)
```

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.

```
1  begin
2      if 0 < r < q and 0 < s < q then
3          h ← H(m);
4          w ← s^{-1} mod q;
5          u_1 ← hw mod q;
6          u_2 ← rw mod q;
7          r' ← (g^{u_1} y_A^{u_2} mod p) mod q ;
8          if r = r' then
9              return Accept;
10         else
11             return Reject;
12     else
13         return Reject;
```

The side-channel attack exploits Algorithm 5 as given below. Specifically, it tries to extract the sequence of square and multiplication operations performed during its execution. Then partial information extracted from the SM sequence is later used in tracing back the key.

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: $b^e \bmod m$.

```
1  // Pre-computation
2  g[0] ← b mod m;
3  s ← MULT(g[0], g[0]) mod m;
4  for j ← 1 to 2^{w-1} do
5      g[j] ←MULT(g[j − 1], s) mod m;
6  // Exponentiation
7  r ← 1;
8  for i ← n to 1 do
9      for j ← 1 to L(e_i) do
10         r ←MULT(r, r) mod m;
11     if e_i ≠ 0 then r ←MULT(r, g[(e_i − 1)/2]) mod m;
12 return r;
```

1.2 Elliptic Curve Digital Signature

The ECDSA is the adaptation of one step of the algorithm from the multiplicative group of a finite field to the group of points on an elliptic curve. The main benefit of using this group as opposed to the multiplicative group of a finite field is that smaller parameters can be used to achieve the same security level. This is due to the fact that the best currently known algorithms to compute the discrete logarithm problem in the finite field are sub-exponential and those used to compute the ECDLP are exponential.

Parameters: An elliptic curve E defined over a finite field $\mathbb{F}_q$; a point $G \in E$ of a large prime order n (generator of the group of points of order n). Parameters chosen are generally believed to offer a security level of $\sqrt{n}$ given current knowledge and technologies. Parameters are recommended to be generated following the Digital Signature Standard. The field size q is generally taken to be a power of 2 or a large odd prime. The implementation of OpenSSL uses both $q = 2^m$ and prime fields.

Public-Private Key pairs: The private key is an integer d, satisfying $1 < d < n - 1$ and the public key is the point $Q = dG$. Calculating the private key from the public key requires solving the ECDLP, which is hard in practice for the correctly chosen parameters. Currently the most known efficient algorithms for solving the ECDLP have a square root run time in the size of the group, hence the aforementioned security level.

The encryption of a given message m is done as follows:

1. Using an approved hash algorithm, compute $e = Hash(m)$, take $\overline{e}$ to be the leftmost l bits of e (where $l = min(\log(q)_2$, bit-length of the hash)).
2. Select $k \leftarrow_R \mathbb{Z}_n$ randomly.
3. Compute $(x, y) = kG \in E$.
4. Calculate $r = x \bmod n$; if $r = 0$ then jump to step 2.
5. Take $s = k^{-1}(\overline{e} + rd_B) \bmod n$; if $s = 0$ then return to step 2.
6. send (m, r, s)

The receiver can validate the message as follows:

1. check whether all received parameters are correct,that $r, s \in \mathbb{Z}_n$ and that sender's public key is valid, that is $Q_b \neq \mathbb{O}$ and $Q_b \in E$ is of order n.
2. Using the same above hash function , compute $\overline{e}$.
3. Compute $\overline{s} = s^{-1} \bmod n$.
4. Compute the point (x,y)= $\overline{e}\overline{s}G + r\overline{s}Q_B$.
5. Verify that $r = x \bmod n$ otherwise reject the signature.

1.3 The Montgomery Ladder

In a number of incidences the scalar is intended to remain secret. This scalar multiplication is most efficiently performed using a double-and-add method (or the related right-to-left method) as outlined in Algorithm 1.

Input: Point P, scalar n, k bits
Output: Point nP
$Q \leftarrow \mathcal{O}$
for *i from k to* 0 **do**
  $Q \leftarrow 2Q$
  if $n_i = 0$ **then**
    $Q \leftarrow Q + P$
  end
end

Algorithm 1: Double-and-add point scalar multiplication

Double-and-add methods, though efficient, are vulnerable to side-channel attacks. The addition law for points on commonly used elliptic curves is not complete. That is, the computation of $P + Q$ differs between the cases $P = Q$ and $P \neq Q$. Consequently, it is possible to distinguish when the if in the loop is executed and hence when a bit of n_i is 0. The Montgomery ladder, described in Montgomery , is presented in Algorithm 2. It is different from Algorithm 1 in that both a doubling and an addition of points occur at each step, regardless of the value of the bit. Thus, the Montgomery ladder thwarts side channel attacks which measure the computation at each bit to determine whether an addition operation was executed. The branching in Algorithm 2 controls where the addition of points is stored and which point is doubled.

Input: Point P, scalar n, k bits
Output: Point nP
$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for *i from k to* 0 **do**
  if $n_i = 0$ **then**
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow 2R_0$
  else
    $R_0 \leftarrow R_0 + R_1$
    $R_1 \leftarrow 2R_1$
  end
end

Algorithm 2: Montgomery ladder point scalar multiplication

1.4 Cache

All modern processors have multiple levels of cache intended to bridge the latency gap between main memory and the CPU. The machine targeted in this paper are Intel Core i7-6500U has three levels of cache (private L1 32KB I-cache and 32KB D-cache, 256KB L2 cache and 3MB L3 cache shared between all cores). 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. This property unlocks the possibility to perform the FLUSH+RELOAD technique, which is described in the following section.

1.5 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 cache-timing attacks exploit timing information leaked from software implementations that run in variable time.

1.5.1 Evict+Time Technique

The Evict+Time technique manipulates the cache state before each encryption and then observes the execution time of the subsequent encryption.the Evict+Time technique in a chosen-plaintext setting and using a plaintext p works as follows:

- *Trigger* an encryption of plaintext p in the target process
- *Evict* memory by accessing appropriate memory blocks
- *Trigger* a second encryption of p and time it

1.5.2 Prime+Probe Technique

The Prime+Probe technique is similar to the Evict+Time technique. In this technique, the state of the cache after encryption is examined instead of using the encryption time as a measurement score to determine the key. In a chosen-plaintext setting and using a plaintext p, the Prime+ Probe technique works as follows:

- *Trigger* an encryption of plaintext p in the target process
- *Prime* by filling up the cache
- *Probe* by reading memory addresses and measure the time

1.5.3 The Flush+Reload Technique

Unlike the Prime+Probe technique which 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. This technique works in three phases :

- *Flush* the monitored memory line from the cache.
- Wait till the victim has time to access the memory line.
- *Reload* the memory line and measures the time it takes to load.

2 Implementation

2.1 Setting up OpenSSL

Trusted old versions of OpenSSL can be found at: `https://www.openssl.org/source/old/`. The specific version used for this attack was version openssl-1.0.2h. After downloading the OpenSSL source, openSSL is configured and installed in the system using the steps given below:

```
./config -g
sudo make
sudo make install\_sw
```

For the selected version i.e. openssl-1.0.2h, the library will be installed in the `/usr/local/ssl` directory. The -g option is used to enable debugging which is later used to find address offsets of the relevant instructions in the executable.

2.2 Fetching the line address in the code-segment

To find the line address, we use an open-source debugging tool for C/C++ called GDB. The labels for square and multiply instructions are bn_sqr8x_mont and bn_mul4x_mont respectively, and these routines can be found in the file `x86_64-mont.s`. Instead of using the address of the first instruction in the above mentioned functions, we select one instruction from each function routine such that the difference between their addresses is maximum. This is done in order to prevent prefetching issues. After selecting the appropriate lines from the code, we find the line address using the following steps:

- Load the openSSL executable file with GDB.
- Run 'break file-name:line-number', where file-name is `x86_64-mont.s` and the line-number is replaced with line number of the selected instruction.

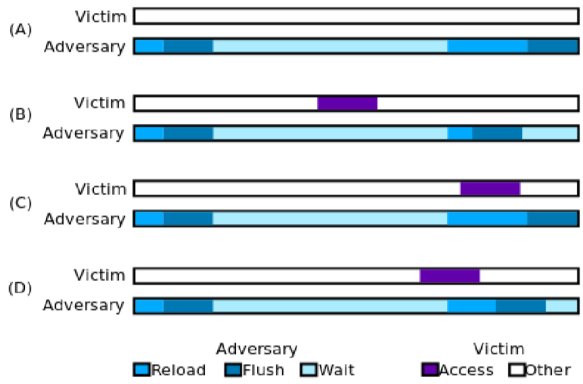

Figure 2.1: Timing of Flush+Reload. (A) No Victim Access (B) With Victim Access (C) With Victim Access Overlap (D) Partial Overlap

2.3 Implementation of spy process for DSA

2.3.1 Sharing the Library

The spy program was implemented in C. The openSSL library is mapped to the user space using the mmap function provided by the library 'sys/mman.h'. The address of the new mapping is returned as the result of the call.

```
void *mmap_base_address = mmap(NULL, size_of_file,
                          prot, flags, fd, offset);
```

2.3.2 Obtaining the VA of the instructions

The virtual address of the square and multiply instructions are found by using the relative instruction addresses obtained from GDB debugging tool and the mmap base address as below :

```
mulProbe = mmap_base_address + 0x4d7360 - text_base;
sqrProbe = mmap_base_address + 0x4d79f6 - text_base;
```

2.3.3 Slot-Time

Overlapping memory accesses by the victim and spy process leads to incorrect timing values as depicted in the Figure 2.1. So, instead of flushing and reloading the memory addresses continuously, we divide time into slots. We take the following cases into consideration while setting the the time-slot value:

- If the time-slot is too small, we face the overlapping problem mentioned above as depicted in the above figure.
- If the time-slot is too large, we may miss multiple bits while recovering.

Taking into account the above discussion, we use a time-slot value, determined using the trial and error method, that gives us the maximum number of cache hits. The time-slot value we use here is 1024.

2.3.4 Attack loop

During each time-slot, the sqrProbe and mulProbe address are accessed, the corresponding access times are recorded, the addresses are flushed and wait till the end of the time-slot. The code for probing is provided below:

```
unsigned long probe_timing(char *adrs) {
    volatile unsigned long time;

    asm __volatile__(
        "    mfence             \n"
        "    lfence             \n"
        "    rdtsc              \n"
        "    lfence             \n"
        "    movl %%eax, %%esi  \n"
        "    movl (%1), %%eax   \n"
        "    lfence             \n"
        "    rdtsc              \n"
        "    subl %%esi, %%eax  \n"
        "    clflush 0(%1)      \n"
        : "=a" (time)
        : "c" (adrs)
        : "%esi", "%edx"
    );
    return time;
}
```

The following is a snippet of our spy program:

```
while (1) {
    buffer[buffer_pos].probe_time[0] = probe_timing(sqrProbe);
    if (buffer[buffer_pos].probe_time[0] <= threshold) hit = 1;

    buffer[buffer_pos].probe_time[1] = probe_timing(mulProbe);
    if (buffer[buffer_pos].probe_time[1] <= threshold) hit = 1;
    .....
    while (rdtsc() < current_slot_end) {}
}
```

2.3.5 Determining threshold

We determine the access time threshold based on the comparison of access times of several runs with only reloading and flush and reloading. From this experiment, we conclude a value of 120, which can be seen in Figure 2.2.

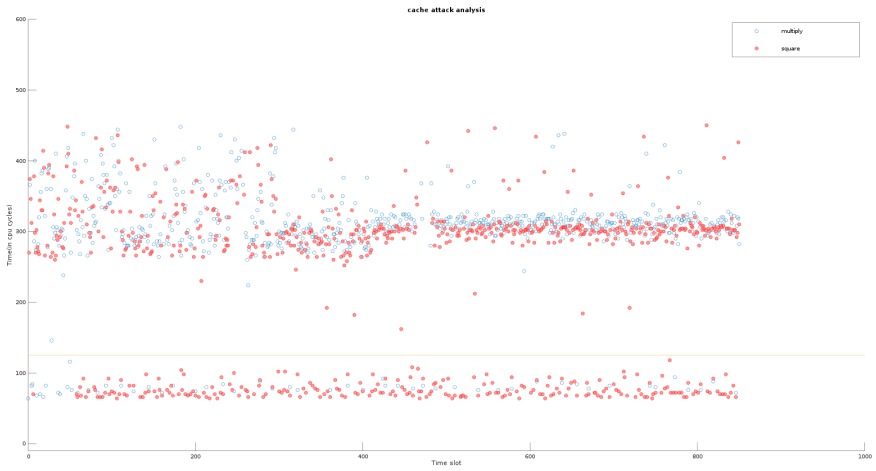

Figure 2.2: Figure showing the access times for square (red) and multiply (blue) operations. The line parallel to the x-axis shows the threshold value.

2.4 Running the Spy program

- Compile the Spy program with the lelf flag as given below:

```
gcc ourspy.c -o ourspy -lelf
```

- Run the Spy program and generate the signature on other terminal using the openSSL command given below:

```
openssl dgst -dss1 -sign
input_key file_to_be_signed > output_sign
```

2.5 Results and Inferences

Figure 2.2 gives the access time values in each slot-time for the multiply and square instructions. The red circles represent the square instruction accesses and the blue circles represent the multiplication instruction accesses. The points below the threshold represent cache-hits and those above represent cache-misses. By using the sequence of cache-hits we can construct parts of the nonce. We can observe that the threshold value set by us is consistent with the results obtained in the figure shown above. Also we can observe that the number of square instructions hits are much more than multiply instructions hits which is expected, as can be seen from the SWE algorithm discussed earlier.

Stepping through Figure 2.2, the initial low access time for the multiply operation is the multiplication for converting the base operand to Montgomery representation. The subsequent low access time for the square operation is

the temporary square value used to build the odd powers for the SWE pre-computation table. The subsequent long period of low access time for multiply operation is the successive multiplications to build the pre-computation table itself. Then begins the main loop of SWE.

2.6 Problems faced

In the initial stages of the project, we faced problems while mapping the openSSL library to the virtual address of our spy program. Later we realized that text base must be subtracted from the address to get the correct virtual address. Moreover, we were unable to simultaneously probe both square and multiply instructions due to prefetching issues. We resolved this issue by selecting instructions in the former half of the multiply function and later half of square function, instead of using the starting instructions of these functions. This effectively suppressed the prefetching of the second instruction while accessing the first one by increasing the gap in memory addresses of both instructions. Still, we were able to observe only around 5% of the possible 2048 (private key length) hits for the square instruction. We tackle the overlap issue described earlier to a certain extent by dividing the time into slots and tuning the time slot value by observing the percentage of square instruction hits. Finally, we were able to observe around 12% of the maximum possible number of square instruction hits. This low value of percentage hits can be attributed to the following reasons:

- The system activity causes the spy program to miss monitoring during some time slots in which the victim process is active. This happens because the spy process might not be scheduled on the processor at exactly the same times as the victim process. We were able to identify such time slots by noting the jumps in the cycle counter values.
- The overlap issue cannot be completely eliminated since each iteration of the for loop in the victim process does not run in exactly the same time. This leads to overlap between victim and spy access and in some cases, multiple victim accesses in the same time-slot.

2.7 Future Work

We will try to improve the captured percentage of instruction hits and implement a key recovery algorithm to construct the complete key with the data obtained from several such signing procedures performed by the same user. We also plan to carry out a similar attack on ECDSA.

2.8 Contributions

All members of the group were involved in reading the related papers, implementation of user-level to user-level process attacks as well as the attack on openSSL Digital Signature Algorithm. During this project, we constantly discussed and improvised upon several ideas.

References

[1] Cesar Pereida Garcia. *"Cache-Timing Techniques: Exploiting the DSA Algorithm", Aalto University*, June 2016.

[2] Yuval Yarom and Naomi Benger. *"Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack, The University of Adelaide"*, February 2014.

[3] Yuval Yarom and Katrina Falkner. *"FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", The University of Adelaide.*

[4] Nepoche. *"A flush-reload side channel attack implementation", GitHub*, October 2017, from `https://github.com/nepoche/Flush-Reload/issues`.

[5] Defuse. *"flush-reload-attacks", GitHub*, October 2016, from `https://github.com/defuse/flush-reload-attacks`.