Flush+Reload based L3 Cache Side-Channel Attack

Meet Kathiriya 160050001

Aman Jain 160050019

Shreyas Pimpalgaonkar 160050024

Shubham Anand 160050060

Phuntsog Wangchuk 160050109

9 November 2018

Abstract

In modern computer systems, lower-level cache pages are shared between processes running on the cores of the system to reduce memory footprint and prevent excessive redundancy. Isolation between the memory spaces of different processes is implemented by making certain shared regions read-only, which somewhat ensures that malicious processes can't make malignant changes to memory or monitor other processes' behaviour. However, due to the nature of this page sharing, certain inferences about memory-access patterns and behaviour can still be made by observing cache access delays over a period of time. This can be exploited in Intel x86 processors to keep track of the behaviour of certain processes and access restricted information. In this report, we first study and implement the intricacies of the abstractions and algorithms that underlie Flush+Reload based side-channel attacks in the implementation of RSA encryption algorithm in GnuPG 1.4.13, and the nature of side-channel attacks on shared cache pages. We evaluate the efficiency of the attacks and made observations regarding the results obtained from the attack. We implement a series of attacks using Mastik API to exploit information regarding file opening. We also test the API to exploit GnuPG and extract the private keys. Furthermore, we propose to implement side-channel attacks in other areas. We critique the drawbacks of the intel x86 processors and some of it's features. Finally, we recommend enhancements to mitigate these side channel attacks.

Contents

List of Figures

1 Introduction

In most architectures, the system software often shares identical memory pages between different processes based on their sources or content, in order to reduce memory consumption. Such sharing can be performed by actively searching and coalescing pages with identical content. The system relies on hardware mechanisms that enforce read only or copy-on-write semantics for the shared pages to maintain isolation between non-trusting processes. While the processor ensures that processes cannot change the contents of shared memory pages, it sometimes fails to block other forms of inter-process interference.

The shared use of the processor cache leads to one such form of interference. Whenever a process accesses a shared page in memory, the contents of the accessed memory location is pulled into the caches. This behaviour can be exploited by a side channel attack technique to extract information on access to shared memory pages. The processor's clflush instruction may be used to evict the monitored memory locations from the cache, and then test whether the data in these locations is back in the cache after allowing the victim program to execute a small number of instructions. We observe that the clflush instruction evicts the memory line from all the cache levels, including from the shared Last-Level-Cache (LLC). Based on this observation, a FLUSH+RELOAD attack is proposed by Yuval Yarom and Katrina Faulkner in their paper[1]. This is a cross-core attack, allowing the spy and the victim to execute in parallel on different execution cores.

Two properties of the FLUSH+RELOAD attack make it more powerful than prior micro-architectural side-channel attacks. The first is that the attack identifies access to specific memory lines, as opposed to larger classes of locations, such as specific cache sets. Consequently, FLUSH+RELOAD has high fidelity. The second advantage of the attack is that it focuses on the LLC, which is the cache level furthest from the processors cores which is shared by multiple cores on the same processor.

To evaluate the power of FLUSH+RELOAD we use it to mount an attack on the RSA implementation of GnuPG[3]. We test the attack in a same-OS scenario, where both the spy and the victim execute as processes in the same operating system.

2 Theory

2.1 Page Sharing and de-duplication

Most modern operating systems use a technique called page sharing to reduce memory footprint of processes. By this technique, the operating system avoids redundant duplication of pages. In a more aggressive form, also known as page deduplication is also employed by many operating systems including Linux and Windows. In page deduplication, the operating system continuously looks for identical pages, combines them and shares the combined page among the different processes. As a result, these processes may be unknowingly sharing the memory pages, a fact that will be used for the attacks discussed later in the report. However, these pages are implemented as copy-on-write, that is if one process wants to write to a block in the shared page, it has to first make a copy for itself and write to that copy. This is however not the case for read operations, where copying is not required.

2.2 Caches

A cache is smaller, faster memory placed closer to the processor, which stores a part of the main memory that can be accessed much faster than from the memory itself. Typical cache functions by transferring data from main memory to cache in blocks of fixed size, called cache lines. When a cache line is copied from memory into the cache, a cache entry is created. When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. If the memory location is in the cache, a 'cache hit' occurs and the data is transferred to processor directly from cache. Otherwise, a 'cache miss' occurs and the cache allocates a new entry and copies data from main memory, then the request is fulfilled from the contents of the cache.

Modern CPUs have have multiple levels of cache, usually L1, L2 and L3 cache. Cache closer to the processor is faster to access than those farther from it. The L3 cache, also called Last Level Cache (LLC), which is of interest for this report, is shared between the cores. In Intel x86 processors, this LLC cache is inclusive, which means that it contains copies of all of the data stored in the lower cache levels. Flushing data from LLC also removes the data from all other levels of cache to maintain the inclusive property.

2.3 RSA Algorithm

RSA is a commonly used cryptographic system used for encryption and signing. It involves the following algorithm:

1. Selection of two primes $p \ and \ q$ and their product n
2. Selecting a public exponent e. For GnuPG, e = 65537
3. Calculation of a private exponent d using the formula: $d = e^{-1}(mod(p-1)(q-1))$

The final encryption system consists of:

1. The public key *(n,e)*.
2. The private key *(p,q,d)*.
3. The encrypting function $E(m) = m^e mod(n)$.
4. The decrypting function $D(c) = c^d mod(n)$.

CRT-RSA is then used to optimize the decryption. By splitting the secret key d into two parts, $d_p = d\, mod(p-1)$ and $d_q = d\, mod(q-1)$, two parts of the message are computed: $m_p = c^{d_p} mod(p)$ and $m_q = c^{d_q} mod(q)$.
m is then calculated from m_p and m_q using the following formulae:

$$h = (m_p - m_q)(q^{-1} modp) modp$$
$$m = m_q + hq$$

2.4 GnuPG

GNU Privacy Guard or GnuPG is a free encryption software that's compliant with the OpenPGP standard. It is used for encryption, decryption and signing messages, and implements public key algorithms like DSA, ElGamal, and RSA. In this report we will use its RSA algorithm's implementation for the attack.

GnuPG 1.4.13 uses the Square-and-Multiply exponentiation algorithm[2] to calculate the encryption and decryption algorithms. Using a binary representation of the exponent e, we calculate a sequence of intermediate values to compute $x = b^e modm$.
Suppose the binary representation of $e = 2^{n-1}e_{n-1} + 2^{n-2}e_{n-2} + ...2^0 e_0$, square-and-multiply computes the binary representation of x $(= 2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + ...2^0 x_0)$ such that

$$x_i = b^{\lfloor e/2^i \rfloor} mod\, m$$

using the formula

$$x_{i-1} = x_i^2 b^{e_{i-1}}$$

2.5 Side Channel Attacks

A side channel attack is an attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm[6]. Classical attacks were based on knowing the plain text or the cipher text or both. Encryption devices were meant only to convert the plain text to cipher text and vice-versa. But today it is known that these devices provides additional information, the side channel information which can be used to break into the crypto-system and get the required secured data[7]. This information includes power, time of operation, electromagnetic field, faults, frequency etc. Side channel attack uses this additional information to attack a system and recover the key.

3 Flush+Reload Attack

A round of F+R attack consists of three phases.

1. The monitored memory line is flushed from the cache hierarchy by the spy program using the *clflush* instruction.
2. The spy waits to grant the victim time to access the memory line.
3. The spy reloads the memory line, measuring the time taken to load it from memory/cache.

If during the waiting phase the victim accesses the memory line, the line will be available in the cache and the reload operation will take a short time. If, on the other hand, the victim has not accessed the memory line, the line will need to be brought in from memory and the reload will take significantly longer.

By monitoring the probe times over a specified period, the spy can determine in what order the target program accessed the probed memory lines, i.e. the pattern of calls made by the target to the relevant functions. The algorithms followed by certain functions are indicative of information held privately by them, and thus by executing a Flush+Reload attack against them can possibly reveal sensitive information, like private keys and encryption schemes.

For the purpose of this project, the probed memory lines are used by the square, multiply and reduce (modulo) functions involved in RSA encryption by GnuPG. Due to the nature of GnuPG's encryption algorithm, the bits of the private key can be determined using a Flush+Reload attack.

```
 1.   function exponent(b,e,m)
 2.   begin
 3.      x <- 1
 4.      for i <- |e|-1 downto 0 do
 5.         x <- \(x^2)
 6.         x <- x mod m
 7.         if(e_i = 1) then
 8.             x <- xb
 9.             x <- x mod m
10.          endif
11.      done
12.      return x
13.   end
```

In the given code snippet, line 5 squares the bit, line 8 performs multiplication and lines 6 and 9 reduce by performing modulo. Depending on the current bit (e_i) of the public exponent e, the algorithm will either perform square and reduce only (if $e_i = 0$), or square, reduce, multiply and reduce subsequently (if $e_i = 1$).
Thus, by monitoring the access times of lines 5,6,8 and 9, we can determine the bits of the exponent e.

4 Implementation Details

4.1 GnuPG

4.1.1 Setup

GnuPG 1.4.13 has been selected for the implementation of this project (newer versions of GnuPG are impervious to F+R attacks of this nature). The following steps are required to setup said version:

1. Download GnuPG 1.4.13 using the following link:
   `http://mirror.switch.ch/ftp/mirror/gnupg/gnupg/gnupg-1.4.13.tar.gz`
2. Unzip the package in an appropriate location and run the command:

   ```
   ./configure --prefix=<location of current directory>
   ```

   This is done to prevent any interference between the current gpg installation and any existing setups.
3. Run the following commands to check for installation of gpg and checking if system has all packages needed by GnuPG:

   ```
   make && make check && make install
   ```

4. To generate a key, run the following command in */gnupg-1.4.13/bin/* and provide the required information (userID, e-mail id, password, size of key) when prompted:

   ```
   gpg --gen-key
   ```

5. To effectively encrypt and decrypt a file, run:

   ```
   gpg -se -r <userID> <file-to-be-encrypted>
   gpg -d <file-to-be-decrypted>
   ```

4.1.2 Probing lines

To find the memory line to be probed:

1. In the *text* segment of the target process's virtual address, locate the line no. of the instruction to be probed. The address of these instructions can be found using gdb (open source debugging tool) to which we provide the line number as breakpoints and use info b command in gdb to get the address of those lines (let's call it Instruction Address).
2. Note the base address from which the target's text segment starts by running the following commands in the background while the target (gpg) is running:

   ```
   cd /proc
   ps -aux | grep "gpg"
   cd <gpg's PID>
   cat maps
   ```

   The first two commands indicate gpg's process ID, using which we can access its directory in /proc and its maps file. In maps, the segment with (r-xp) tags indicate the text segment (readable, non-writable and executable), and we can clearly observe its starting address (let's call it text base).
3. In the spy process, run:

   ```
   void *mmapbase = mmap(NULL, <size of target executable>,
       PROT_READ, MAP_PRIVATE, <fd of target executable>, 0);
   ```

   This maps the target's text segment to the spy process's virtual address space and returns the *mmapbase*, the base address of the spy's newly mapped text section.
4. The line to be probed is then given by the equation:
   Probe Address = (Instruction Address - text base) + mmapbase

4.2 C Implementation

4.2.1 Concept

We implemented a spy program in C to extract the private key information used in the decryption algorithm in GnuPG[4]. We probe the following lines using the above mentioned method for probing addresses in each time slot.

```
probe 0x494b00 S #mpih-mul.c:270
probe 0x4939b5 R #mpih-div.c:329
probe 0x494620 M #mpih-mul.c:121
```

We set a threshold value to distinguish between the time taken to fetch address from the main memory and the cache. Then we record the time taken to access these addresses and compare it with the threshold value to determine its presence in the cache. We consider the probe as a hit if the time taken is less than the threshold time. Otherwise its a miss. A hit implies that the address was accessed by some other program between the previous and the current probe[5].

When GnuPG uses its decryption algorithm, we mainly get two kinds of probes. The first kind has two hits for Square and Reduce function respectively and a miss for Multiply function. The second one is when we get all three hits. The first one signifies that the secret key bit currently accessed by GnuPG in the exponential calculation for loop is set to 0. The second case occurs when the bit is set to 1.

4.2.2 Running the program

Our spy program takes a file as input which contains the directory in which our gpg executable file is located which uses RSA Encryption/Decryption, the base address of text segment of gpg process in its own virtual address space, the addresses of multiply, square and reduce operation of gpg processes in its own virtual address space on which probing will be done and the slot time which specifies the time our spy will wait between probes. We let the victim access the probe addresses.

Once input is read and the executable mapped to spy's virtual address space, it enters an indefinite loop in which it probes the calculated probe addresses and prints if any probe was hit. It then waits for the remaining slot time duration before the next probe.
Here is a snippet of our probe sequence.

```
hit = 0;
for (i = 0; i < noffsets; i++) {
          times[i] = probe(ptrs[i]);
          touched[i] = (times[i] < CUTOFF);
          hit |= touched[i];
      }
if(hit) print....
```

The probe function above takes an virtual memory address as input and calculates the time elapsed in accessing that address, followed by flushing that address from caches.

To test out implementation we will run the spy program in one terminal and leave it running. The spy will not emit any prints as there is no other program which is accessing those probe addresses. We then start our gnupg program on another terminal which decrypts a file. This results in spy printing the addresses hit in that time slot. We stop the spy program after the GnuPG decryption ends and collect the output dumped in terminal to parse it and extract key information from it.

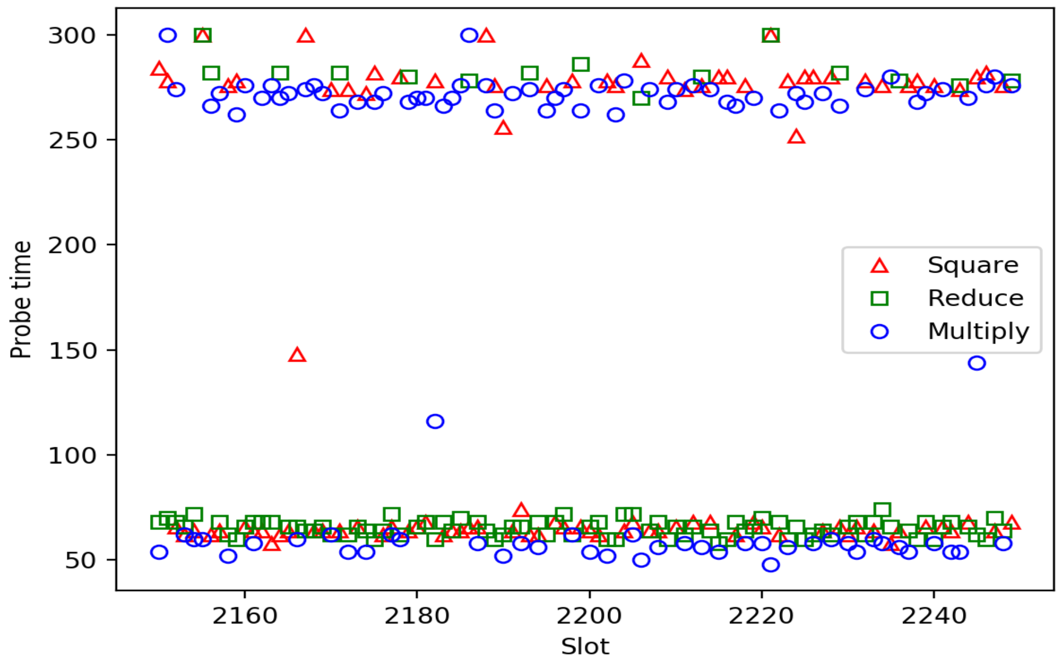

Figure 1: Attack on GnuPG using C implementation

4.2.3 Results and Inferences

It is apparent from Figure 1, that blue points (Multiply operation) which are below threshold (here 150) implies that secret key bit of exponent is set. and if point is above than threshold the bit is 0. we should also discards the slots where square operation points are above the threshold as that is false hit, since for both cases (set or unset) of exponent square must be calculated. This attack is a very high resolution attack because there is clear difference between hits and misses of the function probes. Thresholding is not a very big problem and therefore, as it can be anything between 75 and 250.

4.2.4 Challenges Faced

Although the theoretical concept of Flush + Reload is pretty straightforward, the implementation aspect of this attack had some subtle challenges. The Hardware Prefetcher which preloads the adjacent cache lines distupts the attack as it gives us false hit of Multiply probe address. We resolve this issue by choosing the address somewhere in between the middle and end of the multiply function. Since prefetchers works on concept of spacial locality it won't prefetch instructions far away from current running instruction. After picking appropriate address we were still unable to get the desired output.In idealistic senario we should only get output of the format SR or SRM, but we were getting some other type of output along with desired output as well (i.e. S or R missing from above outputs). The possible reason for getting this outputs is clashing of the probe access calculations in spy with the access in victim. This particular issue is the main reason for bit loss and can't be prevented in c and hence we tried to implement using Mastik API.

4.3 Implementation using Mastik

4.3.1 Introduction

We present the results of our experiments on Flush+Reload attack with Mastik[8], a toolkit for experimenting with micro-architectural side-channel attacks. Firstly, we demonstrate an elementary spy program using this API that detects if a file was opened by probing the memory location at the start of the file. Then we spy on a victim process that is using gnupg for carrying out it's decryption and extract the keys by probing on strategic lines in the code. The main functions in this API are documented here - `https://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`.

4.3.2 Setup

1. sudo apt-get install libdwarf-dev binutils-dev libelf-dev
2. Download and extract the archive from `https://cs.adelaide.edu.au/~yval/Mastik/`
3. execute ./configure && make && make install

4.3.3 Details

We have a file test1.c and it's executable test1, and we have a spy program that constantly probes (using flush+reload) the first line of the program. Whenever the file is opened by a user, the spy program's probe function returns a value less than the threshold. The values of the threshold is 100 and the spy returns a value of 50 when the file is opened and more than 100 otherwise. This indicates that the line is already in cache and is brought up by some other process. So by extending this mechanism of spy spying on a line whether a line was accessed we have shown a proof of concept of the flush+reload technique. Extending this technique to other lines in the program can exploit the vulnerabilities and we can thus know what lines have been exactly accessed. Thus, we extend this spy program to catch the exact bits of the private exponent of decryption in the RSA algorithm by knowing the exact lines of the functions used by it in GnuPG.

The spy program that we have created to extract the private keys of the GnuPG's implementation of RSA algorithm is almost identical to one of the demo programs provided by the Mastik API. It does the exact work that we intend to do, extract the exponent. We have changed the strategic locations of lines to be probed to get better results. Also, as the times of cache are system dependent, we had to tweak those parameters to get our results.

We create the spy program in the following steps; Firstly, create and initialise the fr_t struct that will contain the addresses of memory locations to be probed. Create a char* array with elements of the type a:b where a is the filename in which the function is and b is the line number in that file to be probed. Obtain the virtual addresses of the lines in that array so that they can be probed by using the function sym_getsymboloffset. Add those virtual memory locations in the struct by using the function fr_monitor. Finally call fr_trace function with the following arguments : maximum number of slots to probe (after which the spy stops execution), duration of every slot, hit threshold (the maximum probe time which qualifies as a hit), maximum slots for which it can be idle before exiting, and the array which stores probe times.

The fr_trace function is the most important function as it does all the probing and calls fr_probe. Initially, it loops and probes the memory locations until it gets a hit. A first hit suggests that encryption has been started by the victim process. So, we begin counting and storing the probe times in the array after that bit. There is a slot_time argument passed to the trace function that discretises the probing into slots. If a slot is missed, we put zero values in the result array, otherwise we wait for the slot to complete and keep spinning. In every slot,

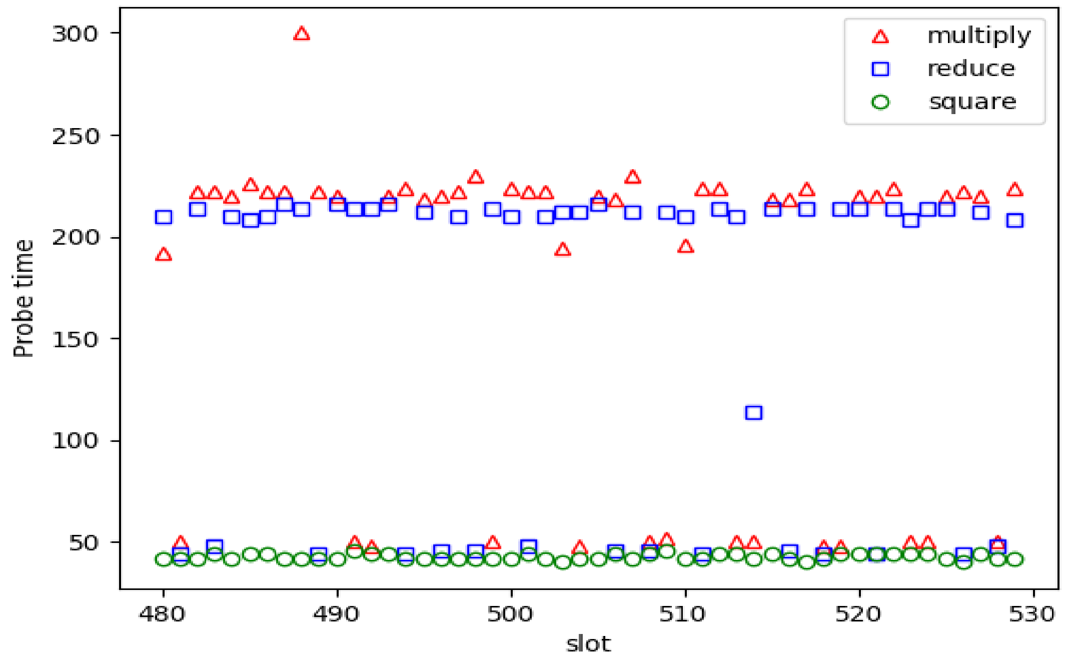

Figure 2: Attack on GnuPG using Mastik

all the specified memory locations are probed and then flushed from the cache. The function finally returns if one of the two conditions are met, either the maximum number of results are obtained, or certain amount of consecutive misses of slots have happened.

4.3.4 Results and inferences

The toolkit provides much more convenience in implementing the attack than the C implementation as described in the earlier section. We observed that the opening of the file locations works correctly, even with multiple files are being probed with multiple locations in those files. On the attack on GnuPG, we get the results as shown in the figure 2. We can clearly observe that in some slots there was a hit in some slots and miss in other slots on the multiply function. A hit is when the time required to probe the line is less than the threshold (100 here). There can be a lot of noise when the code is run due to various reasons. Eventually, we were able to get the hit slots of various functions and therefore extract the information regarding the private key used by the RSA algorithm in GnuPG.

4.3.5 Challenges faced

The toolkit was challenging to run as various intricacies were involved. Many of the challenges faced were same as in the implementation of the above section. The code was tested on a couple of machines, however ran in only one of them. We got in touch with the creators and solved the problems that we faced.

4.3.6 Future Work

We plan to implement and test other categories of side channel attacks using this API. Also, we plan on looking into cross-VM attacks[9], where the spy and the victim execute in separate, co-located virtual machines. This is made possible due to the Flush+Reload's cross-core attacking capabilities.

5 Mitigation Techniques

The Flush+Reload attack relies on a combination of four factors for its operation: data flow from sensitive data to memory access patterns, memory sharing between the spy and the victim, accurate, high-resolution time measurements and the unfettered use of the clflush instruction. Preventing any of these blocks the attack.

The main problem is that the operating system and the architecture provide us with a guarantee that the pages that are being shared are read only but in reality we are able to do operations such as flushing those lines and calculating highly accurate access time to those lines. This violates the meaning of read only. Instead the pages should be called not-write pages.

The lack of permission checks for using the clflush instruction is a weakness of the X86 architecture. To mitigate this, the operating system should check whether the process which is flushing that line or finding the access time for that instruction indeed has the write permission or not. The process violating the privilege rules either should be terminated or the operating system should consider it as a write operation and use mechanism of copy on write on those pages.

Another flaw in the architecture design which leads to the success of this attack revolves around the inclusive property of cache design. The page is shared only in the L3 cache. Flushing any line from the shared page in the spy process removes that line from the L1 and L2 cache of spy process (which is fine) but also evicts that line from the L1 and L2 cache of the core on which the victim process is running. This implies that we have access to the L1 and L2 cache of some other core. To mitigate this, x86 should either shift to non-inclusive cache or should share pages with the privileges as described above.

6 Contributions

We have two major partitions in this project, the C implementation and the one using Mastik API. All of the team members have more or less equal contribution to both the parts as they were implemented one after the other.

References

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", *23rd USENIX Security Symposium* , July 18, 2013

[2] D.E. Knuth, "Seminumerical Algorithms", *The Art of Computer Programming*, vol. 2, Edition. 3, 1997

[3] D. Ge, D. Mally and N. Meyer, "PLUNGER: Reproducing FLUSH+RELOAD: A High-Resolution, Low-Latency Side Channel Attack On GnuPG", *University of Pennsylvania*, Dec. 17, 2014

[4] J. Phukan, K.F. Li and F. Gebali, "Hardware Covert Attacks and Countermeasures", *Advanced Information Networking and Applications (AINA) IEEE 30th International Conference*, pg. 1051-1054, 2016

[5] H.J. Mahanta, S. Ahmed and A. Kumar Khan, "A randomization based computation of RSA to resist power analysis attacks", *Intelligent Systems and Control (ISCO) 11th International Conference*, pg. 328-331, 2017.

[6] B.M. Damian, Z. Hascsi and A.B. Săndulescu, "Presilicon evaluation on correlation power analysis attacks and countermeasures", *Design and Technology in Electronic Packaging (SIITME) IEEE 23rd International Symposium*, pg. 313-317, 2017.

[7] M.A.M. Sayed, L. Rongke and Z. Ling, *Communications and Networking*, vol. 210, pg. 355, 2018.

[8] Y. Yarom, "Mastik: A Micro-Architectural Side-Channel Toolkit", *The University od Adelaide*, Sept. 2016. [Online]. Available: `http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`

[9] G.I. Apecechea, M.S. Inci, T. Eisenbarth and B. Sunar, "Fine grain Cross-VM Attacks on Xen and VMware are possible!", *Cryptology ePrint Archive, Report 2014/248*, 2014. [Online]. Available: `http://eprint.iacr.org/`