INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

COMPUTER ARCHITECTURE PROJECT

Software-based Microarchitectural Attacks

Suraj Soni : 160050092
Aditya Mahto : 160050094
Sai Teja Talluri : 160050098
Chaithanya Naik : 160050102
Jagadeep Sai : 160050104

November 23, 2018

Abstract

Modern processors are highly optimized systems where every single cycle of computation time matters. Many optimizations depend on the data that is being processed. Software-based microarchitectural attacks exploit effects of these optimizations. Microarchitectural fault attacks exploit the physical imperfections of modern computer systems. We show that imperfections of the hardware, introduced by optimizations on a microarchitectural level, undermine system security and software security. The hardware leaks part of its internal state including potentially secret information through differences in behavior and timing. We discuss FLUSH+RELOAD attack, Cache Template attack and Cross-CPU attacks exploiting DRAM addressing.

Contents

Chapter 1

FLUSH+RELOAD: A L3 Cache Side-Channel Attack

1.1 Introduction

Sharing memory pages between non-trusting processes is a common method of reducing the memory footprint of multi-tenanted systems. Due to a weakness of X86 processors, this exposes processes to information leaks. Here we'll discuss and demonstrate FLUSH+RELOAD, a cache side-channel attack technique that exploits this weakness to monitor access to memory lines in shared pages on a victim running GnuPG, and thereby extracting private encryption keys.

1.2 Preliminaries

Here we'll discuss on page sharing and the implementation of RSA in GnuPG.

1.2.1 Page Sharing

OS employ page sharing in order to reduce memory footprint by avoiding replicated copies of identical content. They are of two types.

In *content-aware sharing*, identical pages are identified by the disk location the contents of the page is loaded from. This is the traditional form of sharing in an operating system, which is used for sharing the text segment of executable files between processes executing it and when using shared libraries.

Content-based page sharing, also called memory de-duplication, is a more aggressive form of page sharing. When using de-duplication, the system scans the active memory, identifying and coalescing unrelated pages with identical contents. This is usually beneficial when performing a cross VM attack.

1.2.2 RSA

RSA is a public-key cryptographic system that supports encryption and signing. Generating an encryption system requires the following steps:

- Randomly selecting two prime numbers p and q and calculating $n = pq$.
- Choosing a public exponent e.

- Calculating a private exponent $d \equiv e^{-1} (\mathrm{mod}\, (p-1)(q-1))$

The generated encryption system consists of:

- The public key is the pair (n, e).
- The private key is the triple (p, q, d).
- The encrypting function is $E(m) = m^e \bmod n$.
- The decrypting function is $D(c) = c^d \bmod n$.

CRT-RSA is a common optimization for the implementation of the decryption function. It splits the secret key d into two parts $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, computes two parts of the message: $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$. m is then computed from m_p and m_q using Garner's formula

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

ALGORITHM : Calculating $b^e \bmod m$

```
1: x ← 1
2: for i ← |e| − 1 downto 0 do
3:     x ← x^2
4:     x ← x mod m
5:     if e_i = 1 then
6:         x ← xb
7:         x ← x  mod m
8:     end if
9: end for
10: return x
```

As can be seen from the implementation, computing the exponent consists of sequence of Square and Multiply operations, each followed by a Modulo Reduce. This sequence corresponds directly with the bits of the exponent. Each occurrence of Square-Reduce-Multiply-Reduce within the sequence corresponds to a bit whose value is 1. Occurrences of Square-Reduce that are not followed by a Multiply correspond to bits whose values are 0. Consequently, a spy process that can trace the execution of the square-and-multiply exponentiation algorithm can recover the exponent.

As GnuPG uses the CRT-RSA optimization, the spy process can only hope to extract d_p and d_q. However, for an arbitrary message m, $(m - m_e^{d_p})$ is a multiple of p. Hence, knowing d_p (and, symmetrically, d_q) is sufficient for factoring n and breaking the encryption.

1.3 The FLUSH+RELOAD Technique

The FLUSH+RELOAD technique relies on sharing pages between the spy and the victim processes. With shared pages, the spy can ensure that a specific memory line is evicted from the whole cache hierarchy. The spy uses this to monitor access to the memory line.

```
1  int probe(char *adrs) {
2    volatile unsigned long time;
3
4    asm __volatile__ (
5      "  mfence             \n"
6      "  lfence             \n"
7      "  rdtsc              \n"
8      "  lfence             \n"
9      "  movl %%eax, %%esi  \n"
10     "  movl (%1), %%eax   \n"
11     "  lfence             \n"
12     "  rdtsc              \n"
13     "  subl %%esi, %%eax  \n"
14     "  clflush 0(%1)      \n"
15     : "=a" (time)
16     : "c" (adrs)
17     :  "%esi", "%edx");
18   return time < threshold;
19 }
```

FIGURE 1.1: Code for the FLUSH+RELOAD Technique

A round of attack consists of three phases. During the first phase, the monitored memory line is flushed from the cache hierarchy. The spy, then, waits to allow the victim time to access the memory line before the third phase. In the third phase, the spy reloads the memory line, measuring the time to load it. If during the wait 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 from memory and the reload will take significantly longer.

The implementation of the attack is in Figure 1.1. The code measures the time to read the data at a memory address and then evicts the memory line from the cache. The threshold used in the attack is system dependent, which can be figured out.

The purpose of the mfence and lfence instructions in lines 5, 6, 8 and 11 is to serialise the instruction stream. The processor may execute instructions in parallel or out of order. Without serialisation, instructions surrounding the measured code segment may be executed within that segment.

To use the FLUSH+RELOAD technique the spy and the victim processes need to share both the cache hierarchy and memory pages. In a non-virtualised environment, to share the cache hierarchy, the attacker needs the ability to execute software on the victim machine. The attacker, however, does not need elevated privileges on the victim machine. For a virtualised environment, the attacker needs access to a guest co-located on the same host as the victim guest.

For sharing memory pages in system that use contentaware sharing, the attacker needs read access to the attacked executable or shared libraries. In systems that support de-duplication the attacker needs access to a copy of the attacked files. De-duplication will coalesce pages from these copies with pages from the attacked files.

1.4 Attacking GnuPG

In this section we describe how we use the FLUSH+RELOAD technique to extract the components of the private key from the GnuPG implementation of RSA

The approach we take is to trace the execution of the victim program. For that, the spy program applies the FLUSH+RELOAD technique to memory locations within the victim's code segment. This, effectively, places probes within the victim program that are triggered whenever the victim executes the code in the probed memory lines. Tracing the execution allows the spy program to infer the internal state of the victim program.

To implement the trace, the spy program divides time into fixed slots of 2,048 cycles each. In each slot it probes one memory line of the code of each of the square, multiply and modulo reduce calculations. To increase the chance of a probe capturing the access, we selected memory lines that are executed frequently during the calculation. Furthermore, to reduce the effect of speculative execution, we avoided memory lines near the beginning of the respective functions. After probing the memory lines, the spy program flushes the lines from the cache and busy waits to the end of the time slot. We used the default build of the gpg program, which includes optimization at -O2 level and which leaves the debugging symbols in the executable. We use the debugging symbols to facilitate the mapping of source code lines to memory addresses. With a little bit of reverse engineering we found the addresses which are most likely be the frequently visited once (loops).

By recognizing sequences of operations, an attacker can recover the bits of the exponent. Sequences of Square-Reduce-Multiply-Reduce indicate a set bit. Sequences of Square-Reduce which are not followed by Multiply indicate a clear bit. We also probed the actual exponentiation function as to find the boundary between the two consecutive exponentianions (d_p and d_q)

1.5 Implementation

Setup

1. Find the executable of GnuPG.
   (a) Path of executable in our experiments: /usr/local/bin/gpg
2. Find the address mappings of functions used in decryption, using objdump (Found the mapping of a instruction in the middle of function rather than function to avoid false positives due to prefetching)
   (a) We found mappings of functons: $mpih\_sqr\_n$ (squaring), $mphihelp\_divrem$(reduce), mul_n(multiplication)
3. In spy, map this binary in memory, get the base virtual address of this.

Parameters

1. Duration of Slot time for Flush+Reload

2. Threshold of cache hits
   Determined by analyzing histograms of cache hits and cache misses to get the boundary threshold between them (calibration.c)>

1.5.1 Attack

In every slot time. Flush+Reload is performed on the four address and the trace of cache hits on the respective addresses is saved.
This trace however needs to be manually analyzed due to outliers and possibly some misses .

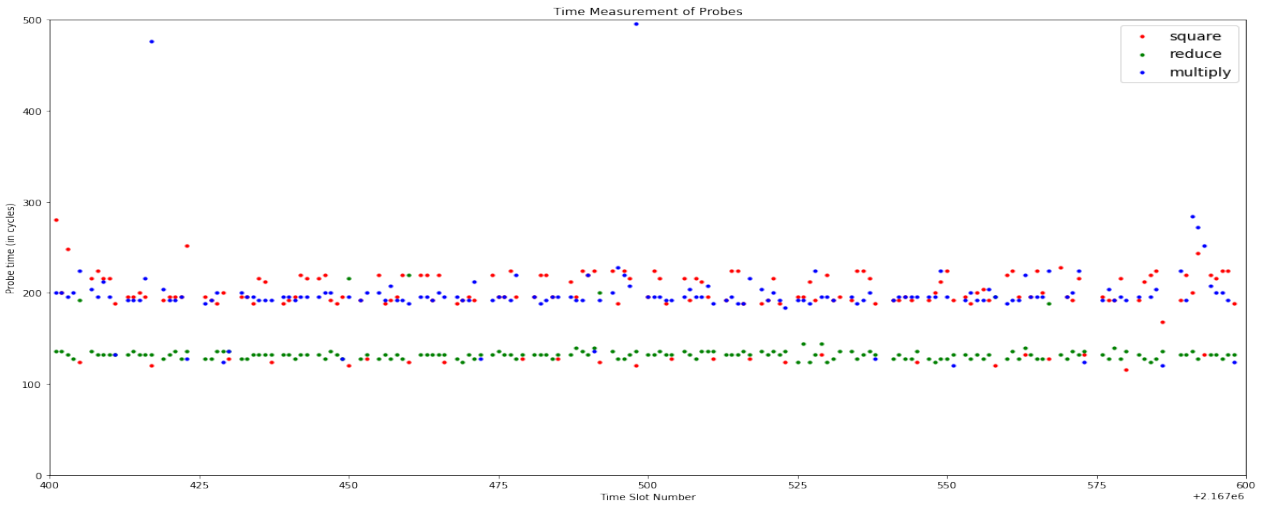


FIGURE 1.2: Plot

FIGURE 1.3: Plot

1.6 Challenges

- Finding address of exact instruction was a challenging task, even by using objdump. We could easly find the address of the function but not the exact instruction from the function. More over the GnuPG code used alot of macros, which confused us a lot.
- Fine tuning the slot size was something which consumed a lot of effort.
- There was no proper ideal threshold. Increasing it injected alot of noise, and decreasing it reduced hit occurrence, which too was happening randomly.
- Writing a script which could automate key prediction seemed impossible with the given noise, but was doable with human observation.

1.7 Conclusions

- In this paper we describe the FLUSH+RELOAD technique and how we use it to extract GnuPG private keys across multiple processor cores.
- The FLUSH+RELOAD technique exploits the lack of restrictions on the use of the clflush instruction. Not restricting the use of the instruction is a security weakness of the Intel implementation of the X86 architecture.

Chapter 2

Cache Template Attack: Automating Attacks on Inclusive Last-Level Cache

2.1 Introduction

Cache Template Attack is a generic attack to exploit cache-based vulnerabilities in any program running on architectures with shared inclusive last-level caches without any prior knowledge of specific system information.
They can be executed online on a remote system without any prior offline computations or measurements.

This attack exploits the following concepts of modern cache architectures and operating systems :

1. Last-Level caches are shared among all CPUs.
2. Cache lines are shared among different processes.
3. Last-Level caches are inclusive, i.e., all data which is cached in within L1 and L2 cache must be cached in L3 cache.
4. The operating system allows programs to map any other program binary or library.

We attack to infer keystrokes and identification of specific keys.

2.2 Cache Template Attack

In this Cache template attack we match generated memory-access templates against the observed memory access to find events.

1. Profiling Phase: We compute Cache Template matrix containing the cache-hit ratio on an address given a specific target event in the binary under attack.
2. Exploitation Phase: We use the generated Cache Template matrix to infer events from cache hits.

In both the phases rely on Flush+Reload. In both the phases attacked binary is mapped into read-only shared memory in the attacker process.
The attacker and the spy program access the same read-only shared physical memory region and as the last level cache is physically indexed, they both access the same cache lines.

2.2.1 Profiling Phase

Measures the cache hits on a specific address during an event. This is done for all addresses for each event and stored in a Matrix known as Cache Template matrix. Column represent event (known as profile) and Row represents address.

1. Step 1: Cache-hit trace and event trace are generated. Binary under attack is executed and the event is triggered constantly. Each address is profiled for a specific duration.
   - Cache-Hit Trace: Value 1 for every timestamp t when cache hit occurs.
   - Event Hit Trace: Value 1 when at timestamps t of event start and end.
2. Step 2 (Main): We compute the cache-hit ratio for each trace and store it in Template. Hit ratio is calculated as ratio of cache hits on address a and the number of times the event is triggered within profiling duration
3. Step 3: Pruning
   - We remove redundant rows from the Template.
   - Merge indistinguishable events based on the rows.

2.2.2 Exploitation Phase

Monitor all address and record cache hits. Use this and Cache Template Matrix to find Event occurred. Detected Events are recorded in log file which is still manually parsed.

ALGORITHM : Profiling Phase

Input: Set of events E, target program binary B, duration d
Output: Cache Template matrix T
Map binary 'B' into memory
for *event e* in E **do**
  for *address a* in binary B **do**
    while *duration d* not *passed* **do**
      Trigger event 'e' and save in event trace $g^E_{a,e}$
      Flush + Reload on address 'a' and save cache hit trace $g^H_{a,e}$
    end while
    Extract cache-hit ratio from trace and store in Template. d.
  end for
end for
Prune the Template

ALGORITHM : Exploitation Phase

Input:Target program binary B, Cache Template matrix T
Map binary 'B' into memory
while True **do**
  for *address a* in binary B **do**
    Flush + Reload on address 'a' and save the 0/1 (miss/hit) in $\bar{h}[a]$
    if $\bar{h}[a]$ is similar to Column i of T **then**
      Event i detected
    end if
  end for
end while

2.2.3 Attack: Key Stroke Logging

Experiment Setup

GDK library handles the default UI in Ubuntu 16.04. We attacked GDK library libgdk-3.so.0.1800.9 . Find the address mapping in /proc/ folder.

Attack: Profiling Phase

Profiling is the most expensive step in the attack. Key events are low frequency events, So we choose duration of d = 0.8 secs. For faster profiling, profile only cache aligned addresses. Problem of Prefetching: Multiple address can be profiled simultaneously but due to prefetching address on same page are loaded beforehand and this leads to false positives. Therefore we profiled addresses on different pages. However in this case of low-frequency event, it is possible to profile all pages within one binary in parallel.

For every address in the mapping, Key Events are triggered and Cache Template Matrix is computed. This matrix is pruned. Events with similar profile are clubbed to one set. Keys in a set are indistinguishable.

Exploitation Phase

A Key event is compared to the profiles in the Cache Template matrix to identify the possible key.

2.2.4 Challenges

1. Since we are attacking GDK, a lot of false positives (noise) arise during profiling due to the user interface. As of now we thought of profiling multiple times to eliminate them
2. The duration for the measurement. Small values will lead to false negatives. Large value means more events are triggered and more accurate computation of cache-hit ratio is done, but it takes too long time. Value to balance this trade off is needed.
3. The profiling phase takes a lot of time, as for one address it takes duration d = 0.8 secs and approx 200 events are triggered. Made it faster by profiling only cache-aligned address and multiple address at a time. Further enhancement required.

Chapter 3

DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks

3.1 Introduction

Due to the popularity of cloud services, multiple tenants sharing the same physical server through different virtual machines (VMs) is now a common situation. In such settings, a major requirement is that no sensitive information is leaked between tenants, therefore proper isolation mechanisms are crucial to the security of these environments. While software isolation is enforced by hypervisors, shared hardware presents risks of information leakage between tenants. Microarchitectural attacks can leak secret information of victim processes, e.g., by clever analysis of data dependent timing differences. Cloud providers can deploy different hardware configurations, however multi-processor systems are becoming ubiquitous due to their numerous advantages. They offer high peak performance for parallelized tasks while enabling sharing of other hardware resources such as the DRAM.

To attack these configurations, successful and practical attacks must comply with following requirements :

1. Work across processors: Attacks that don't work across processors are severely limited and can be trivially mitigated by exclusively assigning processors to tenants or via the scheduler.
2. Work without any shared memory: With memory deduplication disabled, shared memory is not available between VMs.

Though there are many attacks based on processor-integrated cache and cache coherency mechanisms, they assume shared memory or fail to work across processors. **Daniel Gruss [1]** presented two methods to reverse engineer the mapping of memory addresses to DRAM channels,ranks and banks. Using this mapping, he introduced DRAMA attacks that exploit the DRAM row buffer that is shared, even in multi-processor systems without failing our configurations.

3.2 Background Learning

We read briefly about the hardware covert and side channels, organization of the DRAM and Row hits and conflicts.

3.2.1 Hardware covert and side channels

Attacks exploiting hardware sharing can be grouped into two broad categories - Covert and Side Channel. In Covert channel, attacker and sender actively cooperate to exchange information in a setting where they are not allowed to i.e., across isolation boundaries. In Side channel, atacker spies on a victim and extracts sensitive information such as cryptographic keys.

3.2.2 DRAM

The main memory of modern computer systems is typically DRAM (Dynamic random-access memory). It is a type of random access semiconductor memory that stores each bit of data in a separate tiny capacitor within an integrated circuit. As the charge on the capacitory leaks off slowly leading to the loss of data, DRAM uses an external memory refresh circuit which periodically rewrites the data in capacitors, restoring their original charge, hence it is called dynamic memory. Unlike flash memory, DRAM is volatile memory since it loses its data quickly when power is removed. Refer **Appendix A** for more information about DRAM organization.

3.2.3 Row hits and conflicts

Every physical memory location maps to one of out of many rows in one amongst several banks in the DRAM. Consider a single row access i in a bank. There are two possible cases :

1. The row i may already be opened in a row buffer. We call this case a *row hit*.
2. A different row $j \neq i$ in same bank is opened. We call this case a *row conflict*.

So by frequent alternating access to two addresses by flushing the the cache in between, the access times in the case when these two addresses map to two different rows in the same bank is high beacuse of the row confilcts. Based on the timing differences between row hits and row conflicts, different attacks have been proposed.

3.3 Salient points and highlights

The crux of the paper is based on the following three important aspects.

3.3.1 Reverse Engineering DRAM addressing

Two approaches have been dicussed, the first one is based on physical probing, whereas the second one is entirely software-based and fully automated. The solving approach discussed in the paper *assume* that the addressing functions are *linear*, i.e., they are XORs of physical-address bits which might not be true in case of triple-channel configurations.

Physical Probing

In this approach they used a standard passive probe to establish contact with the pin at the DIMM slot and then repeatedly accessed a selected physical address to measure the voltage and subsequently deduce the logic value of the contacted pin. After repeating this experiment with all pins of interest and various physical addresses, we start the solving phase. In this phase for each DRAM addresing function we create an over-defined system of linear equations in physical address bits and then test them on the physical addresses

used and compare the logic values of the output address and the pins of our interest. The solution is the corresponding DRAM Addressing function.

This approach has some drawbacks - expensive measurement equipment and physical access to the internals of the tested machine. But the advantage of this approach is that the address mappings can be reconstructed for each control signal individually. Thus, we can determine the exact individual functions for the bus pins.

Software-based and Fully Automated

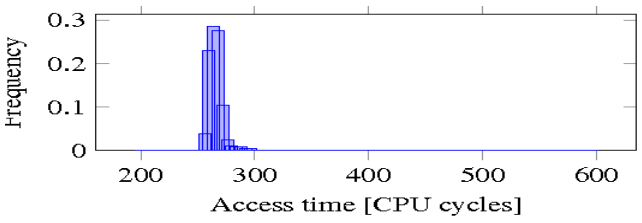
In this approach we exploit the fact that row conflicts result in high memory access time. In first step we find sets of address that belong to the same DIMM,rank and bank by checking peaks in access times when alternately accessing two addresses from a pool of addresses. In the second phase, we use the identified address sets to reconstruct the addressing functions. This reconstruction requires resolution of the tested virtual addresses to physical ones which can be done through pages allocated for large arrays. We then generate all linear functions that use exactly n bits as coefficients and then apply them to all addresses in one randomly selected set. We start with n = 1 and increment n subsequently to find all functions. Only if the function has the same result for all addresses in a set, we test this potential function on all other sets. We verify the correctness either by comparing it to the results from the physical probing, or by performing a software-based test, i.e., verifying the timing differences on a larger set of addresses.

This approach has some advantages - does not require any additional measurement equipment and can be executed on a remote system. But this approach cannot recover the exact labels of the functions, though we don't require this in any of subsequent attacks.

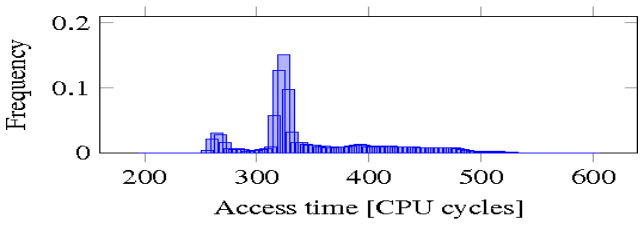
3.3.2 A high-speed cross-CPU covert channel

This attack does not require shared memory and the only pre-requisite is that the communicating processes have access to the same memory module. In this covert channel attack, the sender and receiver occupy different rows in the same bank. If the sender continously accesses a row it is equivalent to sending a logical bit value 1 and if he is doing nothing then its a logical bit value 0. The receiver continously accesses a chosen physical address in the same bank but a different row and measures the average access time. If the access time is large it means that there is row conflict, so the sender must have sent a logical 1 else it must be 0, thus infering the bit values from sender. The speed of this channel is attributed the fact that each combination of (CPU, channel, DIMM, rank, bank) is a unidirectional transmission channel. So we can use a maximum of 8 banks in parallel. However higher number of parallel channels result in increased noise, so to attain optimal performance we use only a subset of tuples.

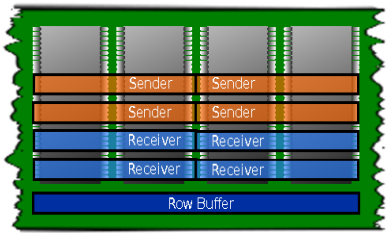
TABLE 3.1: Timing differences between active and nonactive sender (on one bank)



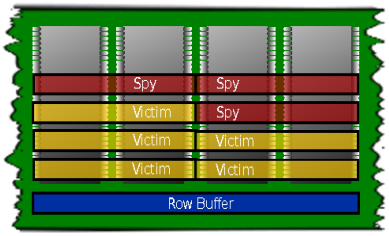
(a) Sender inactive on bank: sending a 0.



(b) Sender active on bank: sending a 1.



(A) The sender occupies rows in a bank to trigger row conflicts. The receiver occupies rows in the same bank to observe these row conflicts



(B) Victim and spy have memory allocated in the same DRAM row

FIGURE 3.1: DRAM Array in the case of Covert channel and Side channel

Implementation

We have an Agreement phase in which the sender and receiver must agree on tuples to be used for communication. Each must find at least one address in their respective address space that maps to each tuple using the DRAM addressing function. Alternatively this process can be done without this function, though this may require an Initialization phase where each party performs timing analysis to determine same-bank addresses. Subsequently, the communicating parties need to synchronize their sets. So a data block is transmitted for a fixed time span which is agreed upon before starting communication. To synchronize the start of these blocks we employ various methods. If sender and receiver run natively, wall clock acts as the means of synchronization. If the sender and receiver run in two different VMs, then a common (or perfectly synchronized) wall clock is typically not available. In this case, the sender uses one of the transmission channels to transmit a clock signal which toggles at the beginning of each block. The receiver then recovers this clock and can thus synchronize with the sender.

3.3.3 A low-noise cross-CPU side channel

In this Side channel attack, the victim and the spy need to have access to memory in the same row in the same bank.This is possible without shared memory due to the DRAM addressing functions. Depending on the addressing functions, a single 4 KB page can map to multiple DRAM rows. Since a general DRAM row is of size 8KB, it contains memory divided among 16 pages.

Side-Channel Attack

To run the side-channel attack on a private memory address t in a victim process, the attacker allocates a memory address p that maps to the same bank and the same row as the target address t. Allthough t and p map to the same DRAM row, they belong to different 4 KB pages (i.e., no shared memory). The attacker also allocates a row conflict address p' that maps to the same bank but to a different row. The side-channel attack then works in four steps:

1. Access the row conflict address p'
2. Wait for the victim to compute
3. Measure the access time on the targeted address p
4. If access time is small, there there is a row hit so victim must have accessed the row.

As p and p' are on separate private 4 KB pages, they will not be prefetched and we can measure row hits without any false positives.

Fully-automated template attack

Now we will automate the above side channel attack. For this attack we do not need to reconstruct the full addressing functions nor determine the exact bank address.To perform a DRAMA template attack, the attacker allocates a large fraction of memory, ideally in 4 KB pages. This ensures that some of the allocated pages are placed in a row together with pages used by the victim. The attacker then profiles the entire allocated memory and records the row-hit ratio for each address. False positive detections are eliminated by running the profiling phase with different events. If an address has a high row-hit ratio for a single event, it can be used to monitor that event in the exploitation phase. After such an address has been found, all other remaining memory pages will be released and the exploitation phase is started.

Improving Attack: Flush + Reload

In calibration of cache-hit threshold, the general method used is using histograms of Cache Hits and Cache misses and concluding the threshold time. If a process accesses any memory location in the same row, a row hit will be misinterpreted as a cache hit.

This introduces a significant amount of noise especially for Flush+Reload attacks on low-frequency events (Like the one done in Cache Template Attack for key logging in Chapter 2) as the spatial accuracy of a cache hit is 64 bytes and the one of a row hit can be as low as 8 KB, depending on how actively the corresponding pages of the row are used. So taking the row hits and conflicts into consideration will improve the performance.

Conclusions

- In this paper, we read about DRAM organization and two methods to reverse engineer the mapping of physical memory addresses to DRAM channels, ranks, and banks. One uses physical probing of the memory bus, the other runs entirely in software and is fully automated.
- We also read about two different DRAMA (DRAM Addressing) Attacks, one based on the covert channel and the other based on the side channel.
- We also read how DRAM can be exploited to improve existing attacks like Flush + Reload from chapter 1.
- We also compared the DRAM template attacks with cache based template attacks from chapter 2.

Appendix A

DRAM Organisation

The main memory of modern computer systems is typically DRAM. DRAM has a significantly higher latency than the various caches inside the processors. The reason for the latency is not only the low clock frequency of DRAM cells but also how DRAM is organized and how it is connected to the processor. As it is difficult to reduce the latency, hardware manufacturers instead focused on increasing the bandwidth of DRAM. The high bandwidth can be utilized to hide the latency, e.g., through speculative prefetching. Modern processors have an on-chip memory controller which communicates through the memory bus with the DRAM as shown in figure below.

The memory subsystem hierarchy is in the order of channels, DIMM, Rank, Chip, Bank and Row/Column. Chips consist of multiple banks and banks share command/address/data buses. Multiple chips are put together to form a wide interface called Dual Inline Memory Module (DIMM). All chips in one side of a DIMM are operated the same way, so each side of the DIMM is called a rank. They all respond to a single command, share addresses and command busses but provide different data.

DRAM bank is a 2D array which consists of DRAM rows and columns (typically 1024). Modern systems organize the memory in a way that a DRAM row typically has a row size of 8 KB. A row can either be opened or closed. If it is currently opened, the entire row is preserved in the row buffer. To fetch a data value from DRAM (refer fig A.2) the processor

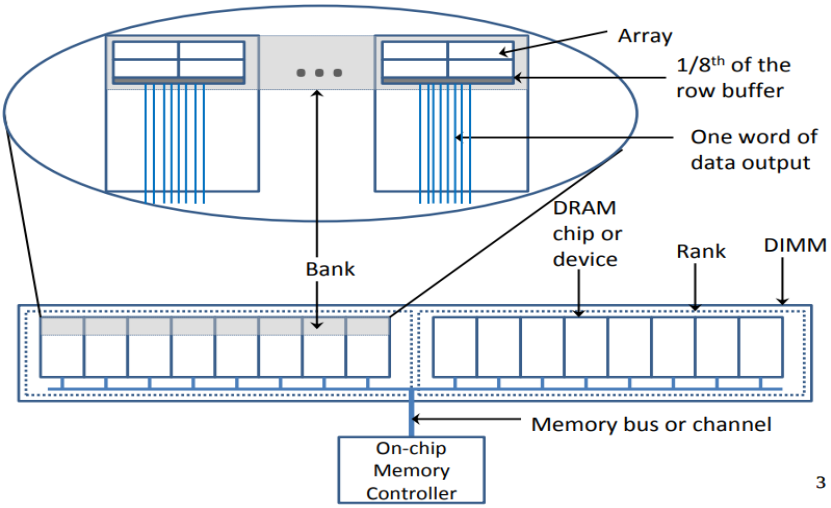


FIGURE A.1: DRAM Organisation

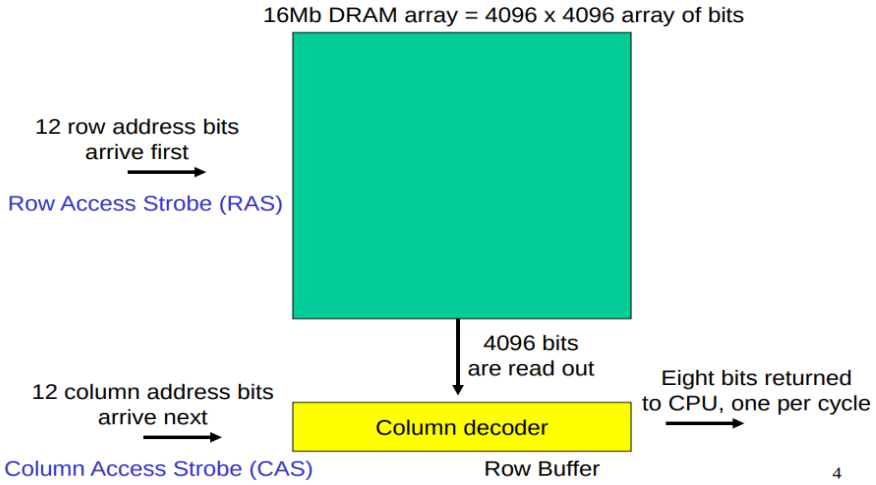


FIGURE A.2: DRAM Array Access

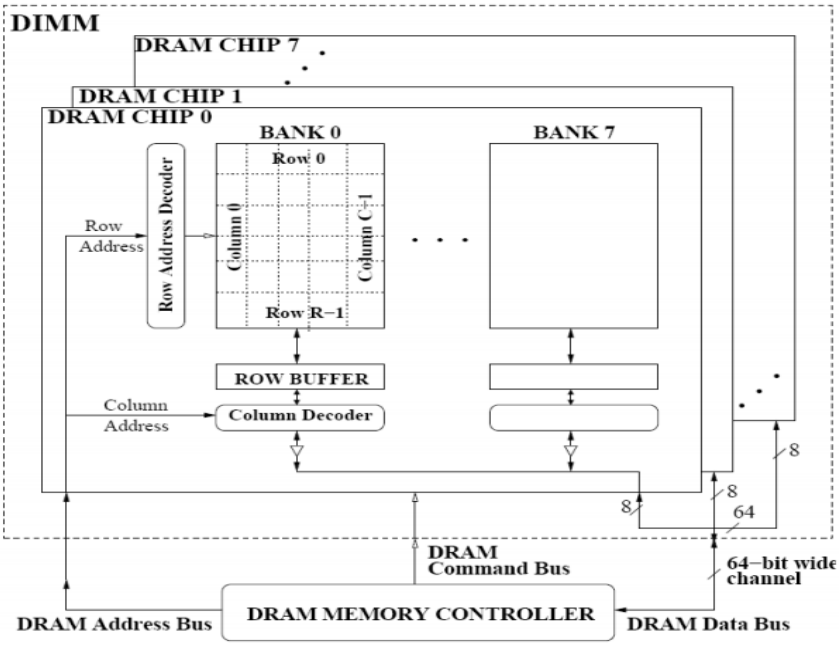


FIGURE A.3: 64-bit Wide DIMM

sends a request to the integrated memory controller. The memory controller then determines a sequence of commands to send over the memory bus to the DRAM to retrieve the data. If the currently opened row contains the data to retrieve, the memory controller just fetches the data from the DRAM row buffer. As this situation is very similar to a cache hit, we call it a row hit. If the currently opened row does not contain the data to retrieve, we call it a row conflict. The memory controller then first closes the current row, i.e., writes back the entire row to the actual DRAM cells. It subsequently activates the row with the data to retrieve, which is then loaded into the row buffer. Then the memory controller fetches the data value from the row buffer. Similar to cache misses, row conflicts incur an increased access latency. The overall organisation od DRAM is depicted for a 64 bit-wide channel in Figure A.3.

A.1 Contributions

Team used to discuss all the papers each read and the problems in implementation. We had discussions, so everyone had read the papers simultaneously. Everyone contributed equally.

Bibliography

[1] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[2] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016, pp. 565–581. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[3] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.