# Breaking VM Isolation – An In-Depth Look into the Cross VM Flush Reload Cache Timing Attack

**Danny Philippe-Jankovic† and Tanveer A Zia††**

School of Computing and Mathematics Charles Sturt University, NSW, Australia

**Summary**
Cloud computing has led to a lot of improvements in the way we manage our IT infrastructure, but this convenience has led to new security challenges. In this paper, we recreate a cross Virtual Machine Flush+Reload cache timing attack and document our attack methodology in depth. Cache timing attacks are highly technical, and executing them in a virtualized environment makes them more complex. We have not come across any literature that documents these attacks adequately, and so this paper aims to deliver detailed insight into the entire lifecycle of these types of attacks. Our attack methodology dissects the life cycle of a side channel attack in a virtualized environment from beginning to end. We present an in-depth analysis of the environment, the attack setup, the attack execution, and how these attacks can be used to gather and analyse results. This research will shed some valuable insight into what is a very technical and complex topic. By documenting our attack environment and methodology we hope to help new researchers in this field gain a foothold in a research topic that has recently gained popularity and may be difficult to enter. Finally, we examine how meaningful results are collected and analysed by the attacker. We believe this insight will also be valuable for cloud administrators and developers, and that they may use their understanding of the attack lifecycle and result analysis tools to mitigate and disrupt attack attempts.

***Key words:***
*Hypervisor, Virtualization, Isolation, Flush+Reload, Cache Timing Attack, Cross VM.*

## 1. Introduction

Cloud computing has quickly become a massive part of the infrastructure of the internet and of corporate systems. With the rise in popularity of private and public clouds security researchers have recently defined a new field of research, cloud security. Cloud security covers any security topic relevant to virtualization technologies and their applications. We have chosen to focus our research in this field on the topic of side channel attacks within virtualized environments. This paper presents the methodology used to execute a cache timing attack in a virtualized environment, as well as an in-depth analysis and recreation of a cross Virtual Machine Flush+Reload cache timing attack.

Our goals in this paper are; to document how we set up our virtualization lab, produce a step by step guide on how we were able to recreate the attack, and to discuss how the results of attacks like these can be collected and analysed. To this end, we have developed our attack methodology in depth which discusses each phase of the attack, from setting up the environment and coding the attack to collecting and analysing the results. This information can be used as an introduction into the field, helping new researchers establish a virtualization lab and the essential software and code required to execute these attacks. This paper also offers insight into the life cycle of these highly technical attacks, which will allow security experts and cloud administrators to better understand and defend against them. There are two main chapters in this paper; Attack Methodology, and Result Analysis.

Our attack methodology explains step by step the details of the lifecycle of a cross Virtual Machine Flush+Reload attack. This includes the specifics of our attack environment, the hardware, the operating systems, the software, the attack code, and the attack execution. We document the details of each of these topics, as well as the theory behind the attack and the specifics of the assembly code that allow the execution of this attack. To deploy our attack, we aimed to create a virtualization environment that was typical of private and public clouds. To do this we implement industry standard software for our operating systems and hypervisor. Our environment was limited by low funds, so we were unable to run our experiments on a wide range of hardware. We were, however, able to utilize open source software to set up our operating systems and software environment. We achieved this with the use of Linux based operating systems that have built in virtualization technology, which has become one of the standard virtualization vendors in the cloud industry.

In chapter 3 we discuss in detail how attackers collect results from these attacks and how these results can be analysed and interpreted by the attacker. To collect accurate results there are a few specific pieces of information that the attacker must acquire about the system which we cover in the chapter. There are also several factors that may contribute to noise within the result data which are also briefly discussed. Once the results are collected they must be analysed in order to extract useful information. We identify two methods used for analysing this data; Result Set Analysis, and Real-time

Manuscript received February 5, 2017
Manuscript revised February 20, 2017

Monitoring. We discuss both methods and their uses before ending the paper with a conclusion.

This paper contains 4 chapters; chapter 1 is the introduction, chapter 2 is our attack methodology, chapter 3 is our results and analysis discussion, and chapter 4 concludes the paper. The introduction introduced our research goals, the scope of our research, and limitations. Our attack methodology outlines the attack environment as well as the entire life cycle of the recreation attack, discussing in detail the attack conditions and each step of the cross VM Flush+Reload cache timing attack. Chapter 3 discusses the specifics of gathering and analysing results from cache timing attacks. Finally, chapter 4 wraps up with a conclusion and a brief discussion of our intended future research.

## 2. Attack Methodology

### 2.1 Theoretical Basis

The attack that we have chosen to replicate targets information leakage in the Last Level Cache (LLC or Level 3 Cache) on Intel CPUs and relies on a few features that are commonly enabled in clouds. We are able to leverage these conditions to execute a high-resolution cache timing attack and monitor code running on third party VMs. Research has shown that these attacks are able to even break encryption algorithms [3,38]. The specific attack that we will be replicating is a cross VM Flush+Reload attack that is able to identify if a specific Operating System (OS) or program is running on a co-located target VM. This section consists of 3 subsections that will explain the theory behind the attack. These sections are; Attack Anatomy, CPU Architecture, and Shared Memory.

#### 2.1.1 Attack Anatomy

A Flush+Reload attack is a type of Micro Architectural attack. These attacks target weaknesses in the architecture of low level computer components such as CPU, RAM, Data Storage devices, etc. The Flush+Reload attack is a cache timing attack that monitors the difference in the access time of a targeted address in memory. We can use the information we gain to determine if and when these addresses are accessed by another process. This can lead to further insights into the state of a targeted system. The Flush+Reload attack has 3 distinct phases; the Setup phase, the Attack phase, and the Analysis Phase.

During the setup phase the attacker will create the attack environment and configure the attack software. In a live attack this will require the attacker to achieve co-location with the victim VM, test background noise of the side channel, and gather information on the expected access times for both Main Memory (MM) and the LLC. For our attack recreation, we deployed our own hypervisor with our own controlled VMs, ensuring co-location and making background noise negligible. We cover exactly how we set up the attack environment in section 2.2.1 and we will discuss how we gather information about access times in section 2.2.3.

The Attack phase is where the attack is deployed, the target is monitored, and data is gathered for the Analysis phase. The Flush+Reload attack has 3 steps; Flush, Wait, and Reload. In the flush step the cache is flushed of the targeted address', ensuring that the next time it is accessed it will need to be loaded from MM. After this, the attack program will wait for a small amount of time so that the victim has an opportunity to access the address. Finally, the address is reloaded and the access time is recorded. If the target had accessed the address within the wait time, then the load time we record will be approximately equal to the LLC access time. If the target did not access the address then we expect to see a longer access time, one approximate to the MM access time. These attack times can either be interpreted immediately, compared to a threshold to trigger some event, or they can be recorded for later analysis.

During the Analysis phase the recorded access times are analysed and interpreted. The way this data is interpreted depends a lot on the behaviour that is being monitored. Some attacks rely on statistical analysis [32,33,38,39] while others can represent the data visually and be interpreted by an expert to gather the desired information [36]. The data gathering process is not perfect and some error correction calculations may be needed in order to complete the attack. Yarom et al and other researchers have been able to show how cache timing attacks can be used to break encryption. They note that even if the data recovered is incomplete what is recovered can be used to drastically reduce the number of guesses needed to crack the encryption [36,38,39].

There are cases where the results we gather from an attack session can contain errors, and depending on the context of the attack these errors may be recoverable. If the attack targets an encryption algorithm there may be ways to retrieve the lost bits of the secret key [36,38,39]. However, if the attack aims to log key presses or other user input there is no way to retrieve the lost data [32,40,41]. There are three main cases that lead to errors in results; The targeted address is loaded into cache by a third-party process, the target address is flushed from LLC before we can probe it, or our attack process is suspended causing us to be inactive for some time. The first case may occur if the targeted code is a specifically popular one and multiple tenants are competing for it, such as network drivers. The

second case can occur when there are periods of high traffic in the cache and the target address is evicted to make room for a running process's data. The third case can also occur when there are high levels of traffic in the cache. If our process has lower priority than other processes running in the stack, then it may be suspended temporarily to allow these other processes access to the CPU. This can also occur if our VM has over used its allotted CPU resources and other VMs have higher priority for that resource, in this case our entire VM may be suspended. In both cases, we should be able to detect this loss of time by recording a very high delay between two attacks, but we will not be able to directly recover the missing data.

### 2.1.2 CPU Architecture

Intel CPUs have two features that enable the Flush+Reload attack; An inclusive cache, and the 'rdtsc' instruction. An inclusive cache is not necessary for all cache timing attacks, but it does enable the majority of these attacks. It allows a program to flush data from the entire cache, including all the L1 and L2 caches within the CPU. This lets attackers circumvent isolation and interact with process or VMs via a shared memory address. The 'rdtsc' instruction is a high resolution timing instruction that allows programs to count the amount of cycles it takes to execute a set of instructions. This functionality is necessary for programs that require fine grained synchronization for tasks such as human input processing, audio processing, or video processing. However, it also allows malicious users to execute high resolution side channel attacks. These two CPU features leave systems vulnerable to most cache timing attacks and are the focus of a lot of cache timing attack research [32,33,35,36,37,38,39,40,41,42].

Each core in a CPU has its own L1 and L2 cache that only it has access too. Additionally, each thread running on a system will be assigned to a single core. Unless hyper-threading is enabled, each thread will run exclusively on a core and must be put into suspension if another thread is scheduled to run. Hyper-threading has some known security issues so has become best practice for it to be disabled [29]. This means that on most systems for most process to have access to another process's L1 and L2 cache they both must be assigned to the same core and one must be suspended immediately before resuming the other. For any useful information to survive this procedure, the swap time and the OS thread management code must be kept to a minimum or else useful data may be evicted. The attacker also has no control over the scheduling of these swaps, making it difficult to ensure that any data observed after a swap was from a specific process. This makes cache timing attacks in the L1 and L2 cache virtually impossible.

In contrast, every core in a CPU has access to the LLC which allows an attacker to flush any shared data from this region. In an exclusive cache, flushing data from the LLC will not evict it from the L1 and L2 cache. If the shared data is still loaded in L1 or L2 cache, a target process may continue to use it without an attacker being able to monitor this behaviour. An inclusive cache ensures that the LLC holds a superset of all memory found in the higher levels of cache. This means that when data is evicted from the LLC, the L1 and L2 caches are checked for the same data and if it is found it will be evicted. These caches allow malicious processes to evict a shared memory address from the LLC, letting them monitor when these addresses are accessed by other processors and enabling cache timing attacks. Modern Intel processors utilize an inclusive cache, and it is these processors that the Flush+Reload attack targets primarily [36]. Fig 3 shows an illustration of the Ivy Bridge cache architecture which is an example of an inclusive Intel cache architecture.

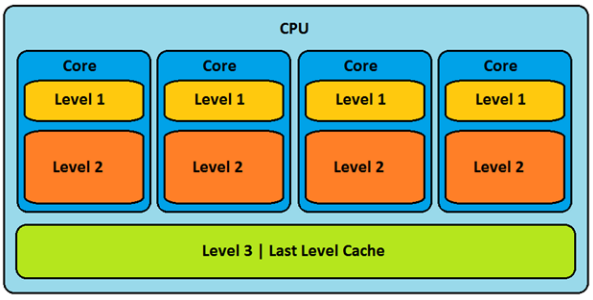


Fig. 3 Cache architecture, isolation and access times

The rdtsc instruction is a timing function that returns the number of clock cycles that have passed since the processor has been reset. As illustrated in Fig 4, an attacker will invoke this instruction before and after loading a targeted memory address and calculate the difference. This becomes a record of the time it took to access the memory address. The resolution of these recorded times allows an attacker to differentiate between data accessed from L1 cache, the LLC, or MM. If the target address is found to be loaded into cache, then an attacker can assume the data is in use by the target. If the loading time is close to the typical MM access time, then that target is idle and not in use. One record alone will hold very little information about a target system's state, but by monitoring important targets over a period of time the result sets gathered can yield very valuable information about a target. Exactly how these result sets are recorded and interpreted will be covered in chapter 3.

```
1       lfence
2       rdtsc
3       mov %eax, %edi
4       mov (%r8), %r8
5       mov (%r8), %r8
6       mov (%r8), %r8
7       mov (%r8), %r8
8       mov (%r8), %r8
9       mov (%r8), %r8
0       mov (%r8), %r8
1       mov (%r8), %r8
2       mov (%r8), %r8
3       mov (%r8), %r8
4       mov (%r8), %r8
5       mov (%r8), %r8
6       lfence
7       rdtsc
8       sub %edi, %eax
```

Fig. 4 x86 code used in the cache timing attack

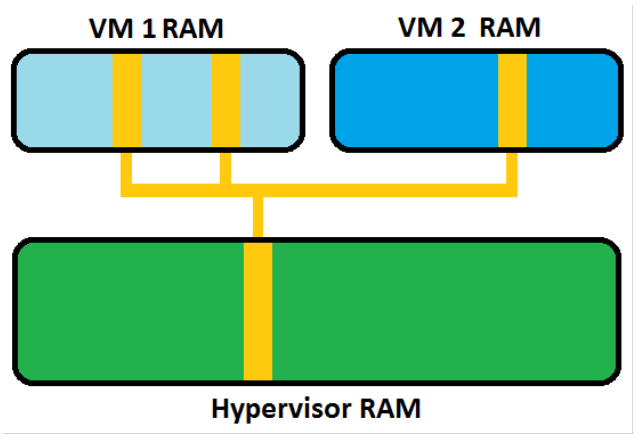


Fig. 5 A Hypervisor consolidating 3 duplicate pages into one

### 2.1.3 Shared Memory

Programmers share and reuse code constantly. Code reuse saves time and reduces programming errors as rewriting functions can be time consuming and may lead to human error. This has led to OS features that help facilitate the reuse of code such as Dynamic Link Libraries. Loading a separate copy of a shared library into memory every time a program is executed is inefficient, so to reduce the amount of redundant memory OSs allow processes to share executable code. This is achieved through the distinction of memory as either 'Read Only' or 'Write'. When a process loads data into MM it can either be loaded as "Read Only" or "Write". Whenever executable code is loaded into MM it must be loaded as "Read Only", this is to help mitigate runtime errors and security vulnerabilities in code. Whenever a process tries to run a piece of code that is not "Read Only" an error is thrown and the process is usually terminated. When a process attempts to load "Read Only" memory into MM, the OS will first check to see if an identical copy of it already exists. If it does, the process is given a pointer to the existing copy of the data instead of creating a separate instance. This is what allows processes to share code or run multiple instances of the same program with a reduced memory footprint. An attacker running on the same OS as its target can exploit this weakness and obtain a pointer to the target's code by attempting to load a copy of its code into MM as 'Read Only'. From there they can launch a cache timing attack on the code and monitor the target's state.

This is enough to launch the attack within a non-virtualized environment, but our attack will take place between two VMs. To overcome the hypervisors memory isolation in our recreation attack we take advantage of page-deduplication. Memory isolation is enforced in hypervisors by segregating the regions of memory that each VM has access too. Each system call that a VM makes must first pass through the hypervisors kernel. Here, the hypervisor can control and monitor which regions of memory each VM has access to. This prevents VMs from accessing a memory region that does not belong to it, making our attack near impossible. However, most hypervisors also have a feature called page-deduplication which works similarly to the "Read Only" deduplication described above. A kernel module will periodically scan running VMs for identical pages of memory. When one is found one of the copies will be unallocated and reclaimed by the hypervisor. The kernel will then point both VMs to the one remaining copy in MM. Fig 5 illustrated this page deduplication mechanism. This feature can save large amounts of memory when multiple similar VMs are running. Unfortunately, this features also gives attackers a shared memory address which can lead to isolation breaches and vulnerabilities such as side channel attacks. The security vulnerabilities associated with page-deduplication are known to the cloud security community who encourage disabling this feature [34]. However, this feature is still enabled by default on QEMU which is the hypervisor that we have chosen to launch our attack on. It is best practice to disable this feature on a cloud to help mitigate attacks such as those discussed here, but cloud system admins may either be unaware or undaunted by these attacks.

## 2.2 Conceptual Basis

### 2.2.1 Attack Environment

We set up our attack on a Linux based hypervisor with two Linux based VMs, one attacker and one victim. There are no Linux specific requirements for this attack, it is just the easiest environment to set up due to its simplicity, its accessibility, and its built-in virtualization features. The KVM Kernel module has been included in the Linux mainline since 2.6.20 making virtualization easily implementable and widespread. The KSM Kernel module has been included in the Linux mainline since 2.6.32 and is enabled by default on KVM hypervisors. This environment can be set up on a wide range of hardware quickly and for free and so it has become common in private and public clouds. Our hypervisor was initialized with default settings making it a typical cloud environment that may be run by a security naïve systems administrator.

For our hypervisor OS, we chose to run Ubuntu LTS v16.04.1, currently the most up-to-date Linux distro released, with full updates. We also ran QEMU-KVM v2.6.2, which is currently the most up-to-date version of this hypervisor. Both the OS and the hypervisor software were configured with default settings and no additional security software was installed. This demonstrates that the current software versions are vulnerable to the Flush+Reload attack we have implemented and that if a cloud environment is set up incorrectly and without the proper expertise it may leave users vulnerable to data theft and further security vulnerabilities. The KSM kernel module was active on this hypervisor by default which, as we covered in section 2.1.3, helps facilitate a Flush+Reload attack. By default, KSM is active on QEMU-KVM which allows the hypervisor to save RAM and deploy additional VMs where it otherwise would not have the necessary RAM. Without page deduplication active, it would be very difficult for an attacker to find a shared physical address between two VMs to exploit, but as RAM is the scarcest resource in the cloud utilizing page deduplication technologies is highly incentivized.

Our attacker VM and our victim VM were also both set up with Ubuntu LTS v16.04.1 with full updates. Again, this is because it is the simplest environment to set up and it demonstrates that current software with default settings is vulnerable to the Flush+Reload attack. The only additional software installed on these systems was the programs and tools necessary to carry out the attack. For our victim VM, we installed and ran our ‘Hello’ program which we describe in section 2.2.2. This program does not assist the attacker in any way, it simply helps us illustrate the attack methodology and the execution of the attack. The attacker VM was set up with the attack program, some analysis tools, and a copy of the ‘Hello’ program. A copy of the targeted program is necessary for the attack as the attack program needs to load an exact copy of it into MM in order to generate the shared memory pointer. The analysis tools helped set up the attack and analyses the results we collected while the attack program itself launched the attack and collected the results. In section 2.2.3 we discuss our attack program in depth and in chapter 3 we discuss the results our attack generated and how they can be analysed.

### 2.2.2 Finding Target Addresses

Cache timing attacks target specific addresses within a program, and these addresses correspond to lines of code that are of interest. It is important to pick the addresses we monitor with care as factors such as proximity and temporal locality of code may affect our results. For example, if the victim program runs a function with a loop in it and the attacker targets an address within or close to the loop they will receive constant feedback from the timing attack for the entire duration of that loop. This can be useful it the attacker wants to monitor the duration of the loop, but if instead they only want feedback once each time the function is run they would need to target an address towards the beginning or end of the function that is only executed once each time the function is run. It is also important to pick an address then isn’t too close to a piece of code in another unrelated function. This may lead to a false positive result if the other function is called instead of the monitored function. This is due to the way the cache loads memory from MM. Instead of loading the specific bytes needed, CPU architecture takes advantage of proximity and temporal locality of code and loads entire pages into the LLC at a time. From there more specific lines of those pages may be loaded into L1 and L2 cache.

There are many ways an attacker could reverse engineer target programs and find the address corresponding to the specific line in the code that they wish to monitor. Some examples include attaching a debugger to the targeted program while it is running and following its execution path, opening the binary of the executable in a disassembler and searching for the functionality, or creating a large list of random addresses throughout the program to watch and monitoring them during runtime. Each method requires a set of skills and experience with specific tools as well as an insight into how the targeted code will behave during runtime. Gruss et al [41] were able to develop a method of automatically locating interesting addresses through a series of refined guesses. Their method started with a set of addresses located throughout the entire targeted program. They then watched each of these addresses while they triggered the functionality they wished to monitor. Any addresses that were not triggered were removed from the test set and their method would continue until their set contained a list

of addresses that were indicative of the functionality they wished to monitor. To find our target addresses we used a more traditional method of running our target program within a debugger. The rest of this section will document the method we used to find our target addresses.

Our first attack was launched on a very simple command line program that we wrote. Our program was called 'Hello' and was comprised of four functions and a main loop. The main loop of 'Hello' will wait for the user to input a character and will test whether that character is either 'H', 'E', 'L', or 'O'. If it is one of these characters, the program will run one of the 4 separate functions that simply output the same character that the user input. By monitoring each of these functions with our attack code we will be able to tell when our victim enters one of these characters and what character they have entered. This is a simple proof of concept attack that allows us to test that our environment is correctly configured for the attack as well as illustrate our attack in a way that is simple to extrapolate to a more useful, every day example. It is easy to see how this method could be applied to any command line interface that branches off to multiple functions based on specific user input.

We wrote and attacked our own code to help illustrate our method. We compiled our code using the GNU Compiler Collection (GCC) and including the '-g' tag. This preserves the debugging information that helped us find the target addresses. Typically, this debugging information is stripped from the binary when a program is constructed or when a program is run as it only assists humans during debugging. Its presence does not change the program binary so any addresses we find with a debugger that we are interested in monitoring will correspond directly to the program running in a live environment. This information is not necessary for finding target addresses but it does make the process easier. It is possible to debug programs without their debug information and find target addresses but it is a lot more difficult and requires a higher level of expertise. It is very easy for an attacker to obtain this information for any open source program, all they need to do is download the source code and compile it locally with debugging information included. Therefore, this method does not impact the efficacy of this attack in a real scenario.

We use this debug information in conjunction with the GNU Project Debugger (GDB). GDB is a debugging tool that allows us to run our program in a sandbox and monitor its execution. We can follow its execution path, watch variables, set breakpoints, and more. These tools help programmers debug their programs so they can fix problems in their code more efficiently. We are using GDB simply to find where the lines of code we are interested in monitoring will load into memory at runtime. The first step is to find the lines in code that we want to monitor with our cache timing attack. In our code, we find the beginning of our four print functions at lines 4, 123, 235, and 369. The reason these lines are so far apart is because our functions are filled with dummy code to avoid false positives created when the cache loads pages of our program into memory. We now load our 'Hello' program in GDB and look up those code lines. This gives us the information we need for our attack; an example of the information we are looking for can be found in Fig 6. The addresses corresponding to our functions are 0x4006AE, 0x400A9C, 0x400E48, and 0x4012B4 respectively.

```
(gdb)
(gdb) info line 4
Line 4 of "hello.c" starts at address 0x4006a6 <printH>
   and ends at 0x4006ae <printH+8>.
(gdb) info line 123
Line 123 of "hello.c" starts at address 0x400a94 <printE>
   and ends at 0x400a9c <printE+8>.
(gdb) info line 235
Line 235 of "hello.c" starts at address 0x400e40 <printL>
   and ends at 0x400e48 <printL+8>.
(gdb) info line 369
Line 369 of "hello.c" starts at address 0x4012ac <printO>
   and ends at 0x4012b4 <printO+8>.
(gdb)
```

Fig. 6 GDB output showing the addresses of the lines we are targeting with our attack.

OSs require metadata to be stored on every running process in order to ensure the stability of the system and to properly allocate resources. In a Linux OS the first 0x400000 of every process holds this data and the base address of the program data stars at this address, meaning from 0x400000 onwards the binary is loaded into memory. This means that when we retrieve our addresses for the target lines of code they will always be above this 0x400000 region of memory. To find the relative address that we should use in our attack we simply subtract this base value to find the relative address. This makes our new target addresses 0x6AE, 0xA9C, 0xE48, and 0x12B4 respectively.

### 2.2.3 Mastik Framework

Our attack code is based on the Mastik (Micro-Architectural Side-Channel Toolkit) framework which is currently being developed by Yuval Yarom et al of The University of Adelaide. According to the developers, "Mastik is a toolkit that aims to provide robust implementations of side-channel attack techniques" [42]. Although Mastik is not developed with the intent of deploying its attacks in a virtualized environment, our research shows that it is entirely possible to deploy Mastik's Flush+Reload attack between two VMs without the need for extra techniques or code. This may also be true for other attacks that Mastik facilitates such as the Prime+Probe attack on the LLC or other attacks that may be introduced to Mastik in the future. Our future research will aim to explore the implementation of more Mastik

attacks within virtualized environments. As of version 0.02, Mastik contains the implementation of the following six side channel attacks with more planned in the future:

- Prime+Probe on the L1 data cache
- Prime+Probe on the L1 instruction cache
- Prime+Probe on the Last-Level Cache
- Flush+Reload
- Flush+Flush (new in 0.02)
- Performance degradation attack (new in 0.02).

To implement our attack, we used Mastic's Flush+Reload (FR) library. This library allows us to create an object that will load the target code into memory, store the target addresses we want to probe, and run the Flush+Reload attack. The functions we will use to achieve this are 'map_offset()' to load the targeted code into memory, 'fr_monitor()' to add a targeted address to the list of monitored addresses, 'and fr_probe()' which will run the attack and return the approximate number of cycles it took to access the targeted addresses. The rest of this chapter is dedicated to the dissection of these elements and the explanation of their implementation in our Flush+Reload attack recreation code.

**map_offset()**

The 'map_offset()' function loads the code we want to target into MM as a "Read Only" file. The function acts as a wrapper for the 'mmap()' function and takes 2 arguments; a file descriptor and an offset. The file descriptor simply points the function towards the file we want to load into memory. The offset must be a multiple of the page size, usually 4096 bytes, and indicates the page at which the program should start loading data into memory from the file. This allows us to copy only specific sections of a file into memory and lower our memory footprint, but this value must be taken into account when determining the target addresses we wish to monitor. The function then uses 'mmap()' to load the file into memory with the PROT_READ flag enabled. This indicates that the data obtained from the file is "Read Only" and cannot be written to, enabling page-deduplication which is a necessary condition for the Flush+Reload attack. The Mastik implementation of mmap() can be seen in Fig 7.

```
void *map_offset(const char *file, size_t offset) {
  int fd = open(file, O_RDONLY);
  if (fd < 0)
    return NULL;

  char *mapaddress = mmap(0, sysconf(_SC_PAGE_SIZE), PROT_READ, MAP_PRIVATE, fd, offset & ~(sysconf
(_SC_PAGE_SIZE) -1));
  close(fd);
  if (mapaddress == MAP_FAILED)
    return NULL;
  return (void *)(mapaddress+(offset & (sysconf(_SC_PAGE_SIZE) -1)));
}
```

Fig. 7 Excerpt from map_offset() showing the mmap() invocations.

The function returns a pointer to where the data was loaded into MM. This pointer is used in conjunction with the targeted addresses, like the ones found in section 2.2.2, to find the bytes of data that correspond to the lines of code that we want to monitor. Typically, the file will be loaded into memory without an offset, which will return the effective address 0 of our file. In this case, to find the location of the targeted address in MM we simply add the base pointer of the file provided by the map_offset() function to the target address stored within the FR object. However, if the file is loaded from a specific offset, the pointer returned will not be pointing at address 0 of the file but at the address equal to the offset of the file. In this case, the targeted addresses must be altered by subtracting the offset used to find the correct location of the targeted code. It is also important to note that any data in the file before the offset address will not be loaded into MM and therefore cannot be accessed. If a targeted address is less than the offset, it will not be reachable and may cause errors during runtime. This concept is illustrated in Fig 8.

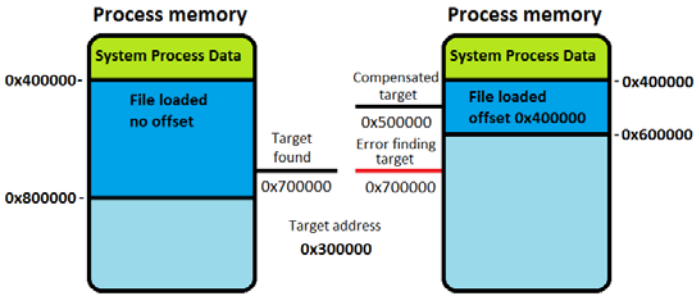


Fig 8 – Illustration of offset calculation and how errors can occur.

**fr_monitor()**

'fr_monitor()' takes an address as an argument and adds it to an array which is used to monitor multiple target addresses at a time. Each address is probed and timed in the 'fr_probe()' function, which returns individual timing results for each probed address. The array is handled by the 'vlist' class which functions as a wrapper for the array, allowing it to be utilized as a vector list. This class lets researchers interact with the array without burdening them with its upkeep, while also supplying the rest of the Mastik framework with a flexible data structure to hold targeted addresses. Fig 9 shows an excerpt of 'vlist.h' which lists the functions contained within this class.

```
vlist_t vl_new();
void vl_free(vlist_t vl);

inline void *vl_get(vlist_t vl, int ind);
void vl_set(vlist_t vl, int ind, void *dat);
int vl_push(vlist_t vl, void *dat);
void *vl_pop(vlist_t vl);
void *vl_poprand(vlist_t vl);
void *vl_del(vlist_t vl, int ind);
inline int vl_len(vlist_t vl);
void vl_insert(vlist_t vl, int ind, void *dat);
int vl_find(vlist_t vl, void *dat);
```

Fig. 9 Excerpt from vlist.h listing its functions

**fr_probe()**

The 'fr_probe()' function is where the attack is run. It is a static function that takes 2 arguments; an FR class that holds the targeted addresses and a pointer to the targeted code, and an array of uint16_t unsinged integers to hold the results of the probes on the targeted addresses. When the 'fr_probe()' function is run, address stored in the FR's 'vlist' are probed one by one and their access times are recorded. Fig 10 is an excerpt of the FR class that shows the 'fr_probe()' function. The structure of this function ensures that each address is probed individually, that a separate access time is generated for each, and that none of the probes interfere with any of the other access times.

```
void fr_probe(fr_t fr, uint16_t *results) {
  assert(fr != NULL);
  assert(results != NULL);
  int l = vl_len(fr->vl);
  for (int i = 0; i < l; i++)  {
    void *adrs = vl_get(fr->vl, i);
    int res = memaccesstime(adrs);
    results[i] = res > UINT16_MAX ? UINT16_MAX : res;
    clflush(adrs);
  }
}
```

Fig. 10 Excerpt of the FR class showing the fr_probe() function

The attack iterates through each of the target addresses using a For loop. During each cycle of the loop there are 4 steps that are executed. First the target address is retrieved from the FR via the 'vl_get()' function. The attack is then run with the 'memaccesstime()' function and the result is retrieved. The result is added to the 'results' array, a pointer of which was parsed into the function. And finally, the targeted address is flushed from the cache in order to prime the address for the next attack. This final step is achieved with this 'clflush()' function which is a very simple one line wrapper function for the 'clflush' assembly instruction.

**Mastik Utilities**

The Mastik framework also contains a set of tools that can help researchers set up their environments and execute their attacks. These tools gather information on the system that is vital to the success of various attacks, such as; the average cache access time for the L1, L2, and LLC, whether or not certain technologies are enabled on the system, and whether or not certain countermeasures are enabled on the system. For our attack, we will be using the 'FR_threshold' tool to help us find a suitable threshold between our systems LLC and MM access timing. This threshold will allow us to interpret the data we gather from our attack and classify individual results as either 'hits' or 'misses'.

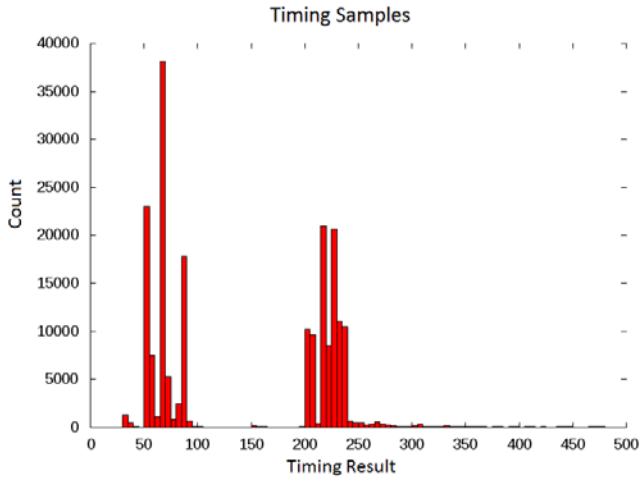


Fig 11 – Graph of timing results from running FR_threshold

'FR_threshold' generates 100000 timing samples, half of which are taken when the target address has been flushed from the cache, and the other half are taken when the target address is loaded into the LLC. The tool then outputs five statistics calculated from these samples for both the MM and the LLC; the minimum, the maximum, the median, and both the top and bottom decile. An example of this output can be seen in Fig 12 and a graph of the access time distribution from a typical FR_threshold execution can be seen in Fig 11. As we can see from the graph in Fig 11, there is a clear range between 100 – 200 where there are almost no samples. We can safely choose a value somewhere around the higher limit of this range to be our threshold for testing whether the target VM has accessed the targeted address or not.

```
                :   Mem  Cache
Minimum         :   189     72
Bottom decile:    195     81
Median          :   204     81
Top decile      :   210     84
Maximum         : 35925   6717
```

Fig 12 – Output from Threshold run

### 2.2.4 Cross VM Cache Timing Attack

Finally, we will discuss our implementation of the Mastik framework and our recreation of the Flush+Reload cache timing attack. Our attack is designed to monitor multiple addresses within a single program simultaneously. It is fundamentally a recreation of other cross VM cache timing attacks [4,21,32,33], and the first attack within the Mastik framework that we aim to test in virtualized environments. Our attack will target a known program that we created that is running on a co-located VM. We wrote our target program to help illustrate the entire attack methodology, from finding target addresses within the program, to

monitoring the target process, and finally interpreting the results.

Our attack code takes in a minimum of two arguments. The first argument received is the file descriptor for the target program that will be monitored. All subsequent arguments are target addresses that will be monitored and are expected to be hexadecimal values. This lets us monitor multiple targets at a time. The command we will be using to invoke our program via command line is '/CTA Hello 6A6 A9C E48 12B4'. This will load the 'Hello' program into memory and set up four monitors, one for each of the functions we are targeting.

There are two main phases in our attack code; The initiation phase and the attack loop. During the setup phase, the user's inputs are collected and verified and our code uses these inputs to set up the FR object. Once this is done the attack loop will run and the collected results will be tested. In turn, each address is probed and the result is tested against our threshold. Fig 13 shows an excerpt of our attack code, illustrating our implementation of the features of the Mastik framework that we described above. Here we can see that the target code is loaded into MM using the 'map_offset()' function on line 39, and the target addresses are added to the FR object using the 'fr_monitor()' function on line 43. Following this the attack loop is initiated and continues until the program is stopped by the user. The 'fr_probe()' function is continuously invoked on line 52.

```
for (int i = 0 ; i < count ; i++ ){
     sscanf(argv[i+2], "%x", &offset[i]);
    fr[i] = fr_prepare();
}

//Add target file to the FR object
void *file = map_offset(target, 0);

//Add our addresses to the FR to be monitored
for(int i = 0 ; i < count ; i++){
    fr_monitor(fr[i], file+offset[i]);
}

//Attack Loop
for (;;) {

  for(int i = 0; i < count; i++){

    //Probe each target address
    fr_probe(fr[i], &time[i]);

    if (time[i] < 175){
        printf("Probe %i was triggered in %u \n", i+1 , time[i]);
    }

  }

}

exit(0);
```

Fig 13 – Excerpt of our recreation attack code

As stated above, our target addresses are 0x6AE, 0xA9C, 0xE48, and 0x12B4, and our cache timing threshold is 175. Our program will monitor each of these addresses and record their access times continuously. Once the access time of one of these addresses is below the threshold an identifier for the address and the access time is printed to the standard output. Fig 14 shows a typical example of what is expected during a successful attack session. In this attack our attack code was running on one VM while the 'Hello' target program was running on another co-located VM. In this example the 'Hello' program received 5 inputs; 'H', 'E', 'L', 'L', and then 'O'. We can see that our attack code was able to record these inputs and output them in our attacker VM. All 'missed' probes are filtered out to help illustrate the attack.

```
Probe 1 was triggered in 55
Probe 2 was triggered in 53
Probe 3 was triggered in 60
Probe 3 was triggered in 51
Probe 4 was triggered in 60
```

Fig 14 – Example of typical attack output.

## 3. Result Analysis

Our experiment shows that an attacker is able to monitor the state of a victim's system in real time, but we haven't yet addressed how an attacker can extract information from the results they collect. The attack environment also influences the results of these attacks, by introducing noise to the result set and potentially erasing results. To successfully interpret these results, the attacker must consider these environmental factors and tailor their analysis to the targets they are monitoring. This chapter will discuss these topics in detail, outlining what the necessary knowledge one needs to analyse these attack results. In the following sections, we will discuss typical results we expect to collect, how the environment affects these results, how each result is interpreted, and how result sets are analysed. Section 3.1 will begin by discussing the details of result collection and how individual results are interpreted. Section 3.2 will cover how information is retrieved from these results via two main methods; teal time monitoring, and results set analysis.

### 3.1 Result Collection

When the fr_probe() command is run the results returned is an integer typically within the range of 10 – 500. This is approximately the number of cycles that it took for the CPU to access the address. Analysing these results requires two steps; first the result needs to be classified as either a 'hit' or

a 'miss' based on whether the target was loaded into cache or not, then these 'hits' and 'misses' need to be interpreted as some expected pattern of operation. The first step

requires a large set of test data and some familiarity with the environment. The second step is covered in section 3.2. In this section, we will discuss how results are classified and how some channel noise can be accounted for.

It is important to correctly classify each result as either a 'hit' or a 'miss'. In most cases this simply means finding a

threshold between the average LLC access time and the average MM access time that has minimal crossover and then testing each result against it. We outlined our method for finding this threshold in section 2.2.3. For example, Fig 11 illustrates that the system we ran our attack on has an empty region between 100 – 200 where results are very uncommon. Most systems will have clearly defined access regions as per our example, but during times of high system load these regions can become blurred resulting in ambiguous results.

One of the motivations behind cache timing attack research is its high time resolution potential. In our experiment, we focus on a timing difference between LLC access and MM access, but using this attack we are also able to identify when addresses are loaded into L1 cache. Each of these regions are usually clearly distinguishable due to the predictable performance differences between them. In our test environment, if the target address is loaded into L1 cache we expect to see an access time between 10 – 30, if the address was loaded into the LLC we expect an access time between 40 – 100, and if the address is in MM we expect an access time to be greater than 200. Table 1 shows statistics compiled from 30000 access timing samples; 10000 from MM, 10000 from LLC, and 10000 from L1. These ranges will differ slightly in each environment depending on the hardware of the system, the architecture of the hypervisor, and the code used to probe the addresses. For this reason, it is important to have a proper understanding of the attack environment and expected results.

Table 1: Access timing region data for MM, LLC, and L1

| Address Location | Minimum | Bottom Decile | Median | Top Decile | Maximum |
|---|---|---|---|---|---|
| **L1** | 21 | 21 | 30 | 33 | 345 |
| **LLC** | 72 | 81 | 81 | 84 | 3592 |
| **MM** | 189 | 195 | 204 | 210 | 6717 |

Noise on a cache side channel can come from 3 main sources; premature cache flushing, monitor process suspension, and unexpected target access. When a high number of processes compete for the CPU at once the cache can experience a high access load, which can affect timing results. The noise generated by this high access load can manifest in two ways, either a third-party process loads data into the cache causing the address we are monitoring to be prematurely evicted, or the OS suspends the attacker process which prevents it from monitoring target addresses for a period of time. The attacker VM can also be suspended at the hypervisors discretion with the same effect. In both cases, there is little we can do to recover the lost data, but when our process is suspended we can at least detect it. This can be done by calculating the time between each cycle of the attack loop. We expect our attack loop to be regular and constant, but if we notice that a single loop takes a much longer time to complete we can infer that our process was suspended and that we lost timing results for that period.

There are also times when a third-party process unexpectedly accesses the target address. The likelihood of this depends greatly on the popularity of the shared code and the attack environment. Consider a cache timing attack targeting a network driver; on a single user PC, this attack will only record the activity of that one user, but on a multitenant hypervisor hosting 8 webservers the results of the attack will be a combination of the activity of all deployed VMs. Our experiments contain an example of this type of noise that will be common to any cache timing attack that relies on hypervisor level page deduplication. The KSM module regularly scans the memory of its deployed VM looking for identical pages. When identical pages are found, the module will merge them and point both VMs to the shared page. To do this the module needs to load the pages it checks into cache and if it checks an address that our attack process is monitoring it will generate a false positive 'hit'. Fig 16 illustrates this noise with an example of the attack output during one of our experiments. In this example both the attacker's VM and victim's VM are idle and the time between each KSM scan was reduced to help demonstrate the noise.

The more an attacker knows about the attack environment the better they can tailor their attack to it. By understanding why this noise was present in our experiments we were able to filter it out of our result sets and increase the accuracy of our attack. What set these false positives apart from our genuine results, and what helped us filter them out, was the fact that they always had a very low access time of approximately ~20 cycles. This stood in contrast to our genuine 'hits' which had an access time of approximately ~50 cycles. This occurred because the KSM kernel module was run on the same CPU core as our attacker's VM, loading the target address into L1 cache. The victim VM was running on a separate CPU core, so that it loaded the target address into the LLC instead. Understanding this we could filter out all results that were too low to indicate that they originated in the LLC.

```
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 33
Probe a was triggered in 198
Probe a was triggered in 24
Probe a was triggered in 198
Probe a was triggered in 24
Probe a was triggered in 33
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 24
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 33
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
Probe a was triggered in 198
```

Fig. 16 Output of the KSM experiment noise experiment

## 3.2 Result Analysis

There are two main methods an attacker can use to interpret the results they gather; real time monitoring, and result set analysis. The majority of cache timing attack research focuses on monitoring encryption algorithms in an attempt to steal secret keys or reduce the time it takes to calculate these keys [38, 39]. This information is retrieved during the result set analysis and takes place after the attack is run. Other applications of this side channel include monitoring user inputs [32], data exfiltration [21], and even reverse shell exploitation [29]. These attacks require a timelier interpretation of attack results and rely on a constant stream of data, therefore they implement the real-time monitoring method. This section will explore the use of each of these methods and discuss what types of information attackers are able to gain from them.

### 3.2.1 Result Set Analysis

Result set analysis aims to interpret a large set of timing attack results collected over a period of time. The aim of the attacker is to record the state changes of a victim system and then analyse these changes to infer some pattern of functionality. This method is the more common of the two, mainly due to the fact that most cache timing attack research aims to monitor encryption algorithms and crack secret keys, which requires this kind of result analysis. The result set is usually analysed by an expert [32, 36, 39], but the analysis can also be automated depending on the specific attack [21, 33, 41]. For example, Yarom et al where able to show how this attack can be used to break AES encryption. Their method requires visualizing the result set as a graph where an expert would be able to extract the generated key bit by bit [36]. Fig 17 illustrates the data they generated from their attack and how the expert is able to interpret it.

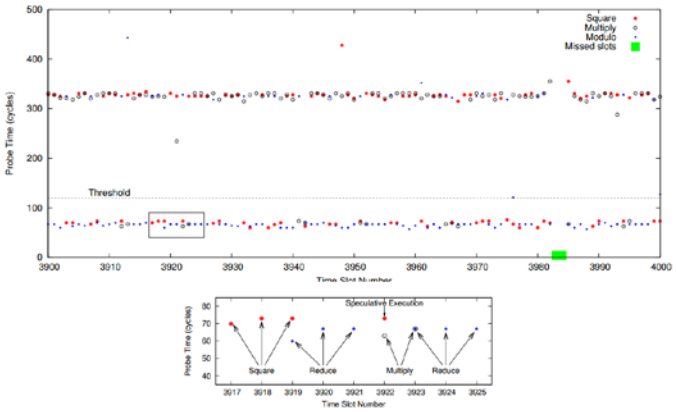


Fig. 17 Results from Yarom et al Flush+Reload attack on AES [36]

The attacker will have a good idea of how the victim should behave and what behavioural patterns they can expect to observe from their victim. They will then begin recording the state of the cache at a time that the victim is expected to be executing the targeted process. Once the initial result set is collected it will be refined to only contain results that are interesting to the attacker. This can be done by scanning the data set until specific targeted addresses are accessed, since the attack should only be targeting a single, specific functionality. From here the data is either visualized in a graph and interpreted by an expert. In theory, this processes could be automated with some analysis code but we were not able to find any solutions of this kind in peer reviewed literature.

### 3.2.2 Real-time Monitoring

Monitoring the cache in real time allows an attacker to respond to state changes in the cache in real time. Here, the attacker will be looking for specific state change patterns in the target system, which they can then interpret in order to trigger some functionality. This method is used primarily in covert channel attacks as a way for two parties to communicate without being detected [19, 20, 21, 23, 24, 29]. This is achieved when two conspiring parties treat this

side channel as a communications channel and attempt to set up either one-way or two-way data transmission. In this way, attackers are able to exfiltrate data [19, 20, 21] and may even allow for a reverse shell connection [29]. Our attack results in section 2.2.4 (Fig 14) are an example of the real-time analysis method and illustrate how an attacker can monitor specific user inputs or system states in real time. This information may prompt the attacker to launch further attacks or it may be used to trigger automated scripts.

## 4. Conclusion and Future Work

In this paper, we accomplish three main objectives; we have documented our attack environment, we have dissected in detail our attack methodology, and we discussed the process of collecting and analysing results from the recreation attack. In order to set up our attack environment, we utilized industry standard Linux based technology. This allowed us to execute a recreation of the cross VM Flush+Reload cache timing attack. We documented this entire process in detail, including the process of collecting and analysing results to extract information. It is clear from the literature that this field of research is gaining popularity as virtualization technology is becoming a more common utility on the internet. From here we aim to further the research in this field with a focus on alternate uses for cross VM side channel attacks.

Our attack methodology discusses in detail the specifics of our attack environment and the execution of our recreation attack. We show that the vulnerabilities that allow this cache timing attack to occur are still present in current hardware and software. Using the Mastik framework, we then successfully recreated a cache timing attack between two VMs. Without correctly understanding and mitigating these vulnerabilities cloud administrators can leave their users vulnerable which may lead to untraceable data theft and other compromises. In our paper, we outline our attack methodology, describing in detail how an attacker is able to carry out these attacks step by step. We believe that this information will be valuable to cloud administrators and new researchers in this field, helping them gain a foothold in this recently popular topic.

We also discussed result collection and analysis. There are a few technical pieces of information that an attacker must obtain in order to correctly gather timing results. By understanding what this information is and how attackers obtain it, cloud administrators and developers will be able to mitigate these attacks by making it harder for attackers to set up their software. Similarly, understanding how these results are analysed and interpreted can help professionals and researchers make it harder for attackers to gain any meaningful information out of their result analysis.

Most research in cache timing attacks has been aimed at cracking encryption algorithms and retrieving secret keys. This has been true since the inception of this research topic in the 1970s. We believe that this is due to the single OS environment of these attacks. The advent of virtualization technologies introduces new applications for these attacks that have yet to be explored. It is only recently that researchers have begun to introduced these new applications with targets such as key stroke timing, partial key logging, mouse activity logging, covert channel attacks, and data exfiltration. Our future research will aim to answer questions such as:

- What shared resources can and can't be used as side channels and what is the extent of their use?
- Can these attacks be used as a form of recognisant, allowing for further comptonization of the target system?
- •Is a malicious VM able to directly alter a target VMs shared resource? If so, to what extent?

We believe that with our current virtualization lab we are well equipped to tackle these questions and continue research in this field well into the future.

## References

[1] Irazoqui, G., Eisenbarth, T., & Sunar, B. (2016, May). Cross processor cache attacks. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (pp. 353-364). ACM.

[2] Hay, B., Nance, K., & Bishop, M. (2011, January). Storm clouds rising: security challenges for IaaS cloud computing. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on* (pp. 1-7). IEEE.

[3] Bonneau, J., & Mironov, I. (2006, October). Cache-collision timing attacks against AES. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 201-215). Springer Berlin Heidelberg.

[4] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., & Sunar, B. (2015). Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint. iacr. org.

[5] Chen, D., & Zhao, H. (2012, March). Data security and privacy protection issues in cloud computing. In *Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on* (Vol. 1, pp. 647-651). IEEE.

[6] Yu, H., Powell, N., Stembridge, D., & Yuan, X. (2012, March). Cloud computing and security challenges. In *Proceedings of the 50th Annual Southeast Regional Conference* (pp. 298-302). ACM.

[7] Cayirci, E., Garaga, A., Santana, A., & Roudier, Y. (2014, December). A cloud adoption risk assessment model. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing* (pp. 908-913). IEEE Computer Society.

[8] van der Veen, V., Andriesse, D., Göktaş, E., Gras, B., Sambuc, L., Slowinska, A., ... & Giuffrida, C. (2015, October). Practical context-sensitive cfi. In *Proceedings of*

*the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 927-940). ACM.

[9] Biedermann, S., Mink, M., & Katzenbeisser, S. (2012, October). Fast dynamic extracted honeypots in cloud computing. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop* (pp. 13-18). ACM.

[10] Gutierrez, E., Kohlenberg, T., Mahankali, S., & Sunderland, B. (2012, January). Virtualizing High-Security Servers in. Intel White Papers.

[11] Ibrahim, A. S., Hamlyn-harris, J. H., & Grundy, J. (2010). Emerging security challenges of cloud virtual infrastructure.

[12] Hamlen, K., Kantarcioglu, M., Khan, L., & Thuraisingham, B. (2012). Security issues for cloud computing. Optimizing Information Security and Advancing Privacy Assurance: New Technologies: New Technologies, 150.

[13] Agrawal, S. (2013). Establishing Trust in Cloud Computing. Journal of Indian Research, 1(1), 91-97.

[14] Takabi, H., Joshi, J. B., & Ahn, G. J. (2010). Security and privacy challenges in cloud computing environments. IEEE Security & Privacy, (6), 24-31.

[15] Anthes, G. (2010). Security in the cloud. Communications of the ACM, 53(11), 16-18.

[16] Kaufman, L. M. (2009). Data security in the world of cloud computing. *Security & Privacy, IEEE*, *7*(4), 61-64.

[17] Mansukhani, B., & Zia, T. A. (2011). An empirical study of challenges in managing the security in cloud computing.

[18] Fernandes, D. A., Soares, L. F., Gomes, J. V., Freire, M. M., & Inácio, P. R. (2014). Security issues in cloud environments: a survey. *International Journal of Information Security*, *13*(2), 113-170.

[19] Bates, A., Mood, B., Pletcher, J., Pruse, H., Valafar, M., & Butler, K. (2014). On detecting co-resident cloud instances using network flow watermarking techniques. *International Journal of Information Security*, *13*(2), 171-189.

[20] Ristenpart, T., Tromer, E., Shacham, H., & Savage, S. (2009, November). Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (pp. 199-212). ACM.

[21] Wu, Z., Xu, Z., & Wang, H. (2015). Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. IEEE/ACM Transactions on Networking (TON), 23(2), 603-614.

[22] Rodero-Merino, L., Vaquero, L. M., Caron, E., Muresan, A., & Desprez, F. (2012). Building safe PaaS clouds: A survey on security in multitenant software platforms. computers & security, 31(1), 96-108.

[23] Younis, Y. A., Kifayat, K., Shi, Q., & Askwith, B. (2015, October). A New Prime and Probe Cache Side-Channel Attack for Cloud Computing. In Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on (pp. 1718-1724). IEEE.

[24] Aciiçmez, O. (2007, November). Yet another microarchitectural attack:: exploiting I-cache. In Proceedings of the 2007 ACM workshop on Computer security architecture (pp. 11-18). ACM.

[25] Bull, R. L., & Matthews, J. N. (2014). Exploring Layer 2 Network Security in Virtualized Environments. Retrieved Oct, 19, 2014.

[26] Nir Valtman, Moshe Ferber. (2015). From 0 To Secure In 1 Minute — Securing IAAS. Defcon 23. Las Vegas: DEFCON.

[27] Flynn, C. (2015). Don't Whisper my Chips: Side channel and Glitching for Fun and Profit. Defcon 23. Las Vegas: DEFCON.

[28] Yuriy Bulygin, Mikhail Gorobets, Alexander Matrosov, Oleksandr Bazhaniuk, Andrew Furtak. (2015). Attacking Hypervisors Using Firmware and Hardware. Defcon 23. Las Vegas: DEFCON.

[29] Martineau, E. (2015). Inter-VM data exfiltration: The art of cache timing covert channel on x86 multi-core. Defcon 23. Las Vegas: DEFCON.

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

[31] TCSEC, D. O. D. (1985). Trusted computer system evaluation criteria. DoD 5200.28-STD, 83.

[32] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., & Keromytis, A. D. (2015). The Spy in the Sandbox--Practical Cache Attacks in Javascript. arXiv preprint arXiv:1502.07373.

[33] Liu, F., Yarom, Y., Ge, Q., Heiser, G., & Lee, R. B. (2015, May). Last-level cache side-channel attacks are practical. In IEEE Symposium on Security and Privacy (pp. 605-622).

[34] VMware Inc., "Security considerations and disallowing inter-virtual machine transparent page sharing," VMware Knowledge Base 2080735 http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735, Oct 2014.

[35] Page, D. (2003). Defending against cache-based side-channel attacks. Information Security Technical Report, 8(1), 30-44.

[36] Yarom, Y., & Falkner, K. (2014). Flush+ reload: a high resolution, low noise, L3 cache side-channel attack. In 23rd USENIX Security Symposium (USENIX Security 14) (pp. 719-732).

[37] García, C. P., Brumley, B. B., & Yarom, Y. " Make Sure DSA Signing Exponentiations Really Are Constant-Time".

[38] Irazoqui, G., Inci, M. S., Eisenbarth, T., & Sunar, B. (2014, September). Wait a minute! A fast, Cross-VM attack on AES. In International Workshop on Recent Advances in Intrusion Detection (pp. 299-319). Springer International Publishing.

[39] Yarom, Y., & Benger, N. (2014). Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. IACR Cryptology ePrint Archive, 2014, 140.

[40] Hornby, T. Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+ RELOAD.

[41] Gruss, D., Spreitzer, R., & Mangard, S. (2015). Cache template attacks: Automating attacks on inclusive last-level caches. In 24th USENIX Security Symposium (USENIX Security 15) (pp. 897-912).

[42] Yarom, Y. (2016, August). *Mastik: A Micro-Architectural Side-Channel Toolkit*. Retrieved from School of Computer Science Adelaide: http://cs.adelaide.edu.au/~yval/Mastik/