Inferring Firewall Rules by Cache Side-channel Analysis in Network Function Virtualization

Youngjoo Shin
School of Computer and Information Engineering
Kwangwoon University
Seoul, Republic of Korea
yjshin@kw.ac.kr

Dongyoung Koo
Department of Electronics and Information Engineering
Hansung University
Seoul, Republic of Korea
dykoo@hansung.ac.kr

Junbeom Hur
Department of Computer Science and Engineering
Korea University
Seoul, Republic of Korea
jbhur@korea.ac.kr

***Abstract*—Network function virtualization takes advantage of virtualization technology to achieve flexibility in network service provisioning. However, it comes at the cost of security risks caused by cache side-channel attacks on virtual machines. In this study, we investigate the security impact of these attacks on virtualized network functions. In particular, we propose a novel cache-based reconnaissance technique against virtualized Linux-based firewalls. The proposed technique has significant advantages in the perspective of attackers. First, it enhances evasiveness against intrusion detection owing to the ability of source spoofing. Second, it allows inference on a wide variety of filtering rules. During experiment in VyOS, the proposed method could infer the firewall rules with an accuracy of more than 90% by using only a few dozen packets. We also present countermeasures to mitigate cache-based attacks on virtualized network functions.**

***Index Terms*—Network function virtualization, Cache side-channel analysis, Firewall reconnaissance**

I. Introduction

In network function virtualization (NFV), a variety of network functions such as a router and firewall run as instances of virtual machines (VMs) on commodity servers. To achieve flexibility in provisioning, networking features such as packet forwarding and filtering are implemented in software rather than in dedicated hardware. Like the other software, execution of those features inherently makes memory access, which in turn leaves footprints on a CPU cache. In the NFV architecture, the CPU cache is shared across VMs. Thus, the internal processes of forwarding and filtering are visible to a co-located VM through *cache side-channel analysis* [1], [2]. This results in a serious security breach as sensitive information about network functions can be leaked to unauthorized users. Although a wide variety of security challenges in NFV have been explored in the literature [3], [4], the real effects of cache side-channel threats on virtualized network functions have never been studied.

In this paper, we study the security impact of cache side-channel attacks on NFV applications in the real world. In particular, we implement a *cache-based reconnaissance* on the policy of virtualized firewalls. Firewall policy reconnaissance is a type of network scanning technique to learn packet filtering rules of a firewall. As a preliminary step of network infiltration, this technique is of crucial importance for attackers as it allows them to know what types of packets (*i.e.,* IP addresses, protocols and port numbers) can go through the firewall. All the existing techniques [5]–[8] are based on the probe-and-response approach, which watches for a response for the probe packet. In this paper, we present a new approach for a more powerful reconnaissance technique. Rather than observing the responses, our technique learns the filtering rules through the cache side-channel.

In our approach, we consider an attacker whose VM is physically co-located in the same host with a victim (*i.e.,* a firewall); however, it is logically located outside the private network that the firewall is protecting. Indeed, such an adversary model is practical in certain applications such as 5G networks, where multiple network slices from different domains co-exist in a host. The co-located attacker performs the cache-based reconnaissance as follows. First, he sends a probe packet to a virtualized firewall, and then obtains traces of cache status while the packet is being handled by the victim. Through classification of the cache trace, he can determine the type of action (*i.e.,* filtering rule) that has been applied to the packet.

This technique has significant advantages over previous probe-and-response methods from the perspective of attackers. First, because the attacker does not require responses, he can easily spoof the source address of a probe packet. Source-spoofing hides the location of the probing hosts; consequently, it enhances the evasiveness against intrusion detection. Furthermore, with no restriction in the range of source addresses, it extends the search space of firewall policies. Second, it provides information not only about primary actions such as allowance or denial but also about various actions such as logging or network address translation (NAT). This obviously

allows more useful information to attackers who aim to infiltrate the network.

We present details of the implementation of our cache-based reconnaissance technique in this paper. The implementation aims at Linux-based virtualized firewall products. In particular, we target *iptables*, a kernel component that provides the main functionality of a firewall. Packet filtering of most Linux-based products such as VyOS, IPFire, Smoothwall and OPNSence are constructed upon iptables. The basic idea is to trace invocations of functions in the Linux kernel and iptables by observing cache behaviors with a Flush+Reload technique [1].

Although our approach utilizes the existing cache side-channel analysis technique, it is not trivial to implement it in the firewall reconnaissance. We have several technical challenges that should be overcome. The first challenge is that all the target functions that have to be traced are located in kernel memory pages, which are subject to kernel address space layout randomization (KASLR). The uncertainty introduced by KASLR results in difficulty in preparing identical pages in the VM of an attacker, which is necessary to mount the Flush+Reload technique. We resolve the unknown part of the page by guessing with a divide-and-conquer-based approach.

The second challenging problem is that learning information through cache side-channel is prone to errors under the existence of background network traffic. Our solution for this problem is to exploit the high linearity between the arrival rate of probe packets and the number of measured cache hits. Based on this idea, we devise a filtering rule inference algorithm which is highly accurate regardless of the noise incurred by the background traffic.

We prove the efficiency of our reconnaissance technique by evaluating its performance against VyOS[1], the most widely used Linux-based virtual router/firewall, on a KVM hypervisor. Experimental results demonstrate that all the target pages can be resolved using only 34 MB of memory. The rule inference algorithm requires only a few dozen packets to figure out the rule with an accuracy of more than 90%. It is to be noted that our technique is not confined to VyOS; further, it is generally applicable to all types of Linux-based systems.

We also present possible solutions to mitigate the cache-based firewall reconnaissance technique. Certain countermeasures can be utilized in designing the next generation intrusion detection systems to cope with a variety of cache-based attacks in NFV environments.

The rest of this paper is organized as follows. Related work and background knowledge are presented in Sections II and III, respectively. Details of the firewall policy reconnaissance through cache side-channel exploitation are presented in Section IV. The performance of the proposed method is evaluated in Section V. Countermeasures against attacks are described in Section VI. Finally, we conclude the paper in Section VII.

II. Related Work

In this section, we discuss certain related work regarding firewall reconnaissance and cache side-channel attacks.

[1]Open Source Linux-based Networking OS, https://vyos.io

Firewall reconnaissance. It is a scanning technique on a firewall to determine which probe packets are permitted to reach the hosts behind the firewall. As there is a large space in the header fields (*e.g.,* IP addresses, port numbers and protocols) of the probe packet, it is certainly inefficient to scan all the possible values by brute force. Therefore, reducing the searching complexity is the main concern of previous works. For this purpose, Kim and Ju [6] proposed a line sweep algorithm to generate probe packets to efficiently cover the search space. Samak *et al.* [8] proposed two different methods based on region growing and split-and-merge strategy to reduce the complexity. Ali *et al.* [5] presented a hybrid approach that combines these two methods to leverage their advantages.

There are also other works that are similar to this study. Schmitt and Schinzel [7] and Lin *et al.* [9] proposed reconnaissance methods against a web application firewall and an SDN controller, respectively. Khakpour *et al.* [10] proposed a fingerprinting technique to identify the implementation of a target firewall rather than inferring its filtering rules.

All the previous works basically followed the probe-and-response approach, *i.e.,* observe an arrival of response for a given probe packet. In this paper, we propose a new type of approach that utilizes the cache behavior instead of observing the response packets. Our approach enables a more powerful reconnaissance than previous methods in several aspects. First, the removal of responses makes an attacker free to spoof the source address of a probe packet. Source-spoofing enables not only to enhance evasiveness against intrusion detection, but also to extend the search space of firewall policies. Second, our approach allows inference on a wide variety of filtering rules rather than packet allowance and denial, which provides more useful information to attackers.

It is also noteworthy that the proposed technique is orthogonal to the previous works. That is, the complexity reduction methods described above can be utilized in combination with our cache-based reconnaissance technique.

Cache side-channel attacks. Cache side-channel attack enables an attacker to obtain private information regarding the victims in a shared physical system such as in a virtualized environment, by monitoring the cache access made by the victim [11]. There are several works that present cache side-channel attacks on various target applications. Most works illustrated their attacks against cryptographic algorithms to prove the security impact of cache side-channels. Specifically, they demonstrated that certain algorithms are vulnerable to Flush+Reload attacks, such as RSA [1], ECDSA [12], ECDH [13], [14] and AES [15], [16]. Several works utilized a Prime+Probe technique and its variants to demonstrate attacks against cryptographic algorithms [17]–[20].

Besides cryptographic algorithms, Gruss *et al.* [21] and Lipp *et al.* [22] illustrated attacks on practical software, such as extracting user inputs from a keyboard and touchscreen, respectively. Zhang *et al.* [23] showed the browsing history in a web browser can be leaked through the cache side-channel.

Weber *et al.* [24] presented a method to construct SSH over a cache-based covert channel.

To the best of our knowledge, our work is the first that exploits a cache side-channel attack to extract information about firewall filtering rules in NFV environments.

III. Background

A. Cache side-channel analysis

Cache is a hardware component to bridge the gap of latency between a register and a memory. Cache side-channel analysis is an attack technique to extract unauthorized data from other VMs by exploiting the cache as a leakage source.

Flush+Reload attack [1] is one of the techniques for cache side-channel analysis. It leverages a page sharing feature (known as memory deduplication) of hypervisors, which merges the same pages across multiple VMs into a single shared page for the purpose of efficient memory utilization. The attack proceeds in three phases. In the *Flush* phase, an attacker flushes a target cache line shared with the victim from the entire cache hierarchy. He can use a clflush instruction in an x86 instruction set to flush the cache line. During the *Wait* phase, the attacker waits for a certain amount of time while the victim performs security sensitive operations. Finally, in the *Reload* phase, he reloads the target line and measures its latency. Lower reload time indicates that the line was accessed from the cache (*i.e.,* cache hit), which implies that the victim accessed it during the Wait phase. Conversely, higher reload time implies that the line was not accessed and still resides in the memory (*i.e.,* cache miss).

The Flush+Reload technique is not applicable when the page sharing is disabled or not supported. In this case, a Prime+Probe technique [2] can be used as an alternative to the Flush+Reload.

B. iptables

iptables is a kernel component that provides the main functionality of a firewall to Linux-based virtualized network functions. It consists of a number of tables, each of which has a specific purpose in packet handling. Among them, a *filter* table is associated with filtering packets according to the firewall policy defined by users. The firewall policy is actually constructed as a list of ACL rules. Each rule defines the shape of packets (*e.g.,* IP addresses, port numbers and protocols) and the target action (*e.g.,* ACCEPT, DENY and REJECT) to perform with a packet matched with the rule. In Linux, iptables is implemented on the top of Netfilter, a hook-based framework for various networking operations. Thus, a filtering rule is internally comprised of a set of function hooks. Whenever an operation (*e.g.,* matching a rule against a packet or triggering an action) is initiated, specific functions registered to the hook will be invoked to complete the operation.

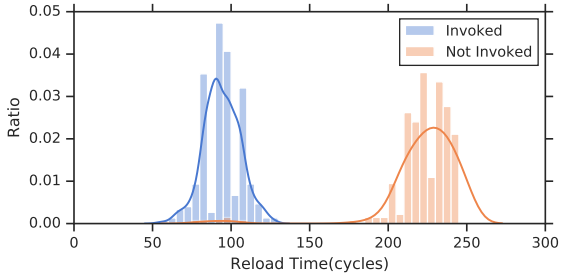

Figure 1. Distributions of reload time on a memory address of ip_forward()

IV. Firewall policy reconnaissance

A. Adversary model

In our adversary model, we consider an attacker whose VM is physically co-located with a victim firewall, but logically located outside the private network that the firewall is protecting. The attacker runs his own VM on the host with privileged access to the guest OS. Therefore, he has an ability to launch cache side-channel attacks against the victim. The attacker can also choose a remote host from arbitrary locations, which is capable of sending probe packets to the victim. In fact, there are many possible ways to realize it in practice. For instance, he can launch another guest VM in the cloud or attempt to occupy benign PCs by distributing malware.

In our assumption, the attacker has prior information about the victim firewall such as its product name and the version. We also assume that a hypervisor supports page sharing, which is necessary to use the Flush+Reload technique. However, our reconnaissance method is not confined to the specific cache side-channel technique.

B. Overview of the proposed method

Linux-based firewalls rely on the kernel networking stack to handle the forwarding, routing, and filtering of network packets. While a packet is traversing through the stack, a series of kernel APIs (including functions in iptables) are invoked to process the packet accordingly. Then, the invocation leaves footprints on a cache, which can be observed through cache side-channel analysis. For instance, Fig.1 shows different distributions of reload time on the memory address of ip_forward(), a packet forwarding function in the kernel. Therefore, through the side-channel, we can learn whether the target function was invoked upon the arrival of the packet.

Our approach for firewall policy reconnaissance basically utilizes cache side-channel on packet processing. Specifically, we send a probe packet to the victim; then, obtain traces of reload time on kernel functions through the Flush+Reload technique. Subsequently, we identify the action that was applied to the probe packet by classifying it according to the invoked functions.

As described in Section III-B, the core functionality of a Linux-based firewall is constructed upon iptables. Therefore, we choose a set of target functions from iptables to trace their invocations for our purpose. For a precise deduction, we also

Table I
TARGET FUNCTIONS FOR FIREWALL POLICY RECONNAISSANCE

Function	Action					Location	Offset†
	A	D	R	L	N		
ip_forward()	O	O	O	O	O	Kernel text	0x818f2800
ip_forward_finish()	O	-	O	O	-	Kernel text	0x818f2790
reject_tg()	-	-	O	-	-	LKM (ipt_REJECT.ko)	0x30
log_tg()	-	-	-	O	-	LKM (xt_LOG.ko)	0x50
nf_nat_ipv4_manip_pkt()	-	-	-	-	O	LKM (nf_nat_ipv4.ko)	0x70

A: ACCEPT, D: DROP, R: REJECT, L: LOG, N: NAT
O: Invoked, -: Not invoked, †: Linux kernel version 4.18.0-20

choose certain additional functions from a Linux networking subsystem in the kernel as well.

Table I enumerates the target functions we selected for the reconnaissance. Two of those functions, ip_forward() and ip_forward_finish(), are closely related to packet forwarding and are chosen from the kernel text segment. The other functions, reject_tg(), log_tg() and nf_nat_ipv4_manip_pkt(), are associated with specific filtering actions of iptables such as REJECT, LOG and NAT, respectively. Hence, they are chosen from the corresponding loadable kernel modules (LKMs) of iptables. All the functions listed in the table comprise the smallest set that allows us to distinguish among different filtering actions. For instance, ACCEPT and DROP are the primary actions in the filtering rules, which can be distinguished by only observing invocations of ip_forward() and ip_forward_finish().

Although the basic idea seems straightforward, it is not trivial to implement the technique in the practical system due to several technical challenges. In the following two subsections, we describe these problems and present our solutions to overcome them in detail.

C. Resolving target pages

All the functions listed in Table I belong to memory pages in the kernel space. Tracing these functions via a Flush+Reload technique is not trivial because kernel memory pages are subject to KASLR. For the sake of security, KASLR randomizes the mapping of kernel pages into the virtual address space. This results in a divergence of addresses of kernel symbols (*i.e.,* functions and global variables). That is, whenever a memory mapping is performed, text segments of the kernel and LKMs should be relocated accordingly so that kernel symbols are correctly referenced. This implies that the content of a kernel page varies for every instance of memory mapping. Fig. 2 shows an example of the divergence of a kernel page associated with the reject_tg() function.

The challenging problem is that such divergence caused by KASLR hinders content-based page sharing of hypervisors, which is necessary to utilize the Flush+Reload technique. As a target page of the victim contains uncertainty in its content, we are required to resolve the unknown part of the page by guessing every possible content.

Assembler code for function reject_tg()	Load 1 (mapped at 0xffffffffc08f6000)	Load 2 (mapped at 0xffffffffc089b000)
...	...	...
mov 0x8(%rsi),%rax	0x35<+5>: 488b4608	0x35<+5>: 488b4608
mov 0x10(%rsi),%rcx	0x39<+9>: 488b4e10	0x39<+9>: 488b4e10
cmpl $0x8,(%rax)	0x3d<+13>:833808	0x3d<+13>:833808
mov (%rcx),%edx	0x40<+16>:8b11	0x40<+16>:8b11
ja 0xcd	0x42<+18>:0f8785000000	0x42<+18>:0f8785000000
push %rbp	0x48<+24>:55	0x48<+24>:55
mov (%rax),%eax	0x49<+25>:8b00	0x49<+25>:8b00
mov %rsp,%rbp	0x4b<+27>:4889e5	0x4b<+27>:4889e5
mov sys_call_table(,%rax,8),%rax	0x4e<+30>:488b04c540708fc0	0x4e<+30>:488b04c540c089c0
jmpq __x86_indirect_thunk_rax	0x56<+38>:e9d5cf90f4	0x56<+38>:e9d57fd6e0
mov $0xd,%esi	0x5b<+43>:be0d000000	0x5b<+43>:be0d000000
callq nf_send_unreach	0x60<+48>:e85bb0ffff31	0x60<+48>:e85b90ffff31
...	...	...

Figure 2. Divergence of a kernel page in a LKM (ipt_REJECT.ko)

Our solution for resolving a target page is as follows. Let us assume that an entropy (*i.e.,* the amount of uncertainty) of the target page is e, and there exists an indicator ISDEDUP(m), which informs whether a given memory page m gets deduplicated (*i.e.,* merged) with other identical pages by a hypervisor. We map all the 2^e candidate pages into the memory space of our VM, then look for one merged with the victim's page by querying ISDEDUP for every candidate.

In order to realize our solution, we should overcome some hurdles. First, we have to devise the effective indicator function ISDEDUP. Second, we should reduce the complexity when a target page has a large entropy e.

Identifying deduplicated pages. Memory deduplication of hypervisors is basically implemented based on a copy-on-write (CoW) mechanism. Specifically, a hypervisor merges duplicate pages into a single physical page and then sets its permission bit with read only access. When a write access occurs at the page, a page fault exception will be triggered. An exception handler then splits the merged pages back into the independent ones so as to deal with the write operation properly. We observed that a write access to the merged page takes a longer time, which is clearly measurable with timestamp counter instructions, such as rdtsc. Fig.3 shows traces of write access time for two different pages on a KVM hypervisor. One of them has its duplicate in memory, so is very likely to get deduplicated. Conversely, the other is filled with random numbers, therefore deduplication will hardly occur. As shown in Fig.3, we can observe that a number of peaks repeatedly occur only on the duplicate page because of the repetition in merging and splitting.

By exploiting such differences in write access time, we devise the indicator function ISDEDUP. That is, if a given page m shows a pattern similar to Duplicate page in Fig.3 in its write access time, ISDEDUP(m) function returns true; else false.

Reducing the complexity. The complexity of guessing is entirely dependent on the entropy of a target page. Basically, the source of the entropy comes from the amount of uncertainty provided by KASLR. In a 64-bit Linux, KASLR introduces 9-bit entropy for a kernel text (base) and 10-bit for LKMs [25], [26]. In addition, for certain pages that belong to LKMs, dependency to external components (*i.e.,* a kernel base or

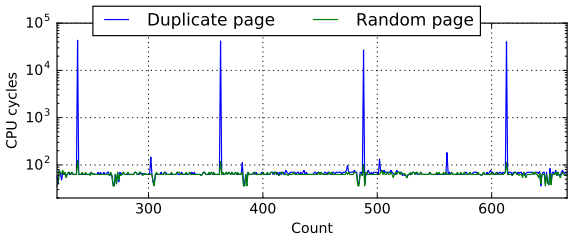

Figure 3. Write access time of two memory pages on KVM hypervisor

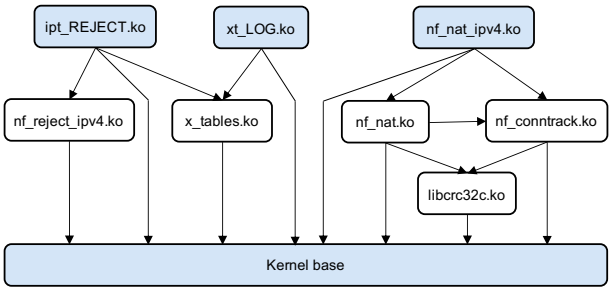

Figure 4. Module dependency of iptables (The LKMs that contain target functions are depicted by shaded blocks)

other LKMs) is another source that augments the entropy. The dependencies of LKMs in iptables are represented in Fig.4.

Based on this, we calculate the complexity of resolving target pages. For instance, a page containing reject_tg() function, which belongs to a module named ipt_REJECT.ko, has an entropy of 39-bit, which is calculated as 10 bits (for the module itself) + 10 bits (for nf_reject_ipv4.ko) + 10 bits (for x_tables.ko) + 9 bits (for a kernel base). In this example, we have to prepare 2^{39} memory-mapped pages for guessing in the worst case. This implies that we need 2 PB ($= 2^{39} \times 4$ KB) of memory in total, which is too huge to resolve the target page in practice through such a naive method.

We propose a divide-and-conquer method to reduce the computational complexity. In particular, we divide the problem of resolving a target page, in which multiple dependencies have to be dealt with at the same time, into a number of smaller problems to resolve a (randomized) base address of each module. By solving each problem one by one, we can successfully resolve the target page with low complexity.

In the above scenario, for instance, the target page containing reject_tg() has dependencies to three external modules (including a kernel text). Initially, we attempt to resolve the base addresses for those modules on a one-by-one basis. Note that with knowledge of base addresses we can directly decide the location of symbols, which in turn determines the content of the target page. Among those external modules, a kernel text has only a dependency to itself. Therefore, the problem of obtaining its base address, which is randomized by KASLR, can be solved with up to 2^9 guessed pages. The other modules, nf_reject_ipv4.ko and x_tables.ko, have dependencies to the kernel text as well as the module itself. As a base address for the kernel text was already resolved, we only need to determine base addresses of modules with up to 2^{10} guessed pages for each. Now all the dependencies have been resolved except the dependency to the last module (*i.e.,* ipt_REJECT.ko). This can also be resolved with 2^{10} guessed pages. Therefore, the maximum amount of required resources is $2^9 + 3 \cdot 2^{10} < 2^{12}$ pages in total, which is approximately 16 MB of memory.

D. Classifying probe packets

Our reconnaissance technique basically infers the filtering rule by classifying a probe packet according to the action it triggered. The arrival of the packet will invoke target functions in the victim. By tracing the function calls via the cache side-channel, we can learn the action that was applied to the packet.

If there are no other packets traversing the firewall, only a single probe packet is sufficient to learn the action. Suppose that the probe packet sent from a remote host reaches the firewall with a latency d. Without any irrelevant packets, a simple method to learn the action is as follows: send a probe packet to the firewall and wait for a minimum time d, then measure the cache status through the Flush+Reload technique.

However, this inference method is highly error-prone. This is because even a single occurrence of an irrelevant packet during the waiting phase would compromise the state of cache and result in a wrong decision. In real environments where there exists background network traffic, this method is likely to suffer from a high amount of noise.

Proposed method. We propose a rule inference method that is able to operate under a high amount of background traffic. Before giving details, we present our observation regarding a relation between the arrival rate of probe packets (*i.e.,* packets per second or pps) and the number of measured cache hits on addresses of target functions. Let us consider a remote host that sends the same probe packets in bulk each time. Its sending rate (*i.e.,* pps), say r, is under the control of the co-located attacker. While probe packets are being sent from the remote host, the attacker measures the number of cache hits, say h, on target memory lines. According to our experiment in Fig.5, we found that there is a strong linearity between the arrival rate of packets (r) and the number of cache hits (h).

In the experiment, the remote host sent probe packets at varying rates, while a co-located VM measured the cache state accordingly. Various types of probe packets were used so that each type of packet matches different actions. As shown in Fig.5, the linearity between the rate of packets and the cache hits is clearly observed in every type of packets. More importantly, we also observe from the graph that each type has a unique and distinguishable pattern in target functions that show cache hits.

Our method for inferring filtering rules on probe packets exploits this linearity. The basic idea is to classify probe packets with respect to target functions that show high linearity in their cache hits with the arrival rate of the packets. The proposed method is presented in Algorithm 1. This algorithm

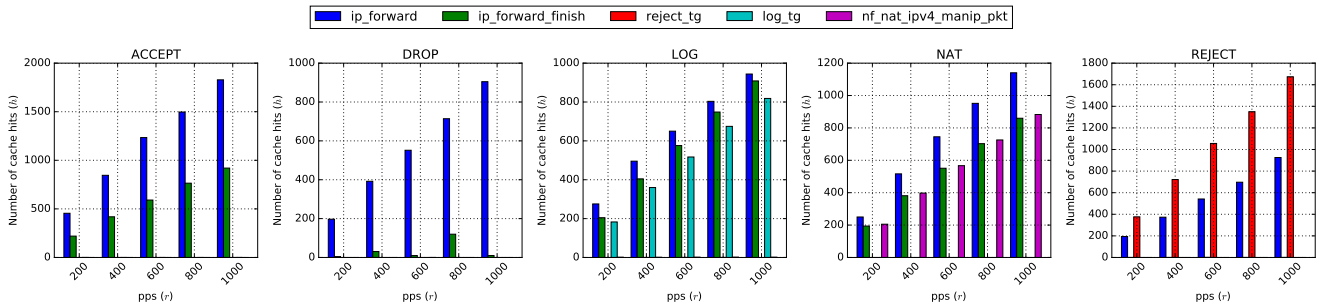

Figure 5. Linearity of the number of cache hits with respect to the arrival rate of probe packets (pps)

Algorithm 1 Inferring Packet Filtering Rule

Require: Probe packet p
(N, $\mathcal{M}_{pps}$ and Δ_T are configurable parameters)
Ensure: Filtering rule on the probe packet

1: **procedure** INFERRULE(p)
2: $R \leftarrow \emptyset$
3: $\forall_{i\in\{1,\ldots,\tau\}} H^i \leftarrow \emptyset$ ▷ R and H^i are ordered lists
4: **for** $n \leftarrow 1$ **to** N **do**
5: $r_n \xleftarrow{R} \{0, 1, \ldots, \mathcal{M}_{pps}\}$
6: $(h_n^1, \ldots, h_n^\tau) \leftarrow \text{FRTRACE}_{\Delta_T}(p, r_n)$
7: $R \leftarrow R \cup r_n$
8: $\forall_{i\in\{1,\ldots,\tau\}} H^i \leftarrow H^i \cup h_n^i$
9: **end for**
10: $\forall_{i\in\{1,\ldots,\tau\}} c_i \leftarrow \text{CORR}(R, H^i)$
11: Rule=CLASSIFY($\vec{c}$) ▷ $\vec{c} = (c_1, \ldots, c_\tau)$
12: **return** Rule
13: **end procedure**

takes as an input a probe packet p and outputs the rule applied to the packet based on the classification. In the algorithm, N, $\mathcal{M}_{pps}$ and Δ_T serve as attack parameters configurable by attackers. Given the packet p, the attacker performs the rule inference according to the algorithm as follows:

1) (Lines 4-9) The following operations a) and b) are executed N times.
   a) Choose the arrival rate of packets ($0 \leq r_n \leq \mathcal{M}_{pps}$) at random.
   b) Invoke a program $\text{FRTRACE}_{\Delta_T}(p, r_n)$. This program consists of two modules, P_{remote} and $P_{colocated}$. Residing in the remote host, P_{remote} sends p in bulk with rate of r_n for a short interval Δ_T. At the same time, $P_{colocated}$, which resides in the co-located VM, measures the number of cache hits on addresses of τ target functions during the interval (*e.g.,* $\tau = 5$ in our example) by using the Flush+Reload technique. Consequently, the program outputs the measurements $h_n^1, h_n^2, \ldots, h_n^\tau$ on these addresses.
2) (Line 10) Ordered lists R and H^i ($1 \leq i \leq \tau$) are obtained as a result of the above iteration, where R is a list of r_n ($1 \leq n \leq N$) and H^i is a list of h_n^i for the i-th target function. Then, a correlation coefficient c_i for each H^i with regard to R is computed by calling $\text{CORR}(R, H^i)$.
3) (Lines 11-12) Given a vector of correlation coefficients $\vec{c} = (c_1, \ldots, c_\tau)$, the probe packet p is classified by calling $\text{CLASSIFY}(\vec{c})$. Based on our observation, the classification is sufficiently simple. For instance, if only correlation coefficients for ip_forward() and reject_tg() are shown as high, then the packet is classified into the class of REJECT action (refer to Fig.5). The filtering rule applied to the packet p is finally decided from the classification.

Complexity analysis. Now we present an analysis of the complexity of the algorithm in terms of the execution time and the total amount of required packets. The vast majority of the time required executing the algorithm comes from the iteration (*i.e.,* lines 4-9 in Algorithm 1). As time Δ_T is required for each iteration to execute FRTRACE, the total amount of time is estimated to be $\mathcal{T} = N \cdot \Delta_T$. Specifically, only $\mathcal{T} = 5$ seconds is sufficient for the algorithm to successfully infer the rule (details will be presented in the subsequent section).

We also calculate $\mathcal{V}$, the expected number of packets in total, which are consumed by the execution of the algorithm, as follows:

$$\mathcal{V} = \sum_{n=1}^{N} E[r_n] \cdot \Delta_T = \frac{\mathcal{M}_{pps}}{2} \cdot N \cdot \Delta_T. \quad (1)$$

The above equation indicates that the total amount of packets is determined by not only $\mathcal{T}(= N \cdot \Delta_T)$ but also $\mathcal{M}_{pps}$. In the proposed method, the parameter $\mathcal{M}_{pps}$ is decided according to the condition of the background network traffic. In particular, the traffic pattern (with respect to pps) observed for the interval Δ_T determines the parameter. For the traffic with low fluctuations, a small value of $\mathcal{M}_{pps}$ is sufficient to infer the rule through the algorithm. However, if the level of fluctuation is high, it will likely disturb the inference because of the high amount of noise. In this case, the parameter needs to be increased, which eventually results in the growth of $\mathcal{V}$.

Improvement. In order to reduce the required amount of packets ($\mathcal{V}$) under highly fluctuating background traffic conditions, we craft a special probe packet so that a specific marker

is embedded in its IP header. Since there are a variety of header fields in a packet, we need to select the most proper field satisfying the following properties: *cache visibility* and *uniqueness*. The former means that the marker should cause invocation of certain functions that are observable through cache side-channel; while the latter means the marker should be unique to only probe packets. The uniqueness helps to achieve high signal-to-noise ratio regardless of the pattern of the background network traffic.

Considering the requirements, we choose IP Options field in IP header as the marker for probe packets. This field contains various options for handling the packet on an IP networking layer. When a received packet is decided to be acceptable by the filtering rules, the Linux kernel examines whether the packet has options. If so, the option is processed by invoking a kernel function ip_forward_options(), and the packet is forwarded to an outgoing interface. Hence, the IP Options field has cache visibility. Moreover, since this field is rarely used in most IP packets of network traffic, the uniqueness is also achieved.

It is noteworthy that there are certain limitations in using the IP Options field as a marker. First, this field provides limited information about filtering rules. Specifically, it allows us to only learn whether the probe packet is accepted or not. While concerning the primary purpose of policy reconnaissance (*i.e.,* to identify the class of packets that can go through the firewall), we emphasize that such information is sufficient to achieve the goal. Second, the probe packets with the marker are likely to be dropped by intermediate routers en route in the Internet due to performance or security issues. This may limit the usage of option field as a marker. However, there are still a fraction of routers that permit the packets with IP Options in practice. Note that in our attack model, we consider an adversary with an ability to utilize a remote host on arbitrary locations. Therefore, it is possible to locate the remote host that achieves zero packet drops by routers on its path.

V. Evaluation

In this section, we analyze performance of the proposed reconnaissance method, and present the experimental results.

A. Experimental setup

The experimental setup consists of two hosts, both of which are connected to each other over the network. One of the hosts is a PC that acts as an attacker-controlled remote host. Upon receipt of the request, it sends probe packets toward the opposite side. The other host is a server that is equipped with an Intel Xeon E5-2620v4 CPU and 16 GB of memory. The server runs a KVM hypervisor, which is enabled with KSM (Kernel Samepage Merging) feature.

There are two guest VMs on top of the hypervisor, one for an attacker and the other for a victim. The VM of the attacker runs our reconnaissance program in a 64-bit Ubuntu Linux 18.04 LTS. The program is implemented based on Mastik [27], a micro-architectural side-channel toolkit, to perform a Flush+Reload analysis. On the side of the victim, an instance of VyOS, a Linux-based versatile networking OS for virtualized network functions, with version 1.2.1 was run.

For our experiment, the victim was configured to operate as a firewall with packet filtering rules. We also built a private network, which consists of a few dummy hosts, directly connected to the firewall. All the probe packets sent from the remote host are destined for the private network through the firewall. Thus, the filtering rules of the victim are applied to those packets en route.

Table II
Memory and time for resolving target pages

Page	Function	Entropy (bit)	Complexity	Memory (KB)	$\mathcal{G}$	Time (min)
P1	ip_forward()†	9	2^9	2,048	1	51
P2	ip_forward_options()	9	2^9	2,048	1	50
P3	reject_tg()	39	$2^9 + 3 \cdot 2^{10}$	14,336	3	151
P4	log_tg()	29	$2^9 + 2 \cdot 2^{10}$	10,240	3	150
P5	nf_nat_ipv4_manip_pkt()	39	$2^9 + 4 \cdot 2^{10}$	18,442	5	251
Total			$2^9 + 8 \cdot 2^{10}$	34,816	5	251

† ip_forward_finish() is contained in P1 as well

B. Resolving target pages

Resolving target pages is a preliminary step to mount our cache-based reconnaissance method. In this subsection, we present an evaluation of the amount of resources (*i.e.,* memory and time) that are required for this step.

By a program analysis of iptables and the kernel of the victim, we identified a total of five 4 KB pages that have to be resolved. All those pages belong to either text segments of LKMs or the kernel text. Note that the page size is relevant to KSM, not to guest OSes. KSM performs deduplication over physical pages with regular size (*i.e.,* 4 KB).

The identified pages are listed in Table II. Each page contains at least one target functions. The amount of uncertainty (*i.e.,* entropy) to the page content is presented in the table as well. Of these pages, P1 and P2 belong to the kernel text. Therefore, entropy on those pages is solely dependent on KASLR of the kernel text. Conversely, P3, P4 and P5 belong to LKMs, thus have additional entropy caused by dependencies to external modules.

In the table, complexity represents the maximum number of guessed pages required for resolving target pages by using our divide-and-conquer method. Due to additional dependencies, the complexity of LKM pages (*i.e.,* P3, P4 and P5) is much larger than that of kernel text pages. Among all the LKM pages, P5 has the highest complexity as there are more modules in its dependency chain than the other pages (refer to Fig.4). The complexity determines the actual amount of required memory. As listed in the table, no more than 20 MB is necessary for resolving each page.

We also evaluated the amount of time required for our method. As described in Section IV-C, we utilize an indicator function IsDedup to identify the right page among candidates. The indicator is implemented by using the CoW mechanism of the hypervisor. Hence, to select the right page, it is

Table III
BACKGROUND NETWORK TRAFFIC

Class	μ_{pps}	σ_{pps} (5 seconds)	σ_{pps} (10 seconds)
T1	364	90	110
T2	3,257	985	1,094
T3	5,424	1,714	1,853

necessary to observe a pattern of write access latency on each candidate for a certain amount of time, until at least one peak is observed to confirm the selection. As shown in Fig.3, a deduplicated page shows a pattern of repetitive peaks in its write access time. The period of peaks is determined by configurations of KSM parameters such as pages_to_scan and sleep_millisecs. In the default configuration, the period is measured to be approximately 50 minutes in our experimental environment. That is, it requires approximately 50 min to execute ISDEDUP for each candidate page. For the optimization, we concurrently observe all the candidates at the same time. It allows us to dramatically reduce the total time.

The execution time of resolving target pages is presented in Table II. P1 and P2 required approximately 50 min to complete executing ISDEDUP for all 2^9 candidate pages in parallel. However, for the other pages, concurrent execution for all the candidates is infeasible as there are certain dependencies among them. We handled this by partitioning candidate pages into several groups so that there is no dependency among each group. (The number of groups, referred to as $\mathcal{G}$, is presented for each target page in the table.) Resolving is then performed sequentially for each group of candidate pages. Consequently, the total execution time is determined according to the number of groups.

Until now, we only considered a method for individually resolving each target page. However, it is also possible to resolve all the target pages simultaneously at the same time. As shown at the bottom of the table, simultaneous resolution requires approximately 34 MB of memory and five hours of execution time in total.

C. Classifying probe packets

We evaluated the performance of the proposed inference method described in Section IV-D. The experiment was conducted under a simulated background where network traffic was introduced to an interface of the victim. To simulate real-world network traffic, we need to collect real-time data of packet arrival rate (*i.e.,* pps) on network devices. Such kind of traffic data is accessible through SNMP. Thus, we gathered the data by repeatedly querying IF-MIB::ifInUcastPkts for ingress interfaces of the device every second.

In order to have available network devices that provide public access to SNMP, we searched them via Shodan[2]. As a result, we identified a total of three available devices on the Internet. They are all in operation under networks of different sizes ranging from SOHO to a university campus with a variety of traffic patterns on their interfaces.

Learning from the collected traffic data, we classified the traffic patterns into three classes, which are listed in Table III. In the table, μ_{pps} refers to the average of the distribution of pps, which represents the volume of the traffic. On the other hand, σ_{pps} refers to the standard deviation of the distribution for a certain period of time (*e.g.,* 5 and 10 seconds), which represents the fluctuation of the traffic.

The background network traffic was simulated considering the traffic patterns described above. It actually consists of dummy TCP packets with no payload, which are created by a packet generator. In the experiment, we set the default policy of the firewall so that all the packets from the background traffic are allowed. Note that such default-allow-rule results in a significantly more noisy environment while conducting the filtering rule inference because permitted packets cause more invocations of internal functions than denied packets.

As described in the previous section, Algorithm 1 has better performance when background traffic has lower fluctuation. According to Table III, the fluctuation (σ_{pps}) becomes smaller in the distribution with the shorter interval (*i.e.,* 5 seconds). Therefore, we set the parameters of the algorithm to $N = 5$ and $\Delta_T = 1$ so as to complete the execution of the algorithm within 5 ($= \mathcal{T}$) seconds.

With this setting, we first measured the correlation coefficients of target functions for each type of probe packet. In Algorithm 1, the coefficients are computed by a CORR function. This function computes the coefficient by internally using a Pearson correlation with a modification that negative values are always converted to zero. Fig.6 shows the experimental results of the measurement with varying values of $\mathcal{M}_{pps}$ under various classes of background traffic. A target function with high correlation indicates that it is highly likely to be invoked upon arrival of a probe packet. From each graph in Fig.6, we clearly observe that different types of probe packets have unique and distinctive combinations of highly correlated target functions. This ensures that the algorithm, especially a CLASSIFY function, can easily classify probe packets. Such distinguishability becomes more precise as the parameter $\mathcal{M}_{pps}$ increases. We see that the fluctuation of background traffic is another factor that affects the distinguishability. It is worth noting that for all types of allowed probe packets (*i.e.,* ACCEPT, LOG and NAT), the correlation of ip_forward_options() remains high regardless of the value of $\mathcal{M}_{pps}$ under any kind of background traffic. It proves the validity of the IP Options field as an appropriate marker of probe packets.

We also evaluated the accuracy of packet classification according to the parameter $\mathcal{M}_{pps}$ and the background network traffic. The accuracy was measured by counting the number of correct answers returned from Algorithm 1 for challenged probe packets. For the classification, we set $c = 0.4$ as the threshold of correlation coefficient for CLASSIFY to determine whether the corresponding function was invoked or not. Table IV presents the experimental results of the accuracy

[2]A search engine for network devices, https://www.shodan.io/

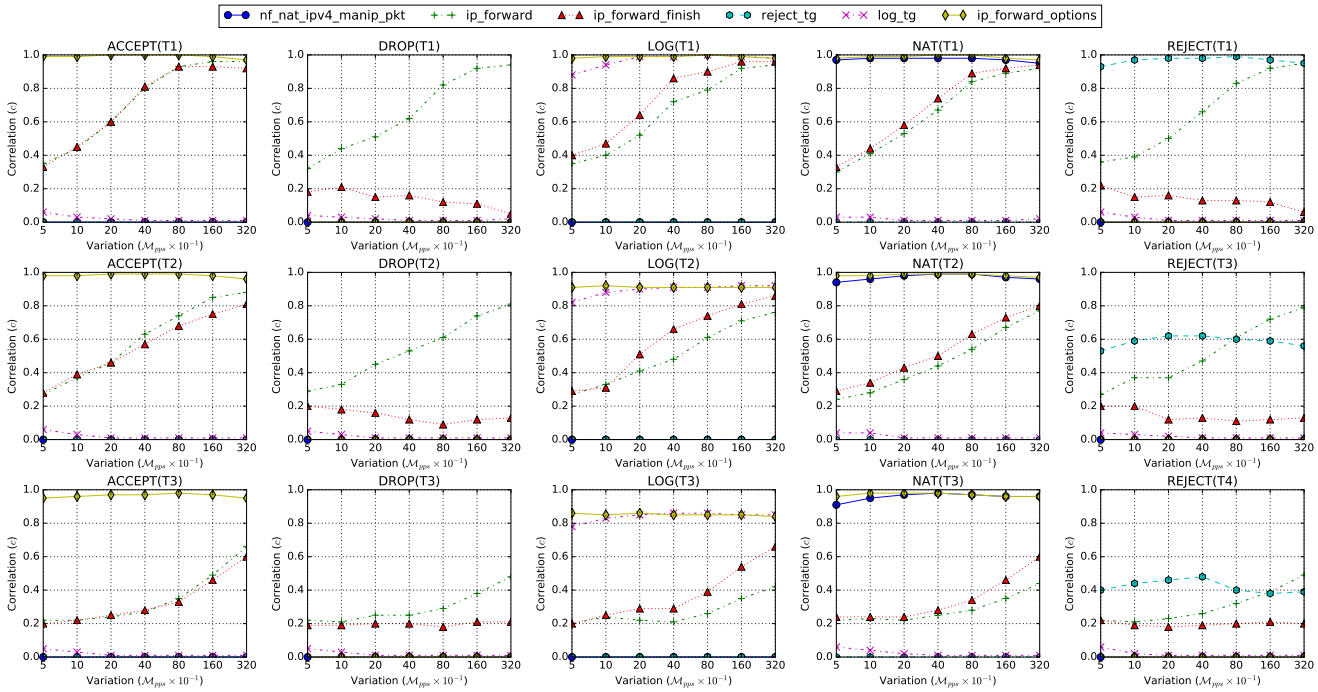

Figure 6. Measurement of correlation coefficients in Algorithm 1 ($N = 5$, $\Delta_T = 1$ seconds)

Table IV
ACCURACY OF PACKET CLASSIFICATION ($N = 5$, $\Delta_T = 1$ SECONDS)

$\mathcal{M}_{pps}$	Total # of probe packets (avg.)	Accuracy			Accuracy (w/ IP Options)		
		T1	T2	T3	T1	T2	T3
25	62	0.54	0.32	0.11	0.92	0.91	0.91
50	124	0.73	0.45	0.2	0.95	0.94	0.94
100	262	0.85	0.77	0.46	0.96	0.96	0.97
200	509	0.92	0.86	0.51	0.99	0.98	0.98
400	1,100	0.93	0.89	0.62	1.0	0.99	0.99
800	1,984	0.95	0.92	0.78	1.0	1.0	1.0
1,600	3,964	0.97	0.95	0.86	1.0	1.0	1.0

measurements. In the table, the average number of packets used for a single execution of the algorithm is also shown according to the parameter $\mathcal{M}_{pps}$. As shown in the table, the classification accuracy is determined from both the parameter and the background traffic. Under the same class of the traffic, more precise classification is achieved as the parameter is increased, which comes at the cost of increasing the total amount of probe packets.

Table IV lists another result of the measurement that uses probe packets with IP Options. In this case, the classification is limited to just two classes, allowed or denied, as described in the previous section. Owing to the uniqueness as well as the cache visibility, probing with IP Options packets achieves a classification accuracy of more than 90%. Furthermore, it only requires about dozens of packets regardless of the class of the background traffic.

VI. COUNTERMEASURE

The cache-based firewall reconnaissance is one of many possible threats of cache side-channel attacks to virtualized network functions. To achieve the robustness of the NFV, it is necessary to mitigate such security threats. In this section, we present possible countermeasures against cache-based attacks.

Resource isolation. The root cause that allows cache side-channel attacks is that the cache is shared between a co-located attacker and a victim. There are hardware-assisted solutions that enable strict cache isolation between VMs. For instance, Intel Cache Allocation Technology (CAT) provides a method to physically divide an L3 cache into distinct fragments. By using CAT, we can enforce each fragment to be exclusively allocated to VMs.

Page sharing is another cause of cache side-channel attacks, especially for the Flush+Reload attack. Since memory deduplication is the memory management technique provided by hypervisors (*e.g.,* Transparent Page Sharing in VMWare and KSM in KVM), it should be disabled by the hypervisor to prevent the attack.

Attack detection. Because of its nature, cache side-channel attacks result in a high amount of cache contention while the attack is in progress. Such contention can be exploited to devise a system that detects the attack in real time. There are already certain detection methods for cache side-channel attacks [28], [29]. These methods utilize a hardware performance counter, which provides a variety of CPU counters such as the number of cache hits/misses. With the aid of these counters, we can build anomaly-based intrusion detection

systems for cache side-channel attacks against virtualized network functions.

VII. Conclusion

In NFV, the benefit of virtualization technology comes at the cost of security risks caused by cache side-channel analysis. In this paper, we investigated the security impact of cache side-channel attacks on virtualized network functions. In particular, we proposed a novel firewall reconnaissance method based on the Flush+Reload cache side-channel attack against a virtualized Linux-based firewall along with major challenges and our solutions. Our method demonstrates significant advantages over the existing probe-and-response based techniques in various ways. Through experiments, we proved its effectiveness in inferring filtering rules in practical network environments. We also presented certain countermeasures that can mitigate our cache side-channel attacks on virtualized network functions.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2019-0-00533, Research on CPU vulnerability detection and validation), (No.2017-0-00184, Self-Learning Cyber Immune Technology Development).

References

[1] Y. Yarom and K. Falkner, "Flush + Reload : a High Resolution , Low Noise, L3 Cache Side-Channel Attack," in *Proceedings of the 23th USENIX Security Symposium*, 2014, pp. 719–732.

[2] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of 2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[3] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi, "NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 4, pp. 3330–3368, 2018.

[4] M. Daghmehchi Firoozjaei, J. P. Jeong, H. Ko, and H. Kim, "Security challenges with network functions virtualization," *Future Generation Computer Systems*, vol. 67, pp. 315–324, 2017.

[5] M. Q. Ali, E. Al-Shaer, and T. Samak, "Firewall policy reconnaissance: Techniques and analysis," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 2, pp. 296–308, 2014.

[6] H. Kim and H. Ju, "Efficient method for inferring a firewall policy," in *13th Asia-Pacific Network Operations and Management Symposium*, 2011, pp. 1–8.

[7] I. Schmitt and S. Schinzel, "WAFFle: fingerprinting filter rules of web application firewalls," in *The 6th USENIX conference on Offensive Technologies (WOOT)*, 2012.

[8] T. Samak, A. El-Atawy, E. Al-Shaer, and L. Hong, "Firewall policy reconstruction by active probing: An attacker's view," in *2006 2nd IEEE Workshop on Secure Network Protocols*, 2006, pp. 20–25.

[9] P. C. Lin, P. C. Li, and V. L. Nguyen, "Inferring OpenFlow rules by active probing in software-defined networks," in *19th International Conference on Advanced Communication Technology (ICACT)*, 2017, pp. 415–420.

[10] A. R. Khakpour, J. W. Hulst, Z. Ge, A. X. Liu, D. Pei, and J. Wang, "Firewall fingerprinting," in *The 31st Annual IEEE International Conference on Computer Communications (IEEE INFOCOM 2012)*, 2012, pp. 1728–1736.

[11] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[12] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack," *IACR Cryptology ePrint Archive, Report 2014/140*, 2014.

[13] Y. Shin, H. Chan Kim, D. Kwon, J. Hoon Jeong, and J. Hur, "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, 2018, pp. 131–145.

[14] D. Genkin, L. Valenta, and Y. Yarom, "May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017, pp. 845–858.

[15] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, cross-VM attack on AES," *Research in Attacks, Intrusions and Defenses (RAID), LNCS*, vol. 8688, pp. 299–319, 2014.

[16] B. Gulmezoglu, M. Inci, G. Irazoki, T. Eisenbarth, and B. Sunar, "Cross-VM Cache Attacks on AES," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 211–222, 2016.

[17] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS 2012)*, 2012, pp. 305–316.

[18] B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES 2016)*, 2016, pp. 368–388.

[19] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing - And its application to AES," in *Proceedings of 2015 IEEE Symposium on Security and Privacy*, 2015, pp. 591–604.

[20] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "PRIME+ABORT: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 51–67.

[21] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 897–912.

[22] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 549–564.

[23] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, 2014, pp. 990–1003.

[24] M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, R. Kay, and S. Mangard, "Hello from the Other Side : SSH over Robust Cache Covert Channels in the Cloud," in *2017 The Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 21–24.

[25] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, 2016.

[26] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," Tech. Rep., 2019. [Online]. Available: http://arxiv.org/abs/1905.05725

[27] Y. Yarom, "Mastik: A Micro-Architectural Side-Channel Toolkit," Tech. Rep., 2017. [Online]. Available: http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

[28] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using Hardware Performance Counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[29] M. Payer, "HexPADS: A platform to detect stealth attacks," *Engineering Secure Software and Systems (ESSoS 2016), LNCS*, vol. 9639, pp. 138–154, 2016.