SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe

Majid Sabbagh
Department of Electrical and Computer Engineering
Northeastern University
sabbagh.m@husky.neu.edu

Yunsi Fei
Department of Electrical and Computer Engineering
Northeastern University
yfei@ece.neu.edu

Thomas Wahl
College of Computer and Information Science
Northeastern University
wahl@ccs.neu.edu

A. Adam Ding
Department of Mathematics
Northeastern University
a.ding@neu.edu

ABSTRACT

Microarchitectural side-channel attacks have posed serious threats to many computing systems, ranging from embedded systems and mobile devices to desktop workstations and cloud servers. Such attacks exploit side-channel vulnerabilities stemming from fundamental microarchitectural performance features, including the most common caches, out-of-order execution (for the newly revealed Meltdown exploit), and speculative execution (for Spectre). Prior efforts have focused on identifying and assessing these security vulnerabilities, and designing and implementing countermeasures against them. However, the efforts aiming at detecting specific side-channel attacks tend to be narrowly focused, which can make them effective but also makes them obsolete very quickly. In this paper, we propose a new methodology for detecting microarchitectural side-channel attacks that has the potential for a wide scope of applicability, as we demonstrate using a case study involving the Prime+Probe attack family. Instead of looking at the side-effects of side-channel attacks on microarchitectural elements such as hardware performance counters, we target the high-level semantics and invariant patterns of these attacks. We have applied our method to different Prime+Probe attack variants on the instruction cache, data cache, and last-level cache, as well as several benign programs as benchmarks. The method can detect all of the Prime+Probe attack variants with a true positive rate of 100% and an average false positive rate of 7.4%.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**;

KEYWORDS

Side-channel analysis, Prime+Probe, PIN tool address analysis

ACM Reference Format:
Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A. Adam Ding. 2018. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe. In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3240765.3240844

1 INTRODUCTION

With the advent of Internet of Things (IoT) and ever increasing demand for connected systems, many more accessible and yet connected points of intrusion could be exposed to cyber criminals for them to launch catastrophic attacks [30][4][19]. These attacks may leak secret system information, breach users' privacy and data confidentiality, disrupt digital transactions, or even cause physical damage to vital equipment and infrastructures. Side-channel attacks (SCAs) have become a realistic threat to many computing systems, where observable side-channel leakage is utilized to infer secret information. Typical side-channel leakage includes timing information, power consumption, electromagnetic emanations, or other physical side-effects [36].

Microarchitectural cache timing attack has become an effective and realistic cyber threat [13][23][3]. The goal of this attack is to infer some information about the victim's sensitive data or code blocks by monitoring the cache state. This attack is possible because the attacker program and the victim program, although in different address spaces and disjoint in memory, share the common on-chip caches at different levels of the cache hierarchy. In fact, the attacker observes cache hits and misses sourced from the victim program, and either finds the set index, as in Prime+Probe, or the exact cache line, as in Flush+Reload, that is accessed by the victim. With such information and knowledge of the victim's program implementation, the attacker can reveal the victim's secret data or operation.

Several methods have been proposed in the literature to detect cache timing attacks. There are mainly two categories, static code analysis and run-time detection. The existing static code analysis methods e.g., a tool called MASCAT, focuses on detecting different instruction groups or attributes that are typically used in cache timing attacks, such as timers, memory barriers, and memory moves, in disassembled program binaries. However, attackers can use different obfuscation techniques and packing schemes to avoid being detected by such static code analysis. The tool also has high false positive rates because there are many benign applications that use the same instructions or attributes that MASCAT relies on. Zhang et. al. in [34] propose another type of cache timing attack detection method: monitoring the hardware performance counters at run-time. It first detects the running of cryptographic applications, and then captures the cache miss rate to identify abnormal behavior and interactions of programs over the cache. This type of approaches [8][11][5][9][26] that use hardware performance counter monitoring or kernel tracing have the advantage of performing runtime or in the best case real-time detection. However, the main challenge that they are facing is that the patterns captured by the very low-level hardware performance counters or kernel probes could be attributed to a mix of programs running in parallel, leading to high false positive rates. Another challenge is that the detection tool (including monitoring or tracing, and analyzing) has to be fast enough to keep up with the execution of the victim program, so as to detect the attack in time.

In this paper, we propose a new methodology and implement it in a tool, SCADET–Side-Channel Attack DEtection Tool. We used Prime+Probe as our case study, while our approach can be extended to detect other cache timing attacks. SCADET accepts the executable binary of a user program as input and identifies any potential for Prime+Probe attack behavior. SCADET consists of two stages, dynamic binary instrumentation and offline pattern detection. SCADET conducts the instrumentation in a controlled (virtual or non-virtual) environment and collects the address traces of the input program. In the current version of SCADET the analysis is performed offline to have higher flexibility in the analysis and to avoid extra overhead on the instrumentation and the host system. Currently our tool only supports x86 architectures, although it can be extended to support different architectures, such as ARM for embedded systems. We view our tool as a side-channel "anti-virus" tool, advocated by Spreitzer et. al. in [29], which performs deep scanning on program binaries in the APP stores to identify threats. Our contributions in this research are summarized as follows:

- Proposed a methodology to detect microarchitectural SCAs based on their memory-access behavior
- Analyzed Prime+Probe attacks from the algorithmic level down to the microarchitectural level for identifying its invariant pattern
- Developed a tool to instrument program binaries and analyze their address traces
- Evaluated the accuracy and the performance of the tool for different types of attack and benign programs

SCADET could successfully detect all of the studied Prime+Probe attack variants with a 100% true positive rate and an average false positive rate of 7.4%.

The rest of the paper is organized as follows. In Section 2, we describe Prime+Probe attack, its implementations, and its primary signature. In Section 3, we elaborate on our detection method and the algorithm we designed for detecting the Prime+Probe signature. We talk about the SCADET components and their tasks in Section 4. In Section 5, we explain our experimental setup and provide the results of our experiments. We discuss the performance, the accuracy, and the limitations of SCADET and some of our future plans to address them in Section 6. Finally, we conclude in Section 7.

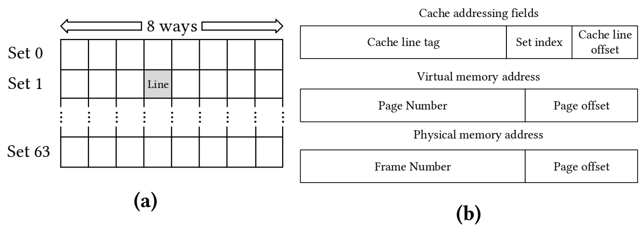

Figure 1: An 8-way L1 cache organization (a), and virtual and physical memory address fields (b).

2 PRIME+PROBE ATTACK ON CACHES

Prime+Probe [25] is one of the most effective cache timing attacks. It can detect which cache sets are accessed by the victim and infer some secret information. Another cache timing attack, Flush+Reload [33], can be more accurate than Prime+Probe because it detects the accesses to specified cache *lines*. However, Prime+Probe is more generic as it does not need to use flush instructions, and it also works on non-shared memory environments. To better explain the Prime+Probe attack, in Section 2.1 we provide some background on the cache organization and memory addressing in modern processors.

2.1 CPU caches and memory addressing

Caches, a hierarchy of small but fast on-chip storage units, play an important role in improving computer performance. However, the access time difference between cache hit and cache miss can leak memory addresses being used by the victim and therefore cause secret leakage.

2.1.1 Cache organization. In a typical cache hierarchy, there are three levels. L1 and L2 caches are private to each core and L3 cache is shared among several cores. Modern caches are typically set-associative, where the cache is organized into multiple cache *sets*, with each set containing multiple cache lines. Each memory address can be mapped onto one set but any line within that. In a set-associative cache, the associativity therefore is denoted by *#ways*. Figure 1a, exemplifies an 8-way set-associative cache, with 8 cache lines in each cache set.

Cache replacement policies determine which cache line in a set should be evicted when a new memory block is brought into the cache. Commonly used policies are the Least recently used (LRU), Pseudo-LRU, and adaptive replacement policy [28].

2.1.2 Memory addressing. Virtual addressing enables processes to view the memory as if they own it all. The operating system allocates a part of memory to each process, and performs the virtual to physical address translation. Figure 1b shows different fields of virtual and physical addresses. Note that, the page offset stays the same in both the virtual and the physical addresses, while the page number get translated to the frame number using page tables. When the CPU wants to read or write a data or an instruction from/to caches, it uses their virtual addresses. However, depending on whether the caches are virtually or physically indexed or tagged, an address translation might happen. Figure 1b shows the address breakdown, where a memory address consists of set index (which cache set the line is mapped onto), the cache line tag (to search the cache line in the set), and cache line offset (which word in a cache line). The number of bits in these fields depends on the cache size, number of sets in the cache, and the cache line size.

2.2 Prime+Probe implementations

Prime+Probe attack can target different levels and types of caches in modern x86 architectures, specifically, L1 data (D) [27], L1 instruction (I) [1], and last-level caches (LLC) [23]. Depending on the level and the type of caches, attacker would have to change the attack procedure although the core behavior may remain the same.

In D-cache attacks, the attacker wants to extract secret information from the victim's data-access patterns in L1 or Last-level caches. In these types of Prime+Probe attacks, an attacker monitors the footprint of a victim in certain cache sets by replacing the victim's data with its own data and evaluating the (re)access time to this data after victim runs its sensitive code. These types of attacks can be either launched on one core or across multiple cores.

In [1] Aciiçmez introduces an I-cache Prime+Probe attack, where the goal is to find the execution path or instruction flow of a victim program, such as RSA [3] or ElGamal encryption [35], which executes different branches depending on a secret value. Similarly, the attacker attempts to create conflicts on certain cache sets that both the attacker code block and the sensitive victim code block map onto. I-cache attacks have a broad range of targets, including not only conditional branches which Branch Prediction Analysis (BPA) [2][16][24] applies to, but also unconditional branches. In I-cache attacks, the spy process is launched either quasi-parallel (by time-slicing and frequent pauses/interrupts in victim operation) or simultaneously (using simultaneous multi-threading) with the victim process on the same core.

In both I-cache and D-cache attacks, cache sets can be primed and probed in several ways. For example, all the sets can be primed at once and then probed one by one, or prime and probe is done with a certain period across a group of sets. Besides that, cache mapping method as well as cache replacement policy are other factors that should be considered in these attacks. In fact, the strategy that the attacker chooses can affect the success or failure of an attack. We evaluated the detection accuracy of SCADET for some of the aforementioned variants of conducting Prime+Probe attacks. SCADET achieved similar accuracy in detecting these attack variants and the default approaches of conducting the attacks.

2.3 Primary signature (invariant pattern) of Prime+Probe attacks

The core behavior of the Prime+Probe attack can be described as coupled two groups of cache accesses in a *short interval of time*, where certain cache sets are first primed completely (i.e., mounting the attacker data in the cache) and after a delay (to let the victim run) a few or all of the primed cache locations are reaccessed and the access time is measured. To prime a cache set completely the attacker should at least perform *#ways* accesses (number of ways in each cache set) each having a unique cache line address for the same set index. If the cache replacement policy is LRU then *#ways* cache accesses is enough. For other policies such as pseudo-LRU or adaptive replacement policy, however, the attacker may need to perform more than *#ways* accesses as some of the access requests may go to the same cache way. The time between consecutive

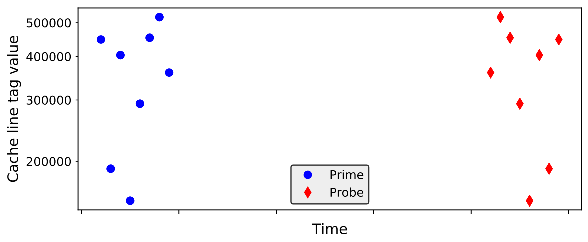

Figure 2: Access pattern of Prime+Probe in one set of an 8-way L1 data cache.

accesses to the same set is identified by an empirically chosen threshold (which also depends on the attack implementation). We call this threshold the *intra-group* threshold. On the other hand, the distance between Prime and Probe access groups (where each group consists of accesses to the same set) is another important factor in attacks which is compared against a predetermined threshold that we name as the *inter-group* threshold. These thresholds enable our tool to correctly identify the groups of accesses. The way that proper *intra-group* and *inter-group* thresholds are chosen is detailed in Section 3.1.1.

We define the primary signature of Prime+Probe attacks as at least two consecutive groups of accesses to at least one cache set, which pass both intra-group and inter-group threshold conditions. Figure 2 illustrates an example of Prime+Probe signature for an L1 data cache attack based on the Mastik library [32].

3 DETECTION METHOD

We propose an algorithm for detecting the Prime+Probe signature in the target program's executable binary by analyzing the footprint (addresses) of its instruction or data access. After instrumenting the binary and capturing the program's address trace, we follow these steps:

(1) Add sequence number: a sequence number is added to each address in the trace as its order in the program which can save memory space than real time-stamps. Also, it makes the trace acquisition faster.
(2) Filter reads/writes: based on the type of analysis, either the read accesses or write accesses are kept (since in the real Prime+Probe attack priming and probing are done by a group of reads or a group of writes and not a mix of both). An exception is I-cache analysis, which consists of reads only.
(3) Calculate the cache set indexes and cache line tags: for each physical address in the trace, we calculate the set indexes and line tags based on the cache configuration.
(4) Group by set indexes: all of the line tags and their corresponding sequence numbers are grouped by the set indexes which they belong to.
(5) Filter the sets: any sets which have fewer than *#ways* unique line accesses are removed.
(6) Cluster the line tags based on the Prime+Probe pattern: for each set, the line tags and their corresponding sequence numbers are organized into different clusters based on a clustering algorithm which is explained in Section 3.1.
(7) Filter the clusters: any clusters which have fewer than *#ways* unique line tags are removed. This is because, we assume

there are at least *#ways* unique line tags for both the prime and probe access groups.

(8) Evaluate the Prime+Probe pattern: if at least two clusters remain, we report this binary as containing a potential attack code with at least one instance of the Prime+Probe signature.

3.1 Clustering the line tags

To organize the line tags and sequence numbers into different clusters we devised a custom light-weight clustering algorithm. Before describing the algorithm, we define the *intra-group* and *inter-group* thresholds.

3.1.1 Intra-group and inter-group thresholds. In Section 2.3, we described the Prime+Probe signature as grouped accesses which exhibit certain properties. First, all of the accesses in a group should go to a single set and they should be in temporal proximity. In other words the temporal distance of two consecutive accesses should be less than or equal to a threshold, the *intra-group* threshold. Second, since in practical attacks the prime and probe access groups should not be far apart, otherwise there may be a large amount of cache access noise by other irrelevant programs (in addition to the victim program), we define another threshold called *inter-group* threshold as the upper bound of this distance. These thresholds are measured by number of accesses (or memory references) as the indicator of temporal distance. For example, an intra-group threshold of 10 for an I-cache attack means that the consecutive targeted accesses (those that are intended to prime or probe a cache set) to the same L1 I-cache set should have sequence numbers differing by at most 10 (i.e., there are at most 9 *other* accesses in between).

Both the intra-group and inter-group thresholds are chosen empirically after evaluating many attack and benign programs. However, to better select the thresholds we also evaluated the source codes of the attacks. For example, to choose a proper intra-group threshold for I-cache attacks we recorded the assembly instructions that are executed by the I-cache attack in the Mastik library [32]. Listing 1 shows a code block that is used by the I-cache attack to prime an 8-way L1 instruction cache. This code block is the body of a function which contains a chain of eight continuous `jmp` instructions and a `ret` instruction that, when called, primes all of the cache lines of the target set (assuming an LRU cache or pseudo-LRU replacement policy). Therefore, we expect the intra-group threshold to be 1 (that is completely back to back accesses). As we show in Section 5 our experiments confirm that 1 is an excellent value for the intra-group threshold in I-cache attacks. Similarly, the best intra-group threshold value for L1 D-cache attacks is 1.

Listing 1: Code snippet from the Mastik I-cache attack.

```
jmp 0x7f2ee3a1c9c0
jmp 0x7f2ee3a1d9c0
jmp 0x7f2ee3a1e9c0
jmp 0x7f2ee3a1f9c0
jmp 0x7f2ee3a209c0
jmp 0x7f2ee3a219c0
jmp 0x7f2ee3a229c0
ret
```

In contrast to L1 cache attacks, based on our experiments the intra-set threshold for LLC attack is about 24 which is significantly larger than 1. We analyzed the reason to be the replacement policy of the LLC as well as the speculative and out-of-order execution in

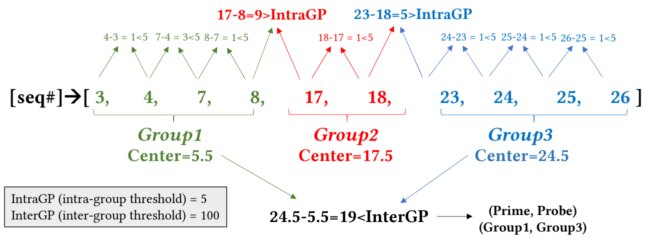

Figure 3: Cache line tag clustering

modern processors. Our analyses are done on a Skylake microarchitecture CPU which we expect to have an adaptive replacement policy for its LLC [14]. To fully evict a set, there need to be more than *#ways* accesses to the same set because some of the accesses may touch the same way in the set. Therefore the attack code has to loop over the eviction procedure multiple times to evict all of the ways in a cache set (e.g. about 9 times for each way in a 16-way cache in libflush L3 Prime+Probe attack [20][21] assuming a uniform distribution for accesses). It means that we have some redundant memory accesses pertaining to the for loop execution between the target accesses. Moreover, in modern processors instructions are executed speculatively and out-of-order, which means different memory accesses are issued between consecutive targeted accesses. In view of these reasons, a larger threshold has to be selected to capture the targeted accesses.

On the other hand, inter-group threshold also affects the performance of the detection method. This threshold is chosen empirically based on the relative distance of Prime and Probe access group *centers*. The value for this threshold could be as large as 10000 for LLC analysis.

3.1.2 Clustering algorithm. Figure 3 illustrates an example of line tag clustering for a 4-way set. To cluster the addresses of accesses to each cache set based on the sequence numbers we first find the temporal distance (difference of sequence numbers) of all consecutive accesses. Then, we group the accesses that their distance are less than or equal to the intra-group threshold. If a distance is larger than the intra-group threshold, a new group of accesses is created. This procedure repeats until all of the line tags are clustered into different groups. The groups that have less than *#ways* unique line tags are removed. Finally, if the differences between the centers (averages of the sequence numbers) of the remaining pairs of groups are less than or equal to the inter-group threshold, these pairs are returned as the Prime+Probe access groups. This algorithm is run for each candidate cache set that is selected by step 5 of detection steps in Section 3.

4 SCADET COMPONENTS

We designed SCADET, a tool which automates the acquisition of address traces from executable binaries as well as the analysis for detecting attack signatures. Figure 4 shows a high-level diagram of SCADET and its components. This tool has two primary parts. First part, as detailed in Section 4.1, is dedicated to trace acquisition through binary instrumentation. The output trace of the first part, is passed to the second part for address analysis and signature detection stage.

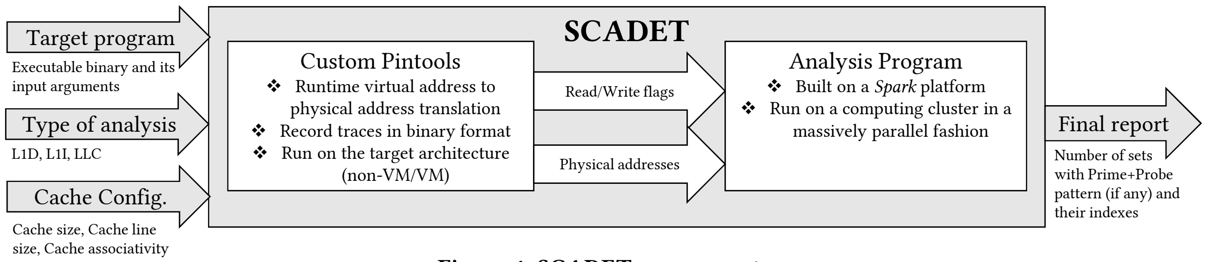
Target program
Executable binary and its input arguments
Type of analysis
L1D, L1I, LLC
Cache Config.
Cache size, Cache line size, Cache associativity
SCADET
Custom Pintools
❖ Runtime virtual address to physical address translation
❖ Record traces in binary format
❖ Run on the target architecture (non-VM/VM)
Read/Write flags
Physical addresses
Analysis Program
❖ Built on a Spark platform
❖ Run on a computing cluster in a massively parallel fashion
Final report
Number of sets with Prime+Probe pattern (if any) and their indexes

Figure 4: SCADET components.

4.1 Trace acquisition

We use Pin, which is a dynamic binary instrumentation framework developed by Intel [17], to acquire memory and instruction address traces of programs. Pin supports IA-32, x86-64 and MIC instruction-set architectures and enables the creation of custom program analysis tools. In particular, we designed two customs tools based on the `itrace` and `pinatrace` Pintools to instrument the instruction addresses, for I-cache analysis, and the memory addresses, for D-cache analysis, respectively. We run and instrument the programs on a Linux operating system.

There are two main differences between our custom-designed tools and the original tools. First, in order to produce a compact trace our tool only keeps essential information and saves the traces in binary format. Second, our tool translates the virtual addresses to physical addresses using the `pagemap` kernel interface. Our Pintool accesses the pagemap file located at `/proc/[pid]/pagemap` to find out which physical frame each virtual page is mapped to, and conduct the translation using a userspace program similar to the one presented in [10].

A major challenge in monitoring memory traces is the sheer volume of data that needs to be collected, maintained, and analyzed [31]. To mitigate this problem, our Pintool encodes the information in a compact format and stores them in a binary file. Besides that, we put an upper limit of 10GB for the size of output traces, which was more than enough for the majority of the programs we evaluated. However, it could be that our instrumentation misses some important parts of a program. This issue can be addressed in future work by a more targeted instrumentation (potentially through pre-static analysis or machine learning based targeted instrumentation). In our current prototype, the upper limit is enforced manually. Another important point is the time it takes to acquire the traces. In our experiments the minimum and maximum acquisition times were 0.38 minutes and 38.71 minutes respectively. Targeted instrumentation also helps reducing the acquisition time. Note that the traces used for D-cache and LLC analyses are identical, because in both types the physical memory addresses are analyzed. The only difference is in finding the set indexes and line tags for the right level of caches. Also, in I-cache traces all recorded addresses are for read accesses as the program only reads the instructions from I-cache during execution; it does not write to it. Figure 5 shows trace structures for different types of analyses.

4.2 Address analysis

After collecting the address traces, we send them to a computing cluster for analysis and pattern detection. We developed a parallel analysis program based on the Apache Spark data processing engine [12]. We used `pyspark` to develop our program and ran it on a cluster of 1 master node with 5 worker nodes connected to it. Each of the nodes had 48 computing cores. This program executes the steps described in Section 3 using a parallel processing approach and outputs the final report in a text format similar to Figure 6. In our experiments, minimum and maximum I-cache analysis time was 1.8 and 312 minutes, D-cache analysis was 0.5 and 296 minutes, and LLC analysis was 0.5 and 84 minutes.

```
17143690309
16294010779
16294010783
16294010786
16294010788
...
```

(a)

```
0,10287225266
1,14661488704
1,6241984576
1,13222883392
1,5051146304
...
```

(b)

Figure 5: SCADET raw address traces (converted to human readable formats). (a) I-cache trace, physical instruction addresses. (b) D-cache or LLC trace, read/1 and write/0 flags and physical memory addresses.

```
READ mode, LLC
Cache parameters: (assoc, intra_group_thrd, line_bits,
set_mask, set_bits, inter_group_thrd) = (16, 24, 6, 2047,
10, 10000)
detected set idx and (#lines, cluster_center): [(17,
[(31, 284691), (31, 293359)], (18, [(31, 465791), (31,
473912)]))
Set idxs: [17, 18]
#Sets: 2
prime+probe pattern is detected!
```

Figure 6: A sample output report of SCADET.

5 RESULTS

In this section, we describe our experiments and present their results. First, we elaborate on the experimental setup. Second, we explain how we evaluated our tool and the metrics we used. Third, we list the benchmarks we used to study the performance of our tool. Finally, we describe our results for each experiment.

5.1 Experimental setup

We performed all of the binary instrumentation and trace acquisition on an Ubuntu 16.04.1 operating system with latest updates and Linux kernel version of 4.13.0-38-generic. As a side study, we also ran the acquisition process on an Oracle VirtualBox virtual machine

(VM), with Ubuntu 16.04 as the operating system, achieving similar results to the non-VM settings. In both VM and non-VM settings, the host desktop computer had Intel Skylake core-i7 6700 CPU, 16GB RAM, 16-way 8MB L3 cache, 8-way 32KB L1 I-cache, and 8-way 32KB D-cache. Note that all of our trace acquisitions are performed when regular background and foreground processes, such as OS scheduling and user applications (browsers, editors, network applications etc.), were running at the same time. This adds real noise to our traces. The computing cluster which we performed the address analysis and pattern detection on had 1 master node and 5 worker nodes connected to it. Each of the 6 nodes had Intel Xeon CPU E5-2690 v3 2.6GHz, 48 logical cores, and 128GB RAM. We launched our analyses on a Apache Spark platform version 1.4.1 (Hadoop 2.4) with 50GB driver memory, 50GB executor memory, and 20GB maximum result size.

5.2 SCADET results evaluation

We run SCADET on a variety of benign and attack programs. All of the attack programs are based on the open-source libraries from experts in SCA research in order to conduct a fair evaluation and allow reproducibility and comparability. We also used publicly available benign programs.

5.2.1 Benchmarks. In total we used 21 benchmark programs, listed in Table 1, for evaluating SCADET. We used two open-source libraries, libflush [20] and Mastik [32], for implementing the attacks. L1 I-cache and L1 D-cache attacks are implemented based on the Mastik library. We evaluated two LLC attacks, one based on libflush and the other from Mastik. For benign applications we used, three types of sorting, insertion sort, merge sort, and quick sort from an open-source repository [6], ten programs from the PARSEC 3.0 benchmarking suit [7], a compression program, Bzip2 which is available on Linux as a tool, and three OpenSSL based cryptographic operations – an AES encryption, a document signing, and an SSH key generation. We chose these benchmarks as they represent a diverse class of programs with different workloads, parallelism, and memory access intensity. Note that to successfully run and accurately evaluate all of the benchmarks we provided necessary inputs for them. For Prime+Probe attack binaries, we set them up for an attack on 16 sets for libflush L3 Prime+Probe and on 64 sets for Mastik L1 instruction, L1 data, and L3 attacks with 100 iterations for each attack. For PARSEC programs we used `-i simlarge` flag to run them with a large simulation input data. For Bzip2 we used `libc-static.a` as the input file of compression with default options. For AES encryption we used 256-bit CBC mode options, for document signing we used a 256-bit SHA hash operation and a 2048-bit RSA encryption, and for SSH key generation we used 521 bit ECDSA.

5.2.2 Analysis results. To measure the accuracy of the tool, we used *true positive rate (TPR)* and *false positive rate (FPR)* metrics. We can tell from TPR and FPR how effective the tool is in detecting the Prime+Probe pattern in attack programs and not in benign programs which may exhibit similar patterns. We studied the detection accuracy of SCADET by running 105 experiments, corresponding to five types of analyses (I-cache, D-cache reads, D-cache writes, LLC reads, and LLC writes) for each of the benchmarks. The SCADET detection result are summarized in Table 1. The results show that, SCADET had 4 true positives, 95 true negatives, 6 false positives (3 for D-cache analysis and 3 for I-cache analysis), and 0 false negatives, achieving a TPR of 100% and an average FPR of 7.4% (FPR is 15% for I-cache analysis, 7.3% for D-cache analysis, and 0% for LLC analysis). This result corresponds to intra-group thresholds of 1 for I-cache and D-cache analysis and 24 for LLC analyses, and inter-group threshold of 3500 for L1 cache analysis and 10000 for LLC cache analysis. Note that, we reported an attack as detected if at least one set with the Prime+Probe pattern is identified in its address trace. However, if we consider the number of identified sets over the number of expected sets (those that are targeted by the attack) the average TPR is about 73.9%.

As shown in Table 1, for the libflush LLC, Mastik LLC, L1-D, and L1-I attacks, SCADET correctly detects 15 LLC, 1 LLC, 64 L1-D, and 64 L1-I cache sets with Prime+Probe patterns respectively (marked in red). Also, SCADET produces 7 false positives (marked in violet) for two types of analyses, L1-I and L1-D, in seven benchmark programs.

l3pp libflush (**15/16 LLC**)	l3pp mastik (**1/48 LLC**)	l1dpp mastik (**64/64 L1-D**)
l1ipp mastik (**64/64 L1-I**)	blackscholes (***1 L1-D***)	canneal (***1 L1-D***)
ferret (***1 L1-I***)	fluidanimate (***1 L1-I***)	raytrace (***1 L1-I***)
vips (***1 L1-D***)	insertion sort	quick sort
merge sort	freqmine	bodytrack
facesim	x264	bzip2
aes256cbc	rsa2048sha256	sshecdsa521

Table 1: Evaluated benchmarks and the numbers of sets with the Prime+Probe pattern (identified/expected) in each program (red for true positives and violet for *false positives*) as detected by SCADET

5.2.3 Trace acquisition times, trace sizes, and analysis times. We considered *trace acquisition time* and *trace size* as metrics for the trace acquisition part of the tool. For instruction address traces (I-cache analysis), the acquisition time had a mean of 22 minutes and for memory address traces (D-cache and LLC analyses), the acquisition time had a mean of 11 minutes. For I-cache analysis, the trace size had a mean of 2.7 GB; and for D-cache and LLC analysis, the trace size had a mean of 2.4 GB. I-cache analysis time had a mean of 22 minutes, D-cache analysis 24 minutes, and LLC analysis 5 minutes.

6 DISCUSSION

SCADET performs L1 attack analysis as well as LLC attack analysis. In both types of analysis, intra-group and inter-group thresholds have a large impact on the accuracy of the detection outcome. The intra-group threshold filters out unwanted patterns in each set which may be similar to Prime+Probe signature in the sense that they are frequently accessing the same set, but are in fact from benign programs as they do not satisfy the temporal proximity constraint of Prime+Probe accesses. Therefore this threshold controls which access patterns can be categorized as priming or probing.

Choosing a very large intra-group threshold may lead to false positive results (false alarms) and choosing a too small value may lead to false negative results. On the other hand, the inter-group threshold captures a more high-level behavior, which is the Prime+Probe period. There could be two groups of access patterns which are similar to prime and probe patterns individually, however they are too far from each other that they cannot be considered as *coupled* entities. It means that the information from one of the access groups could not reach to the second one, as there would be a complete disruption by other programs that may exceed the span of a sensitive code in the victim program (the accesses of the victim are mixed or obstructed by other active code blocks or other programs).

6.1 SCADET accuracy evaluation

With the best settings, SCADET achieves a TPR of 100%, and an average FPR of 7.4%. The false alarms produced by SCADET were triggered by I-cache analysis of the benchmarks `ferret`, `fluidanimate`, and `raytrace`, D-cache writes analysis of the benchmarks `blackscholes`, `canneal`, and `vips`, and also D-cache reads analysis of the `l3pp mastik` program (that is also correctly detected as an LLC attack). For I-cache analysis, the false positives are because of code blocks which have many short distance jumps. For false positives in D-cache write analysis, some programs initialize large data structures in memory which generates many write memory accesses in a short interval of time, creating a pattern very similar to the Prime+Probe signature in write accesses. The false positive in D-cache read analysis of Mastik LLC Prime+Probe attack could be because of the eviction set preparation steps in the attack program which have frequent memory accesses to the memory to find the congruent addresses and create the eviction set.

Based on our analyses and experiments the intra-group threshold plays a more important role in detecting the attacks accurately, compared to the inter-group threshold. This is both predictable and desirable because the intra-group threshold is capturing the core pattern of the Prime or Probe, while the inter-group threshold is a knob for reducing the false positive and it depends on how the attack is implemented.

SCADET has better accuracy in LLC analysis as it produces no false positives, compared to I-cache and D-cache analysis for which it produces a total of 6 false positives. This is reasonable, because LLC have a much larger size and is not accessed as frequently as L1 caches, therefore the Prime+Probe signature stands out more clearly among other memory accesses.

6.2 SCADET trace size vs. analysis time

Figure 7 illustrates, on logarithmic scales, the analysis time variations for different types of analyses for all of the benchmark programs as the trace size (denoted as the *number of addresses* stored in the trace) increases. About 90% of the analysis times are spent on the file I/O as well as data movement to/from the memory and between the compute nodes in the cluster. For each of three types of analyses, the analysis time generally increases when the trace size increases. However, there are notable irregularities when the trace size exceeds 10^8. We labeled on Figure 7 a few benchmarks which pertain to these irregularities. For example for I-cache analysis (dashed lined plot with red circles), although the trace size of `l3pp mastik` is much smaller than PARSEC benchmarks' trace sizes, it takes more analysis time. Similarly for the D-cache analysis and the LLC analysis, the `bzip2` program has the largest traces size, while much longer analysis times are used by the `l3pp mastik` program and the `insertion sort` program respectively.

These irregular trends are due to the fact that the analysis times significantly depend on the address filtering. As described in Section 3, steps 1-5, SCADET performs a few levels of filtering to remove the noise in address traces before clustering the accesses and searching for the Prime+Probe pattern. Some benchmarks, such as `insertion sort` and `l3pp mastik`, have very memory intensive computations which involves many accesses to the same cache sets, hence they have many addresses to be analyzed even after filtering. For example, although `bzip2` is a memory intensive benchmark, about 37% of its memory accesses gets filtered out before the pattern detection step.

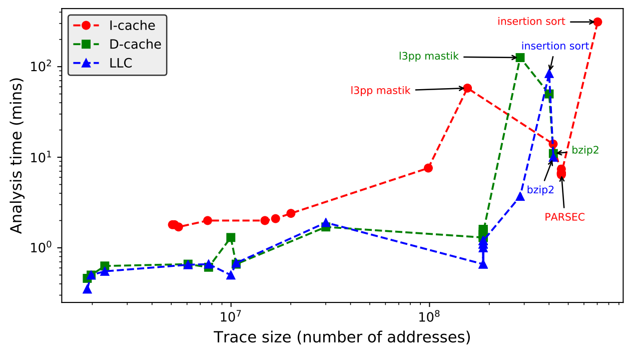

Figure 7: Analysis time vs. trace size for different types of analysis.

6.3 Limitations and future work

In some types of analyses, SCADET generates false alarms for a few of the benchmark programs. To fix that we are considering using a more adaptive pattern detection methodology which tunes the threshold based on each program to detect the patterns more accurately. Besides that, for a number of benchmarks, SCADET acquisition and detection times are very long. This could be a problem when many programs should be analyzed in large repositories or APP stores. Selective instrumentation is another major step for us to reduce the acquisition time and the analysis time as well as to reduce the noise and enhance the detection capabilities of SCADET. Furthermore, we think SCADET can be integrated into the OS kernel as a run-time address tracking and pattern detection framework. Currently, SCADET only supports the x86-64 ISA. We plan to extend our tool to support different ISAs and architectures.

Furthermore, we are planning to upgrade our tool to detect other types of microarchitectural SCAs. Attacks which exploit set-associative caches, such as Evict+Time [25] and Evict+Reload [15] could be detected by our tool. Also, SCADET can be extended to detect the covert cache transmission channels used by the recent Meltdown [22] and Spectre [18] attacks.

7 CONCLUSION

In this paper, we proposed a methodology to detect microarchitectural SCAs by inspecting their root cause and detecting their primary signature. We developed SCADET, a tool which collects address traces of programs by dynamic binary instrumentation and performs offline analysis using a parallel processing framework to detect the Prime+Probe signature as our case study. SCADET can detect different variants of the Prime+Probe attack with 100% TPR and 7.4% average FPR. We view SCADET as a side-channel "anti-virus", which can be deployed in APP stores and enterprise servers as well as local machines.

8 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants SaTC-1563697 and MRI-1337854.

REFERENCES

[1] O. Aciiçmez. 2007. Yet Another MicroArchitectural Attack:: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture (CSAW '07)*. ACM, New York, NY, USA, 11–18.

[2] O. Aciiçmez, Ç. K. Koç, and J.P. Seifert. 2007. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security (ASIACCS '07)*. ACM, New York, NY, USA, 312–320.

[3] O. Acıiçmez and W. Schindler. 2008. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *Topics in Cryptology – CT-RSA 2008*, Tal Malkin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–273.

[4] Akamai. 2018. Internet of Things and the Rise of 300 Gbps DDoS Attacks. (2018). https://www.akamai.com/us/en/multimedia/documents/social/q4-state-of-the-internet-security-spotlight-iot-rise-of-300-gbp-ddos-attacks.pdf "[Online; accessed 22-April-2018]".

[5] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya. 2017. Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks. *IACR Cryptology ePrint Archive* 2017 (2017), 564.

[6] B. Karakan. 2017. Benchmarking Sorting Algorithms. (2017). https://github.com/karakanb/sorting-benchmark "[Online; accessed 13-April-2018]".

[7] C. Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[8] M. Chiappetta, E. Savas, and C. Yilmaz. 2016. Real Time Detection of Cache-based Side-channel Attacks Using Hardware Performance Counters. *Appl. Soft Comput.* 49, C (Dec. 2016), 1162–1174.

[9] D. Fiser and W. G. Sanchez. 2018. Detecting Attacks that Exploit Meltdown and Spectre with Performance Counters. (2018). https://blog.trendmicro.com/trendlabs-security-intelligence/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters/ "[Online; accessed 17-April-2018]".

[10] D-man. 2014. How to translate virtual to physical addresses through /proc/pid/-pagemap. (2014). http://fivelinesofcode.blogspot.com/2014/03/how-to-translate-virtual-to-physical.html "[Online; accessed 13-April-2018]".

[11] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. 2013. On the Feasibility of Online Malware Detection with Performance Counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 559–570.

[12] Apache Software Foundation. 2018. Apache Spark data processing engine. (2018). https://spark.apache.org "[Online; accessed 13-April-2018]".

[13] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering* 8, 1 (2018), 1–27.

[14] D. Gruss, C. Maurice, and S. Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). Springer International Publishing, Cham, 300–321.

[15] D. Gruss, R. Spreitzer, and S. Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 897–912.

[16] N. A. Howgrave-Graham and N. P. Smart. 2001. Lattice Attacks on Digital Signature Schemes. *Des. Codes Cryptography* 23, 3 (July 2001), 283–290.

[17] Intel Pin. 2012. Intel Pin dynamic binary instrumentation tool. (2012). https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool "[Online; accessed 13-April-2018]".

[18] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[19] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. 2017. DDoS in the IoT: Mirai and Other Botnets. *Computer* 50, 7 (2017), 80–84.

[20] M. Lipp. 2018. libflush. (2018). https://github.com/IAIK/armageddon/tree/master/libflush "[Online; accessed 13-April-2018]".

[21] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564.

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622.

[24] Nguyen and Shparlinski. 2002. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptol.* 15, 3 (June 2002), 151–176.

[25] D. A. Osvik, A. Shamir, and E. Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, David Pointcheval (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[26] A. W. Paundu, D. Fall, D. Miyamoto, and Y. Kadobayashi. 2018. Leveraging KVM Events to Detect Cache-Based Side Channel Attacks in a Virtualization Environment. *Security and Communication Networks* 2018 (2018), 4216240:1–4216240:18.

[27] C. Percival. 2005. Cache missing for fun and profit. In *Proc. of BSDCan 2005*.

[28] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 381–391.

[29] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. 2018. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys Tutorials* 20, 1 (Firstquarter 2018), 465–488.

[30] S. Tweneboah-Koduah, Knud Erik Skouby, and Reza Tadayoni. 2017. Cyber Security Threats to IoT Applications and Service Domains. *Wireless Personal Communications* 95, 1 (01 Jul 2017), 169–185. https://doi.org/10.1007/s11277-017-4434-6

[31] Z. Xu, S. Ray, P. Subramanyan, and S. Malik. 2017. Malware Detection Using Machine Learning Based Analysis of Virtual Memory Access Patterns. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '17)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 169–174.

[32] Y. Yarom. 2018. Mastik: A Micro-Architectural Side-Channel Toolkit. (2018). https://cs.adelaide.edu.au/~yval/Mastik/ "[Online; accessed 13-April-2018]".

[33] Y. Yarom and K. Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[34] T. Zhang, Y. Zhang, and R. B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses*, Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 118–140.

[35] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 305–316.

[36] Y. Zhou and D. Feng. 2005. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. (2005).