Evaluation of Cache Attacks on Arm Processors and Secure Caches

Shuwen Deng, Nikolay Matyunin, Wenjie Xiong, Stefan Katzenbeisser, Jakub Szefer

Abstract—Timing-based side and covert channels in processor caches continue to be a threat to modern computers. This work shows for the first time, a systematic, large-scale analysis of Arm devices and the detailed results of attacks the processors are vulnerable to. Compared to x86, Arm uses different architectures, microarchitectural implementations, cache replacement policies, etc., which affects how attacks can be launched, and how security testing for the vulnerabilities should be done. To evaluate security, this paper presents security benchmarks specifically developed for testing Arm processors and their caches. The benchmarks are evaluated with sensitivity tests, which examine how sensitive the benchmarks are to having a correct configuration in the testing phase. Further, to evaluate a large number of devices, this work leverages a novel approach of using a cloud-based Arm device testbed for architectural and security research on timing channels and runs the benchmarks on 34 different physical devices. In parallel, there has been much interest in secure caches to defend the various attacks. Consequently, this paper also investigates secure cache architectures using proposed benchmarks. Especially, this paper implements and evaluates secure PL and RF caches, showing the security of PL and RF caches, but also uncovers new weaknesses.

Index Terms—Processor Caches, Side Channels, Covert Channels, Security, Arm, Secure Caches

1 Introduction

Over the last two decades, many timing-based attacks in processor caches have been exploited to show that it is possible to extract sensitive information across the logic boundaries established by the software and even hardware protection mechanisms, e.g., [1], [2], [3], [4], [5], [6]. Even though a variety of secure processor architectures have been proposed [7], the caches in the proposals are still vulnerable to timing channel attacks. Further, most recently, Spectre [8] and Meltdown [9] attacks have been presented, which attack commercial processors. Many of their variants depend on cache timing covert channels to extract information. They exploit speculative execution to access sensitive data and then make use of cache covert channels to actually extract the data. In most of the attacks, cache channels are thus critical to actually make the attacks work.

Despite cache timing channel threats, most of the research has previously focused on x86 processors. Specifically, there is no previous, systematic evaluation of Arm devices, despite over 100 billion Arm processors being sold [10].

Consequently, this work fills the research gap by analyzing the security of Arm processors through new security benchmarks developed for testing timing channels in Arm processor caches. The benchmarks are built to evaluate 88 types of vulnerabilities previously categorized for processor caches in our conference paper [11]. To gain an understanding of the scope of the vulnerabilities in Arm, this work provides the first, large-scale study of Arm processors, by testing over 34 different physical devices through three cloud-based device farms: the Visual Studio App Center [12], the Amazon AWS Device Farm [13], and the Firebase Test Lab [14]. For the three cloud-based device farms, we develop the first *cloud-based cache security testing platform*. We further develop and perform sensitivity tests to evaluate how incorrect cache configuration information (resulting from misconfiguration or a malicious misinformation) affects the results of the benchmarks, and which types of tests are most affected by incorrect cache configurations. As a result, we demonstrate that many of the tests (and attacks), especially for address-only-based and set-or-address-based vulnerabilities (explained in Section 7), do not require precise knowledge of the cache configuration. On the other hand, this means that attackers can attack the system even when the cache configuration is unknown – hiding or intentionally misleading an attacker about the cache configuration is not a useful defense that one can use.

Compared to our prior conference paper [11], the benchmarking effort in this paper presents new insights and a number of new solutions we developed to effectively analyze the Arm processors. Arm uses the *big.LITTLE* architecture, which has heterogeneous caches and CPUs; we are the first to consider this aspect (Section 4.1) and the first to show the *big.LITTLE* architectures provide a larger attack surface by systematically evaluating different cross-core and cross-CPU vulnerabilities in these devices (Section 6.2). Our work further considers the pseudo-random replacement policy for caches used by Arm, while our prior paper [11] only considered LRU on x86. The replacement policy affects the eviction and probing steps used for 48 out of the 88 types of vulnerabilities and requires new approaches for testing.

Understanding the threats on Arm further requires overcoming a number of challenges. Cycle-accurate timings are not accessible without root access on Arm, while x86 provides accurate assembly instructions to record timing (e.g., `rdtscp`). Our benchmarks closely resemble real attacks by, for example, not assuming root privileges, but using code that can get reliable timing in user-level

- *S. Deng, W. Xiong and J. Szefer are with the Department of Electrical Engineering, Yale University, New Haven, CT, 06511. E-mail: shuwen.deng@yale.edu, wenjie.xiong@aya.yale.edu, jakub.szefer@yale.edu*
- *N. Matyunin is with Technical University of Darmstadt, Darmstadt, Hesse, Germany. E-mail: matyunin@seceng.informatik.tu-darmstadt.de*
- *S. Katzenbeisser is with University of Passau, Passau, Bayern, Germany. E-mail: stefan.katzenbeisser@uni-passau.de*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

programs. Our cache timing attack benchmarks use automatically-composed assembly code sequences specialized for Arm. This allows for testing different implementations of the assembly for the use in specific attack steps, and to obtain the final, more accurate vulnerability tests. We propose the first Arm benchmarks that utilize statistical tests to differentiate distributions of timings to check if vulnerabilities can result in attacks, with each benchmark run 30,000 times to better understand the timing distributions and minimize potential noise in the measurements.

We also found specific new insights about CPU features affecting security (Section 6.1). For example, we show that the Snoop Control Unit (SCU) in Cortex A53 contains buffers that handle direct cache-to-cache transfers; consequently, vulnerabilities related to differentiating cross-core invalidation timing occur much less frequently on Cortex A53 than on the other cores. Meanwhile, the Store Buffer (STB) implemented in Kryo 360 Gold/Silver core pushes the write accesses into a buffer, resulting in different timings of accesses to clean and dirty L1 data and resulting in more vulnerabilities. These are examples of units that help security, e.g., SCU, and hurt security, e.g., STB. Only through benchmarking of real devices can such insights be discovered.

Given the existing threats due to cache timing attacks, there has already been a number of works on secure caches. However, none of the cache designs have been systematically evaluated using benchmarks, such as ours. Consequently, having developed the benchmarks, we further analyze secure cache designs to understand if they can enhance security of Arm devices. This work shows the security of PL [15] and RF [16] caches, but also uncovers new weaknesses. Especially, we find a new attack related to eviction-based attacks in the PL cache because it fails to consider write buffer impacts when locking data in the cache. Further, we found that the RF cache is secure when setting a large neighborhood window (for selecting the randomly fetched cache line). A small random-fill neighborhood window, however, may be better for the performance, but with high probability can leak information about the victim's cache access.

1.1 Contributions

In summary, the contributions of this work compared to our prior conference paper [11] are as follows:

- Design of the first security benchmark suite and evaluation framework specifically for Arm, to systematically explore cache timing-based vulnerabilities in Arm devices (considering the *big.LITTLE* architecture, pseudo-random cache replacement policy, etc.)
- Use of a new sensitivity testing approach to evaluate how incorrect cache configuration information can affect the benchmarks, and consequently which vulnerability types can still be successful if the cache configuration is incorrect or unknown.
- The first large-scale cloud-based test platform allowing to uncover the security characteristics of a large number of different Arm devices.
- The first set of cache security benchmarks which can run on the *gem5* simulator. This allows to test microarchitectural features, such as write buffer and MSHR sizes, which cannot be changed on real devices, and provides an understanding of how they affect the security of the system.
- Implementation of secure caches in *gem5* simulation, and use of the benchmarks to find a new write-based attack on the PL cache and problems with the RF cache if the random-fill neighborhood window is not sufficiently large.

1.2 Open-Source Benchmarks

The Arm benchmarks and the code for the cloud-based framework will be released under open-source license and made available at https://caslab.csl.yale.edu/code/arm-cache-security-benchmarks/.

1.3 Additional Data and Results

An arXiv version of this paper is available at https://arxiv.org/abs/2106.14054. There will be more evaluation data results added in an appendix to the arXiv paper, which is not present in this version.

2 Related Work and Background

This section provides background on prior cache timing-based side-channel attacks in Arm devices, and gives an introduction to our three-step model used as foundation for the benchmarks and the evaluation given in this paper.

2.1 Cache Timing-Based Attacks on Arm

Most of the existing work so far has focused on x86 processors. For Arm, we are aware of six papers [17], [18], [19], [20], [21], [22] that specifically explore security of caches. Table 1 lists the related work and compares it to this paper. AutoLock [18] explores how the *AutoLock* feature found in some Arm processors could be used to thwart some cache timing attacks; the paper also shows how attackers can overcome the feature and perform timing attacks. This work explores previously proposed Evict+Time [3], Prime+Probe [3], and Evict+Reload [23] attacks. ARMageddon [17] focuses on cross-core cache timing attacks using Prime+Probe [3], Flush+Reload [24], Evict+Reload [23], and Flush+Flush [25] strategies on non-rooted Arm-based devices. TruSpy [19] analyzes timing cache side-channel attacks on Arm TrustZone. It exploits cache contention between the normal world and the secure world to leak secret information from TrustZone protected code. The work only considers the Prime+Probe [3] attack strategy. Zhang et al. [20] give a systematic exploration of vectors for Flush+Reload [24] attacks on Arm processors and Lee et al. [22] explore Flush+Reload [24] attacks on the Armv8 system. iTimed [21] makes use of Prime+Probe [3], Flush+Reload [24], and Flush+Flush [25] to attack Apple A10 Fusion SoC.

While existing works do a good job testing a few vulnerabilities, they fail to systematically analyze all possible types of cache timing attacks in Arm processors, as does this work.

2.2 Three-Step Model for Cache Attacks

Based on the observation that all existing cache timing-based side and covert channel attacks have three steps, a three-step model has been proposed previously by the authors [11]. In the three-step model, each step represents the state of the cache line after a memory-related operation is performed. First, there is an initial step (*Step1*) that sets the cache line into a known state. Second, there is a step (*Step2*) that modifies the state of the cache line. Finally, in the last step (*Step3*), based on the timing, the change in the state of the cache line is observed. Among the three steps, one or more steps comprise the victim's access to an address that is protected from the attacker (denoted by V_u), and timing is observed in *Step3*. In the model, there are three possible cases for the address

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

TABLE 1: Comparison to related work exploring Arm processors and cache timing attacks.

	Num. Vuln. Explored	Num. Devices	Cloud-Based Framework	`gem5` Testing	Secure Cache Testing
AutoLock [18]	3	4	✗	✗	✗
ARMageddon [17]	4	4	✗	✗	✗
TruSpy [19]	1	1	✗	✗	✗
Zhang et al. [20]	1	5	✗	✗	✗
Lee et al. [22]	1	1	✗	✗	✗
iTimed [21]	3	1	✗	✗	✗
This Work	88	34	✓	✓	✓

of V_u: (1) a, which represents an address known to the attacker, (2) a_{alias}, which refers to an address that maps to the same cache set as a, but is different from a, and (3) Not In Block (*NIB*), which refers to an address that does not map to the same cache set as a. If a vulnerability is effective, the attacker can infer whether V_u is the same as a, a_{alias}, or *NIB* based on the access timing observations. The soundness analysis of the three-step model in our prior work [26] showed that it covers all possible timing-based attacks in set-associative caches. Our recent conference paper [11], upon which this journal paper improves, presented a benchmark suite based on the three-step model to evaluate vulnerabilities in x86 processors – it did not evaluate Arm processors nor secure cache designs.

2.3 Cache Vulnerability Types

We previously identified 88 vulnerability types in caches [11]. To better summarize them, this work categorizes them into different attack types, as shown in Table 2. *AO*-Type (address-only-based), *SO*-Type (set-only-based), and *SA*-Type (set-or-address-based) categorize the vulnerabilities based on the information that the attacker can gain from the timing observation. Note that our prior work [11] defined the three types as *A*-Type, *S*-Type, and *SA*-Type, respectively; we rename the types in this paper to better convey their meanings. Furthermore, we also categorize them as *I*-Type (internal-based) and *E*-Type (external-based) based on whether the interference is within the victim process or between the victim process and the attacker process. These two types of categories are orthogonal to each other. One specific vulnerability can be both one of *AO*-Type, *SO*-Type, or *SA*-Type, and one of *I*-Type or *E*-Type. For example, vulnerability #43 (see Figure 3) belongs to the *E*-*SO*-Type. Here the E-Type and SO-Type are merged into a combined vulnerability E-SO-Type.

3 Threat Model and Assumptions

We assume that there is a victim that has secret data which the attacker tries to extract through timing of memory-related operations. The victim performs some secret-dependent memory accesses (V_u) and the goal for the attacker is to determine a particular memory address (or cache index) accessed by the victim. The attacker is assumed to have some additional information, e.g., he or she knows the algorithm used by the victim, to correlate the memory address or index to values of secret data.

In addition to regular reads, writes, and flush operations, we assume that the attacker can make use of cache coherence protocol to invalidate other core's data, by triggering read or write operations on the *remote* core as one of the steps of the that attack.

A negative result of a benchmark means there is likely no such timing channel in the cache or the channel is too noisy to be observable. Meanwhile, a positive result may be due to structures other than cache, such as prefetchers, Miss Status Handling Registers (MSHRs), load and store buffers between processor and caches, or line fill buffers between cache levels. Our benchmarks focuses on L1 data caches, but we consider that timing results could be due to all the different structures. Detailed benchmarks for these structures or other levels of caches are left for future work.

4 Arm Security Benchmarks

In this section, we present the first set of benchmarks which is used to evaluate L1 cache timing-based vulnerabilities of Arm processors. To implement the security benchmarks on Arm, as listed below, we developed solutions to key challenges accordingly.

4.1 Heterogeneous CPU Architectures

Arm processors implement the *big.LITTLE* architecture with *big* and *little* processor cores having different cache sizes. This presents a new challenge, as the architecture is fundamentally different from multi-core systems where all cores have identical cache sizes and configurations. This was not considered in our previous work [11] which only dealt with x86, nor in previous studies [17], [18], [19], [20] which only tested attacks on one core type.

The *local* core is the one wherein is located the target cache line that the attacker wants to learn. Meanwhile, the *remote* core is a different core where the target cache line is not located, but which could affect the *local* core and its caches, e.g., via cache coherence protocol. Thus, both cross-core and cross-CPU vulnerabilities are considered in our work by testing the victim and attacker operations on different combinations of *local* and *remote* cores. Especially, with different *big* and *little* processor cores, a *local* or *remote* core can be either of *big* or *little* core type, resulting in four combinations.

Because we consider different core types, unlike prior work, and caches are not even between the *big* and *little* cores, we define how to correctly specify the cache configurations for the benchmarks when running the tests:

- If the first two steps of the three-step model describing a particular vulnerability both occur in the remote core, use the remote core's cache configuration.
- In all other cases, use the local core's cache configuration.

In the three-step model, when testing for vulnerabilities, main interference (leading to potential timing differences) occurs within the first two steps, while the final, third step is used for the timing observation used to determine if there is possible attack or not. Therefore, the above method of choosing the cache configuration focuses on where the main interference is occurring in the three steps.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

TABLE 2: Attack vulnerability types, following [11].

Attack Type	Description
AO-Type (address-only-based)	In vulnerabilities of this type, the attacker can observe that the timing for victim's access $V_u = a$ is different from the timing for victim's accesses $V_u = a_{alias}$ or $V_u = NIB$, so the attacker can infer if the address of V_u is equal to a known address a or not. Vulnerabilities of this type usually differentiate timing between L1 cache hit and DRAM access, which is usually large and distinguishable. Sample vulnerabilities of this type are Flush+Reload (vulnerability benchmarks #5-#8 shown in Figure 3).
SO-Type (set-only-based)	In vulnerabilities of this type, the attacker can observe that the timing for victim's access $V_u = a$ or a_{alias} is different from the timing for victim's access $V_u = NIB$, or the attacker can observe that the timing for victim's access $V_u = a_{alias}$ is different from the timing for victim's accesses $V_u = NIB$ or $V_u = a$. In this case, the attacker can infer the cache set of the address of V_u. Vulnerabilities of this type usually differentiate timing between L1 cache hit and L2 cache hit, which is usually small. Sample vulnerabilities of this type are Evict+Time (vulnerability benchmark #41 shown in Figure 3).
SA-Type (set-or-address-based)	In vulnerabilities of this type, the attacker can observe different timing for victim's accesses $V_u = a$, $V_u = a_{alias}$, and $V_u = NIB$. For example, in Prime+Probe (vulnerability #44), if in *Step1*, attacker reads data in address a; then in *Step2*, the victim writes to V_u; and then in *Step3*, the attacker tries to read data in address a, data can be read from the write buffer (due to the write in the second step if $V_u = a$) instead of being read directly from the L1 cache (if $V_u = NIB$ or $V_u = a_{alias}$) and attacker can observe the timing difference of the two cases.
I-Type (internal-based)	Vulnerabilities of this type only involve the victim's behavior in *Step2* and *Step3* of the three-step model. One example of this attack is the Bernstein's Attack (vulnerabilities #33-#36).
E-Type (external-based)	Vulnerabilities of this type are the ones where there is at least one access by the attacker in the second or third step, e.g., Flush+Reload (vulnerabilities #5-#8).

4.2 Random Replacement Policy in Arm

Modern Arm cores use the random replacement policy in the L1 cache [17]. This policy is significantly different from the Least Recently Used (LRU) replacement policy, and poses fundamental challenges for eviction and probing steps in 48 out of 88 vulnerability types.

In particular, this makes the set-only-based vulnerabilities (*SO*-Type) harder to implement. The reason is that occupying a cache set in caches using a random replacement policy is not as easy as in caches using LRU or similar policies, where accessing a certain number of ways (denoted as `cache_associativity_num`) of cache lines in a cache set is able to evict all data in the set. In caches using the random replacement policy, the *cache set thrashing problem* [27], referring to self-evictions within the eviction set, which affects accessing all the ways of the cache set in eviction-based vulnerabilities. To avoid this problem, we use a smaller set size to avoid set thrashing in our benchmarks. We set the eviction set size to `cache_associativity_num-1` and then repeat each step's memory operations 10 times. Using this technique, we are able to reduce set thrashing significantly given the random replacement policy. However, in this case, exactly one way will not be occupied after the repeated memory operations. This will cause victim's access in one out of `cache_associativity_num` ways to be not detectable, but this is acceptable as vulnerabilities can still be detected as we show in our evaluation.

4.3 Measuring and Differentiating Timing

For benchmarking Arm cache timing-based vulnerabilities, this work is the first to utilize statistical tests – Welch's t-test [28] – to differentiate distributions of timings to check if vulnerabilities can result in attacks. The *pvalue* is the threshold used to judge the effectiveness of the vulnerabilities. Based on our evaluation, we select 0.00049 for the *pvalue* in our tests, improved from our previous work on x86 [11], and use this to determine if different timing distributions are distinguishable. We chose Welch's t-test since it is generally used in attack evaluations [29], [30], [31]. There is also Kolmogorov-Smirnov's two-sample test [32] that can be used to differentiate distributions. However, in the case of cache timing side channel, there is only two possible timing observations

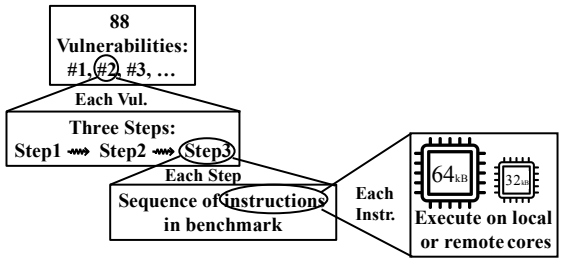


Fig. 1: Relationship of the 88 vulnerabilities, each of which is described using three steps from the three-step model. The steps are further translated into sets of assembly instructions for the benchmarks, and the code can be run on either *big* or *small* cores in the tested systems.

(i.e., hit or miss), t-test is sensitive to the mean of distributions, and thus fit in this case.

The statistical tests are used to differentiate timings of memory related operations. However, cycle-accurate timings are not accessible without root access on Arm, while x86 provides accurate assembly instructions to record timing (e.g., `rdtsc`). Consequently, we developed code that can get reliable timing measurements in user-level applications using the `clock_gettime()` system call. We experimented with other different performance counters and thread timers, but these proved not to be applicable or accurate enough for our benchmarks.

When performing timing measurements, in our experience, Arm devices further exhibit a lot of system noise when running the tests on real devices in the cloud-based device farms, possibly due to OS activity, or other background services. Therefore, we set the benchmarks to run more than $30,000$ repetitions for each benchmark for each device to average out the noise. Further, when running each step operated by either the victim or the attacker, we isolate the core to avoid influence of other application processes from user-level applications.

4.4 Summary of Benchmark Structure

Following the above features, we developed benchmarks for all 88 vulnerabilities. As shown in Figure 1, there are three steps for each

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

Algorithm 1 Read/Write Access Code Sequence

```
1: asm __volatile__ (
2: "DSB SY \n"
3: "ISB \n"
4: "LDR/STR %0, [%1] \n"
5: "DSB SY \n"
6: "ISB \n"
7: : "=r" (destination)
8: : "r" (array[i]));
```

Algorithm 2 Flush Code Sequence

```
1: asm __volatile__ (
2: "DSB ISH \n"
3: "ISB \n"
4: "DC CIVAC, %0 \n"
5: "DSB ISH \n"
6: "ISB \n"
7: : : "r" (array[i]));
```

vulnerability, and each step is realized by a sequence of instructions. The instruction sequences from each step can execute on local or remote cores. When performing the steps, there are two possible cases for the victim's or attacker's memory related operation: read or write access for a memory access operation; and flush or write in the remote core for an invalidation-related operation. Thus, for each vulnerability, there are in total of $2^3 = 8$ types considering different cases of each step's operation. Further, if a vulnerability being tested has both the victim and the attacker running on one core, these two parties can run either time-slicing or multi-threading. Consequently, the 8 cases are doubled to account for both time-slicing and multi-threading execution. Thus, for each vulnerability being tested, there are correspondingly 8-16 cases depending on the specific vulnerability. Each vulnerability is realized as a single benchmark program. In total there are 1094 benchmarks for all 88 types of vulnerabilities.

The 1094 benchmarks are automatically generated. The basic code sequences, e.g., Alg. 1 and 2, are composed into programs, with one program for each benchmark. Additional instructions are used in the benchmarks to pin execution of the code to different processor cores when testing different configurations. The resulting 1094 programs are compiled and executed on the devices under test as detailed in the next section.

5 Cloud-Based Framework

In this section, we report on the first cloud-based platform for testing cache channels on Arm devices. Our prior work only considered x86 [11] with several processors manually set to test, and work by others only manually tested only few Arm devices [17], [18], [19], [20].

5.1 Android Device Testbeds

We build our evaluation framework using testing platforms for mobile devices, namely the Visual Studio App Center [12], the Amazon AWS Device Farm [13], and the Firebase Test Lab [14]. We developed a framework which allows us to run custom binary benchmarks and retrieve the results in an automated manner.

In these cloud deployments, it is not possible to execute benchmark files through a remote shell and download the results. Instead, the entire functionality must be implemented as a user-level native Android application. Consequently, the benchmark executables are inserted into the application package (APK) of a custom Android application we developed. Figure 2 illustrates the resulting test setup, which will be open-sourced.

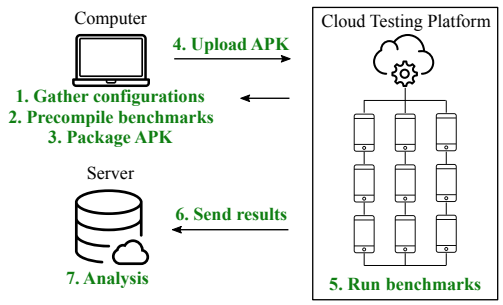


Fig. 2: Overview of the evaluation framework using the cloud-based testing platforms for Android mobile devices.

5.2 Extracting Cache Configurations

To build the benchmark, cache and CPU configuration information are needed. The configuration can be automatically identified by reading the corresponding system information located at */sys/devices/system/cpu/cpux/* (where *x* stands for the CPU core number) on each tested device. However, depending on the SELinux policies applied by the vendor and Android version, access to these files is restricted on some devices [33]. For these device models, we manually identify and verify their cache configurations from public resources. Finally, we store both automatically- and manually-extracted cache configuration parameters in a single database, and include this database into the APK, so that it can be used when running the benchmarks.

5.3 Packaging Security Benchmarks

Starting from Android 9, the operating system does not allow to execute files from an arbitrary writeable location on the filesystem [34]. Instead, only native library dependencies within an Android application can be executed. Consequently, we pre-compile and place the benchmark files in the resource subfolder of the APK package which contains native libraries (*src/main/resources/lib/arm64-v8a*), as the OS grants read-and-execute permissions for all binary files in this subfolder.

5.4 Running Benchmarks

We give an overview of our evaluation framework in Figure 2. Once the cache configuration is extracted (step 1), the corresponding benchmarks are precompiled (step 2) and packaged (step 3), we upload the application package to the cloud testing platforms (step 4). The implemented application does not require any user interaction. Instead, it contains an instrumented unit test which automates the execution of benchmarks. The tests can be run simultaneously on multiple devices (step 5). The process of uploading and running the application is automated using the APIs provided by the cloud platform provider.

On each device, the application first identifies the device model by accessing the *Build.MODEL* property. This information is used to look up the corresponding cache configuration parameters in the database. Afterwards, the application executes the precompiled benchmarks one by one, using the corresponding parameters. In order to automatically retrieve the results of benchmarks from

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

Core Name	Core Freq.	L1 Cache Config.	SoC Name	Vul. Num.
Kryo 585[1] Gold/ Silver	2.42-2.84/ 1.8	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 865	88
Kryo 385[2] Gold/ Silver	2.5-2.8/ 1.6-1.7	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 845	87
Kryo 360[3] Gold/ Silver	2.0-2.2/ 1.7	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 670/ 710	87
Cortex A53[4]	1.9-2.2	32 KB 4-way	Nvidia Tegra X1/ Qualcomm Snapdragon 625/ 630	81
Kryo 280[5] Gold/ Silver	2.35-2.5/ 1.8-1.9	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 835	79
Kryo 260[6] Gold/ Silver	1.8-2.2/ 1.6-1.8	64 KB 16-way/ 32 KB 4-way	Qualcomm Snapdragon 636/ 660	76

TABLE 3: CPUs and SoC types found in the evaluated devices. The *Core Name* (with corresponding number used in Figure 3), *Core Freq.*, and *L1 Cache Config.* columns show the processor core names, their frequency ranges, and typical cache configurations. The *Vul. Num.* column shows the average number (out of 88) of vulnerabilities that show up during tests; smaller value is better.

multiple devices, we implement an HTTP server which can receive POST requests from Android applications. Each request contains the results in textual or binary format. As the execution time of the whole set of benchmarks on a device can take several hours, the application periodically sends the intermediate results to the server. In this way, we can precisely monitor the current state of the execution on each device. Finally, the results are collected from the server (step 6) for further analysis (step 7).

6 Evaluation

We tested a total of 34 different devices. The corresponding processor core types are shown in Table 3 – note that some devices use the same processor or SoC configuration so there are less than 34 processors in Table 3. The results of the tests are shown in Figure 3, which shows the vulnerabilities that can possibly be exploited on the device, based on sufficient timing differences in the memory operations corresponding to each three-step attack. Figure 3 consists of 88 columns, each corresponding to one of the three-step vulnerabilities. The vulnerabilities are colored based on the different types.

In addition to smartphones, we further tested other Arm cores, leveraging Amazon EC2 [35] with an X-Gene 2 core and Chameleon cloud [36] with a Neoverse core to test Arm processors on servers. Arm server chip results generally have similar patterns as the mobile devices. Therefore, in this work, we show only results for the mobile devices from the cloud-based testbeds.

6.1 Microarchitectures' Impacts on the Vulnerabilities

Below we list some of the observations gained from our evaluation. Only through the extensive benchmarking of caches on a large set of devices, can such insights be discovered.

6.1.1 Store Buffer

The STB (STore Buffer) is used during write accesses to hold store operations. This structure makes clean and dirty L1 data access timing easier to be distinguished. For example, $I-SA$-Type vulnerability #33 differentiates timing between reads of dirty L1 data and reads of clean L1 data, or between writes of dirty L1 data and writes of clean L1 data, which is a typical vulnerability that allows STB to make it more effective. From the evaluation results, Kryo 360 Gold/Silver cores are more susceptible to vulnerabilities such as #33, compared to Cortex A53 core, which confirms the fact that the STB is presented in Kryo 360 Gold/Silver cores but not in Cortex A53 core, based on reference manuals.

6.1.2 Snoop Control Unit

The Snoop Control Unit (SCU) contains buffers that can handle direct cache-to-cache transfers between cores without having to read or write any data to the lower cache by maintaining a set of duplicate tags that permit each coherent data request to be checked against the contents of the other caches in the cluster. With the SCU, when comparing the timing between remote writes to invalidate local L1 data and remote writes to invalidate local L2 data, the SCU will accelerate the coherence operations. This makes the different cache coherence influence non-differentiable in timing on the cores that have the SCU.

For example, $I-SO$-Type vulnerabilities #78-#79 mainly use timing differences between flushing of L1 data and flushing of L2 data, or between remote writes to invalidate local L1 data and remote writes to invalidate local L2 data. From the evaluation results, vulnerabilities #78-#79 occur much less frequently on Kryo 280 Gold/Silver cores and Cortex A53 cores compared to Kryo 360 and Kryo 385 Gold/Silver cores. This supports the observation that the Kryo 280 Gold/Silver cores and Cortex A53 cores have a Snoop Control Unit (SCU), which helps prevent these types of vulnerabilities, while Kryo 360 and Kryo 385 Gold/Silver cores do not have it.

6.1.3 Transient Memory Region

Transient Memory Region allows for setting a memory region as transient. Data from this region, when brought into L1 cache, will be marked as transient. As result, during eviction, if this cache line is clean, it will be marked as invalid instead of being allocated in the L2 cache.

Although this may help avoid polluting the cache with unnecessary data, internal and external SO-Type vulnerabilities #33-#44 that we are able to differentiate between L1 and L2 cache hits can now differentiate between an L1 cache hit and a data access from DRAM. This makes this type of vulnerability more effective on cores that support this feature, which are Kryo 360/385 Gold/Silver cores, compared to other cores, such as Cortex A53.

6.2 Heterogeneous Caches' Impact on Vulnerabilities

We also evaluated how Arm's *big.LITTLE* architecture impacts the attacks, where we set *local* and *remote* core to be either *big* or *little* processor core. In Figure 4, we present evaluation results for one example device, Google Pixel 2. A similar pattern was observed for all other tested devices.

SO-Type and SA-Type vulnerabilities which differentiate L1 and L2 cache hit timings (#41-#44) are mostly vulnerable to the case when the *local* core uses the *big* core. This is mainly because the bigger cache (e.g., 64K 16-way vs. 32K 4-way) of the *big* core results in larger timing differences for the vulnerabilities that require priming each cache set, reducing the proportion of system noise at the same time. SO-Type and SA-Type vulnerabilities which differentiate writing to remote dirty L1 and L2 cache data (#73-#76) are successful when *local* and *remote* core both use the *little* core. Dirty data are usually not stored in the cache line but stored in other locations such as write buffer. Write buffer is possibly processed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

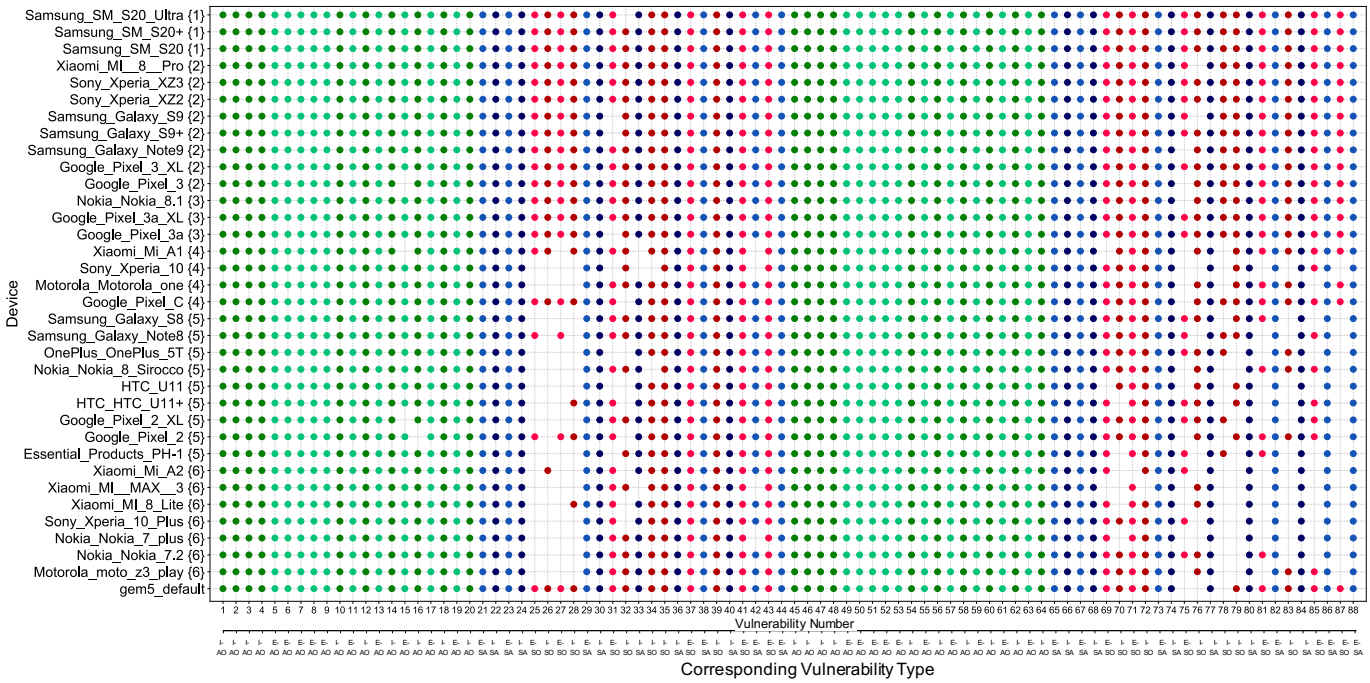


Fig. 3: Evaluation of the 88 types of vulnerabilities on different Arm devices.[a] A solid dot means the corresponding processor is found to be vulnerable to the vulnerability type. The "*I-SO*" (colored by dark red) and "*E-SO*" (colored by light red) are internal-interference set-only-based and external-interference set-only-based vulnerabilities, respectively. The "*I-AO*" (colored by dark red) and "*E-AO*" (colored by light red) are internal-interference address-only-based and external-interference address-only-based vulnerabilities, respectively. The "*I-SA*" (colored by dark red) and "*E-SA*" (colored by light red) are internal-interference set-or-address-based and external-interference set-or-address-based vulnerabilities, respectively. The devices are grouped according to their core types. Each device's core is labeled by a number shown after the device name, with corresponding cores shown in Table 3. The order is from the most vulnerable core to least vulnerable among the cores. The last line shows `gem5` testing results of default `gem5`, to show that `gem5` simulation gives similar results to real devices.

a. We further tested other Arm cores, including an X-Gene 2 core and a Neoverse core to test Arm processors on servers. The results generally have similar patterns as the mobile devices so we show only results for the mobile devices from the cloud-based testbeds.

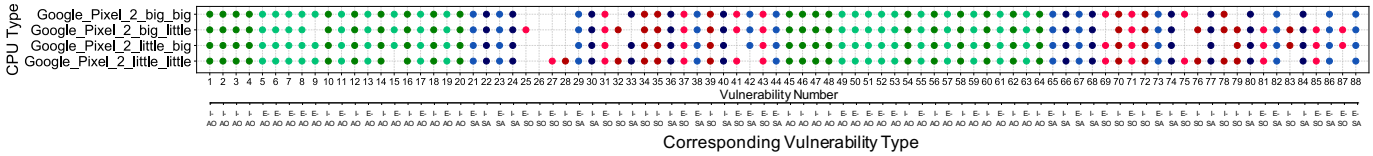
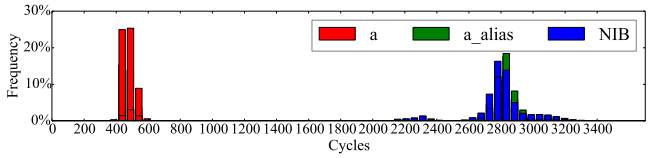
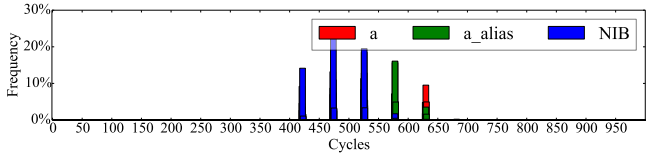


Fig. 4: Evaluation of the 88 types of vulnerabilities on different cores of Google Pixel 2. "big_big" means running both local and remote core on big cores, "big_little" means running local core on the big core, remote core on the little core. Same naming is applied to "little_big" and "little_little". Dot coloring is the same as in Figure 3.



(a) Sample histogram of *AO*-Type vulnerability



(b) Sample histogram of *SO*-Type vulnerability

Fig. 5: Samples of different types of vulnerabilities' timing histograms for different candidate values for V_u.

in an out-of-order way. Therefore, fewer number of writes due to fewer number of ways in *little* core are more likely to have relatively differentiable timing. *SO*-Type and *SA*-Type vulnerabilities which differentiate writing remote L1 and remote L2 cache data (#77-#88) are mostly successful when *local* and *remote* cores use different core types (*big* or *little*). This is due to the fact that *big* and *little* cores are often in different quad-core clusters in the SoC, where coherence time across quad-core cluster results in higher timing differences when accessing data located in the remote cluster.

6.3 Core Frequency's Impact on Vulnerabilities

High clock frequency tends to make long memory operations more differentiable, and will make timing attacks easier to exploit the difference. From the evaluation results, we found that devices with

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

higher clock frequency will likely have more effective timing-channel vulnerabilities.

This is especially visible in *SO*-Type vulnerabilities, most of which differentiate between L1 and L2 cache hits, which have a relatively small cycle difference, e.g., less than 10 cycles. However, if the core's frequency increases, the timing difference is also increased, which makes cycle distributions more differentiable and an attack possibly easier to execute.

6.4 Influence of Write Buffer and MSHR Sizes

We design our benchmarks so they can also be used in simulation. We use the Arm *big.LITTLE* configuration to run the benchmarks in Full System (FS) mode or Syscall Emulation (SE) mode on `gem5`. The simulator is configured to use the Exynos [37] configuration to model real Android devices and uses the O3CPU model with a 5-stage pipeline. The last line of Figure 3 shows the benchmark results when using the default configuration on the `gem5` simulator. Overall, we find that baseline `gem5` results have good correspondence with real CPUs in terms of the cache timing vulnerabilities.

Next, we evaluate different configurations of the Miss Status Holding Register (MSHR) and the write buffer (WB), both tested on `gem5`. Results are shown in Figure 6: A larger MSHR size leads to more vulnerabilities to be observed. MSHR is a hardware structure for tracking outstanding misses. Larger MSHR sizes lead to more outstanding misses that can be handled, which may stabilize the memory access timings and give more consistent results.

Changing the size of WB does not have an explicit influence on the vulnerability results. WB stores the write request, which frees the cache to service read requests while the write is taking place. It is especially useful for very slow main memory, where subsequent reads are able to proceed without waiting. We use the "SimpleMemory" option of `gem5`, which is relatively simple compared with the implementation of real devices and may not have the same slow memory timing in this case. As the result shows, bigger WB may improve performance and can be added without degrading security, while bigger MSHR may improve performance but at some cost to security.

6.5 Patterns in Vulnerability Types

It can clearly be observed from the colored dots in Figure 3 that *AO*-Type vulnerabilities are observable in almost all devices and in the simulation, because these types of vulnerabilities, e.g., differentiate L1 cache hits and DRAM hits, which have large timing differences. Such timing distribution results can be observed in Figure 5a. *SA*-Type vulnerabilities also occur relatively often, but are much more unstable compared with *AO*-Type vulnerabilities, which shows that different devices have large but quite variable timing differences among different memory operations, e.g., between clean abd dirty L1 data invalidation or between local access of remote clean and dirty L1 data. *SO*-Type vulnerabilities are least effective. This is because the timing differences between the observations such as L1 and L2 cache hits are so small that they are sometimes indistinguishable due to system noise. For example, timing distribution evaluation result shown in Figure 5b have small timing difference.

I-Type and *E*-Type vulnerabilities do not show explicit evaluation differences. In this case, another take-away message is that protecting only the external-interference vulnerabilities is not enough at all. Internal-interference vulnerabilities can be as effective as the external-interference vulnerabilities for attacks.

6.6 Estimating the Real Attack Difficulty

To estimate the real attack difficulty, we can leverage the distance and likelihood (using p-value) of different timing measurement distributions. As is shown in Figure 5 in Section 6.5, *AO*-Type or *SA*-Type vulnerabilities are easier to exploit since they depend on timing differences of L1 cache hits vs. DRAM accesses; meanwhile *SO*-Type vulnerabilities are more difficult to exploit, since they depend on the timing differences between L1 and L2 cache hits, which are much smaller compared to the former.

Further, our benchmarks show the overall attack surface. If a motivated attacker only needs to use one attack to derive sensitive information, he or she will likely start with *AO*-Type or *SA*-Type vulnerabilities. However, the bigger the attack surface is, the more options he or she has, and if there are defenses for *AO*-Type or *SA*-Type types of vulnerabilities, attackers could still leverage *SO*-Type vulnerabilities. The goal of this work is to show the whole attack surface on Arm devices, including vulnerabilities and attack types that are not previously presented in the literature. Which attack could be used in practice depends on the attacker's motivation and resources, but thanks to this work, the overall attack surface is better understood.

6.7 Results Compared with Other Work

For our benchmark results shown in Figure 3, strategies exploited by existing Arm attacks – Evict+ Time (#41-#42 in the Figure), Prime+Probe (#43-#44 in the Figure), Flush+Reload[1] (#5-#8 in the Figure), and Flush+Flush (#47-#50 in the Figure) – all indeed show up as effective vulnerabilities for 30 out of the 34 mobile devices tested. This confirms that our benchmarks can cover existing work. Note that the 5 types of vulnerabilities explored by prior work, e.g., the Evict+ Time, etc., can be realized using more than one vulnerability from the 88 types, thus prior work covers 12 types, leaving 76 types not considered, for the total of 88 vulnerabilities that are possible.

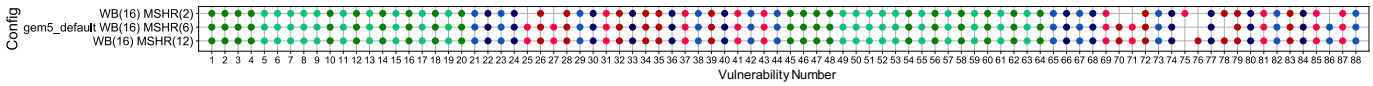
6.8 Summary of Vulnerability Trends

To summarize, the patterns of the vulnerabilities uncovered thanks to the systematic benchmarking on 34 devices are:

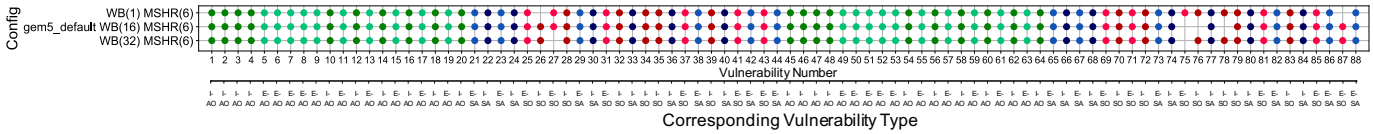
- Microarchitectural features: performance increasing features such as the store buffer can degrade security, while features such as the snoop control unit can be helpful, indicating that security and performance are not always at odds with each other, and some features can help both.
- Heterogeneous cache size: larger coherence timing for accesses involving cores in different clusters, compared to within same cluster, may lead to more vulnerabilities being effective.
- Core frequency: larger core frequency generally correlates with more vulnerabilities.
- WB and MSHR sizes: WB size does not impact security, while larger MSHR may allow more vulnerabilities to be effective.
- Vulnerability type effectiveness: relations of number of effective vulnerabilities showed are: *AO*-Type $>$ *SA*-Type $>$ *SO*-Type; meanwhile, *I*-Type and *E*-Type vulnerabilities are similarly effective on the tested devices.
- Tested device results: relations of number of effective vulnerabilities showed are: Kryo 585 $>$ Kryo 385 $\approx$ Kryo 360 $>$ Core A53 $>$ Kryo 280 $>$ Kryo 260.

1. Our Flush+Reload benchmarks test for a stronger variant of the Evict+Reload vulnerability shown in [17], [18].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers



(a) Benchmark `gem5` simulation results for different MSHR sizes.



(b) Benchmark `gem5` simulation results for different write buffer sizes.

Fig. 6: Evaluation of 88 types of vulnerabilities on different number of write buffer (WB) and MSHR sizes. A solid dot means the corresponding processor is found to be vulnerable to the vulnerability type. The "*SO*" (colored red) and "*AO*" (colored green) are set-only-based and address-only-based vulnerabilities, respectively. "*SA*" (colored blue) are the ones that are set-or-address-based. The "*E*" (colored in lighter color) and "*I*" (colored in darker color) are internal- and external-interference vulnerabilities, respectively.

7 Sensitivity Testing of Benchmarks

To understand how the benchmarks are affected by possible misconfigurations, we performed a number of sensitivity tests. In addition to evaluating how the benchmarks behave, the sensitivity study allows us to understand how knowledge (or lack of knowledge) of the correct cache configuration affects the attacker's ability to attack the system.

7.1 Analysis of Sensitivity Testing

The most important cache parameters for sensitivity tests are: *associativity*, *line size*, and *total cache size*. We use $asso_d$, $line_d$, and tot_d to respectively denote the value of the parameters of the actual target device. Meanwhile, $asso_b$, $line_b$, and tot_b denote the cache parameters used by the benchmarks. The parameters used in the tests are varied and are different from the actual, correct parameters to test the sensitivity of the results to misconfiguration. As we show, setting the configuration incorrectly in the benchmarks changes the mapping of the addresses used by the benchmarks, and influences the number of vulnerabilities judged to be effective on a device.

We implement the sensitivity tests in the following way. A large array is maintained to locate three different candidates of the secret value (a, a_{alias}, or NIB). We consider two addresses that only differ in the low $log_2(line_b)$ bits to belong to the same cache line, and two addresses that are a distance of $C \times tot_b/asso_b$ (C is a integer) apart to map to the same cache set. For each step, we access $asso_b$ number of addresses for each cache set to occupy or cause collision in the whole cache set. To increase the signal to noise ratio in our measurements, rep cache sets are accessed in each of the steps of a benchmark (in our setting this number is 8).

When $asso_b$, $line_b$, or tot_b deviates from $asso_d$, $line_d$, or tot_d, the following situations could happen: ① the number of addresses being accessed in one cache set is less than $asso_d$, so interferences that should happen are not observed; ② the addresses that should map to a target cache set actually map to several cache sets, and contention in the target cache set might not happen or will become contention in several sets; and ③ the addresses that should map to different cache sets actually map to the same cache set, introducing noise to the channel. We show later that the total number of attacks judged to be effective is less when an incorrect configuration is used – however, there are still attacks that are effectively independent of the configuration setting.

In the following, we denote one L1 cache hit timing as t_{L1} and one L2 cache hit timing as t_{L2}. When the configuration of the benchmark is correct, if the secret maps to the same cache set as some known address that was accessed, t_{L2} will be observed, while if they are not mapped, t_{L1} will be observed. In this case, timing observations for mapped and unmapped cases are $asso_d \times t_{L2}$ and $asso_d \times t_{L1}$.

7.1.1 Cache Associativity

Associativity usually influences the number of accesses that map to a target cache set. We distinguish two cases:

- $asso_b < asso_d$: In this case, due to smaller number of ways accessed in each step, fewer evictions will occur (situation ①). If a data address maps to the same set as the secret data, timing observation will be $n \times t_{L2} + (asso_d - n) \times t_{L1}$ instead of $asso_d \times t_{L2}$. Here, $0 < n < asso_b$. Due to the random replacement policy, only n (not all $asso_b$) cache lines will be evicted. This will make the timing less distinguishable compared with the unmapped case, in which timing should be equal to $asso_d \times t_{L1}$.
- $asso_b > asso_d$: When $tot_b = tot_d$, this setting will lead to accesses that should map to one cache set actually mapping to several cache sets (situation ②). This will result in measuring more than rep of cache sets for one step, which possibly introduces more noise.

7.1.2 Cache Line Size

Line size generally influences which cache set is chosen within an attack (benchmark) step. Again, we distinguish two cases:

- $line_b < line_d$: In this setting, the accesses that should map to different cache sets in the benchmark actually map to the same cache set (situation ③). This will lead to the result that the benchmark measures less than rep cache sets effectively, causing a reduced signal to noise ratio. For example, when choosing $line_b = line_d/2$, then two addresses that differ in $line_b$ will map to the same cache line instead of different lines in difference sets. This results in having more L1 cache hits, from $asso_d \times t_{L2}$ to $asso_d/2 \times t_{L2} + asso_d/2 \times t_{L1}$, which makes it less distinguishable compared with unmapped case where timing is $asso_d \times t_{L1}$.
- $line_b > line_d$: In this setting, since we always access the first 64 bits in a cache line, the addresses that should map to the same sets in the benchmark (with the incorrect configuration) still map to the same set (if the correct configuration was used). However, when $line_b$ is larger or equal to $cache\_set/rep$[2]

2. In the example of Section 7.2, this number is equal to $128/8 = 16$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

times of $line_d$, the address for *NIB* in the benchmark will wrap back and map to the same cache set as a and a_{alias} (situation ③), causing a false negative result.

7.1.3 Total Cache Size

Cache size mainly influences the data addresses accessed in each step of an attack (benchmark).

- $tot_b < tot_d$: In this setting, the accesses that should map to one cache set in the benchmark actually map to several cache sets (situation ②), because $tot_b/asso_d < tot_d/asso_d$. This further causes the number of data accesses in each set to be less than the number of ways being accessed in the target cache set, i.e., $asso_d$ (situation ①). Thus, for the mapped case, it is equivalent to observing $n \times t_{L2}$ timing instead of $asso_d \times t_{L2}$ timing for this cache set, where $0 < n < tot_b/tot_d \times asso_d$ due to the random replacement policy. This could decrease the signal to noise ratio.
- $tot_b > tot_d$: Let $C' = tot_b/tot_d$. In most cases, C' will be an integer, assuming a cache size (both tot_b and tot_d) of 2^N bytes. In this setting, the cache addresses that are different by $tot_b/asso_d = C' \times tot_d/asso_d$ in the benchmark, will still map to a different cache set in target device[3]. Further, if C' is too large, this will cause unexpected system noise if prefetching, copy-on-write, etc., functions are enabled in the device.

7.1.4 Analysis by Vulnerabilities Types

For *AO*-Type and *SA*-Type Vulnerabilities, the timing observation for $V_u = a$ is different from $V_u = a_{alias}$ or $V_u = NIB$. In these types of vulnerabilities, the attack does not rely on the interference between different cache lines in a cache set. How the addresses map to the cache set does not affect the result, and the cache configurations will not influence the effectiveness of the vulnerabilities. Also, these types usually rely on relatively larger timing differences, so the signal to noise ratio is large.

SO-Type vulnerabilities usually derive the V_u information by observing evictions of the originally accessed data in a prior attack step. For *SO*-Type vulnerabilities, we need to access all the $asso_d$ ways to prime the whole cache set in order to observe the timing difference, therefore, *SO*-Type vulnerabilities will actually be influenced by the setting of parameters including *associativity*, *line size*, and *total cache size*.

7.1.5 Summary

Based on the above, we make three observations about the configurations' impact on the benchmarks and the corresponding attacks and how easy they are to perform:

1. Attackers can still attack the system even when they are uncertain about the cache configuration. This is especially true for *AO*-Type or *SA*-Type attacks since they are not impacted much by the (mis) configuration.
2. Most of the differences are due to *SO*-Type attacks, which do not work well when incorrect setting is selected.
3. Setting correct configurations causes more vulnerabilities to be judged effective for a device. Incorrect settings can cause an underestimation of the total number of vulnerabilities.

3. When C' is not an integer, e.g., $C' = 1.5$, then the address to set mapping will be different than the case when $tot_b = tot_d$, which is equivalent to having addresses mapped to other cache set, resulting in fewer number of addresses mapped to the target cache set (situation ①).

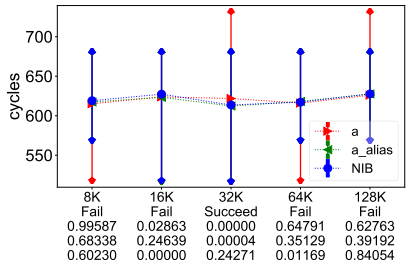


Fig. 7: Timing histogram of a vulnerability case when changing the cache size. The error bar shows the range of timing distribution and the dot shows the average timing cycles. "Succeed" under the configuration means the vulnerability is effective while "Fail" means not. Three values under "Succeed" or "Fail" are the *pvalue* for each two timing distributions out of three. If it is smaller than 0.00049, we judge the two timing distributions to be differentiable, otherwise not.

TABLE 4: Configuration test results for cache associativity, line size and cache size of Google Pixel 2. Black bold numbers show the largest effective number of vulnerabilities for each category. Middle column shows the correct configuration values for this device, other columns show smaller (left side) and bigger (left side) values that were tested for each parameter of the cache.

Config.		Effective Vul. Num. for Diff. Config.				
Associativity	$asso_b$ Value	1	2	4	8	16
	Total Vul. Num.	75	78	**82**	75	75
	SO-Type Num.	17	17	**20**	13	12
Line Size	$line_b$ Value	16	32	64	128	256
	Total Vul. Num.	77	75	**82**	80	79
	SO-Type Num.	14	12	**18**	17	17
Cache Size	tot_b Value	8192	16384	32768	65536	98304
	Total Vul. Num.	79	77	**82**	79	77
	SO-Type Num.	16	15	**20**	16	14

7.2 Evaluation of Sensitivity Testing

We tested a wide range of devices and found similar trends among the results. Here we give results for one example phone, Google Pixel 2, to show how the sensitivity test is implemented and evaluated.

The L1 cache configuration of small core of Google Pixel 2 is 32KB, 4-way set-associative with line size to be 64B. We test this configuration by changing one of the three parameters (associativity, line size or cache size), while keeping the other two the same to avoid interference between different parameters. The different configuration values we choose in our evaluation are listed in Table 4.

In the example test case shown in Figure 7, timing distribution differences between three candidates are larger for the correct configuration, compared to the wrong configurations. The vulnerability is effective under the correct configuration while it fails for the incorrect configuration.

As shown in Table 4, we found that differences between the number of correct configuration and incorrect configuration for all effective vulnerabilities and *SO*-Type only effective vulnerabilities are roughly the same. For example, when changing the associativity, difference of all effective vulnerability numbers between 4 (82) and 8 (75) is 7, which is the same as difference of *SO*-Type numbers (between 4 (20) and 8 (13)). This also shows that wrong configurations will still lead to *AO*-Type and *SA*-Type vulnerabilities to be effective even if the configuration is wrong.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

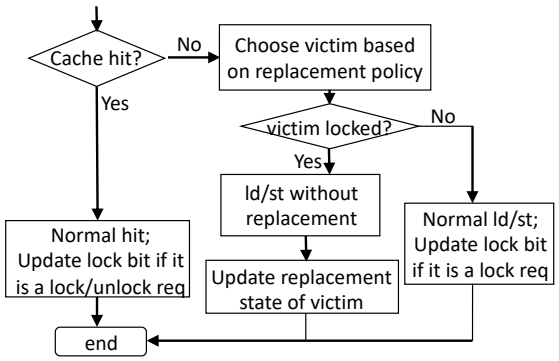


Fig. 8: PL cache replacement logic flow-chart, as proposed in [15].

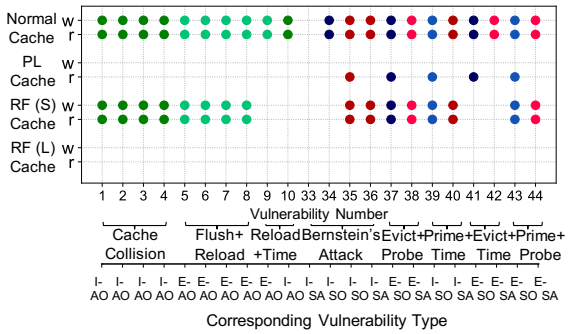


Fig. 9: Evaluation results of security benchmarks on PL cache, RF cache, and a normal set-associative cache, for comparison. Solid dots, half solid dots or empty dot mean all of the, part of the, or no vulnerability cases are vulnerable to the cache, respectively.

As shown in Table 4 as well, attacks are most effective under the correct configuration. When setting the wrong value for either one of the three cache configurations, the number of vulnerabilities that are effective decreases. On the other hand, this shows that hiding the cache architecture information or giving wrong configurations on-purpose is not a reliable defense.

8 Evaluation of Secure Caches

As shown in the previous sections, current commercial Arm architectures are indeed vulnerable to most of the attack types. A potential defense are secure caches. To help understand if existing secure cache designs could help defend the attacks in Arm processors, we implemented and evaluated the Partition-Locked (PL) [15] and Random Fill (RF) [16] caches together with our benchmarks in the `gem5` simulator. We show that they can defend many of the attacks, but we also uncover new vulnerabilities in the secure cache designs. In this section, we focus on the security analysis of the secure cache designs. Performance evaluations of PL cache and RF cache can be found in [15] and [16], where reasonable overhead is shown.

8.1 PL Cache Design and Implementation

Cache replacement is considered as the root cause of many cache side-channel attacks, and partitioned caches were proposed to prevent the victim's cache line from being evicted by the attacker. PL cache [15] is a flexible cache partitioning design, where the victim can choose cache lines to be partitioned. In the PL cache, each cache line is extended with a lock bit to indicate if the line is locked in the cache. When a cache line is locked, the line will not be evicted by any cache replacement until it is unlocked. Figure 8 shows the replacement logic of the PL cache. If a locked cache line is selected to be evicted, the eviction will not happen, and the incoming cache line will be handled uncached. If the victim locks the secret-related address properly and the cache is big enough to hold all the locked cache lines, the PL cache is secure against all types of timing-based vulnerabilities, because the secret-related address will always be in the cache.

To evaluate the PL cache against different vulnerabilities, we implement it in the L1 data cache and add new instructions to lock (and unlock) cache lines in the `gem5` simulator. The evaluation in `gem5` is run in SE mode using a single O3CPU core, where each benchmark has an additional `lock` step for locking the victim's cache line.

8.2 Security Evaluation of the PL Cache

Figure 9 shows the results of evaluation of the PL cache (and the RF cache, as well as the baseline set-associative cache). For the PL cache, *AO*-Type vulnerabilities such as Flush+ Reload fail, because the sensitive data is locked in the cache, and cannot be evicted by the benchmark steps that simulate the attacker. Without locking, a normal cache is vulnerable to these attacks, as shown in Figure 9.

For *SO*-Type or *SA*-Type vulnerabilities such as Bernstein's attack, theoretically the PL cache should prevent all of them as well. However, from the experimental results we find that when the steps are implemented using write (store), some of the attacks will still be successful in the PL cache. This is mainly due to the write buffer structure, which is not considered in original design of the PL cache [15]. These attack strategies all require conflicts of known and unknown secret cache lines. Although being locked before the attack runs, the secret cache lines may be further brought into the write buffer due to a write access and then leave the cache structure to "bypass" the locking features, making the attack successful. On the other hand, without the influence of the write buffer, we find that the attack cases that have all three steps to be non-write accesses to be always prevented on PL cache, as expected. The vulnerabilities leveraging the cache coherence states and multiple cores were not considered in original PL cache design, but can be tested in future.

The PL cache evaluation highlights the need for systematic security evaluation using benchmarks. Thanks to the approach, the original PL cache design is found to have a new write-based attack. More importantly, our benchmarks can be useful for designing future secure caches and testing them in *gem5*.

8.3 RF Cache Design and Implementation

To prevent interference caused by cache replacement, Random Fill (RF) cache [16] has been proposed to de-correlate the cache fill that causes the cache replacement and the victim's cache access. On a cache miss, the missing cache line will be handled without being fetched in the cache, instead a cache line in the neighborhood window $[addr - RF\_start, addr - RF\_start + RF\_size]$ will be fetched, as shown in Figure 10. In this way, the memory access pattern is de-correlated from the cache lines fetched in the cache. Since fetching cache lines in the neighborhood window may still carry information about the original *addr*, the security of RF cache depends on the parameters *RF_start* and *RF_size*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2021.3126150, IEEE Transactions on Computers

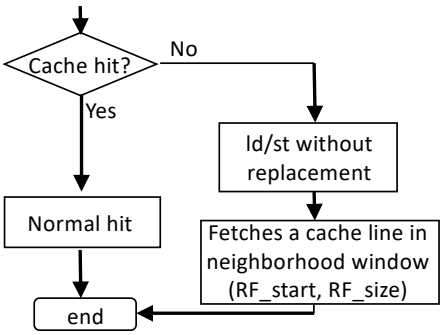


Fig. 10: RF cache replacement logic flow-chart, as proposed in [16].

We implement the RF cache in the L1 data cache, as suggested by the work [16]. Note that here the cache line will still be fetched into L2 cache, but vulnerabilities targeting the L1 cache should be defended. Parameters *RF_start* and *RF_size* can be configured in `gem5`. The benchmark suite for evaluation is identical to the normal three-step benchmarks, no additional step is required for the RF cache, e.g., no special locking step is needed.

8.4 Security Evaluation of the RF Cache

RF cache can potentially defend all attacks because the victim's access to a secret address will not cause the corresponding cache line to be fetched into cache, but a random cache line in a neighborhood window will be fetched instead. However, fetching a cache line in the neighborhood window still can transfer information about the victim's cache access. We tested two different RF cache configurations, one with small neighborhood window (5 cache lines) and one with large neighborhood window (128 cache lines[4]).

To reduce noise in the tests, the benchmarks test 8 contiguous cache lines and measure the total timing. When the neighborhood window of the RF cache is small, the cache line fetched into the cache will be not far from the address being accessed, and can still be observed by the third step of the benchmark with a high probability. As shown in Figure 9, for a small neighborhood window (S), a number of vulnerabilities are still effective, such as Flush+Reload and Prime+Probe.

For a large neighborhood window (L), no effective vulnerabilities are detected by the benchmark. For *SO*-Type vulnerabilities, the large neighborhood window de-correlates the memory access and the cache set to be accessed, so that the vulnerabilities can be prevented. For *AO*-Type vulnerabilities, the channel capacity of the cache side channel decreases with the window size due to the reduced probability of the desired cache line being fetched into cache, as analyzed in [16]. The neighborhood window of 128 cache lines is enough to mitigate the channel in our setting where there are 128 cache sets.

The evaluation of the RF cache shows how the benchmark suite can be used to help choose the design parameter, and the benchmark can quickly evaluate the design prototypes.

8.5 Security Evaluation of Other Secure Caches

CEASER [38] is able to mitigate conflict-based LLC timing–based side-channel attacks using address encryption and dynamic remapping. The CEASER cache does not differentiate whom the address belongs to and whether the address is security critical. When a memory access tries to modify the cache state, the address will first be encrypted using a Low-Latency BlockCipher (LLBC) [39], which not only randomizes the cache set it maps to, but also scatters the original, possibly ordered, and location-intensive addresses to different cache sets, decreasing the probability of conflict misses. The encryption key will be periodically changed to avoid key reconstruction. CEASER-S [40] allows CEASER to divide the cache ways into multiple partitions of all the cache ways and allows the line to be mapped to a different set in each partition via principles of skewing. The modified "skew" idea of CEASER-S cache assigns each partition a different multiple instance of CEASER to determine the set mappings to strengthen the random mapping. These two caches, focusing on randomizing cache set mapping, targets *SO*-type or *SA*-type attacks and cannot prevent *AO*-type vulnerabilities.

ScatterCache [41] uses cache set randomization to prevent timing-based attacks. It builds upon two ideas. First, a mapping function is used to translate memory addresses and process information to cache set indices. The mapping is different for each program or security domain. Second, the mapping function also calculates a different index for each cache way. The mapping function can be keyed hash or keyed permutation derivation function – a different key is used for each application or security domain resulting in a different mapping from addresses to cache sets. Software (e.g., the operating system) is responsible for managing the security domains and process IDs, which are used to differentiate the software processes and assign them with different keys for the mapping. As hardware extension, a cryptographic primitive such as hashing and an index decoder for each scattered cache way is added. ScatterCache is able to prevent *SO*-type or *SA*-type vulnerabilities by assigning a different index for each cache way and security domain. It encrypts both the cache address and process ID when mapping into the cache index, therefore, ScatterCache is able to prevent *E-AO*-type vulnerabilities such as Flush+Reload, but not *I-AO*-type vulnerabilities such as Cache Collision vulnerabilities.

Time-Predictable Secure Cache (TSCache) [42] relies on random placement to exhibit randomized execution times. To achieve side-channel attack robustness, random placement must also decouple cache interference of the attacker from the victim. Memory addresses from victim and attacker's processes must not contend systematically in the same cache set. Instead, each memory address from each process must be randomly and independently placed in a set, thus randomizing interference. This is achieved by operating the address (tag and index bits) together with a random number called random seed. Each task is forced to have a different seed so that conflicts between attacker's and victim's cache lines are random and independent across runs, thus defeating any contention-based attacks. The same seed is given to allow the communication between runnables of a given software components of an application via shared memory. TSCache exploits random placement to de-correlate set mapping with the corresponding address index bits. Therefore, it can be used to prevent *SO*-type or *SA*-type vulnerabilities but may not be able to prevent *AO*-type vulnerabilities.

9 Conclusion

This paper presented for the first time a large-scale evaluation of 34 Arm devices against 88 types of vulnerabilities. In total, three different cloud platforms were leveraged for the evaluation, and

4. There are 128 cache sets in the evaluated L1 cache.

gem5 was used for further analysis of certain microarchitectural features. Based on the evaluation results, the work uncovered a number of components of the microarchitectual design that influence the effectiveness of different types of the vulnerabilities. Further, sensitivity tests were used to understand impacts of possible misconfiguration on the outcome of the benchmarks, and also showed that even with uncertain cache configuration, number of attack types can be successful. To help defend the attacks, the PL and RF secure caches were implemented and evaluated on gem5. Based on the benchmarking results of the secure caches, a new attack on PL cache, and possible issues due to small window size in the RF cache were uncovered.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 1651945 and 1813797, and through SRC task 2488.001. The authors would like to acknowledge Amazon Web Services for cloud research credits used for some of the testing.

REFERENCES

[1] J. Bonneau and I. Mironov, "Cache-Collision Timing Attacks against AES," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2006, pp. 201–215.

[2] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005.

[3] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[4] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games–Bringing Access-Based Cache Attacks on AES to Practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 490–505.

[5] C. Percival, "Cache Missing for Fun and Profit," 2005.

[6] O. Acıiçmez and Ç. K. Koç, "Trace-Driven Cache Attacks on AES (short paper)," in *International Conference on Information and Communications Security*. Springer, 2006, pp. 112–121.

[7] J. M. Szefer, "Architectures for secure cloud computing servers," Ph.D. dissertation, Princeton University, 2013.

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

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[10] Kevin Carbotte, "The Next 100 Billion ARM-Powered Devices Will Feature ARM DynamIQ Technology," https://www.tomshardware.com/news/arm-dynamiq-multicore-microachitecture,33947.html, accessed online.

[11] S. Deng, W. Xiong, and J. Szefer, "A benchmark suite for evaluating caches' vulnerability to timing attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 683–697.

[12] Microsoft corp., "Visual Studio App Center," https://appcenter.ms/, accessed online.

[13] Amazon.com Inc., "AWS Device Farm," https://aws.amazon.com/device-farm/, accessed online.

[14] Google LLC, "Google Firebase," https://firebase.google.com/, accessed online.

[15] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 494–505.

[16] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 203–215.

[17] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 549–564.

[18] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "Autolock: Why cache attacks on {ARM} are harder than you think," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1075–1091.

[19] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices." *IACR Cryptology ePrint Archive*, vol. 2016, p. 980, 2016.

[20] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented flush-reload side channels on arm and their implications for android devices," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 858–870.

[21] G. Haas, S. Potluri, and A. Aysu, "itimed: Cache attacks on the apple a10 fusion soc." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 464, 2021.

[22] H. Lee, S. Jang, H.-Y. Kim, and T. Suh, "Hardware-based flush+ reload attack on armv8 system via acp," in *2021 International Conference on Information Networking (ICOIN)*. IEEE, 2021, pp. 32–35.

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

[24] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." in *USENIX Security Symposium*, 2014, pp. 719–732.

[25] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ Flush: a Fast and Stealthy Cache Attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[26] S. Deng, W. Xiong, and J. Szefer, "Analysis of secure caches using a three-step model for timing-based attacks," *Journal of Hardware and Systems Security*, Nov 2019. [Online]. Available: https://doi.org/10.1007/s41635-019-00075-9

[27] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[28] B. L. Welch, "The Generalization of Student's Problem When Several Different Population Variances are Involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[29] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[30] S. Bhattacharya and I. Verbauwhede, "Exploring micro-architectural side-channel leakages through statistical testing," *yet to receive the details*, 2020.

[31] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, pp. 1–29, 2009.

[32] N. V. Smirnov, "On the estimation of the discrepancy between empirical curves of distribution for two independent samples," *Bull. Math. Univ. Moscou*, vol. 2, no. 2, pp. 3–14, 1939.

[33] Android Open Source Project, "Security-Enhanced Linux in Android," https://source.android.com/security/selinux, accessed online.

[34] Android Issue Tracker, "[Android Q Beta] Apps can no longer execute binaries in their home directory," https://issuetracker.google.com/issues/128554619, accessed online.

[35] Amazon.com, "Elastic Compute Cloud (EC2)," Cryptology ePrint Archive, Report 2019/167, http://aws.amazon.com/ec2.

[36] J. Mambretti, J. Chen, and F. Yeh, "Next generation clouds, the chameleon cloud testbed, and software defined networking (sdn)," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*. IEEE, 2015, pp. 73–79.

[37] B. Burgess, "Samsung exynos m1 processor," in *2016 IEEE Hot Chips 28 Symposium (HCS)*. IEEE, 2016, pp. 1–18.

[38] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[39] J. Borghoff, A. Canteaut, T. Güneysu, E. Kavun, M. Knezevic, L. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger *et al.*, "A low-latency block cipher for pervasive computing applications-extended abstract." Asiacrypt, 2012.

[40] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 360–371.

[41] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 675–692.

[42] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache side-channel attacks and time-predictability in high-performance critical real-time systems," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

Shuwen Deng (S'18) received her B.Sc. in Microelectronics from Shanghai Jiao Tong University in 2016. She is currently a Ph.D. candidate at the department of Electrical Engineering at Yale University, working with Prof. Jakub Szefer. Her current research includes developing and verifying secure processor microarchitectures by self-developing timing side-channel vulnerability checking schemes, as well as proposing languages and tools for practical and scalable security hardware and architectures verification.

Nikolay Matyunin (S'20) received his Dipl. in Computer Science from the Lomonosov Moscow State University in 2014. He is currently a Ph.D. candidate at the department of Computer Science at Technical University of Darmstadt, Germany, working with Prof. Dr. Stefan Katzenbeisser. His research interests include covert channels and physical side-channel attacks, security and privacy of mobile and embedded systems.

Wenjie Xiong (S'17) received her B.Sc. in Microelectronics and Psychology from Peking University in 2014. She is currently a Ph.D. student at the department of Electrical Engineering at Yale University, working with Prof. Jakub Szefer. Her research interests comprise Physically Unclonable Functions, and cache side channel attacks and defenses.

Stefan Katzenbeisser (S'98–A'01–M'07–SM'12) received the Ph.D. degree from the Vienna University of Technology, Austria. After working as a Research Scientist with the Technical University of Munich, Germany, he joined Philips Research as a Senior Scientist in 2006. After holding a professorship for Security Engineering at the Technical University of Darmstadt, he joined University of Passau in 2019, heading the Chair of Computer Engineering. His current research interests include embedded security, data privacy and cryptographic protocol design.

Jakub Szefer (S'08–M'13–SM'19) received B.S. with highest honors in Electrical and Computer Engineering from University of Illinois at Urbana-Champaign, and M.A. and Ph.D. degrees in Electrical Engineering from Princeton University where he worked with Prof. Ruby B. Lee on secure hardware architectures. He is currently an Associate Professor in the Electrical Engineering department at Yale University, where he leads the Computer Architecture and Security Laboratory (CASLAB). His research interests are at the intersection of computer architecture, hardware security, and FPGA security.