SmokeBomb: Effective Mitigation Against Cache Side-channel Attacks on the ARM Architecture

Haehyun Cho[1], Jinbum Park[3], Donguk Kim[3], Ziming Zhao[2],
Yan Shoshitaishvili[1], Adam Doupé[1], Gail-Joon Ahn[1,3]
[1]Arizona State University [2]Rochester Institute of Technology [3]Samsung Research
{haehyun, yans, doupe, gahn}@asu.edu, zhao@mail.rit.edu, {jinb.park, donguk14.kim}@samsung.com

ABSTRACT

Cache side-channel attacks abuse microarchitectural designs meant to optimize memory access to infer information about victim processes, threatening data privacy and security. Recently, the ARM architecture has come into the spotlight of cache side-channel attacks with its unprecedented growth in the market.

We propose SmokeBomb, a novel cache side-channel mitigation method that functions by explicitly ensuring a private space for each process to safely access sensitive data. The heart of the idea is to use the L1 cache of the CPU core as a *private space* by which SmokeBomb can give consistent results against cache attacks on the sensitive data, and thus, an attacker cannot distinguish specific data used by the victim. Our experimental results show that SmokeBomb can effectively prevent currently formalized cache attack methods.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Mobile platform security**.

ACM Reference Format:
Haehyun Cho, Jinbum Park, Donguk Kim, Ziming Zhao, Yan Shoshitaishvili, Adam Doupeé, Gail-Joon Ahn. 2020. SmokeBomb: Effective Mitigation Against Cache Side-channel Attacks on the ARM Architecture. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386901.3388888

1 INTRODUCTION

Cache side-channel attacks exploit time differences between a cache hit and a cache miss to infer sensitive data [51]. These attacks are very effective in stealing cryptographic keys (*e.g.*, the secret keys used in AES) from victim programs and even other virtual machines, in tracing the execution of programs, and in performing other malicious actions [5, 25, 46, 49, 52, 55, 59, 60, 63–65].

The bulk of research into cache attacks was started on, and has continued on, the Intel architecture. However, as mobile devices (*e.g.*, smartphones and watches) experience unprecedented growth, the mitigation of cache attacks on non-Intel architectures has drastically risen in importance. Thus, our community needs a defense against cache side-channel attacks that can protect against both shared cache (L2) attacks and dedicated cache (L1) attacks, and can do so regardless of the specific architecture in question. While cache attacks are, generally, more difficult to carry out on the ARM architecture, new techniques have been developed to make them more effective on mobile platforms [31, 46, 61, 62].

The fundamental causes of cache side-channel attacks are twofold: (1) a cache is a hardware resource shared by multiple processes; and (2) there is a noticeable difference in access time between a cache hit and a cache miss. Therefore, to fundamentally solve this problem, a new architecture or cache design is needed. Such hardware-based approaches can provide strong security features against cache attacks with relatively small performance overhead [28, 43, 45, 48, 57]. Hardware-based solutions, however, require considerable cost and time to be deployed in a practical manner, *and no such concerted effort has yet been undertaken.* Thus, current systems remain vulnerable to current cache attacks, and even if a hardware solution is undertaken, legacy devices will not be secured.

However, software-based approaches are relatively cheap and easily deployable: we can deploy them quickly and broadly through software patches. Therefore, understandably, many software solutions have been proposed to mitigate cache attacks [34, 41, 44, 47, 66, 67]. Some techniques target the protection of the *shared CPU cache* (*i.e.*, L2 cache in the ARM architecture), meaning that they fail to protect programs from emergent attacks against the *dedicated core cache* (*i.e.*, L1 cache) [31, 41, 44, 46, 47, 61, 66, 67]. Crane et al. [26] reduce, but does not eliminate, side-channel information leakage by randomizing the program's control flow. Other techniques, including recent work targeting the protection of L1 cache, use specific hardware features available only on *certain Intel processors* and have uncertain efficacy under heavy system load [34]. In analyzing these techniques, we realized that most current cache side-channel protection mechanisms attempt to mitigate attacks by implicitly creating a *private space*—not shared with any other process—in which constant-time (and thus, side-channel-immune) access to sensitive data is assured.

In this paper, we propose SmokeBomb, a software cache side-channel mitigation method for commonly-used CPU cache configuration, that *explicitly* ensures a private space for each process to safely access critical data—the actual L1 cache of the CPU core on which the process is executing. SmokeBomb reserves the L1 cache for a sensitive operation's exclusive use and *denies attackers the ability to find timing differences between used and unused sensitive data.* Without access to measurable time differences, attackers are unable to carry out cache-based side-channel attacks.

MobiSys '20, June 15–19, 2020, Toronto, ON, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7954-0/20/06…$15.00
https://doi.org/10.1145/3386901.3388888

SmokeBomb employs additional OS-level functionality (uninvasively implemented as a kernel module), which has zero performance impact when there is no sensitive data to protect, and negligible impact on the rest of the system. While SmokeBomb requires a recompilation of the sensitive code that needs to be protected, it assists developers in adopting the protection as a compiler extension, requiring developers to *only annotate the sensitive data.*

We demonstrate the effectiveness of SmokeBomb by applying it to two different ARM processors with different instruction set versions and cache models and carry out an in-depth evaluation of the efficiency of the protection. Our experimental results show that SmokeBomb effectively prevents information leakage against known cache side-channel attacks. To our knowledge, SmokeBomb is the first cache side-channel defense that functions on the ARM architecture and covers both the L1 and L2 cache layers. The contributions of this paper are, thus, as follows:

(1) We propose a novel software-based approach to protect against cache side-channel attacks for inclusive and non-inclusive caches based on the creation of a private space in the L1 cache.
(2) We implement SmokeBomb using the LLVM compiler infrastructure and a kernel module that enables its application to most ARM Cortex-A processors. We open source the prototype.[1]
(3) We show how SmokeBomb provides robust protection against cache side-channel attacks, and we demonstrate its effectiveness and efficiency by evaluating it on devices.

2 ARM CACHE ARCHITECTURE

A cache is low-capacity low-latency memory located between a CPU and main memory. Because access to the cache is significantly faster than to main memory, the presence of caches dramatically improves the runtime performance of a system. Modern processors usually have two or more levels of hierarchical cache structure. In the ARM architecture, each core has its own "L1" data *and* instruction cache. In addition, all of the cores share a larger, unified "L2" cache. When the CPU needs data from a specific memory address, and that data is not in a cache (this is termed a *cache miss*), a cache linefill occurs. Otherwise, the CPU loads the data from its own L1 cache, its own L2 cache, or even the L1 cache of other CPUs using the directory protocol [41].

Set Associativity and Cache Addressing. Operations between a cache and main memory are done in chunks of a fixed size, called a *cache line*, for improved performance. A data memory address is divided into three parts: tag, index, and offset. The index determines in which *cache set* the data should be stored. The tag contains the most significant bits of an address, which is stored in the cache along with the data so that the data can be identified by its address in main memory.

A *set-associative cache* is divided into several sets that consist of the same number of cache lines. The cache is called an *N-way associative cache* if a set has *N* cache lines, or *ways*. The data at a specific main memory address can be fetched into any cache line (*way*) of a particular set. In the ARM architecture, caches are always set-associative for efficiency reasons [12].

Either virtual or physical addresses can be used for the tag and index. In ARM CPUs, the L1 data cache is indexed using the physical address whereas the L1 instruction cache is indexed with the virtual address [10, 13]. The L2 cache is usually physically-indexed.

Replacement policy. While the Intel architecture employs the least-recently-used (LRU) replacement policy [32], the ARM architecture generally uses a pseudo-random replacement policy [12]. However, some Cortex-A processors support other cache replacement policies, such as LRU policy and round-robin policy, which can be chosen by the system developer [8, 17, 20, 21].

Inclusiveness. A cache architecture can be categorized based on whether or not a higher-level cache continues to hold data loaded into a lower-level cache. In *inclusive* caches, cache lines of the L2 cache will not be evicted by new data as long as the data is stored in the L1 cache, which is called *AutoLock* in prior work [31]. In *non-inclusive* caches, a line in L2 cache can be evicted to make space for new data even if the line is present in L1 cache. In *exclusive* caches, there is only one copy of data in the whole cache hierarchy—that is, when a line is loaded into L1, it is flushed from L2. Most ARM processors employ a non-inclusive cache [46].

By reviewing the technical reference manuals of the Cortex-A series [6–9, 11, 14–18, 20–23] and performing experiments on our test environments shown in Table 1, we confirmed the inclusiveness of the following Cortex-A CPUs: (1) only A55 is exclusive;[2] (2) A15, A57, and A72 are inclusive;[3] and (3) A53 is non-inclusive.[4] Judging from the manuals, the other CPUs also would use the non-inclusive cache as the Cortex-A53.

3 CACHE SIDE-CHANNEL ATTACKS

Cache side-channel attacks are possible because (1) the cache is a shared resource by multiple processes and (2) there is a noticeable difference in access time between a cache hit and a cache miss. The specific techniques to attack the cache differs based on the attacker's capabilities in three main areas: whether or not the attacker can reliably control the scheduling of the attack process relative to the victim process, the level of the CPU cache that the attack is targeting, and whether or not the attacker process shares memory with the victim process.

Cache Attack Terminology. In cache side-channel attacks, the attacker uses side-channel information, such as access time, to infer which data has been accessed by the victim. Throughout the paper, we use the term *sensitive data* to denote data that is secret or could be used to infer secrets. For instance, the T-tables of the AES algorithm are sensitive data because the access pattern of T-tables can be used to infer the secret key. Because only a (key-dependent) subset of T-table entries are actually used during encryption, not all of the entries will be put into the cache during an encryption operation. We call the subset of sensitive data that is actually put into the cache during execution *key data.* Thus, the attacker's goal is to use a cache side-channel attack to infer which sensitive data is key data. In addition, we refer to the code that uses sensitive

[1] https://github.com/samsung/smoke-bomb

[2] The Cortex-A55 technical reference manual states that "L2: Strictly exclusive with L1-D caches [22]." Also, the Cortex-A55 uses the *private per-core* unified L2 cache [22].

[3] The Cortex-A15, 57, 72 technical reference manuals state that "L2: Strictly enforced inclusion property with L1-D caches [8, 16, 17]."

[4] The Cortex-A53 technical reference manual and the other ones *do not* state the inclusiveness of the L2 cache [6, 7, 9, 11, 14, 15, 18, 20, 21, 23].

Table 1: Test Environments.

CPU (# of cores)	Instruction Set	L1-D Cache	L2 Cache	Inclusiveness	Cache Replacement Policy	Kernel
Cortex-A72 (4)	ARMv7 32-bit	32 KB, 2-way, 256 sets	2 MB, 16-way, 2048 sets	Inclusive	Least-recently-used	Linux Kernel 4.1.10
Cortex-A53 (4)	ARMv8 64-bit	32 KB, 4-way, 128 sets	512 KB, 16-way, 512 sets	Non-inclusive	Psuedo-random	Linux Kernel 4.6.3

data as *sensitive code*. For example, the implementation of the AES algorithm would be sensitive code.

Attacker Memory Access. Shared memory increases memory efficiency of the system by allowing one copy of the memory contents to be shared by many processes. In addition, this feature enables the system to deploy the cache efficiently. For example, the cache addressing scheme described in § 2 ensures that if one process has loaded data of a shared library, other processes using the same library are able to get the data from the cache quickly [12]. Unfortunately, this feature makes shared libraries *inherently vulnerable* to cache side-channel attacks by allowing an attacker to measure the loading time of sensitive data in shared memory.

Attack Scheduling. Cache attacks generally include a *setup phase* and a *measurement phase* carried out by the attacker process. Control over the timing of these phases in relation to the execution of sensitive code by a victim process sorts these attacks into two categories: synchronous and asynchronous attacks [54]. *Synchronous attacks* are possible when the attacker is able to schedule the victim process's sensitive code between the setup and measurement phase. By doing so, the attacker can significantly reduce measurement noise and increase the accuracy of the attack. In an *asynchronous attack*, however, the attacker tries to leak information by relying on the expected execution time of the victim process without control over its execution. The accuracy of such attacks, thus, is much lower than that of synchronous ones.

Attacks on Different Cache Levels. Attackers can target different levels of the CPU cache, depending on the specific circumstances of the attack. Traditional techniques targeted the L2 cache, which is shared by multiple cores in a CPU [44, 47, 66, 67]. Thus, side-channel attacks against the L2 cache can be carried out both synchronously and asynchronously.

To attack the L1 cache, attackers have mostly been confined to synchronous, same-core attacks, as this reduces the L1 cache to a shared space between the attacker and victim processes. Recently, however, Irazoqui et al. [41] demonstrated an attack that uses a feature that allows the exchange of cached data between L1 caches (*i.e.*, *directory protocol*). These techniques allow attackers to target the L1 cache even on multi-core systems, and they represent a challenge that SmokeBomb must overcome. The ARM architecture calls this feature the *AMBA Coherent Interconnect* [19].

3.1 Attack Methods

Evict+Time. This attack method can determine which cache sets have been used by the victim process [30, 46]. In the first step, the attacker measures the execution time of the victim process. Then, the attacker evicts a target cache set and measures the execution time of the victim program again. From the difference in the execution time, the attacker can figure out the cache set and, thus, the memory that it represents has been used by the victim program.

Prime+Probe. This attack is also used to determine specific cache sets accessed by the victim. It has been studied and implemented in various environments [24, 36, 38–40, 46, 49, 53, 61, 64].

In the Prime phase, the attacker occupies a certain range of cache sets by loading their own data. After the victim process has been scheduled, the attacker probes which cache sets are used by the victim. Because the ARM architecture uses a set-associative cache, a set consists of several *ways*. For example, the L2 cache of the Cortex-A53 has 16 *ways*. Thus, the attacker decides a set has been used if one of the *ways* was refilled by other data, which might not be loaded by the victim. The pseudo-random cache replacement policy of the ARM architecture makes the Prime+Probe attack much more difficult [46].

Flush+Reload and Evict+Reload. These attacks operate by measuring the data reload time of the cache, which are available only when the attacker shares memory with the victim process [37, 41, 60, 62]. The attacker must map a target shared object into its address space. Then, the attacker flushes/evicts a cache line within the shared area. In the Reload phase, attackers reload previously flushed/evicted data after waiting for the victim to access the shared object and checks the time it takes to reload. Based on this reloading time, attackers can infer if the victim accessed the data.

Commonly, attacks can check whether specific data is in the cache after the execution of the victim process. These attacks are more accurate and easier to conduct than the Prime+Probe and Evict+Time attacks. Also, the simplicity of these methods makes asynchronous attacks possible. The only difference is in the way for flushing the data from the cache before the victim has been scheduled. If the flush instruction is available, the attacker could flush data using the virtual address. Otherwise, the attacker needs to evict the data by loading other data [35, 46].

Flush+Flush. This attack utilizes the timing difference of the flush instruction [33, 46]. The execution time of the flush instruction is different depending on whether data is cached or not. If data is cached, the flush instruction takes more time to execute. The first phase is identical to the Flush+Reload attack. In the last step, however, the attacker flushes the data once again to check whether the data has been accessed by the victim.

3.2 Our Threat Model

In this paper, we consider multi-core computing environments that use the inclusive and the non-inclusive caches on the ARM architecture, in which processes, including malicious ones, use shared libraries, such as OpenSSL. The attacker can use all of the cache side-channel attacks mentioned in § 3.1 to extract secret information including cryptographic keys. We do not consider recently proposed Prime+Abort, because the ARM architecture currently does not support transactional memory, and therefore the attack is not available [27]. In addition, we assume the worst case scenario in which attackers can use the flush instruction on ARMv8 CPUs, even though the ARMv8 architecture restricts userland applications from executing the flush instruction by default. Assuming this worst-case attack model allows us to develop as strong a defense as possible, which we present in the rest of the paper.

Throughout this paper, we discuss and perform experiments on two environments as listed in Table 1.

4 OVERVIEW

We present, implement, and evaluate SmokeBomb—a cache side-channel mitigation method that prevents attacks on every cache level mentioned in § 3, especially for inclusive and non-inclusive caches. With SmokeBomb, an attacker attempting to measure data access times of sensitive data will be met with consistent timing results for all sensitive data, and *will thus be unable to infer which sensitive data is actually used.* To this end, we design SmokeBomb to achieve particular defensive goals:

(D1) It defends against cross-core L2 data cache attacks.
(D2) It defends against directory protocol based cross-core L1 data cache attacks.
(D3) It defends against single-core L1 data cache attacks.

To accomplish these goals, SmokeBomb first instruments applications (during compilation time) to find and patch any sensitive code (§ 5), which puts the sensitive data involved under SmokeBomb's protection (where the developer annotates the sensitive data in the source code). Then, SmokeBomb carries out three steps, at different points before, during, and after the execution of the patched sensitive code. Figure 1 depicts the effect of SmokeBomb in terms of the amount of data loaded in the cache, and we reference it through the following description.

Preloading Sensitive Data (§ 6): Before executing the sensitive code (at time-point t_1 in Figure 1), SmokeBomb preloads the sensitive data into the L1 cache. By preloading the sensitive data, SmokeBomb prevents the attacker from identifying which specific cache *sets* have been used by the victim (*e.g.*, Prime+Probe).

Preserving Sensitive Data (§ 7): SmokeBomb ensures that the preloaded data exists in the private L1 cache *throughout the sensitive code's execution* (t_1~t_2 in Figure 1). In a *non-inclusive* cache, the preloaded data will be maintained only in the L1 cache. In an *inclusive* cache, the preloaded data will be maintained in the L1 and the L2 cache. Thus, an *asynchronous* attacker from another core cannot infer any information regarding the key data.

Flushing Sensitive Data (§ 8): At the termination of the sensitive code (t_2 in Figure 1), SmokeBomb flushes all data from the cache so that no information on what data was used is revealed to an attacker (between time points t_2 and t_3).

Without SmokeBomb, as the sensitive code executes from t_1 to t_2, the amount of key data in the cache increases gradually. At t_2, all the key data may have been fetched into the cache and will stay there until being replaced gradually as shown from t_2 to t_3. Thus, attackers can infer which sensitive data is the key data from t_1 to t_3. With SmokeBomb, however, entire or only a certain amount of sensitive data is fetched into the cache when sensitive code starts execution at t_1 and flushed when sensitive code exits at t_2. Consequently, SmokeBomb will cause consistent timing results for all sensitive data. We note that SmokeBomb is not able to defend instruction cache attacks, which will be discussed in § 10.

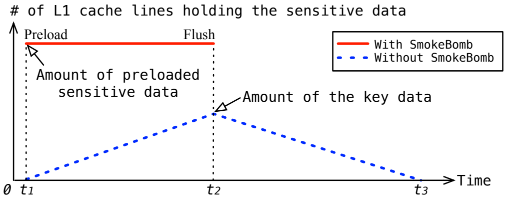


Figure 1: The difference in cache usage with and without SmokeBomb. The x-axis denotes the time of code execution and the y-axis represents the number of cache lines holding sensitive data. Any observable changes on the y access represent a potential cache side-channel attack vector.

Instructions Used. SmokeBomb uses the data prefetcher by calling preloading data (PLD) instruction to preload the sensitive data. We have confirmed that *all* the Cortex-A processors support PLD instructions and their effects through the technical reference manuals of each Cortex-A processor [6–11, 13–18, 20–23].

The DC CISW instruction flushes a designated cache line in a specific cache level [10, 13]. It requires a set number, a *way* number, and a cache level as operands. SmokeBomb uses the DC CISW instruction to bypass the psuedo-random replacement policy. Also, this instruction is used to keep sensitive data away from the L2 cache in inclusive caches.

SmokeBomb APIs. In applications, SmokeBomb is initiated and finalized by the following two APIs in user-space programs: (1) `init_smokeBomb`, and (2) `exit_smokeBomb`. The `init` API has two parameters: the start address and the size of the sensitive data. When this API is called by a process in the system, SmokeBomb preloads the sensitive data into the cache and changes the scheduling policy of the process. The `exit` API, which flushes the sensitive data from the cache and restores the scheduling policy, does not have any parameters. Between these two APIs—during the execution of the sensitive code, SmokeBomb-defined instructions execute to preserve the preloaded sensitive data in the L1 cache.

SmokeBomb-defined Instructions. Because most ARM processors have no cache locking instructions, we utilize the undefined instruction exception handler to implement our own cache locking instructions that are software-emulated by the handler. SmokeBomb finds and patches cache-relevant instructions (such as LDR or STR) that access *non-sensitive* data, because they can change the cache state by fetching *non-sensitive* data to the L1 cache. At runtime, those instructions will be trapped and handled by SmokeBomb's exception handler. We call the patched instructions as xSB instructions, such as LDRSB, which performs the intended operation of the original x instruction, but also ensures the preservation of the sensitive data only in the L1 cache.

5 INSTRUMENTING SENSITIVE CODE

SmokeBomb requires two modifications to the sensitive code. First, the two API calls mentioned in § 4 must be inserted before and after the sensitive code. Second, cache changing instructions in the sensitive code must be modified to SmokeBomb-defined instructions which have opcodes that do not exist in the ARMv7 and ARMv8 instruction sets. SmokeBomb software-emulates them through the undefined instruction handler.

SmokeBomb automates this process for developers by requiring only an annotation of the sensitive data (in our implementation, using attribute syntax annotations [1]), and SmokeBomb derives all necessary code modifications during compilation. Developers can annotate static data directly or a data pointer for dynamically allocated data. Note that, as SmokeBomb is a *protection* mechanism, we rely on the developers to identify the data that *should* be protected. However, approaches exist for the automated identification of such data, and this compilation process could be modified to automatically insert even the annotations themselves [56]

Provided these annotations, SmokeBomb uses a compiler extension (*i.e.*, an LLVM pass) to instrument the application. First, SmokeBomb identifies the sensitive code, which is straightforward due to the annotation. By analyzing each IR instruction, SmokeBomb can identify all memory operations that reference (annotated) sensitive data. Specifically, when data is annotated, SmokeBomb can identify all memory operations that reference annotated sensitive data by analyzing operands of load and store instructions. When a pointer is annotated, SmokeBomb checks whether memory operations dereference the annotated pointer or not. All such instructions are identified as *sensitive code.*

Once sensitive code is found in a function, SmokeBomb identifies the dominator and post dominator of the basic blocks in which the sensitive code exists. It then inserts a call instruction which invokes the `init` API at the dominator node (and specifies the reference to the sensitive data in the API call) and another call instruction for the `exit` API at the post dominates node. If there are other call instructions in the basic blocks, SmokeBomb additionally places instructions calling the `exit` API and the `init` API before and after the other call instructions, respectively.

When the size of the sensitive data is larger than the L1 cache, SmokeBomb takes the first part of the sensitive data as *selected* sensitive data that will be preloaded and preserved. The selected sensitive data is bigger than *a way* of the L1 cache so that the selected part can cover all *sets* for preventing attacks which try to identify which set is used (*e.g.*, Prime+Probe). Note that *unselected* sensitive data is neither preloaded nor preserved: it is explicitly kept *out* of the cache, achieving the same protection.

Next, SmokeBomb patches all cache changing instructions that are located between the two APIs. When the size of sensitive data is smaller than the L1 data cache, it only patches instructions that access *non-sensitive* data, to preserve sensitive data in the L1 cache by enforcing additional cache maintenance operations after these instructions execute. The insight here is that instructions that access *sensitive* data do not change the cache state of that data. By patching only *non-sensitive* instructions, SmokeBomb can avoid unnecessary undefined instruction exceptions, thus minimizing performance degradation. However, if the size of sensitive data is larger than the L1 data cache, SmokeBomb patches *all* cache changing instructions, because it cannot statically determine which sensitive code might access the *unselected* sensitive data.

6 PRELOADING SENSITIVE DATA

Before sensitive code executes, SmokeBomb preloads the sensitive data (or, if the sensitive data larger than the L1 cache, the selected sensitive data). One way to preload data into the cache is to simply access it (e.g., using the `LDR` instruction). However, this is slow, as the CPU will wait until the data actually arrives in a register or memory. For better performance, SmokeBomb employs a hardware feature called *data prefetching*, by using the preloading data (PLD) instruction, which is available in the Cortex-A series. SmokeBomb triggers the prefetcher by using the `PLD` instruction in ARMv7 and the `PRFM PLD` instruction in ARMv8 [10, 13]. For brevity, we use "`PLD` instructions" to refer to both instruction forms. `PLD` instructions execute much faster than `LDR` to fetch data into the cache.

Bypassing the Pseudo-random Replacement Policy. The `PLD` instruction loads data (of the size of a cache line) from memory to the cache. However, with ARM's pseudo-random cache replacement policy, sensitive data loaded earlier in the process might be evicted by sensitive data loaded later in the process. SmokeBomb must ensure that this does not happen, so that the entire sensitive data can be safely loaded. Our experiments on both testing environments reveal that the pseudo-random replacement policy only triggers when there is no empty cache line available. If we can make one cache line available by flushing it in the set that the data is supposed to reside, the data is guaranteed to occupy the empty cache line instead of evicting any other line in the set.

We conducted preliminary experiments to confirm this behavior for both the L1 and L2 caches: we first flush a particular cache line of a set using the `DC CISW` instruction, which takes a set number and *a way* number as operands [10, 13]. Then, we load data using the `PLD` instruction from an address whose index fields match the set number. We then flush the same cache line again. At the last step, we load the same data again on the same core and check the cache refill event using the performance monitor unit (PMU) [10, 13]. If a cache refill event occurs, the data has been loaded in the cache line we selected, and vice versa.

Keeping Sensitive Data Away from L2 Cache. For non-inclusive caches, `PLD` instructions load data into L2 cache automatically as well, which enables cross-core cache attacks because an L2 cache line is evictable. To prevent this, we use the same approach described in the previous subsection to ensure that sensitive data is always loaded into a known *way* in the cache. Then, we flush the sensitive data from the L2 cache. When SmokeBomb preloads the sensitive data, other processes' data in the L2 cache can be evicted. To minimize this impact, SmokeBomb uses only the last *way* of the L2 cache. For inclusive caches, SmokeBomb loads the sensitive data into the L2 cache.

Figure 2 illustrates how SmokeBomb preloads the sensitive data into the cache, bypassing the pseudo-random replacement policy. We first translate the virtual address of the sensitive data to physical address and compute its set number for L1 and L2 cache respectively, because the cache is physically indexed and tagged. We then flush the one way of this set in L1 and L2 cache respectively to make room for sensitive data as shown in (1) – (2) of Figure 2. For convenience, we flush from the last *way* of L1 to the first *way* in this step. Then, we use `PLD` instructions to load data into the cache as shown in (3) of Figure 2. Because there is one cache line available in L1 and L2 cache, the data goes to that available line. We flush the just loaded L2 cache line as shown in (4) of Figure 2. We repeat this procedure until the entire sensitive data or the selected sensitive data is loaded into the cache. In this loop, if a *way* of the L1 cache is fully occupied with sensitive data, we start to fill the previous *way*

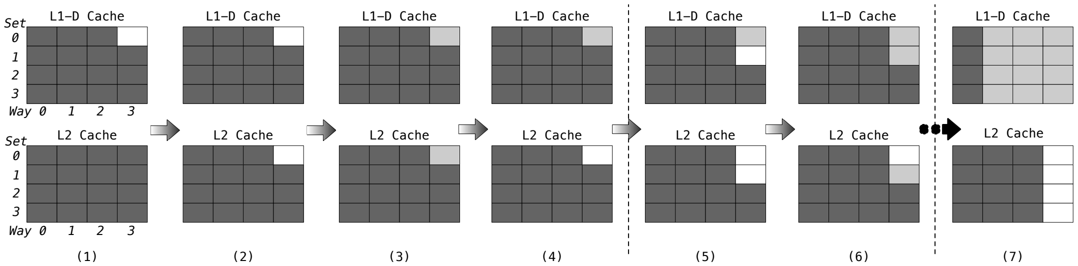


Figure 2: Example of the cache state changes in the non-inclusive cache model when preloading the sensitive data. In this figure, we assume that there are 4 sets and 4 *ways* in both the L1 and L2 caches. Light gray means *sensitive data* and dark gray means *normal data.* White represents a *flushed cache line.* (1): Flush an L1 cache line. (2): Flush an L2 cache line of the last *way.* (3): Load the sensitive data. (4): Flush the sensitive data from the L2 cache. (5)–(7): Repeat the prior 4 steps.

instead as shown in (5)–(7) of Figure 2. For inclusive caches, we can omit to flush an L2 cache line—step (4) of Figure 2. SmokeBomb changes a *way* of the L2 cache similar to the L1 from the last *way* in descending order, if a *way* is full with sensitive data.

7 PRESERVING SENSITIVE DATA

After preloading the sensitive data (or the selected sensitive data, if the sensitive data is larger than the size of the L1 cache), it is critical to preserve it in the cache during the execution of sensitive code to prevent side-channel attacks. Additionally, unselected sensitive data must *not* be in the cache. By preserving only preloaded sensitive data, SmokeBomb achieves a consistent cache state throughout the execution of the sensitive code.

In our experimentation environment, we tested how many cache lines were evicted during AES encryption after preloading the T-tables as the sensitive data into the L1 cache. We used OpenSSL (v.1.0.2) and 128-bit AES algorithm to encrypt an 8-byte plaintext. Even with an intentionally-chosen small plaintext, the results show that around 89% of cache lines holding non-key sensitive data were evicted during the encryption procedure, which opens the door for cache side-channel attacks. The experiments clearly demonstrate the need to preserve sensitive data in the cache.

Unfortunately, most ARM Cortex-A series processors do not support hardware cache locking techniques. Thus, to preserve the sensitive data in the L1 cache, SmokeBomb must hook all instructions that could influence the state of the cache after preloading the sensitive data. If the sensitive data is smaller than the L1 cache, SmokeBomb only needs to hook cache changing instructions that access *non-sensitive* data, because these instructions can evict the preloaded sensitive data from the L1 cache to the L2 cache. When the sensitive data is larger than the L1 cache, SmokeBomb must hook all cache instructions that occur in sensitive code regardless of which data they access.

To design a software cache locking technique, we use the undefined instruction exception handler, which can be used to implement custom "soft-instructions."

Handling xSB Instructions. SmokeBomb installs an undefined instruction handler for each `xSB` instruction. We use a handler for `LDRSB` to illustrate how the handlers work with the non-inclusive cache model (as shown in Figure 3).

SmokeBomb first loads data referenced by the original instruction. If the address of this data is already in the L1 cache, no matter if the data is sensitive or not, the handler returns immediately to the sensitive code. This is because if *non-sensitive* data is already in the cache after preloading the sensitive data, it means the data exists in the cache with the sensitive data. We determine if the data is in the L1 cache by checking the L1 data cache refill event. If the event did not occur (*i.e.*, an L1 cache hit occurred), the memory system does not fetch the data from the main memory. Consequently, the preloaded sensitive data is still only in the L1 cache.

If the data is neither non-sensitive data that is already in the L1 cache nor preloaded sensitive data, the data will be fetched into the L1 and L2 caches as shown in (1) of Figure 3, which may result in the eviction of a cache line where the sensitive data or selected sensitive is stored. Because we cannot determine which *way* has been evicted due to the pseudo-random replacement policy, SmokeBomb simply reloads the sensitive data *in the set* as in the preload procedure. However, if the address of the loaded data is not congruent with any preloaded sensitive data, SmokeBomb returns to the sensitive code without reloading.

To reload the sensitive data, SmokeBomb flushes it located in the *set* using its virtual address. Once the sensitive data has been evicted from the L1 cache, SmokeBomb cannot know which *way* of the L2 cache has the data. Therefore, SmokeBomb entirely removes the sensitive data in the *set* from a cache as shown in (2) of Figure 3. Then, SmokeBomb reloads the sensitive data following the preloading method to fill the *set* again as shown in (3) – (7) of Figure 3. As a consequence, the process has the same sensitive data that was preloaded only in the L1 cache.

In the inclusive cache, everything is identical except that we omit flushing an L2 cache line when reloading the sensitive data.

Handling Preemption. Modern operating systems have a preemptive kernel and provide preemptive multitasking features. These systems allow scheduled processes to execute only for a time slice. For example, the Completely Fair Scheduler (CFS), which is the default scheduler of the Linux kernel, interrupts a process when the time slice of its thread is expired. Context switches can also occur when a thread voluntarily yields control of the CPU by making system calls, such as `sleep` and `yield`.

Unfortunately, a context switch can cause an attacker process to be executed by the core that was previously running the sensitive code, allowing it to influence the state of the L1 cache. To avoid potential attacks stemming from this phenomena, SmokeBomb must do one of two things during context switches: (1) it must flush

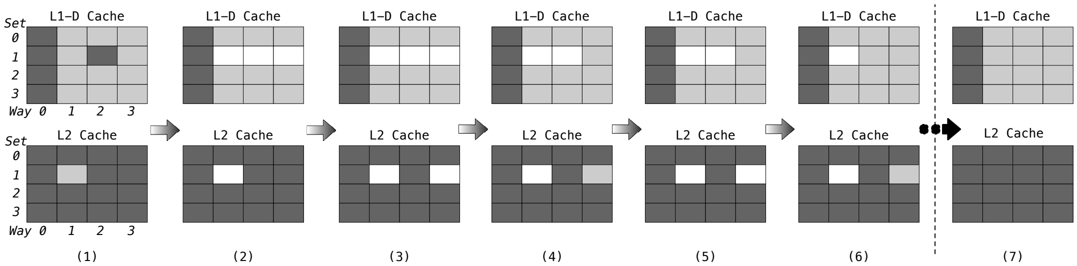


Figure 3: Example of the cache state changes in the non-inclusive cache model when SmokeBomb preserves the sensitive data. In this figure, we assume that there are 4 sets and 4 *ways* in both the L1 and L2 caches. Light gray means *the sensitive data* and dark gray means *normal data.* White represents a *flushed cache line.* (1): Data, which is not in the L1 cache, is loaded. (2): Flush the sensitive data located in the set of the L1 cache. (3) – (7): Refill the set with the sensitive data again.

(on preemption of sensitive code) and re-preload (on resumption of sensitive code) the sensitive data or (2) it must prevent preemption from happening in the first place. The latter represents minimal change to the running system itself.

SmokeBomb overcomes this challenge by executing the sensitive code of a process on the same core until it returns (from `init_smokeBomb` to `exit_smokeBomb`). This *guarantees* that the L1 cache subordinated to the core is not being used by any other processes while executing the sensitive code between the two APIs.

To prevent preemption, SmokeBomb temporarily changes the scheduling policy of only *the single process* when it starts to execute the sensitive code (other processes on the system continue running under the default scheduling policy). Unlike kernel threads, which can manipulate the preemption strategy itself, user-level threads cannot be free from preemption. However, a user-level process can run until it relinquishes the CPU voluntarily by using the First-In, First-Out (FIFO) scheduling policy with the highest static priority. Among the available scheduling policies in the Linux kernel, the FIFO scheduling is the only policy that will not schedule a thread in the time slice manner [4].

It is worth noting that SmokeBomb is only activated in a function where sensitive code exists. If other functions are called from the function in which SmokeBomb is started, `exit_smokebomb` call will be invoked, restoring the scheduler to the default setting. When a thread returns to the function, SmokeBomb is activated again. While this design choice increases the latency on the single application that executes sensitive code, it minimizes the performance impact on the rest of the system.

8 FLUSHING SENSITIVE DATA

After the sensitive code finishes or when the protected process is scheduled out, the entire sensitive data will be gradually evicted if SmokeBomb does not flush it. Even though it seems harmless to leave sensitive data in the cache after the sensitive code terminates, we designed experiments to verify the necessity of flushing: in particular, whether cache exploitation by an attacker is possible when the LRU replacement policy is used. Usually Cortex-A series use the pseudo-random replacement policy, but the LRU policy can be chosen alternatively, and ARM Cortex-A57 and A72 processors employ the LRU replacement policy for the L1 cache by default [8, 16, 17, 17, 20, 21].

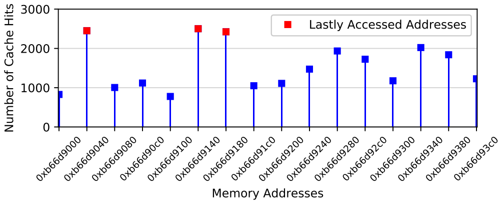


Figure 4: The cache attack results of exploiting the LRU cache replacement policy on the Cortex-A72. The results show that the remaining sensitive data in a cache can be resulted in the key data leakage.

We conducted experiments on the Cortex-A72, which uses the LRU replacement policy for the L1 cache. The victim then terminates itself right after accessing three different memory addresses. Then, the attacker loads data that is congruent with the sensitive data to evict recently used cache lines. At the last step, the attacker checks access times of the sensitive data.

We ran this experiment 3,000 times. Figure 4 shows the attack results where the three addresses that the victim actually accessed have the largest number of cache hits (*red squares*) among the preloaded sensitive data. *Blue squares* stand for the preloaded memory addresses that the victim did not access, which have lower number of cache hits than *red squares.* To protect against this attack, SmokeBomb flushes the sensitive data from cache upon termination of sensitive code to prevent information leakage.

9 EVALUATION

Our experimental environments consist of a Samsung Tizen device and Raspberry PI3 using the Cortex-A72 and A53 processor models, respectively, as listed in Table 1. These devices have different instruction sets: ARMv7 and ARMv8. We implemented proof-of-concept prototypes of SmokeBomb for both instruction sets. The prototypes consist of two parts: (1) an LLVM pass with a binary patching tool and (2) a loadable kernel module. SmokeBomb, which can be deployed without requiring changes to the operating system beyond loading the kernel modules and can be adopted by developers by annotating sensitive data in their applications (or using an approach for automatic identification of it, such as CacheD [56]).

Of the discussed attacks, we evaluate against Flush+Reload and Evict+Reload for the Cortex-A53 (non-inclusive cache) and the Cortex-A72 (inclusive cache), respectively, because these are the fastest and most accurate attack methods. We also used Prime+Probe. Except when Prime+Probe is used, all experiments were conducted on a *cross-core* environment, using two processes: the attacker and the victim. By utilizing a multi-core environment, SmokeBomb's defense against the directory protocol of the ARM architecture is evaluated as well [41]. Flush+Flush was not used for the evaluation because the effectiveness of the method and results are very similar to Flush+Reload and Evict+Reload. Also Evict+Time was not evaluated, because SmokeBomb always loads and flushes the same amount of sensitive data into the cache and the sensitive data does not exist in the cache when SmokeBomb is inactivated, which implies the attack is unavailable.

For the Cortex-A53 processor, which uses the non-inclusive cache, we assume the flush instruction is unlocked for user-land applications, *which strengthens the attacker and makes SmokeBomb's job more difficult.* The Cortex-A72 processor employs the inclusive cache, known as AutoLock, and thus data cannot be evicted by other processes while it is in the L1 data cache [31]. Furthermore, the flush instruction is *not available* in user-land, because the device uses the ARMv7 instruction set. To conduct cache side-channel attacks on the Cortex-A72 processor, we use the two different assumptions introduced by Green et al., depending on the attack method [31]. The victim process provides several services to other processes, and the attacker can request the services. This assumption makes the victim perform the L1 data cache line evictions itself by requests from the attacker. Eventually, the sensitive data of the targeted service can be evicted by the other services requested by the attacker so that Evict+Reload is possible. For Prime+Probe on the Cortex-A72, we assume the attacker and the victim run on the same core. This assumption is theoretically possible, because preemption is disabled only when the sensitive code is executing.

Throughout this section, we present a number of figures describing the difference in cache measurement opportunities for attackers with and without SmokeBomb. In these figures, attack results are shown by *blue squares* in all figures for the non-SmokeBomb case, and by *red circles* for the results of SmokeBomb's application.

9.1 Effectiveness of L1 Cache as a Private Space

By using the cache refill event of the PMU, we tested if the approaches proposed in § 6 and § 7 to preload sensitive data to cache and preserve it in L1 cache alone works in the non-inclusive cache model using the following two experiments [10, 13].

(1) We first loaded 8 KB of data using the `PLD` instruction. Second, we flushed L2 cache using the `DC CISW` instruction. Third, we loaded the same data again with the `LDR` instruction, checking the L2 cache refill event. As expected, no L2 cache refill event occurred because all data accesses triggered L1 cache hit.

(2) In the second experiment, the first and the second steps are as same as in the first experiment. The third step was done by *a different core* and used the `PRFM PLDL2KEEP` instruction for loading the data into L2 cache. This instruction does not fetch data to L1 cache but only to L2 cache for data preloading. The L2 cache refill event occurred, which confirmed that the data was successfully flushed in step 2. If the L2 cache had the data, the event counter would not increase. These experiments clearly demonstrate that SmokeBomb can keep sensitive data in L1 data cache alone.

9.2 Security Analysis

Non-inclusive Caches. SmokeBomb achieves each defensive goal described in § 4: **D1**—by keeping the sensitive data away from L2 cache, the attacker cannot observe any sensitive data in L2 cache. If sensitive data exists in L2 cache, the other processes can evict the sensitive data, which in turn results in key data leakage; **D2**—by preloading the sensitive data and keeping the sensitive data during sensitive code execution, the attacker cannot find the key data using the directory protocol; **D3**—by flushing the sensitive data and protecting the sensitive code from preemption.

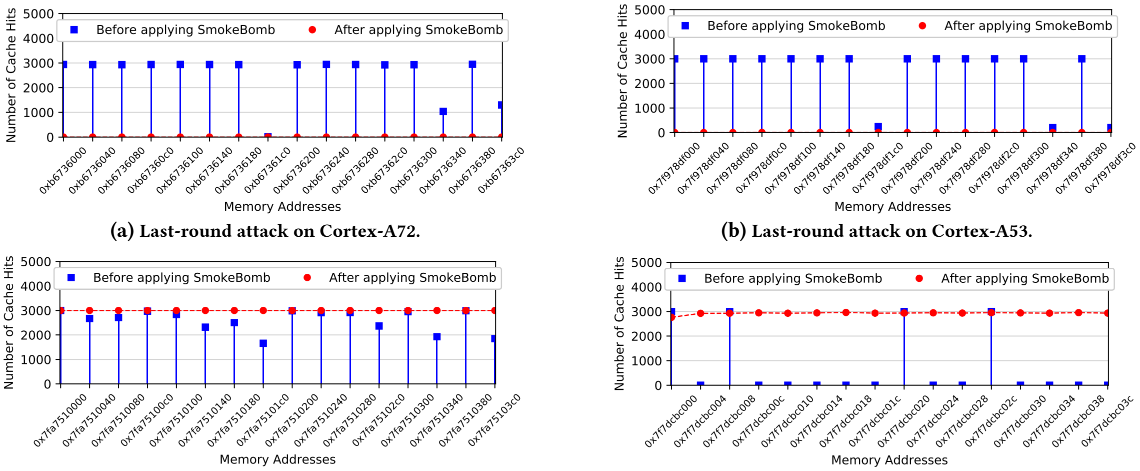
Inclusive Caches. D2 and **D3** are guaranteed in the non-inclusive cache model. However, SmokeBomb does not need to drive the sensitive data out from L2 cache to achieve **D1**. In inclusive caches, the sensitive data cannot be evicted from L2 cache as long as the sensitive data is in L1 cache. Therefore, SmokeBomb can achieve **D1** for the inclusive cache model by keeping the sensitive data in a cache until the sensitive code execution finishes.

The size of protected data. SmokeBomb can provide the same level of defense even when the size of the sensitive data is larger than L1 data cache. If SmokeBomb detects large sensitive data at compile time, it marks a subset of the sensitive data as *selected* sensitive data, and this data is preloaded and preserved. Next, while SmokeBomb is activated at runtime, it flushes all *unselected* sensitive data *out* of L1 cache during the same operation that maintains the selected sensitive data *in* L1 cache. The *unselected* sensitive data, thus, cannot remain in the cache. Consequently, SmokeBomb can always produce consistent results against cache side-channel attacks by preserving the *selected* sensitive data only. However, with caching essentially disabled for the unselected sensitive data, operations on this data will understandably be slow.

Against asynchronous Flush+Reload/Flush attacks. Sensitive data leakage might occur even with the use of SmokeBomb, which must meet the following conditions: (1) immediately after the preloading phase, an attacker can flush the sensitive data on another core and (2) the attacker can reload/flush the data, checking access/flushing times, before the flushing phase. The attack results will be as follows: (1) if the data is in a cache, the attacker will think that it is the key data, however, the data could be data that the sensitive code loaded (the key data) or data that has been reloaded by SmokeBomb (not the key data); (2) if the data is not in a cache, the attacker will think that it is not the key data, however, the data could be data that the sensitive code did not load (not the key data) or data that has been flushed by SmokeBomb (the key data). As the possible attack results show, such attacks would have false-positive errors caused by SmokeBomb, and we believe that the attacks would be extremely difficult to trigger. Also, ARM CPUs do not support a flush instruction except for ARMv8-A CPUs, and the flush instruction is typically not available to user-land applications. We note that, except for the previously discusses case, SmokeBomb can prevent all other cases of asynchronous attacks.

9.3 Case Studies

Case 1: OpenSSL—AES algorithm. The AES implementation of the OpenSSL library is a well-known target for cache side-channel attacks targeting its T-tables [42, 54]. AES T-tables are pre-computed lookup tables used to get a round key for each round of the AES



(a) Last-round attack on Cortex-A72.

(b) Last-round attack on Cortex-A53.

(c) Last-round Prime+Probe attack on Cortex-A72.

(d) One-round attack on Cortex-A53.

Figure 5: The attack and protection results on the AES algorithm. In (a), (b), and (d), the Evict+Reload was used on the Cortex-A72 and the Flush+Reload was used on the Cortex-A53. (c) is the results when the Prime+Probe was used on the Cortex-A72

algorithm. There are four 1 KB T-tables, for a total of 4 KB of sensitive data. If the secret key length is 128-bits, AES encryption and decryption processes have 10 rounds and the key is expanded into 10 round keys as well. This key expansion uses lookups against the T-tables, and determining these lookups via a cache side-channel allows an attacker to recover key data. We used two well-defined attack methods for this experiment: the last-round attack [42] and the one-round attack [54]. With the last-round attack method, it is possible to recover the full secret key. For the one-round attack, we can recover 4 bits of every key byte, since our experimental devices have a 64-byte cache line.

We first performed the last-round attack [42] and the one-round attack [54] without SmokeBomb. The attacks were conducted using the 128-bit AES algorithm (version 1.0.2 of the OpenSSL library). To demonstrate the effectiveness of SmokeBomb, we annotated the 4 KB T-tables of the OpenSSL library as the sensitive data (requiring four lines of code to annotate each of the T-tables), then SmokeBomb was applied to the library automatically.

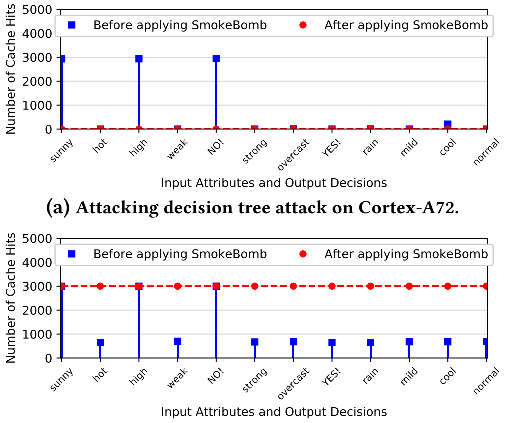
In the last-round attack scenario, the attacker checks the cache state before and after the victim process executes the AES encryption function. Figure 5a and 5b show the attack results of Evict+Reload and Flush+Reload. The attacker can distinctly identify the addresses accessed by the victim without SmokeBomb. Without SmokeBomb's defense, we successfully recovered the secret key after 150 iterations of the attack. However, after SmokeBomb was applied, the attacker cannot observe any timing differences for all entries of the T-tables on both test devices, as shown in Figures 5a and 5b.

Similarly, Figure 5c shows the Prime+Probe attack results, in which we also can identify memory addresses accessed by the victim. The protection results in Figure 5c seem (to the attacker) to indicate that every entry of the T-tables was accessed by the victim. This is because *all cache sets where the sensitive data can be loaded have been occupied in the preloading step*. Thus, the attacker cannot understand *what data was accessed* and *cannot differentiate key data from sensitive data*. We ran the attacker and the victim processes on different cores concurrently.

To simulate the one-round attack, we sent a signal from the victim to the attacker process so that the attacker can perform the Flush+Reload attack after the first round of the AES encryption function—the attacker flushed the sensitive data before the `init` API executes and reloaded the sensitive data after the first round. Also, to avoid unnecessary impediments for conducting the attack, we paused the victim process before the second round—in the middle of the sensitive code execution. This makes *defense more difficult* by giving the attacker an advantage. With the one-round attack, we cracked half of the secret key over 500 iterations of AES encryption, on average. Figure 5d shows that the attack can reveal the T-table entries used in the first round of the AES encryption function.

Conversely, attacks against the SmokeBomb-protected implementation resulted in cache hits for *all* entries—successfully protecting the sensitive data against attacker measurement. This prevention result shown in Figure 5d implies that the key data can be revealed by the attack using the directory protocol *unless* SmokeBomb preserves the sensitive data. After the first round is finished, the victim process still holds the sensitive data in the L1 cache but not in the L2 cache for executing the next rounds. However, access times to the sensitive data must be faster than the L2 cache miss by means of the directory protocol. Thus, the attacker has no choice but to think there was an L2 cache hit—the victim process has used that data.

Case 2: Decision Tree. A decision tree algorithm is used to make a decision according to some input data (called the attributes). Parameters are attributes and the output is the algorithm's decision. Each node of the decision tree is a point where an attribute is tested and a branch is taken according to the result of the test. A leaf node represents a final decision made by the attributes. Because different memory addresses are accessed depending on the attribute, this can result in information leakage via cache side-channel attacks [50].



(a) Attacking decision tree attack on Cortex-A72.

(b) Attacking decision tree on Cortex-A53.

Figure 6: The attack and protection results on the decision algorithm. Evict+Reload was used on the Cortex-A72 and Flush+Reload was used on the Cortex-A53.

For this experiment, we created a decision tree using the ID3 algorithm [29]. We also implemented a shared library which provides a service using the decision tree. The attack scenario is as follows: the victim calls the function within a shared library to get a result from the decision tree. To call the function, the victim needs to select specific information as attributes. A set of attributes is used as a parameter of the function. The attacker tries to identify the attributes selected by the victim and the decision made by the tree.

SmokeBomb was applied in the function that traverses the decision tree by annotating the tree as sensitive data (one line of code change). Because each of the different nodes tests the unique attributes and makes the final decision, there is a one-to-one correspondence between memory addresses of the nodes and attributes (or the final decision). Without protection, the attacker can clearly figure out the input records and the final decision as shown in Figure 6. SmokeBomb forces a consistent cache state for the sensitive data, and thus, the attacker cannot classify data as key data using access time. Figure 6b particularly shows the results of an attack in the middle of sensitive code execution. We simulated the attack to reload the sensitive data before the `exit` API executes. The attack results are all cache hits because of the *directory protocol*, which indicates that the sensitive data is fully preserved.

Case 3: Large sensitive data. We show the effectiveness of SmokeBomb's defense when it protects application with sensitive data larger than L1 cache. In these experiments, the victim accesses 48 KB of sensitive data using a regular pattern and the attacker uses the Flush+Reload attack. The sensitive data consists of 48 entries and each data entry is separated by 1 KB (one line of code change to annotate the sensitive data). SmokeBomb selects only the first 8 KB as *selected sensitive data* if the sensitive data is larger than the L1 data cache. We conducted the attack in such a way that the attacker can check the data reloading time after the victim finishes accessing the sensitive data (before flushing it). Figure 7 shows the results: the attacker cannot infer the actual access pattern, only seeing cache hits on the first 8 entries and cache misses on the rest. This consistency protects against cache side-channel attacks *even when the sensitive data is larger than the L1 cache.*

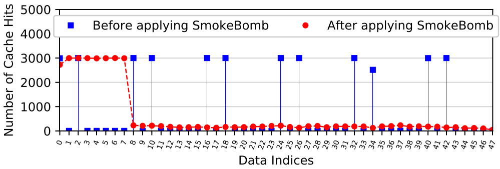


Figure 7: The attack and protection result on the large sensitive data. Flush+Reload was used on the Cortex-A53 processor. After applying SmokeBomb, we cannot find the access pattern.

9.4 Performance

Performance of software instructions. We evaluated the overhead of each xSB instructions emulated by SmokeBomb to keep the sensitive data in L1 cache (as discussed in § 7). For the convenience of the experiment, we implemented a function that has only one instruction (either loading or storing data from/to a memory address) and measured its execution time in nanoseconds. We cannot measure CPU cycles directly because the cycle counter does not increase while execution is in the exception handler. Table 2 shows the execution times of xSB instructions on average across 5,000 executions with execution times of the orginal instructions.

Execution times of xSB instructions are substantial, when compared to the original LDR or STR instructions' execution times. Naturally, the more xSB instructions that execute, the larger the resulting performance overhead. SmokeBomb handles the performance overhead of executing xSB instructions by patching *only the cache changing instructions in the sensitive code that accesses* non-sensitive *data.* This optimization process helps to avoid unnecessary performance degradation. Using the decision tree (case study 3), the performance overhead when SmokeBomb patched *all* cache changing instructions increases by about 104% compared with the optimized patching.

Performance of SmokeBomb APIs. We evaluated the execution times of SmokeBomb API enter and exit APIs (which involves prefetching and flushing the sensitive data). As the performance overhead caused by SmokeBomb APIs is determined by the size of sensitive data, we measured execution times using different sizes of sensitive data up to a size the same as the L1 cache size (sensitive data sizes larger than the L1 cache size will only load the selected data to the L1 cache, therefore the upper bound is L1 cache size). Figure 8 shows the execution times of SmokeBomb APIs. The execution time of each API increases with the size of the sensitive data. When the size of sensitive data is 32 KB, the execution time of the two APIs is about 450 microseconds in total on the Cortex-A53.

Single-application overhead. To understand the impact of SmokeBomb on the performance of sensitive code, we evaluated a SmokeBomb protected HTTP Secure (HTTPS) protocol implementation. For this experiment, SmokeBomb was applied on the AES algorithm, and we used the top 500 web pages selected by the Moz [3]. Then, we compared average execution times required to download the 500 web pages between the normal HTTPS protocol and SmokeBomb-protected one across 5,000 experiments. Table 3 shows SmokeBomb-protected HTTPS protocol has very low performance overhead of around 4.02% for Cortex-A53 and 5.91% for Cortext-A72, making it unnoticeable to users during web browsing and quite acceptable for serving web content.

Table 2: Comparison of the execution times between xSB instructions and original instructions.

CPU	Instruction Group	L1 Hit Original	L1 Hit xSB	Cache Miss Original	Cache Miss xSB
Cortex-A72	LDR	612 ns	1,634 ns	897 ns	1,946 ns
	STR	622 ns	1,678 ns	729 ns	1,802 ns
Cortex-A53	LDR	321 ns	1,209 ns	480 ns	1,916 ns
	STR	365 ns	1,251 ns	540 ns	1,420 ns

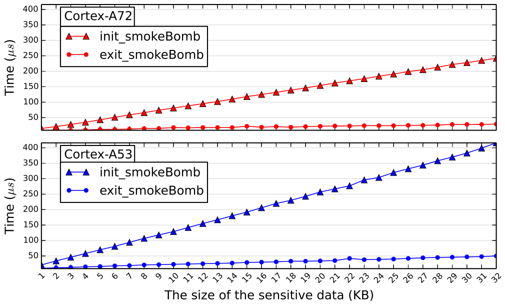


Figure 8: The execution times of SmokeBomb APIs in microseconds.

We also applied SmokeBomb on the AES algorithm of 7zip application and measured execution times required to compress various files secured with AES encryption. As shown in Figure 9, on the Cortex-A53, the latency increased by just 1.58 percent, when 10 KB file is used. However, as the size of the input file increases, the latency also increases. The AES algorithm is a block cipher, and thus, its encryption function to which SmokeBomb was applied operates on a single block. Consequently, the performance overhead brought by SmokeBomb APIs and xSB instructions has to be overlapped as an input file has more blocks.

Performance Impact on Systems. Lastly, we evaluate the impact of SmokeBomb on systems throughput by comparing: (1) when SmokeBomb is not running; with (2) when SmokeBomb is *running continuously* by a process executing the AES encryption function with SmokeBomb applied. The only difference between the two situations is the status of SmokeBomb functions in the testing systems. Thus, it can show how the system performance changes when SmokeBomb is activated. For the evaluation, we used the common UnixBench benchmarking utility (version 5.1.3) [2].

Figure 10 shows the overheads of each benchmark application and the average on the test devices. On the Cortex-A72 of a Samsung Tizen device which uses the inclusive cache model, and where many service processes are executing, the average overhead with SmokeBomb is almost zero. We suspect that the negative overhead of the Whetstone benchmark, which tests how many floating-point operations can execute within a limited time (without using a blocking syscall), is because the benchmark threads could occupy cores a longer time than when SmokeBomb is not running. The average overhead on the Cortex-A53 of Raspberry PI3, which uses the non-inclusive cache model, is about 2.8%. The results illustrate that, when SmokeBomb is activated, the average performance overhead that SmokeBomb imposes on the overall system is negligible.

Table 3: The performance overheads of SmokeBomb-protected HTTPS to load a web page.

Cortex-A72 Baseline	Cortex-A72 SmokeBomb (overhead)	Cortex-A53 BaseLine	Cortex-A53 SmokeBomb (overhead)
7,223 ms	7,650 ms (5.91%)	7,177 ms	7,466 ms (4.62%)

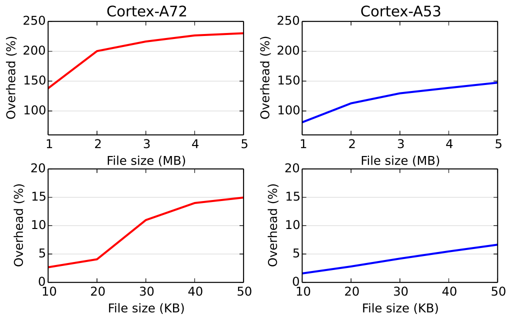


Figure 9: The performance overheads of SmokeBomb-protected 7zip application.

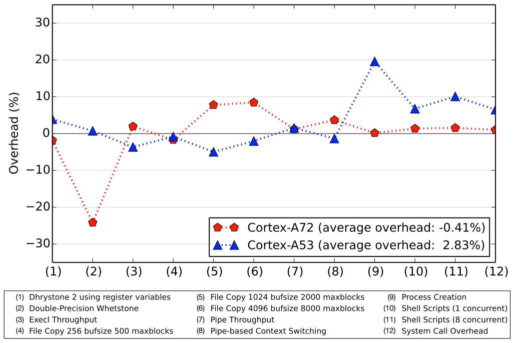


Figure 10: The performance overheads on each benchmark applications when SmokeBomb is activated.

10 LIMITATIONS

Instruction cache attacks. The first limitation of SmokeBomb is that the *instruction cache* is not protected, potentially allowing an attacker to understand which instructions are executed by the victim process (though not what sensitive data was accessed). There is a fundamental problem in applying SmokeBomb to fully protect the instruction cache: ARM has a preloading instruction (PLI) instruction, but in practice we cannot use this instruction to fetch instructions into the L1 instruction cache. The effect of the PLI instruction is not explicitly defined in the ARM architecture reference manuals [10, 13]. The pre-loading instruction (PLI) instruction is treated as a NOP instruction in several Cortex-A processors [15–17, 20, 21], or it fetches instructions to the L2 cache instead of the L1 instruction cache [22]. Thus the only general way to load the L1 instruction cache is to execute the instructions.

However, SmokeBomb can be extended to have functionality for preloading instructions, and flush them, from L2 cache. Given that a combination of preloading and flushing is enough to prevent synchronous instruction cache attacks, a future direction is to defeat asynchronous attacks that inspect an instruction cache in parallel with a running victim process. Note that preloading and flushing make this kind of attack very difficult to perform. In addition, most ARM Cortex-A processors employ a virtually indexed, physically tagged (VIPT) implementation for the L1 instruction cache [6, 7, 9, 11, 14, 15, 18, 20, 21, 23]. Therefore, it is a reasonable assumption that instructions are safe from attacks using the directory protocol because modern operating systems implement ASLR. Aside from this limitation in terms of attacks to determine the execution of sensitive *code*, SmokeBomb can stop known attacks with respect to the sensitive *data*.

Protection for exclusive caches. SmokeBomb cannot provide complete defense for exclusive caches. This is because if an attacker loads the sensitive data into their L1 cache, and evicts the data from the L1 to the L2 cache, the data could also be evicted from the victim's L1 cache. By the definition of the exclusive cache, there is only one copy of the data in the whole cache. Therefore, if an exclusive L2 cache is shared by multiple cores, cache lines of the L1 cache could be affected by the other cores' data usages. As a result, the attacker might deduce information related to the key data. Such information leaks are not theoretically impossible but are difficult to practically achieve, because an attacker must identify whether the data is reloaded by a victim after evicting it within a very short execution time between two SmokeBomb APIs. Moreover, the reloaded data may not be the key data. Unfortunately, we could not conduct any experiment on the exclusive cache model, because there are no available devices on the market. Only the Cortex-A55 employs the exclusive L2 cache among ARM CPUs, but, that is the *private per-core* unified L2 cache which is not shared between cores [22].

Implementation details. The instruction handler might *evict some sensitive data from the L1 to the L2 cache* as the size of sensitive data reaches the size of L1 cache. We evaluated L1 cache refill events caused by the instruction handler during its execution and determined that three cache lines were used by the instruction handler due to the accesses to the stack and global variables. Such evictions could be handled by a dynamically allocated temporary stack only for the execution of sensitive code. This temporary stack can, then, be cached into the L1 data cache with the sensitive data to prevent any eviction of the sensitive data at a small cost of the private area. We note that even if the eviction of sensitive data occurs it would not be critical to SmokeBomb's effectiveness, as the attacker's detecting of eviction does not necessarily lead to the leakage of key data. In addition, the current version of the SmokeBomb implementation protects static data. However, SmokeBomb could provide the same defense for dynamically allocated data with additional improvements to the compiler extension, which would require annotations on pointers that are used to point to sensitive data. In addition, albeit SmokeBomb only requires an annotation of the sensitive data, it has to re-compile source code, and thus, cannot be applied to compiled binaries. Lastly, the implementation of SmokeBomb is dependent on hardware specifications such as the size and inclusiveness of the cache. Hence, minor changes are required to implement SmokeBomb for each different CPU.

Architecture dependence. While the concept of a *private space* in L1 cache is not ARM-specific, SmokeBomb is implemented for, and heavily uses specific functionality of, the ARM architecture. Unfortunately, in our investigation of the current state of the Intel x86_64, architecture, it does not appear to be possible to ensure that data is present *only* in L1 cache, preventing us from implementing SmokeBomb for this architecture. Likewise, the RISC-V architecture currently has no instruction-level control of the cache [58]. However, as both cache control and hardware-based security measures are an actively-evolving field, this could change in the future.

11 RELATED WORK

Because the shared feature of hardware resources is one of the fundamental reasons behind cache side-channel attacks, many proposed countermeasures attempt to isolate shared resources to mitigate such attacks. These countermeasures can be categorized as a hardware approach or a software approach. Previously, most of the software approaches are deployed on cloud systems with Intel architecture. SmokeBomb is the only cache side-channel defense without architecture-specific hardware dependencies, covering the L1 and the L2 cache together. Thus, SmokeBomb is the first defense applicable to the ARM architecture. It achieves this without invasive OS changes. Furthermore, it can be applied to applications automatically by annotating the sensitive data.

SmokeBomb's closest related works are as follows. Kim et al. [44] proposed isolation of the last level cache using a dedicated memory page on each core. Even though it can prevent information leakage via the last level cache efficiently, this approach cannot prevent timing attacks using upper-level cache [41]. Zhang and Reiter [66] proposed periodic cache cleansing mechanism, which prevents information leakages by flushing data used by the previous process in the cache. It cannot address cache attacks targeting the last level cache. Zhou et al. [67] introduced the copy-on-access technique which copies the page when another process accesses memory simultaneously to disable memory sharing. Also, it limits the cacheability of memory pages per process, and thus, each process only can have a limited number of cache lines. Liu et al. [47] presented CATalyst which uses the Intel Cache Allocation Technology to partition the last level cache. It disallows sharing the cache as in STEALTHMEM [44] to defeat the last level side channel attack.

Most recently, Gruss et al. [34] proposed a technique that uses hardware transactional memory (HTM) to prevent cache misses during execution. Though it provides strong cache side-channel protection, the protection range is limited by the size of the CPU's caches. Gruss et al.'s approach requires hardware support, and the ARM architecture does not support the HTM. Another concern is the possibility of these protected transactions failing, which happens frequently under heavy system load (and could be induced by attackers and result from attacks) [34].

12 CONCLUSION

We presented SmokeBomb: a novel, systematic software approach to defeat cache side-channel attacks on the ARM architecture. Our mitigation approach protects access patterns on the sensitive data from attackers easily by providing the protection mechanism to applications as a compiler extension. Our experimental results show that SmokeBomb protects sensitive information leakages against cache attack methods known to us effectively—and with minimal overhead—on overall system throughput.

ACKNOWLEDGMENT

Many thanks to the anonymous referees for their thoughtful reviews. We would also like to thank our shepherd, Kurtis Heimerl.

This material is based upon work supported in part by Samsung Research, Samsung Electronics, the National Science Foundation (NSF) under Grant No. 1703644 and No. 1948175, the Defense Advanced Research Projects Agency (DARPA) HR001118C0060, the Office of Naval Research (ONR) KK1847, the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence), and a grant from the Center for Cybersecurity and Digital Forensics (CDF) at Arizona State University.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of United States Government or any agency thereof.

REFERENCES

[1] (accessed Jan 29, 2019). *Attribute Syntax.* https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html.

[2] (accessed Jan 29, 2019). *Byte-unixbench: A Unix benchmark suite.* https://github.com/kdlucas/byte-unixbench.

[3] (accessed Jan 29, 2019). *Moz's list of the top 500 domains and pages on the web.* https://moz.com/top500.

[4] Josh Aas. 2005. *Understanding the Linux 2.6. 8.1 CPU scheduler.*

[5] Onur Acıiçmez and Werner Schindler. 2008. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA).* San Fancisco, CA, 256–273.

[6] ARM. 2010. *Cortex-A8 Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html.

[7] ARM. 2012. *Cortex-A9 Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100511$_0$401$_1$0$_e$n/index.html.

[8] ARM. 2013. *Cortex-A15 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/index.html.

[9] ARM. 2013. *Cortex-A7 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0464f/index.html.

[10] ARM. 2014. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[11] ARM. 2014. *Cortex-A17 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0535c/index.html.

[12] ARM. 2015. *ARM Cortex-A Series Programmerś Guide for ARMv8-A.* http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/index.html.

[13] ARM. 2016. *ARMv8-A Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487b.b/index.html.

[14] ARM. 2016. *Cortex-A5 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0434c/index.html.

[15] ARM. 2016. *Cortex-A53 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/index.html.

[16] ARM. 2016. *Cortex-A57 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488h/index.html.

[17] ARM. 2016. *Cortex-A72 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100095$_0$003$_0$6$_e$n/index.html.

[18] ARM. 2016. *Cortex-A73 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100048$_0$002$_0$5$_e$n/index.html.

[19] ARM. 2017. *AMBA Protocol.* https://developer.arm.com/products/architecture/amba-protocol.

[20] ARM. 2017. *Cortex-A32 Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100241$_0$001$_0$0$_e$n/index.html.

[21] ARM. 2017. *Cortex-A35 Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100236$_0$002$_0$0$_e$n/index.html.

[22] ARM. 2017. *Cortex-A55 MPCore Processor Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100442$_0$100$_0$0$_e$n/index.html.

[23] ARM. 2017. *Cortex-A75 Core Technical Reference Manual.* http://infocenter.arm.com/help/topic/com.arm.doc.100403$_0$200$_0$0$_e$n/index.html.

[24] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv preprint arXiv:1702.07521* (2017).

[25] Billy Bob Brumley and Risto M Hakala. 2009. Cache-timing template attacks. In *Proceedings of the 15th the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT).* Tokyo, Japan, 667–684.

[26] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity.. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS).* San Diego, CA.

[27] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+abort: A timer-free high-precision l3 cache attack using intel TSX. In *Proceedings of the 26th USENIX Security Symposium (Security).* Vancouver, BC, Canada, 51–67.

[28] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 35.

[29] Shekhar R Gaddam, Vir V Phoha, and Kiran S Balagani. 2007. K-Means+ ID3: A novel method for supervised anomaly detection by cascading K-Means clustering and ID3 decision tree learning methods. *IEEE Transactions on Knowledge and Data Engineering* 19, 3 (2007), 345–354.

[30] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Christiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS).* San Diego, CA.

[31] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. 2017. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *Proceedings of the 26th USENIX Security Symposium (Security).* Vancouver, BC, Canada, 1075–1091.

[32] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in javascript. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA).* San Sebastian, Spain, 300–321.

[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA).* San Sebastian, Spain, 279–299.

[34] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the 26th USENIX Security Symposium (Security).* Vancouver, BC, Canada, 217–233.

[35] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Security Symposium (Security).* Washington, DC, 897–912.

[36] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland).* San Jose, CA, 38–55.

[37] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games–bringing access-based cache attacks on AES to practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland).* Oakland, CA, 490–505.

[38] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC).* Santa Clara, CA, 299–312.

[39] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. Cryptology ePrint Archive, Report 2015/898. (2015). https://eprint.iacr.org/2015/898.

[40] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S $ A: A shared cache attack that works across cores and defies VM sandboxing-and its application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland).* San Jose, CA, 591–604.

[41] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS).* Xi'an, China, 353–364.

[42] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, Cross-VM attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* Gothenburg, Sweden, 299–319.

[43] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N Serpanos, and Stefanos Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* 12, 3 (2008), 221–230.

[44] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (Security).* Bellevue, WA, 189–204.

[45] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. 2008. Deconstructing new cache designs for thwarting software cache-based side channel

attacks. In *Proceedings of the the 2nd ACM workshop on Computer security architectures*. Alexandria, VA, 25–34.

[46] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX, 549–564.

[47] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*. Barcelona, Spain, 406–418.

[48] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B Lee. 2016. Newcache: Secure cache architecture thwarting cache side-channel attacks. *IEEE Micro* 36, 5 (2016), 8–16.

[49] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, 605–622.

[50] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX, 619–636.

[51] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA)*. San Jose, CA, 1–20.

[52] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL, 199–212.

[53] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Bonn, Germany, 279–299.

[54] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 1 (2010), 37–71.

[55] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC, 913–928.

[56] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. Cached: Identifying cache-based timing channels in production software. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada, 235–252.

[57] Zhenghong Wang and Ruby B Lee. 2008. A novel cache architecture with enhanced performance and security. In *Proceedings of the the 41st annual IEEE/ACM International Symposium on Microarchitecture*. Como, Italy, 83–93.

[58] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. 2017. *The RISC-V Instruction Set Manual*. https://github.com/riscv/riscv-isa-manual.

[59] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive* 2014 (2014), 140.

[60] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA, 719–732.

[61] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive* 2016 (2016), 980.

[62] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. 2016. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria, 858–870.

[63] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA, 313–328.

[64] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. Raleigh, NC, 305–316.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, AZ, 990–1003.

[66] Yinqian Zhang and Michael K Reiter. 2013. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. Berlin, Germany, 827–838.

[67] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria, 871–882.