Attack Directories on ARM big.LITTLE Processors

Zili KOU
Hong Kong University of Science and Technology
zkou@connect.ust.hk

Sharad Sinha
Indian Institute of Technology Goa
sharad@iitgoa.ac.in

Wenjian HE
Hong Kong University of Science and Technology
wheac@connect.ust.hk

Wei ZHANG
Hong Kong University of Science and Technology
wei.zhang@ust.hk

***Abstract*—Eviction-based cache side-channel attacks take advantage of inclusive cache hierarchies and shared cache hardware. Processors with the template ARM big.LITTLE architecture do not guarantee such preconditions and therefore will not usually allow cross-core attacks let alone cross-cluster attacks. This work reveals a new side-channel based on the snoop filter (SF), an unexplored directory structure embedded in template ARM big.LITTLE processors. Our systematic reverse engineering unveils the undocumented structure and property of the SF, and we successfully utilize it to bootstrap cross-core and cross-cluster cache eviction. We demonstrate a comprehensive methodology to exploit the SF side-channel, including the construction of eviction sets, the covert channel, and attacks against RSA and AES. When attacking TrustZone, we conduct an interrupt-based side-channel attack to extract the key of RSA by a single profiling trace, despite the strict cache clean defense. Supported by detailed experiments, the SF side-channel not only achieves competitive performance but also overcomes the main challenge of cache side-channel attacks on ARM big.LITTLE processors.**

***Index Terms*—directory, side-channel attacks, ARM processors**

I. Introduction

Cache side-channel attacks represent approaches of exploiting the timing difference between cache hits and cache misses. Researchers have proposed various applications to exploit the cache side-channel, e.g., covert channel communications [1], [2], extracting cryptographic keys [2], [3], and speculative execution attacks [4], [5].

A. Background and Previous Works

Coherence-based cache side-channel attacks [6], [7] exploit the cache coherence protocol between cores. They need to load the same data owned by victims, so the virtual memory space must be shared. Flush-based attacks [8], [9] further need the privilege of cache-cleaning instructions to clear victims' cache lines. Unlike coherence-based and flush-based attacks, eviction-based attacks [2], [10] do not require shared memory or any special instructions. Instead, such attacks construct cache eviction sets to occupy target cache sets.

Flush-based attacks can be avoided using one of two methods. One is to forbid cache-cleaning instructions. Some works suggest restricting the usage of the `clflush` instruction on x86 platforms [11], [12]. On ARM platforms, cache-cleaning instructions can naturally be disabled by configuring the `PMUSERENR_EL0` register. The second method is avoiding shared memory between attackers and victims, which defends against both flush-based and coherence-based attacks. This is usually true as user programs are isolated by the operating system (OS), which is also why previous attacks appear only in the shared library scenario, or when attackers exploit memory deduplication [13]. Eviction-based attacks are therefore more practical, as they do not require the shared virtual memory space with victims but rely on their own data to evict a cache set.

There are two prerequisites to make eviction-based cache side-channel attacks feasible. The first is the existence of shared cache hardware, and the second is that cache hierarchies should be inclusive otherwise attackers cannot evict the private cache line of remote victims. To the best of our knowledge, modern commercial x86 processors are usually implemented with a shared last level cache (LLC) and most adopt an inclusive cache hierarchy. For the increasing trend of adopting non-inclusive cache hierarchies on today's x86 server processors, Yan et al. [14] support our statements that eviction-based cache side-channel attacks fail to deal with non-inclusive caches. Instead of caches, attackers turn to attack directories, which are inclusive between cores.

TABLE I: Existing Cache Side-channel Attacks on ARM.

Work	Single-core	Cross-core	Cross-cluster
[20]	Flush,Evict	Evict[a],Flush,Coherence	Coherence
[21]	Evict	—	—
[22]	—	Flush[a]	—
[23]	Flush	—	—
[24]	Evict	—	—
[25]	Evict	—	—
[26]	Evict	—	—
This paper	Evict	Evict	Evict

[a]However, limited by [15].

ARM processors tell a different story. As licensed designers are free to modify ARM's intellectual property (IP) blocks, ARM processors, however, are board-specific and are much more diverse; therefore, we should never assume that an ARM processor adopts an inclusive cache hierarchy. In fact, AutoLock [15] demonstrates that a group of ARM processors "behave non-inclusively" so that cache side-channel attacks are harder than we think. On the other hand, the ARM big.LITTLE architecture design [16] integrates big and little clusters without a shared cache, in which case, the first precondition is lacking. Despite the big.LITTLE architecture design being one of the most famous ARM IPs that is widely authorized [17], [18] and currently on sale [19], there is no systematic analysis of eviction-based side-channel attacks on ARM big.LITTLE processors. The capabilities of conventional approaches, the feasibility of cache eviction between cores and clusters, and the existence of an inclusive structure remains to be explored.

B. Challenges of Cache Attacks on ARM

While cache side-channel attacks are well studied on x86 platforms, only a few research works exist on ARM platforms. We define three attack scenarios: the single-core scenario, the cross-core scenario, and the cross-cluster scenario. The cross-cluster scenario is the toughest one of the three as it does not conform to the two prerequisites. A summary of the existing cache side-channel attacks on ARM platforms is shown in Table I.

Existing works have not systematically discussed cache side-channel attacks on ARM big.LITTLE processors, so there is no attack in the cross-cluster scenario except Armageddon [20], which presents a coherence-based covert channel. Instead, most of the attacks focus on the single-core scenario, which does not face obstacles in evicting remote cache lines since the attackers always share the same private cache with the victim. For the cross-core scenario, Armageddon [20] successfully implements eviction-based attacks between cores, indicating that their tested platforms comply with the inclusiveness prerequisite. The instruction-side flush-based attack [22] relies on inclusive caches as well.

However, AutoLock [15] limits the success of [20] and [22] by unveiling a lockdown feature of private cache lines found in many

ARM processors. The feature even takes effect in a cache hierarchy that is announced to be inclusive. We agree with AutoLock and conclude that it is still hard to conduct eviction-based attacks in the cross-core scenario because inclusive cache hierarchies on ARM processors should never be assumed, not to mention the existence of AutoLock feature. Additionally, there is no eviction-based attack in the cross-cluster scenario.

This paper systematically analyzes the ARM big.LITTLE architecture design and unveils the SF, an unexplored directory built in ARM interconnects to keep caches coherent. The SF is inclusive between cores and clusters and therefore meets the two prerequisites of eviction-based attacks. We propose to attack directories, i.e., SF, on ARM big.LITTLE processors, and make the following contributions:

- Discuss the directory on ARM platforms, revealing the structure and property of the SF by our systematic reverse engineering.
- Demonstrate the applicability of eviction-based cache side-channel attacks on ARM big.LITTLE processors, despite there being no shared cache hardware between clusters.
- Present a comprehensive methodology to exploit the SF side-channel, including
  - a probabilistic approach to construct SF eviction sets
  - an SF covert channel between cores and clusters
  - eviction-based attacks on AES and RSA, and we are the first to attack RSA on ARM processors in the user space
  - an interrupt-based side-channel attack on RSA in the TrustZone scenario, despite the strict cache clean defense
- Quantitatively compare the SF with the cache on real-world ARM big.LITTLE processors, resulting in competitive performance and better capability.

II. Preliminaries

In this section, we introduce the template ARM big.LITTLE architecture, the general concept of the SF, and the technique of eviction-based attacks. Finally, our threat models are presented.

A. ARM big.LITTLE Architecture

The big.LITTLE architecture formally involves a "big" core cluster and a "LITTLE" core cluster. The big cluster is designed to provide maximum performance, while the little cluster aims to achieve maximum power efficiency. The two clusters have different specifications based on their purpose, i.e., little cores usually have lower core frequencies, smaller cache hierarchies, and less power consumption than big cores, which makes ARM big.LITTLE processors more suitable for the dynamic usage pattern of today's devices. The template ARM big.LITTLE architecture is shown in Fig. 1, and it

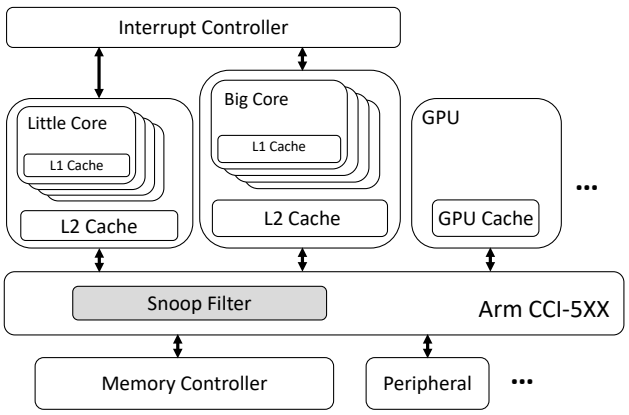

Fig. 1. The template ARM big.LITTLE architecture design.

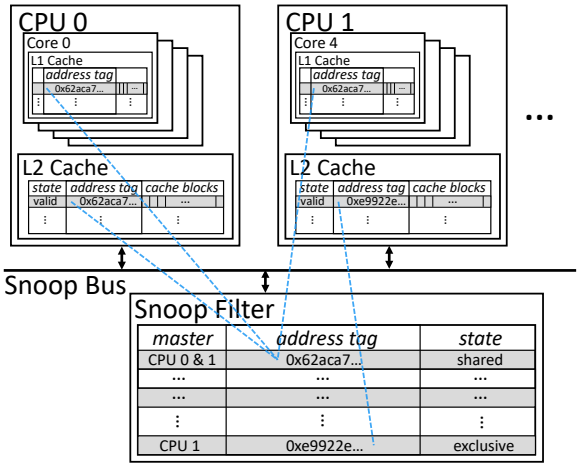

Fig. 2. The general design of the SF.

usually comes with an ARM cache coherent interconnect (CCI) to offer hardware-managed coherency.

In such a system, two clusters and other coherent hardware are connected to master interfaces, and the main memory and system peripherals are connected to slave interfaces. Snoop requests are broadcast by masters to keep caches coherent, like the implementation in CCI-400 [27]. However, CCI-400 suffers a lot from redundant snoop requests between masters. To reduce the workload, the SF is then imported into CCI-500 [28] and CCI-550 [29] to provide efficient snoop requests management by keeping records of cached data. In the following discussion, we target the template ARM big.LITTLE architecture design with CCI-5XX as it is widely used and currently on sale [19].

B. General Concept of SF

The bus snooping scheme keeps caches coherent by broadcasting snoop requests to every master, but it suffers a lot from a heavy bus workload. Broadcast overhead goes higher with more masters that own private caches. Directory-based protocols [30], [31] deal with the scalability issue by storing the cache line status of all masters so that cache coherence is maintained by point-to-point transactions. The SF is one of the directory-based implementations proposed by academia [32], [33] and industry [28], [29]. The general design of the SF is shown in Fig. 2, for which an extra memory block is imported to record the status of cache lines, including owners, address tags, and coherent states (e.g., MOESI [34]). Thanks to the SF, masters now no longer need to broadcast a write exclusive request but check contents of the SF; it directly writes if snoop misses, or invalidates the cache lines of specific masters if snoop hits. Other requests are managed similarly, leading to less power consumption and lower snoop latencies.

C. Technique of Eviction-based Attacks

Prime+Probe [2], [10] is the most common technique of eviction-based cache side-channel attacks. The attacker first needs to construct an eviction set, i.e, a group of addresses that are mapped to the same cache set. In every monitoring epoch, there are three procedures: *prime*, *wait*, and *probe*. In the *prime* procedure, the attacker loads the eviction set to fully occupy a cache set. During the *wait* procedure, if the victim loads data that is mapped to the same cache set, data cached by the attacker will be evicted. The attacker finally reloads the eviction set in the *probe* procedure. A longer reloading time can be detected due to cache misses, indicating that the cache set has been previously accessed by the victim.

D. Threat Models and Assumptions

There are two threat models in this paper. The probabilistic approach, the covert channel, and the side-channel attacks are implemented in the user space. When conducting reverse engineering and attacking TrustZone, we assume attackers have kernel privileges.

In the user space, we make minimal assumptions that attackers are not allowed to share the virtual memory with victims, and cache-cleaning instructions are disabled for user programs as well. Due to the context switching mechanism of the OS, we do not assume that a malicious program can operate on a specific core. We use the cycle counter `PMCCNTR` as the timing source for more precise evaluations. Though `PMCCNTR` may be disabled in the user space, this causes no loss of generality since equivalent timing sources are still available [20].

In the kernel space, we assume victims are built as services in the secure world of TrustZone, while the OS in the normal world is malicious. Malicious programs are implemented as kernel drivers and can assign any core to invoke a secure service. Such a threat model is common for kernel privileged attackers and totally obeys the assumptions made by TrustZone [35].

III. Reverse Engineering

In this section, we review accessible information about the SF, design systematic reverse engineering, and finally evaluate the capability of the SF as a new side-channel.

A. Hints and Assumptions

Hints. We have exhaustively reviewed the documents published by ARM [28], [29], [36] and present all the details as follows.

- The SF is 8-way set associative and ARM recommends setting the SF directory to be 0.75-1 times the total size of the caches.
- There exist performance counters to count the access times of SF banks, from the 1^{st} to the 8^{th} bank.
- Conflicts happen when an SF set has no available position to insert a new entry and an existing entry must be evicted.
- CCI issues a `CleanInvalid` snoop to processors that are holding the evicted lines. This type of eviction is called "back-invalidation" and it is expected to occur rarely.

Assumptions. In our reverse engineering, we only consider the most practical scenario that the SF is configured to keep caches coherent between two clusters. The conservative assumptions are

- The SF has an 8-way set associative structure. We refer to its sets as "SF sets", each of which can record the status for 8 address tags of cached data.
- Conflicts in an SF set trigger the back-invalidation, forcing the corresponding cache lines to be evicted.
- It may or may not be bank-sliced, i.e., the bank number can be 1 or above.
- It is strictly inclusive that every cached data must have its tag recorded in the SF.

B. Experiments Platforms

Directory structures may not be implemented on every processor, like the x86 work [14] only presents one server processor for evaluating the directory. Not to mention that ARM processors are much more diverse. Therefore, we decide to work on the template design of the ARM big.LITTLE architecture. Two system-on-chips (SoCs), namely Hisilicon Kirin 960 and Kirin 970, are selected because they are officially announced with CCI-550. Apart from two developer-friendly boards called Hikey960 and Hikey970 [37], [38], we further verify some of our implementations (like the construction of SF eviction sets) on a modern smart phone, Honor View 10 [39], to justify the practicality. The software and hardware specifications are listed in Table II. Explicitly speaking, around 100 phones [40], [41] with the two SoCs and any other edge devices that adopt CCI-550 are directly affected by our works. At this time, we do not reproduce our works on other SoCs due to the accessibility or confidentiality limits from vendors. However, we design general reverse engineering and algorithms to ensure the extendability. In the following experiments, we only show the results on Hikey960 if results are similar between devices.

TABLE II: Experiment Platforms.

	Hikey960	**Hikey970**	**Honor View 10**
SoC	Kirin 960	Kirin 970	Kirin 970
Processor	4 Cortex A73 as big cores, 4 Cortex A53 as little cores.		
L1-Data	A73: 4-way, 256 sets	A53: 4-way, 128 sets	
L2 Cache	A73: 16-way, 2048 sets	A53: 16-way, 512 sets	
OS	Buildroot Linux 5.5	Debian Linux 4.9	Android 9.0

C. Reverse Engineering

1) Find the First Eviction Set: Researchers usually investigate the structure and property of a set associative structure by observing its conflict behaviors, i.e., how and under what conditions conflicts are triggered. To reverse engineer the SF, we need to find the first SF eviction set (EV) in the kernel space, which is the entry of SF conflicts. An EV refers to a collection of addresses that are mapped to a specific set. For a better demonstration, we extend the traditional definition [2] by calling an EV with the size exactly equal to the set associativity a **"minimized EV"** and an EV with a larger size than a minimized EV a **"conflict EV"**. A cache set can only accommodate a cache minimized EV and any coming data triggers cache conflicts. Therefore, traversing a cache conflict EV results in cache misses and takes a longer time. The same for the SF, SF conflicts that happen in an SF set trigger back-invalidation, which finally results in cache misses. As a result, an SF conflict EV can also be detected by timing difference, as shown in Fig. 3.

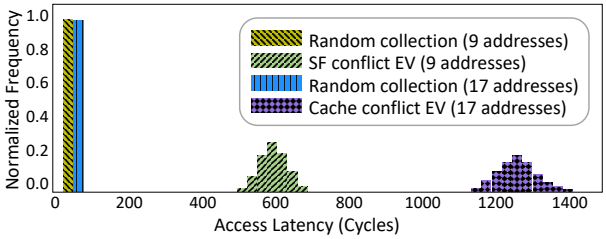

Fig. 3. Latencies of accessing a cache (or SF) conflict EV.

There exist previous approaches [2], [14], [42] for constructing cache EVs, however, they cannot be directly applied to the SF. In Section IV, we will explain the reasons and propose our approach for constructing SF EVs in the user space. To find the first SF EV before we know the detailed structure, we rely on some engineering techniques and empirical attempts. Since we have no idea of the set index portion of the SF, one technique is to allocate a candidate set with addresses that have the same lower n bits. We set n to be 23, which is very likely to cover the possible set index portion. The SF may have a bank-sliced structure, indicating that addresses with the same set index portion may still not be mapped to the same bank [43]. We therefore propose Algorithm 1 to randomly pick 9 addresses from the candidate set and evaluate until SF conflicts are detected. We can successfully get an SF conflict EV for any initial value of lower n

Algorithm 1: Search an SF conflict EV

Input: Candidate set D with the same lower n bits
Output: SF conflict EV T
while 1 **do**
  $T \leftarrow \{\}$
  repeat 9 **times**
    $addr \leftarrow random\_pick(D \backslash T)$
    $T \leftarrow T \cup \{addr\}$
  end
  $access(T)$ // *Load to cache it well*
  $t \leftarrow access\_latency(T)$
  if $t > SF\_conflict\_threshold$ **then**
    $D \leftarrow D \backslash T$ // *remove T from D*
    return T
  end
end

bits, and an SF minimized EV can be obtained by removing one address from the obtained SF conflict EV.

2) Set Associativity: For any address, we can know its owner set by checking if it can be evicted by the corresponding minimized EV. We prepare a large enough EV, from which we can construct EVs with different sizes. Average access latencies of EVs of different sizes are shown in Fig. 4. The change of lines' slopes obviously depicts the capacity of an SF set and a cache set, which supports our assumption of the 8-way set associative structure.

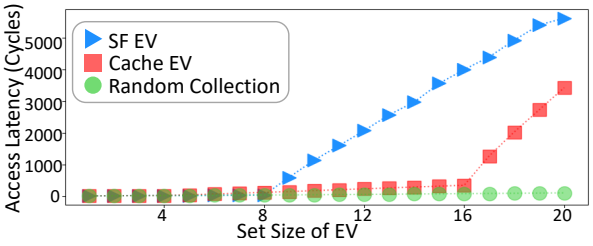

Fig. 4. Access latencies of EVs of different sizes. SF conflicts happen when the size of an SF EV is larger than 8.

3) Replacement Policy: Our verification starts with an SF conflict EV of 9 addresses. Firstly, access 8 addresses several times to cache them well. Secondly, access them "in order", and then access the 9^{th} address. Now one of the 8 addresses must be evicted due to SF conflicts. We observe an even distribution of the 8 addresses, indicating that the SF is more likely to use a random replacement policy as every entry in an SF set has the same possibility of being selected, regardless of the sequence of loading. Indeed, the true replacement policy may be much more complex than we expect, however, the claim of a "somewhat" random policy is reliable enough for side-channel attackers.

4) Set Size: The number of sets can be figured out by counting how many mutually exclusive EVs are in a contiguous memory space. By using an SF minimized EV, we can filter out all the addresses that also belong to the same EV. Such a filtering procedure is iterated within a contiguous memory space until the last SF EV is identified. The uniform results of experiments with different pool sizes are shown in Table III, indicating that the set size is 16384.

TABLE III: NUMBER AND SIZES OF SF EVS.

Contiguous Memory	Number of SF EVs	Set Size
8 MB	16384	8
32 MB	16384	32
512 MB	16384	512
1 GB	16384	1024

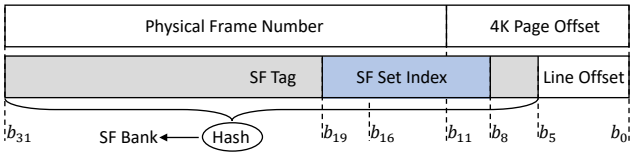

Fig. 5. Memory for a 4 GB address space from the aspect of the 4 KB page and the SF index.

5) Set Index Portion: The set index portion of the SF can be unveiled by checking identical bits of addresses in an SF EV. We try to cover as much memory space as we can, though the maximum size of an SF EV is limited by the main memory's capacity. The overall memory space we can cover is roughly 3 GB in Hikey960 or 5 GB in Hikey970 and Honor View 10, and we iterate such collections with reboots to also import the randomness from Kernel Address Space Layout Randomization (KASLR) [44]. Based on the reliable collections, we find out that addresses mapped to the same SF set all have the same value of the 9^{th} to the 19^{th} bits, as shown in Fig. 5.

6) Bank Hash Function: There are totally 16384 SF EVs, which contradicts our reverse engineered SF set index portion as it only has 11 bits. Additionally, a random collection of addresses with the same SF set index portion can not always form an effective SF EV. This phenomenon is not weird for sliced set associative structures [43], [45], [46]. The number of SF banks is revealed by the gap between the set size and the set index portion, i.e., $16384/2048 = 8$. The bank of an address is determined by an undocumented hash function H:

$$H : \{0,1\}^{32-6} \rightarrow \{0,1\}^3 \tag{1}$$

The input is a physical address. We reconstruct the hash function for a 4 GB memory space and then subtract 6 bits for the cache line offset. The output is the decision from the 1^{st} to the 8^{th} bank. Works in [45], [46] further indicate that H can be separated by independent one-bit hash functions and each function can be expressed as a series of XORs of the input bits:

$$\begin{aligned} H &= \overline{h_2 h_1 h_0} \\ h_2 &= b_{28} \oplus b_{26} \oplus b_{18} \oplus ... \\ h_1 &= b_{11} \oplus b_7 \\ h_0 &= b_{28} \oplus b_{26} \oplus ... \end{aligned} \tag{2}$$

where b_i denotes the i^{th} bit of the input, and we call it an **"effective bit"**. In our reverse engineering, we hypothesize that the hash function of the SF has a similar form to that of sliced LLC on x86 processors, which is verified true in the following evaluations.

Maurice et al. [45] reconstruct the ground true hash function with the help of performance counters, however, the performance counters offered by CCI are not satisfying enough: despite there being 8 SF banks, we are only offered 4 events and each event is fixed to count the access time of two banks. E.g., the first event counts *"access to snoop filter bank 0 or 1, any response."*. As a consequence, we propose our own method to reconstruct the hash function, which is more general and does not rely on dedicated performance counters.

As all SF EVs can be obtained from a contiguous memory space, we can choose a specific value of the set index portion to get 8 SF minimized EVs. For any address, it can only be evicted by one of the 8 SF minimized EVs. We simply number the 8 SF minimized EVs in order, so the "eviction test" of an address results from bank $\{000\}_2$ to bank $\{111\}_2$. As shown in Algorithm 2, we conduct the eviction test twice, one for the original address and another one for the address with one bit reversed. Whether a bit is the effective bit of hash functions is indicated by the difference between resultant

TABLE IV: REVERSE ENGINEERED BANK HASH FUNCTIONS.

	H	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	
Kirin 970	h_2				⊕		⊕								⊕										⊕			$h_2 = b_{28} \oplus b_{26} \oplus b_{18} \oplus b_8$
	h_1																					⊕				⊕		$h_1 = b_{11} \oplus b_7$
	h_0				⊕		⊕							?	?	?	?	?	?	?	?	?	?	?	⊕		⊕	$h_0 = b_{28} \oplus b_{26} \oplus b_8 \oplus b_6$
Kirin 960	h_2				⊕										⊕										⊕			$h_2 = b_{28} \oplus b_{18} \oplus b_8$
	h_1							⊕			⊕			⊕		⊕										⊕		$h_1 = b_{25} \oplus b_{22} \oplus b_{19} \oplus b_{17} \oplus b_7$
	h_0						⊕							?	?	?	?	?	?	?	?	?	?	?			⊕	$h_0 = b_{26} \oplus b_6$

*Bits with ? can be revealed if we have perfect performance counters, which are lacking in CCI.

Algorithm 2: Evaluation of the effective bit

Input: 8 SF minimized EVs $B[8]$, the k^{th} bit to evaluate
Output: $\{0,1\}^3$, indicating the effectiveness for $\overline{h_2 h_1 h_0}$
addr ← *random_pick*()
$addr_{re}$ ← $addr \oplus 2^k$ // *reverse the* k^{th} *bit*

```
bank_id ← 0
for i ← 0 to 7 do
    access(addr)
    access(B[i])
    t ← access_latency(addr)
    if t > cache_miss_threshold
      then
        bank_id ← i
        break
    end
end

bank_id_re ← 0
for i ← 0 to 7 do
    access(addr_re)
    access(B[i])
    t ← access_latency(addr_re)
    if t > cache_miss_threshold
      then
        bank_id_re ← i
        break
    end
end

return {bank_id}_2 ⊕ {bank_id_re}_2
```

banks. The result is shown in Table IV. Note that the obtained hash functions are not the only solution because we may not number banks as the real hardware implementation. Additionally, the 9^{th} to the 19^{th} bits cannot be evaluated since they must be fixed to maintain the same set index portion. However, this is all attackers need due to its effectiveness of distinguishing addresses into 8 banks. Nevertheless, considering that the ground true hash functions can be reconstructed with the help of dedicated performance counters, we also leverage the imperfect performance counters offered by CCI to reveal more details. The result is also shown by the red symbols in Table IV, which is believed to be useful for others.

D. SF Side-channel

In conclusion, the SF has an 8-way set associative structure with a random replacement policy. There are 8 SF banks and each bank holds 2048 sets. Given any address, it can be mapped by checking the hash functions and the SF set index portion, and this is how we construct SF EVs in the kernel space, too. We design verification routines to justify the SF as a new side-channel: 1) Randomly collect an address and construct its corresponding SF minimized EV. 2) Access the address to cache it well and then access the SF minimized EV. 3) Check if the data is still cached. A successful verification should return the data evicted. The verification gets passed after enough attempts between cores and clusters. To exploit the SF side-channel, we propose SF-based Prime+Probe attacks, in which we leverage SF EVs instead of cache EVs. In the following discussion, we refer to Prime+Probe as a general technique, i.e., *prime*, *wait*, and *probe* procedures. We call SF-based Prime+Probe SF-Prime+Probe, while the previous attacks are called Cache-Prime+Probe.

IV. CONSTRUCTING SF EVICTION SETS

In this section, we establish the necessary block for eviction-based SF side-channel attacks in the user space, i.e., constructing SF EVs. We first analyze why existing approaches fail, and then propose our probabilistic approach.

A. Cache EVs Construction

Acknowledged approaches to construct cache EVs are proposed in [2], [14], [42]. We follow their efforts and become able to construct a cache EV for any victim address in only seconds.

Limitation: The key of the existing approaches is to check whether an address conflicts with a candidate set. Unfortunately, we cannot naively transfer such an idea to the SF, since 1) we cannot directly deal with the SF, and instead, we rely on caches to interact with the SF, i.e., we allocate new SF entries by caching data, and we observe the back-invalidation by cache misses. 2) There is no way to distinguish SF conflicts from cache conflicts as they both behave as cache misses. 3) SF conflicts appear much less often than cache conflicts due to the larger set size and the bank-sliced structure. After all, the design goal of the SF is to evenly map common memory activities. To construct SF EVs, we suggest to **avoid cache conflicts first, and then collect SF conflict samples**.

B. Probabilistic Approach

We here define the cache hierarchy we operate on: it has a W-way set associative L2 with S sets and the line size is L. The specifications of the SF are the same as what we reverse engineered. Originally, user space attackers can only access the lower 11 bits of a physical address with the help of 4 KB pages. Thanks to cache EV construction, we can further increase the scope to the lower $log_2 S + log_2 L$ bits. However, there are still $log_2(16384/S)$ bits we cannot cover, as shown by the green shadow area in Fig. 6, and we call them **"unknown bits"**. Therefore, it turns out to be a probabilistic event: "addresses picked from a cache EV have the same SF set index portion".

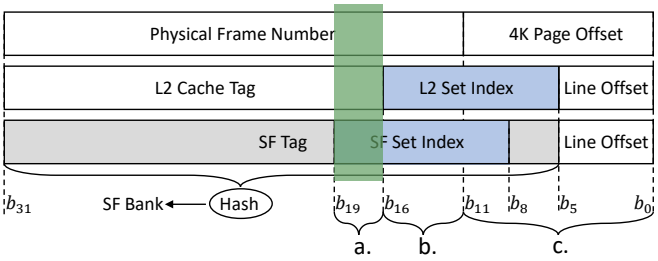

Fig. 6. Set index portions of the L2 cache and the SF. The *unknown bits* in a. cannot be covered.

First, we prepare a cache EV of size N, denoted as C. Now, considering the ideal model when we construct a large enough cache EV with an infinite N, for any address in C, it has the same possibility for $(16384/S)$ different values of *unknown bits*:

$$P(\textit{unknown bits} = \{00...0\}_2) = ...$$
$$... = P(\textit{unknown bits} = \{11...1\}_2) = \frac{S}{16384} \quad (3)$$

Second, we randomly collect k $(8 < k \leq W)$ addresses from C and get $C' = \{c_1, c_2, ..., c_k\}$. The setting of the lower bound is because we wish to find SF conflicts so k must be larger than the SF

set associativity, and the upper bound is set to avoid cache conflicts. A **"success collection"** is when we get a collection containing more than 8 addresses with the same SF set index portion (so the same value of *unknown bits*). In the ideal model, it becomes k independent and identical processes that the possibility is

$$\begin{aligned} &P(\textit{success collection}) \\ &= \sum_{i=9}^{k} P(\textit{success collection: i addresses case}) \\ &= \sum_{i=9}^{k} \frac{16384}{S} \binom{i}{k} \left(\frac{S}{16384}\right)^{i} \left(1 - \frac{S}{16384}\right)^{k-i} \end{aligned} \tag{4}$$

The possibility increases monotonically with k so we may simply adopt $k = W$. We expand every address in C' into 8 addresses by setting the 6^{th} to the 8^{th} bits from $\{000\}_2$ to $\{111\}_2$ to get $C'' = \{(c_{10}, ..., c_{17}), ..., (c_{k0}, ..., c_{k7})\}$. According to the reversed hash functions, such an expansion not only keeps the SF set index portion unchanged but also imports extra addresses that are mapped to all 8 SF banks. More importantly, we can now check whether a collection is a *success collection* by timing difference, since cache misses are triggered "only" by SF conflicts. In case a *success collection* is discovered, we can easily obtain the first SF EV via similar approaches as constructing cache EVs. Remaining SF EVs can be searched out as the obtained SF EVs help filter out irrelevant addresses to significantly increase the probability of later *success collections*.

C. Results

The most essential point is whether our probabilistic approach works on real hardware. For big cores on ARM big.LITTLE processors, i.e., A73 with $S = 2048, k = W = 16$, and $L = 64$, $P(\textit{success collection}) = 2.97 \times 10^{-4}$ in the ideal model. Therefore, the expected number of attempts is $1/P(\textit{success collection}) = 3367$, which can soon be done by big cores. In the real case, however, we cannot construct an infinite cache EV. For a specific value of N, the arithmetic expression of $P(\textit{success collection})$ becomes complicated, so we use simulation to analyze it. For each different N, we let the simulation try no more than $10,000$ collections, and record the number of attempts once succeed. The results are shown in Table V. Note that the averaged number of attempts is only calculated by the success cases, so the success rate also makes sense. When operating on an A73 core with $N = 256$, our probabilistic approach gets a *success collection* after 3082 attempts on average, with a success rate of 0.921, which is consistent with the simulations. Given a victim address, our approach takes 7.8 seconds to construct its cache EV with $N = 256$ and 3.57 seconds to construct its SF minimized EV, which is practical and efficient enough as the baseline cache EV construction also finishes in seconds [42].

However, when the probabilistic approach operates on little cores, i.e., A53 with $S = 512$. The number of *unknown bits* is quadrupled so that $P(\textit{success collection}) = 8.52 \times 10^{-9}$, which is computationally infeasible. To sum up, the construction of SF EVs depends on the specifications of the hardware implementations, which is usually feasible on big cores but infeasible on little cores. Nevertheless, once an SF EV is constructed, following attacks will not be influenced even if the OS schedules the attacker's thread to a little core. Besides, for the SF covert channel in Section V-A, the sender or receiver on a little core can smoothly construct SF EVs with the help of the receiver or sender on a big core.

TABLE V: ATTEMPTS FOR A SUCCESS COLLECTION.

N (Size of Cache EV)	Averaged Attempts	Success Rate
64	3224.22	0.564
128	2936.57	0.826
256	2887.29	0.924
512	2821.41	0.929
1024	2818.92	0.944
[a] 256	3082.00	0.921
[b] 256	3557.54	0.89

[a]evaluated on Hikey960. [b]evaluated on Honor View 10.

TABLE VI: PRIME+PROBE COVERT CHANNELS.

	Sender & Receiver	N	T_s / us	T_r / us	Error Rate	[a]BW / bps	Cross-cluster?
Cache	A53→A53	92	12.5	12.5	0.050	21937	✗
	A73→A73	92	4.5	4.5	0.049	51901	✗
SF	A53→A53	90	9.0	9.0	0.053	21459	✗
	A73→A73	90	6.5	6.5	0.052	42726	✗
	A53→A73	78	23.0	16.0	0.051	18612	✓
	A73→A53	78	18.0	26.0	0.047	18561	✓

[a]Bandwidth

V. SF SIDE-CHANNEL ATTACKS

In this section, we demonstrate the SF as a new side-channel by applications including covert channel and side-channel attacks against AES and RSA. Note that various countermeasures [47]–[50] have been proposed; however, AES and RSA are still being analyzed by researchers [9], [14], [20]. Our presented implementations are more for quantitatively comparing the cache and the SF, so we put less emphasis on the applications in more practical and general scenarios, i.e., we omit the efforts for addressing obstacles, such as synchronizing with victim threads, figuring out the target cache (or SF) sets, etc. The performance comparison accounts only for the monitoring and recovering procedures.

A. SF Covert Channel

Relying on Prime+Probe, our core implementation is similar to that of previous works [1], [2], [51], while additional operations are needed when in the cross-cluster scenario. The program on a big core first constructs and repeatedly loads an SF EV, causing the target SF set to be occupied. For any address, the program on a little core loads it, waits for a while, and reloads it. If the reloading results in a cache miss, the program realizes the address is from the target SF EV. The procedure is iterated until a minimized SF EV is obtained. In the transmission step, all parameters except two are fixed: N bits are sent for every packet, and T denotes the *prime* time (T_s for the sender and T_r for the receiver). On the tested platform, both the cache and SF adopt the random replacement policy so accessing EVs multiple times is needed. For a fair comparison, when in the cross-cluster scenario, we ensure that the sender and receiver always access EVs with the same number of iterations instead of the same *prime* time.

The larger N and the smaller T are, the higher bandwidth but also the higher error rate can be. We implement cache covert channels in the cross-core scenario and SF covert channels in both the cross-core and cross-cluster scenarios. As shown in Table VI, the results (averaged by 1,000 traces) reveal that the SF covert channel reaches 82% of the bandwidth of the cache covert channel between big cores, and almost the same between little cores. In the cross-cluster scenario, the performance is still satisfying regardless of the core the receiver is running on: 85% of the bandwidth of the cache covert channel between little cores. Hence, we recommend to implement the SF

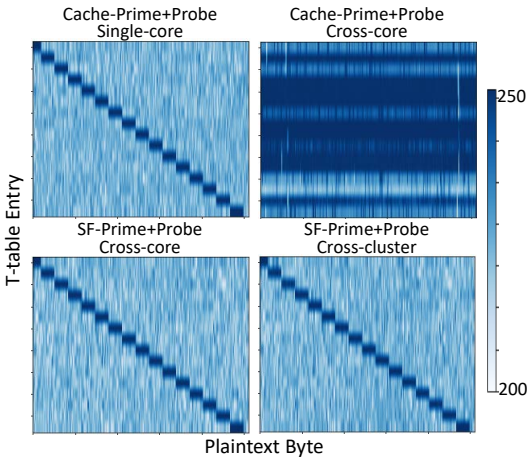

Fig. 7. Secret-dependent access patterns of T-table based AES.

covert channel, as it works with competitive performance, while the cache covert channel would immediately fail once a user program is rescheduled to another cluster by the OS.

B. T-table Based AES

The T-table implementation is one of the optimizations of AES, which precomputes round operations as T-tables. However, T-tables are accessed based on the value of the plaintext and key, forming a secret-dependent access that can be monitored by side-channel attackers. Targeting the 128-bit AES encryption of OpenSSL 1.1.1a, we first implement the first round attack [52] to vividly demonstrate the secret-dependent access of T-tables, as shown in Fig. 7. The access pattern of T-table elements is generated by Prime+Probe, and we iterate the detection 256 times. The results show that SF-Prime+Probe works well between cores and clusters, while Cache-Prime+Probe fails in the cross-core scenario.

To systematically analyze the SF side-channel, we implement the last round attack [53] to fully recover the key and compare SF-Prime+Probe with Cache-Prime+Probe in different scenarios. The implementation of the last round attack mainly follows the previous design, and for every encryption, we use Prime+Probe to monitor all the target cache (or SF) sets. The recovered bits increase with the number of the encryption, as shown in Fig. 8, in which we mark the scenario, the side-channel, and the core the attacker locates. Our attacks depict that SF is a reliable and practical side-channel, as there is no significant difference when using the SF than using the cache in the single-core scenario, while SF-Prime+Probe only takes less than three times the encryption to overcome the bound between clusters.

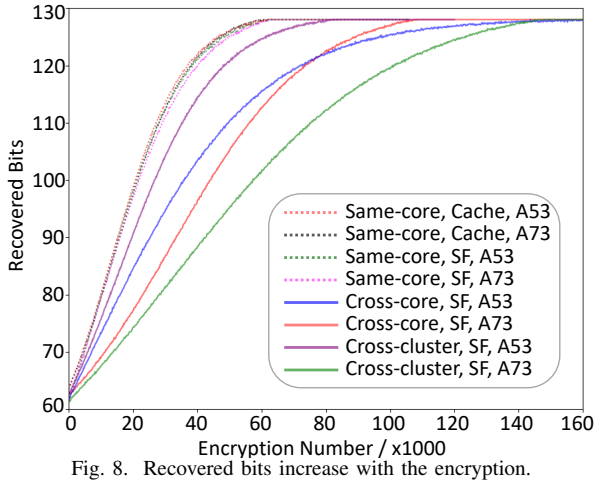

Fig. 8. Recovered bits increase with the encryption.

Algorithm 3: Sliding-window RSA implementation

Input: base b, modulo m, exponent $e = (e_{n-1}...e_0)_2$
Output: $b^e \ mod \ m$
precompute: multipliers $M[2^{w-1}]$ to $M[2^w - 1]$
$r \leftarrow 1$
for $i \leftarrow 0$ **to** $n-1$ **do**
  if $(e_{i+w-1}...e_i)_2$ *matches any* $j \in [2^{w-1}, 2^w - 1]$ **then**
    repeat w **times**
      $r \leftarrow r \times r \ mod \ m$
    end
    $r \leftarrow r \times M[j] \ mod \ m$ // *Secret-dependent access of* M
    $i \leftarrow i + w$
  else
    $r \leftarrow r \times r \ mod \ m$
  end
end
return r

C. Sliding-window Based RSA

The sliding-window algorithm of RSA is shown to be vulnerable to side-channel attacks [2]. We attack RSA in MbedTLS 2.26.0, and its implementation is shown in Algorithm 3. To release the burden of modular computing, the multipliers are pre-computed and stored well, denoted as $M[2^w - 1]$, where w is the window size. The algorithm accesses different multipliers based on the bits in the window, and forms a secret-dependent memory access. Side-channel attackers can monitor the memory access pattern to figure out when and which multiplier has been loaded by the victim. To attack the RSA decryption, the attacker and victim need to operate simultaneously and the attacker keeps conducting Prime+Probe on the cache (or SF) sets that the multiplier is mapped to. In the user space, we target the 4096-bit RSA decryption with $w = 1$. Whenever the bit in the window is '1', the victim needs to load the multiplier, so that the cache (or SF) conflicts can be detected in the *probe* procedure.

To the best of our knowledge, there is no user space cache side-channel attack of RSA on ARM processors. This is because attacks on RSA can only be conducted in the cross-core and cross-cluster scenarios, which are still difficult on ARM platforms. Thus, we are the first to implement cross-core and cross-cluster RSA attacks on ARM big.LITTLE processors. The access latency of 60 profiling samples is shown as Fig. 9, in which the victim is running on the little core, and the attacker, on the big core, monitors the SF set by SF-Prime+Probe. The higher latency means this multiplier has been previously loaded by the victim, indicating that the bit in the window is '1'. When attacking RSA in the cross-cluster scenario, we load the SF EV 8 times in the *prime* procedure and wait for 350 cycles before

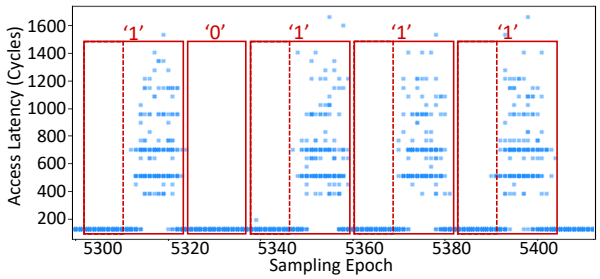

Fig. 9. Secret-dependent memory access of the sliding-window based RSA. From the access latency generated by SF-Prime+Probe, we can recover the key bits as '10111'.

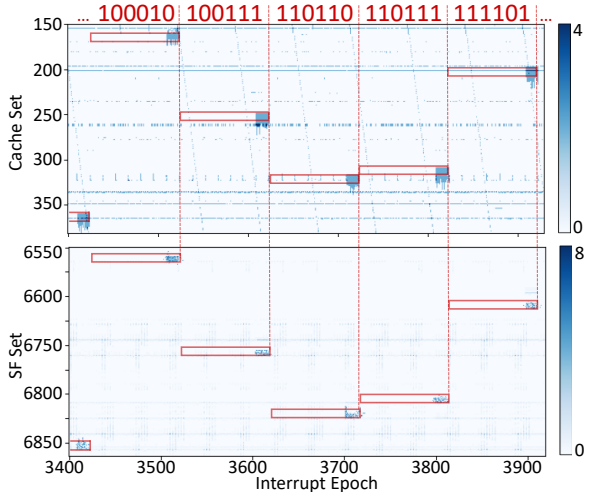

Fig. 10. Secret-dependent access pattern of the sliding-window based RSA. The upper one is generated by Cache-Prime+Probe in the single-core scenario. The lower one is generated by SF-Prime+Probe in the cross-cluster scenario when the strict cache clean defense is applied.

the *probe* procedure. After our evaluation, only 37 samples on average are sufficient to fully recover the private key without error.

D. Attack TrustZone

Trusted execution environments (TEEs) are imported into modern processors to offer stronger protections. Until now, Intel SGX [54] and AMD SEV [55] are the most common TEEs on x86 platforms, and ARM TrustZone [56] is built in most high-end ARM processors. However, previous works [21], [57] reveal that TEEs are still vulnerable to cache side-channel attacks. Presently, there are two attackers [25], [26] exploit the interrupt mechanism to significantly improve the performance of side-channel attacks on TrustZone.

However, a naive cache cleaning defense has already been presented. OPTEE [58] invokes cache-cleaning instructions to flush the core's private caches before world switching. According to our investigation, cache-cleaning instructions on ARM platforms offer the core the ability to flush and invalidate L1 and L2 caches; however, they cannot directly flush a remote cache hierarchy. To allow a core in the secure world to smoothly flush the cache lines of another cluster, we believe rigorous and low-level modifications of TrustZone are needed. Here, we define such a defense as the "strict cache clean defense", which significantly reduces the performance, though, indeed defends the interrupt-based attacks in [25], [26].

We implement a similar framework in which the attacker periodically interrupts the victim's program and then conducts side-channel attacks. We configure our TrustZone environment to be the same as the reference implementation by Trusted Firmware [59], i.e., TF-A + OPTEE + MbedTLS. The victim is running the 4096-bit RSA decryption with the same implementation in Algorithm 3 but with $w = 6$, which means the attacker needs to monitor 32 different multipliers. We interrupt the victim program 52471 times and conduct Prime+Probe attacks during every interrupt epoch. The access patterns of the cache (or SF) sets are shown in Fig. 10. Note that this is the raw data generated by only a single profiling trace, in which we can find out the obvious secret-dependent memory accesses. Despite the strict cache clean defense being enabled, SF-Prime+Probe still generates a clear enough access pattern. We can then easily recover the private key based on attackers' knowledge [2], [26], which is also visualized as the red symbols in Fig. 10. Hence, we propose to exploit the SF side-channel in the kernel space as it achieves the same performance as the single trace recovery in [26] while bypassing the strict cache clean defense.

VI. Mitigation

Hardware-based: There exist hardware implementations to mitigate cache side-channel attacks, e.g., SHARP [11] alters the cache replacement policy and ScatterCache [60] randomizes the mapping of cache sets. With such countermeasures, cache conflicts are expected to be suppressed, and constructing a cache EV becomes harder. However, they may ease SF side-channel attacks since SF EVs are easier to construct if cache conflicts are suppressed. Alternatively, we can implement such countermeasures on SF by preventing SF conflicts, for which we believe case-by-case studies are needed.

Software-based: Software-only defenses [61], [62] usually utilize performance counters to monitor malicious memory access. These defenses will perform better if the anomaly-based detection is further optimized for the SF. In addition, SF conflicts come with the back-invalidation, which can be monitored by the performance encounter of CCI. Other software defenses from the program side, such as the software diversity techniques [63] and the source code level defenses [64], [65], should also work because the SF side-channel attacks still rely on the secret-dependent memory access. Unfortunately, software-based defenses unavoidably degrade the performance and may also destroy the lightweight design of TrustZone.

VII. Conclusion

In this paper, we proposed to attack directories on ARM big.LITTLE processors by unveiling an unexplored structure called SF in ARM CCI. Our systematic reverse engineering and detailed experiments revel that SF is sufficiently capable as a new side-channel, which not only shows competitive performance but also overcomes the difficulties of the cache side-channel in the cross-core and cross-cluster scenarios.

Disclosure

ARM confirmed our work and indicated that our unveiled structure overcomes the lack of shareability in ARM designs that normally thwart cache side-channels between cores and clusters. They believe the threat model is unchanged, thus, similar to dealing with the cache side-channel attacks, they recommend adopting software-based mitigation as we discussed in Section VI.

References

[1] C. Maurice *et al.*, "Hello from the other side: SSH over robust cache covert channels in the cloud." in *NDSS*, 2017.
[2] F. Liu *et al.*, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015.
[3] Y. Yarom *et al.*, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack." *IACR Cryptol.*, 2014.
[4] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019.
[5] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Sec.*, 2018.
[6] G. Irazoqui *et al.*, "Cross processor cache attacks," in *ASIACCS*, 2016.
[7] F. Yao *et al.*, "Covert timing channels exploiting cache coherence hardware: Characterization and defense," *International Journal of Parallel Programming*, 2019.
[8] Y. Yarom *et al.*, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Sec.*, 2014.
[9] D. Gruss *et al.*, "Flush+flush: a fast and stealthy cache attack," in *DIMVA*, 2016.
[10] M. Kayaalp *et al.*, "A high-resolution side-channel attack on last-level cache," in *DAC*, 2016.

[11] M. Yan *et al.*, "Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel attacks," in *ISCA*, 2017.

[12] "Disallow the x86 "clflush" instruction due to dram "rowhammer" problem." [Online]. Available: https://bugs.chromium.org/p/nativeclient/issues/detail?id=3944

[13] B. Gulmezoglu *et al.*, "A faster and more realistic flush+reload attack on AES," in *COSADE*, 2015.

[14] M. Yan *et al.*, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE S&P*, 2019.

[15] M. Green *et al.*, "Autolock: Why cache attacks on ARM are harder than you think," in *USENIX Sec.*, 2017.

[16] "ARM cortex-a series programmer's guide for ARMv8-A." [Online]. Available: https://developer.arm.com/documentation/den0024/a/

[17] S. Wang *et al.*, "High-throughput cnn inference on embedded ARM big. little multicore processors," *TCAD*, 2019.

[18] E. L. Padoin *et al.*, "Performance/energy trade-off in scientific computing: the case of ARM big. little and intel sandy bridge," *IET Computers & Digital Techniques*, 2015.

[19] "ARM flexible access." [Online]. Available: https://www.arm.com/en/products/flexible-access

[20] M. Lipp *et al.*, "Armageddon: Cache attacks on mobile devices," in *USENIX Sec.*, 2016.

[21] N. Zhang *et al.*, "Truspy: Cache side-channel information leakage from the secure world on ARM devices." *IACR Cryptol.*, 2016.

[22] X. Zhang *et al.*, "Return-oriented flush-reload side channels on ARM and their implications for android devices," in *CCS*, 2016.

[23] H. Lee *et al.*, "Hardware-based flush+reload attack on Armv8 system via ACP," in *ICOIN*, 2021.

[24] G. Haas *et al.*, "itimed: Cache attacks on the apple a10 fusion soc," *IACR Cryptol.*, 2021.

[25] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone," in *CCS*, 2019.

[26] Z. Kou *et al.*, "Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict," in *DAC*, 2021.

[27] "Arm corelink CCI-400 cache coherent interconnect technical reference manual." [Online]. Available: https://developer.arm.com/documentation/ddi0470/k/preface

[28] "Arm corelink CCI-500 cache coherent interconnect technical reference manual." [Online]. Available: https://developer.arm.com/documentation/100023/0000/

[29] "Arm corelink CCI-550 cache coherent interconnect technical reference manual." [Online]. Available: https://developer.arm.com/documentation/100282/0100/

[30] A. Agarwal *et al.*, "An evaluation of directory schemes for cache coherence," *SIGARCH*, 1988.

[31] A. Basu *et al.*, "Cmp directory coherence: One granularity does not fit all," *Technical Report# CS-TR-2013-1798*, 2013.

[32] N. Agarwal *et al.*, "In-network coherence filtering: Snoopy coherence without broadcasts," in *MICRO*, 2009.

[33] R. Ulfsnes, "Design of a snoop filter for snoop based cache coherency protocols," Master's thesis, Institutt for elektronikk og telekommunikasjon, 2013.

[34] D. J. Sorin *et al.*, "A primer on memory consistency and cache coherence," *Synthesis lectures on computer architecture*, 2011.

[35] D. Rosenberg, "Reflections on trusting trustzone," *BlackHat USA*, 2014.

[36] J. Jamshed *et al.*, "Snoop filter for cache coherency in a data processing system," in *US Patent 10 157 133B2*, 2015.

[37] "Hikey960." [Online]. Available: https://www.96boards.org/product/hikey960/

[38] "Hikey970." [Online]. Available: https://www.96boards.org/product/hikey970/

[39] "Honor View 10." [Online]. Available: https://www.gsmarena.com/honor_view_10-8938.php

[40] "Smartphones with hisilicon kirin 970 processor." [Online]. Available: https://www.kimovil.com/en/list-smartphones-by-processor/huawei-hisilicon-kirin-960

[41] "Smartphones with hisilicon kirin 970 processor." [Online]. Available: https://www.kimovil.com/en/list-smartphones-by-processor/huawei-hisilicon-kirin-970

[42] P. Vila *et al.*, "Theory and practice of finding eviction sets," in *IEEE S&P*, 2019.

[43] G. Irazoqui *et al.*, "Systematic reverse engineering of cache slice selection in intel processors," in *DSD*, 2015.

[44] J. Edge, "Kernel address space layout randomization," 2014. [Online]. Available: https://lwn.net/Articles/569635/

[45] C. Maurice *et al.*, "Reverse engineering intel last-level cache complex addressing using performance counters," in *RAID*, 2015.

[46] R. Hund *et al.*, "Practical timing side channel attacks against kernel space aslr," in *IEEE S&P*, 2013.

[47] G. o. Irazoqui, "Wait a minute! a fast, cross-vm attack on AES," in *RAID*, 2014.

[48] S. Gueron, "Intel advanced encryption standard (AES) instructions set," *Intel White Paper*, 2010.

[49] R. Könighofer, "A fast and cache-timing resistant implementation of the AES," in *Cryptographers' Track at the RSA Conference*, 2008.

[50] MbedTLS, "Tech updates security advisories." [Online]. Available: https://tls.mbed.org/tech-updates/security-advisories

[51] Z. Wu *et al.*, "Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud," *TON*, 2014.

[52] D. A. Osvik *et al.*, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*, 2006.

[53] E. Tromer *et al.*, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, 2010.

[54] V. Costan *et al.*, "Intel SGX explained." *IACR Cryptol.*, 2016.

[55] W. Arthur *et al.*, "Platform security technologies that use TPM 2.0," in *A Practical Guide to TPM 2.0*, 2015.

[56] "ARM security technology building a secure system using trustzone technology," 2017. [Online]. Available: https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture

[57] A. Moghimi *et al.*, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[58] "OPTEE." [Online]. Available: https://www.op-tee.org/

[59] "Trusted firmware." [Online]. Available: https://www.trustedfirmware.org/

[60] M. Werner *et al.*, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX Sec.*, 2019.

[61] T. Zhang *et al.*, "Cloudradar: A real-time side-channel attack detection system in clouds," in *RAID*, 2016.

[62] M. Chiappetta *et al.*, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, 2016.

[63] S. Crane *et al.*, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015.

[64] A. Rane *et al.*, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Sec.*, 2015.

[65] B. Coppens *et al.*, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *IEEE S&P*, 2009.