Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone

Haehyun Cho
Arizona State University
haehyun@asu.edu

Penghui Zhang
Arizona State University
pzhang57@asu.edu

Donguk Kim
Samsung Research
donguk14.kim@samsung.com

Jinbum Park
Samsung Research
jinb.park@samsung.com

Choong-Hoon Lee
Samsung Research
choonghoon.lee@samsung.com

Ziming Zhao
Rochester Institute of Technology
zhao@mail.rit.edu

Adam Doupé
Arizona State University
doupe@asu.edu

Gail-Joon Ahn
Arizona State University
Samsung Research
gahn@asu.edu
gailjoon.ahn@samsung.com

ABSTRACT

The security of ARM TrustZone relies on the idea of splitting system-on-chip hardware and software into two worlds, namely normal world and secure world. In this paper, we report cross-world covert channels, which exploit the world-shared cache in the TrustZone architecture. We design a Prime+Count technique that only cares about *how many* cache *sets* or *lines* have been occupied. The coarser-grained approach significantly reduces the noise introduced by the pseudo-random replacement policy and world switching. Using our Prime+Count technique, we build covert channels in single-core and cross-core scenarios in the TrustZone architecture. Our results demonstrate that Prime+Count is an effective technique for enabling cross-world covert channels on ARM TrustZone.

CCS CONCEPTS

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

KEYWORDS

Cache side-channel, Covert channels, ARM TrustZone

ACM Reference Format:
Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, San Juan, Puerto Rico, USA, 12 pages. https://doi.org/10.1145/3274694.3274704

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274704

1 INTRODUCTION

ARM Security Extensions, marketed as TrustZone, have been introduced in ARMv6 and later profile architectures, including Cortex-A (mobile) and Cortex-M (IoT) [2–4]. The idea of TrustZone is to split the system-on-chip hardware and software into two security states or worlds, namely *normal world* and *secure world.* Hardware barriers are established to prevent normal world components from accessing secure world resources.

Two legitimate channels exist at the hardware level that a normal world component and a secure world component can use to communicate with each other. The first channel is that either world can put messages in the general registers when a world switching is performed. The second channel is the secure world can directly read and write to a region of physical memory that normal world can also access.

Previous studies have shown that these legitimate channels are vulnerable to an attacker who has the normal world kernel privileges and keeps sending crafted arguments to probe the vulnerabilities of the secure world [18, 19, 26, 33]. There are two ways to protect these channels from being abused:

(1) Prior work, SeCReT [18], has aimed at restricting the access to the communication channels and secure world resources to normal world components on an access control list (ACL). SeCReT ensures only predefined and legitimate normal world components can communicate and access secure world resources. To this end, SeCReT authenticates a normal world component by verifying its code and control integrity when it initiates communication with secure world. Consequently, unauthenticated normal world components cannot access the cross-world communication channels.

(2) It is possible to deploy a strong monitor, similar to a network intrusion detection or deep packet inspection system, in legitimate communication channels, including parameters passed by registers and shared memory, between the normal and secure world to inspect all transmitted data and block illegal communication when it is detected. Even though how to design such strong monitors is a research problem itself, and no practical solutions exist to the best of our knowledge, we assume that they could exist in the future.

In this paper, we are interested in building cross-world covert channels in the TrustZone architecture that (1) enable unauthenticated normal world and secure world components to communicate even when solutions like SeCReT are deployed; (2) enable normal world and secure world components to communicate even when strong monitors that can inspect all transmitted data in legitimate channels are deployed in the future. As a result, a secure world component can always smuggle sensitive information that is not supposed to leave the secure world to the normal world, such as private keys, user passwords, etc. And, a normal world component can send secret messages (e.g., command and control messages) to secure world.

The emergence of downloadable Trusted Applications (TAs) gives such covert channels even more practical use-scenarios [39], where a malicious TA can steal sensitive information that does not belong to it in the secure world and send to its counterpart in the normal world, hence circumventing SeCReT and strong monitor.

We propose to build covert channels using a trade-off between performance and cost in the TrustZone hardware, which are not governed by any software solution built on top of TrustZone, such as SeCReT or strong monitors. We notice that even though many system-on-chip resources are separated in the TrustZone architecture, there is only one copy of cache in the system that is shared between the two worlds.

Even though it is easy to assume covert channels must exist given that cache is shared between worlds, there is no comprehensive study on the practicality and bandwidth of cross-world covert channels in the TrustZone architecture. In this paper, we identify several challenges in building such cross-world covert channels: (1) the pseudo-random replacement policy on ARM makes Prime+Probe less reliable [21]; (2) the cross-world context switching also introduces much noise. Our work confirms that Prime+Probe is not reliable in the cross-world scenario; (3) low noise and fine-grained cache line-level attacks, such as Flush+Reload [43] and Flush+Flush [12], require sharing memory objects between the *Sender* and the *Receiver*, which does not fit in a practical attack model.

To cope with these challenges, we need a novel cache attack approach that does not require memory sharing and introduces less noise in the cross-world scenario. In this paper, we leverage an overlooked ARM Performance Monitor Unit (PMU) feature named "L1/L2 cache refill events" and design a Prime+Count technique that only cares about *how many* cache *sets* or *lines* have been occupied instead of determining *which* cache *sets* have been occupied as in Prime+Probe. The coarser-grained approach significantly reduces the noise introduced by the pseudo-random replacement policy and world switching. Even though some performance counters in PMU, such as cycle counter, have been used to carry out and detect cache-based side-channel attacks in the ARM and Intel architecture [10, 42], to the best of our knowledge it is novel to use "L1/L2 cache refill events" to perform attacks.

We leverage the Prime+Count technique to build covert channels in single-core and cross-core scenarios in the TrustZone architecture. To evaluate the efficacy of the covert channels, we test our implementations on two devices, one of which is a Samsung Tizen TV with ARMv7 CPU and the other is a Hikey board with ARMv8 CPU. The evaluation results show that the bandwidth could be as high as 27 KB/s in the single-core scenario and 95 B/s in the cross-core scenario.

2 BACKGROUND

2.1 ARM Architecture and TrustZone

Processor Modes. An ARM processor has up to 9 different modes depending on if some optional extensions have been implemented. The user (`usr`) mode has a privilege level 0 and is where user space programs run. The supervisor (`svc`) mode has a privilege level 1 and is where most parts of kernel execute.

TrustZone and Processor States. TrustZone is a hardware security extension of the ARM processor architecture, which includes bus fabric and system peripherals. When TrustZone is implemented, a processor has two security states or worlds, namely the secure world (`s`) and the normal world (`ns`). The distinction between the two states is orthogonal to the processor modes. The partitioning of all the System on Chip (SoC)'s hardware and software into two worlds may be physical and/or virtual. For instance, a processor core is shared by the normal and secure world in a time-sliced fashion. World switching is done in the monitor mode after calling the secure monitor call (SMC) instruction in either world. The SMC instruction forces the running core to enter the monitor mode.

2.2 Legitimate Channels between the Normal and Secure Worlds

At the hardware layer, there are two ways for a normal world and a secure world component to communicate with each other. Firstly, messages can be stored in the general registers when a world switching happens, which is triggered by the `SMC` instruction. For instance, secure monitor call calling convention [5] defines how parameters are passed through the general registers, and it is implemented in firmware, such as ARM Trusted Firmware [6]. Previous projects, such as SeCReT [18], attempted to add extra layers of authentication and verification to make sure only predefined and legitimate components can use this channel.

Secondly, the secure world kernel can directly map a memory region that is accessible by the normal world. Hence, this shared memory region can be used by the normal and secure world to communicate. Secure world OSes, such as OP-TEE [28], have implemented shared memory. Usually, a physical memory region is first allocated by the normal world kernel. The physical address, the size of the shared memory, and other important information are then transferred to the secure world OS through the `SMC` interface, so the secure world can configure its MMU table entries to access the region directly. Since important information is still passed through the `SMC` interface, solutions such as SeCReT can also monitor this channel. Besides SeCReT, we can assume strong monitors can be implemented in the future to inspect all transmitted data in these channels.

2.3 ARM Cache Architecture

A cache is a relatively small but fast array of memory, which is usually placed between a CPU core and the main memory. In the ARM architecture, each core has its own dedicated L1 cache, which

is separated into instruction cache (I-cache) and data cache (D-cache). The separation of instruction and data cache allows transfers to be performed simultaneously on both instruction and data buses and increases the overall performance of L1 caches. In the ARM architecture, a unified L2-cache is shared among the cores in a core cluster.

Inclusiveness. Depending on if a high-level cache holds the data of a lower level cache, a cache architecture can be categorized into the following three classes: (1) *inclusive cache*: a line in L2 cache will be evicted when new data is fetched even even though the line is in L1 cache; (2) *strict inclusive cache*: a cache line in L2 cache cannot be evicted by new data if the data is stored in L1 cache, which is known as *AutoLock* by a prior work [11]; (3) *exclusive cache*: a cache line will only be stored in one of the cache levels. Inclusiveness of a cache is important for cross-core cache attacks. In the ARM architecture, cache is not exclusive mostly, which enables cross-core cache-based attacks.

Set Associativity. For efficiency reasons, multiple adjacent words in memory are moved in or out from the cache in a single cache load or eviction. And, the smallest unit in a cache is called a *cache line*. In the modern cache architecture, a cache is organized in multiple *cache sets*. And, adjacent memory data with the size of a cache line can be stored into any cache line in the same set. If a cache set has N cache lines, the cache is called an N-way associative cache.

Replacement Policies. In set-associative caches, to decide which specific cache line to use in a particular set several policies can be utilized: (1) *Least-recently-used replacement policy*: the least recently used cache entry in a cache set will be replaced. Intel architecture uses this policy [22]; (2) *Round-Robin replacement policy*: the cache lines that are first filled will be cleared first; (3) *Pseudo-random replacement policy*: a random cache line will be evicted. In the ARM architecture, a pseudo-random replacement policy is used, which usually makes cache-based attacks harder to implement [21, 31].

Cache in TrustZone Architecture. Unlike some banked system registers, there is only one copy of cache that is shared between normal and secure world. Each cache line has one bit to indicate if its content is from a secure or normal world memory region. Even though this extend bit can prevent normal world components from accessing cache contents of secure world, the design of shared cache still makes some cross-world cache attacks possible.

2.4 Previous Cache Attacks

Previous cache attacks utilize time differences between a cache hit and a cache miss to infer whether specific code/data has been accessed. We briefly overview several attacks that have been widely exploited on both Intel and ARM architectures.

Both Evict+Time and Prime+Probe can be used to determine which cache *sets* have been accessed by a victim [29]. Both of them have been used to reconstruct cryptography key in a victim program [15, 22, 29] and perform cross-VM attacks [24, 44–46]. In these two approaches, attackers can only achieve set-level granularity, but they do not need to map objects in the memory space of the victim into their own memory space. Previous research effort also showed that the pseudo-random replacement policy on ARM makes Prime+Probe much harder than it is on Intel architectures [21, 42].

The objective of Flush+Reload is to determine which specific cache *lines* are accessed by a victim program. First, the attacker maps objects in the victim's address space into its own. The attacker then flushes specific cache lines, schedules the victim program, and checks which the cache lines that were flushed have been reloaded. This technique was first implemented using the `CLFLUSH` instruction provided in the Intel architecture [41], and it has been used to extract cryptographic keys [14, 16, 17]. Evict+Reload was proposed for ARM by replacing the flush action with eviction [13, 43].

Because the cache references and misses caused by Flush+Reload and Prime+Probe could be monitored by hardware performance counters, Gruss et al. [12] proposed Flush+Flush that only relies on the execution time of the flush instruction to detect if a cache *line* has been loaded by a victim program.

Even though Flush+Reload, Evict+Reload and Flush+Flush provide finer-grained attacks at the cache line level, they all need shared memory between an attacker program and a victim program. In this paper, we assume the secure world and normal world communication parties do not share memory. Therefore, these techniques cannot be adopted.

3 ASSUMPTIONS AND ATTACK MODEL

We assume a solution, such as SeCReT [18], that only allows authenticated normal world components to use the communication channel, is running in secure world monitor mode. Such a solution safely maintains a list of predefined normal world components that are allowed to use the legitimate channels. We also assume that there is a strong monitor that can understand all transmitted data between the normal and secure world and block illegal communications.

The goal of an attacker is to smuggle sensitive information that is only accessible in the secure world to the normal world. To this end, the attacker runs a component, namely *Receiver* in the normal world and another component, namely *Sender* in the secure world. Because legitimate communication channels, including parameters passed by registers and shared memory, between the normal world and the secure world are under inspection by a SeCReT or a strong monitor, it is impossible for the *Sender* and the *Receiver* to transfer sensitive data from the secure to the normal world using such channels without being detected. To bypass this kind of cross-world communication monitoring, the *Sender* and the *Receiver* need to use channels that are not governed by the sentries implemented in the monitor mode.

We assume the attacker has kernel privileges in the normal world, so the *Receiver* can use privileged instructions to access the PMU. This constraint can be loosened if the *perf_event_open* system call is provided to monitor "L1/L2 cache refill events" in userland. The *Sender* can simply be a secure world application (trusted application), and it is not necessary for it to have kernel privileges. This is because the *Sender* will only need to influence cache by reading/writing memory regions and does not need to access the PMU. However, having the *Sender* running in the kernel space enables it to steal information that is not available for userland processes. Running an application in the secure world is very feasible for the attacker who can either leverage vulnerabilities of the secure

Device	SoC	CPU (cores)	L1 Cache	L2 Cache	Inclusiveness	Secure World OS	Normal World OS
Samsung Tizen TV	N/A	32-bit Cortex-A17 (4)	32KB, 4-way, 128 sets	1024KB, 16-way, 1024 sets	Inclusive	SecureOS	Tizen OS on Linux 4.1.10
Hikey board	HiSilicon Kirin 620	64-bit Cortex-A53 (8)	32KB, 4-way, 128 sets	512KB, 16-way, 512 sets	Inclusive	ARM Trusted Firmware, OP-TEE	Linux 4.1.0

Table 1: Test Environments.

world interfaces as shown in [20, 27] or bypass application vetting mechanisms [9]. The use of downloadable TAs, which are predicted to be used widely [39], would increase the chance as well.

In summary, in our attack model attackers are not stronger than their counterparts in previous events [26, 33] or in the attack model presented in SeCReT [18], except that the *Sender*, which can be a userland application, is a must. Our implementation suggests such an application could be implemented in hundreds of lines of C code. Moreover, our attack can be carried out even when mechanisms, such as strong monitors, that are more powerful than normal world component authentication, such as SeCReT, are deployed between the two worlds.

Depending on the hardware the attack is performed on and resources the attacker possesses, we articulate two attack scenarios: single-core and cross-core.

(1) *Single-core scenario*: This scenario occurs when either the targeted device only has a single-core CPU or the attacker can only control one of the cores in a multi-core CPU. Because there is only one core available to the attacker, the attacker needs to use the SMC instruction to switch between the normal and secure world. In addition, in this scenario the attacker can use either L1 cache or L2 cache. Note that even if the attacker can use the SMC instruction in this scenario, it is not possible to send sensitive information directly using the SMC instruction or shared memory due to the sentry in the monitor mode;

(2) *Cross-core scenario*: In this scenario, the attacker can execute the *Receiver* in the normal world and the *Sender* in the secure world on two different cores at the same time. Because different cores do not share L1 cache, the covert channel can only be constructed using the L2 cache. Therefore, the inclusiveness of L2 cache affects the result. In this scenario, there is no need for the attacker to use the SMC instruction to switch between the worlds.

In this paper, we attempt to solve the challenges in building cross-world covert channels in both aforementioned scenarios. All experiments are performed on the two environments as listed in Table 1. In addition, we use a TRACE32 hardware debugger[1] to trace cache operations on the Tizen TV.

4 CROSS-WORLD COVERT CHANNELS

At a high level, to build cache-based covert channels, the *Receiver* first makes the whole cache or some specific cache lines enter a known state. To this end, the *Receiver* can fill the cache with contents from its own address space. In the second step, the *Sender* carefully changes states of some cache lines by evicting the contents of those lines and placing its own contents there. In the third step, the *Receiver* detects such changes to decipher the message the *Sender* transmits. Note that, in almost all the platforms, neither the *Sender* nor the *Receiver* can directly read the content of any cache line. Therefore, the message is actually delivered using channels such as *which* specific cache lines or sets have been changed in previous projects [21, 32, 37]. To receive such information, the *Receiver* accesses its own address space again and uses cache hit or miss to detect how many cache lines have been changed.

Algorithm 1: Prime+Count-based Cross-world Covert Channels. x is the message to be sent.

```
   /* Receiver: Prime                                          */
1  if Single-core covert channel then
2      for Each L1-D cache line do
3          Clean & Invalidate the L1-D cache line
4          Load data to fill the L1-D cache line
5      Yield control to the secure world by executing SMC
6  if Cross-core covert channel then
7      for Each L2 cache line do
8          Clean & Invalidate the L1-D cache line
9          Clean & Invalidate the L2 cache line
10         Load data to fill the L1-D & L2 cache lines
11     Clean & Invalidate the whole L1-D cache
   /* Sender: Write to covert channel                          */
12 if Single-core covert channel then
13     Occupy x L1-D cache lines
14     Yield control to the normal world by executing SMC
15 if Cross-core covert channel then
16     Occupy x L2 cache lines
   /* Receiver: Count                                          */
17 Determine how many cache lines are changed by Sender
18 Apply bucket method for further noise reduction
```

Our approach follows this general idea with some changes that are tailor-made for the TrustZone architecture. In particular, we propose Prime+Count, it uses *the number* of changed cache lines as the covert channel instead of *which* cache lines or sets. Algorithm 1 demonstrates the overall workflow of building cross-world covert channels using Prime+Count. As shown in Lines 2–4 and 7–11, the *Receiver* first Primes the cache. Because covert channels are based on the number of cache misses, the results of the Prime step can have a strong influence on the reliability and bandwidth of the covert channel. Due to the pseudo-random cache replacement

[1] http://www.lauterbach.com/

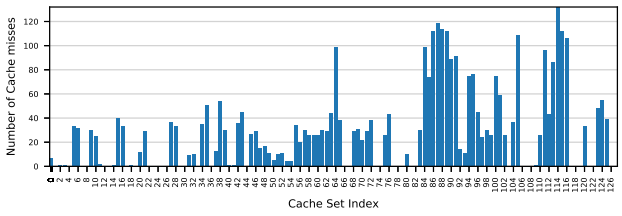


Figure 1: Cache misses introduced by world switching.

policy, an effective and efficient PRIME method is not very straightforward. We discuss the PRIME method in detail in Section 4.2.

In the single-core scenario, the *Receiver* then needs to yield control to the secure world so the *Sender* can execute as shown in Line 5. In the cross-core scenario, this step is omitted. Then, as shown in Lines 13 and 16, the *Sender* writes data to the covert channel by occupying x cache lines, where x is the message to be sent. In this step, the cache replacement policy could be the obstacle again. Consequently, a similar method in PRIME is used for accurate message writing. In the single-core scenario, the *Sender* then yields control to the normal world so that the *Receiver* can decode the message as shown in Line 14. Lastly, the *Receiver* COUNTS how many cache lines are changed as shown in Line 17 and uses a simple noise reduction method to get the message as shown in Line 18.

The main difference in single-core and cross-core scenario is that the L2 cache is used in the cross-core scenario instead of the L1-D cache. We discuss the details of the differences in Section 4.5.

4.1 PRIME+COUNT Overview

4.1.1 Why not PRIME+PROBE? Intuitively, PRIME+PROBE can be used to build cross-world covert channels in our attack model. It is not the best option due to the following reasons:

(1) *Noisy:* Due to ARM's pseudo-random replacement policy, Lipp et al. demonstrated that PRIME+PROBE is not reliable [21]. The world switching introduced by TrustZone increases the ineffectiveness of PRIME+PROBE. In addition, during the time when the normal world part of the covert channel is working, other kernel code executing on the same core can introduce extra noise.

We conducted several experiments on both devices to show how much noise can be introduced on each set of the L1-D cache during the world switching after the PRIME. In the experiments, the secure world simply yields control to the normal world after loading a specific number of cache sets. Figure 1 shows how many cache misses occurred for each cache set in 200 world switchings on the Hikey board. The x-axis is the index of the cache sets from 0 to 127, and the y-axis is the accumulated number of cache misses. The experiments suggest the noise is widely dispersed on the cache sets and the average number of cache misses per world switching is around 18 over 128 cache sets. Even though Figure 1 shows some cache sets, such as cache set 1, are never used during the world switching in our experiments on the Hikey board, it does not mean that those cache sets are guaranteed to stay intact when other hardware devices or different firmware and OS are used. Hence, it is not feasible to use this observation to build generic covert channels for a variety of hardware and software environments.

Algorithm 2: An Alternative Method.

```
/* The following lines replace Lines 2-4 in
   Algorithm 1                                  */
1 for Each L1-D cache set do
2     Clean & Invalidate the L1-D cache set
3 for Each L1-D cache set do
4     Load data to fill the L1-D cache set
```

(2) *Difficult to choose threshold:* One way to tell a cache hit from a cache miss is to use the Performance Monitors Cycle Count Register (PMCCNTR) that increments from the hardware processor clock. By subtracting a previously recorded PMCCNTR value (p_1) from its current value (p_2), the number of elapsed processor cycles ($\Delta = p_2 - p_1$) can be easily computed. To distinguish between cache hit and miss for a memory access, PMCCNTR is read before and after the memory access attempt. If the number of elapsed cycles is greater than some predefined threshold ($\Delta > \theta$), the attempt is classified as a cache miss; otherwise, it is considered as a cache hit. This approach has been used in PRIME+PROBE and other cache attacks. However, *the thresholds used for decision making are contingent upon the implementation of the CPU*, which means there is no one-size-fits-all threshold value for all available devices on the market. Even though Lipp et al. proposed a mechanism to automatically compute the threshold at run-time [21], it inevitably increases the size of the attack code base and the chance to be discovered.

4.1.2 WHY PRIME+COUNT? PRIME+COUNT counts *how many* cache *sets* or *lines* have been occupied instead of determining *which* cache *sets* have been occupied. PRIME+COUNT, as a coarser-grained approach than PRIME+PROBE, significantly reduces the noise introduced by the random replacement policy and world switching. In addition, PRIME+COUNT does not require shared memory space or shared memory objects with a victim. PRIME+COUNT only cares about how many cache sets/lines have been changed. Therefore, it may be difficult to use it for some attacks other than building covert channels, such as stealing cryptographic algorithm keys.

4.2 PRIME the Cache

Ineffective PRIME affects the accuracy of COUNT and adds noise to the covert channel. It is suggested that the pseudo-random cache replacement policy is a significant obstacle in PRIME [21, 42]. Taking a 4-way set associative cache as an example, based on the index of the physical address newly fetched data can be loaded to any of the 4 ways. Therefore, even if we load as much data as the size of the L1-D cache, there is no guarantee that the cache will be completely occupied.

4.2.1 Previous PRIME Method. Previous approaches to achieve high cache coverage in PRIME for userland programs load data repeatedly using various access patterns [21, 42]. However, this type of approach costs thousands of CPU cycles even when it is only used to prime a small portion of the cache [21].

Also, our experiments confirm that repeating the data loading at kernel level is costly. We perform a systematic analysis using the TRACE32 hardware debugger to dump the content of cache on

a Samsung Tizen TV. To this end, we prepare 32 KB of memory space, the same size of the L1-D cache. Then, we access the first byte of the memory and keep accessing data at the address that is 64 bytes (size of cache line) away from the one before. After repeating this operation for 512 times (128 sets × 4 ways), we dump the content of cache using the TRACE32 debugger. To minimize possible interference, we use a spinlock to give our experiment code exclusive use of the core. Our results show that, on average, only 372 of 512 cache lines were occupied after accessing the 32 KB memory once. Only by repeating this procedure for more than 50 times could it achieve around 95% cache occupation.

4.2.2 Our Prime Method. Obviously, a faster Prime method could significantly increase the bandwidth of covert channels and reduce the chance of being discovered. In this paper, as shown in Lines 3–4 in Algorithm 1 we clean and invalidate each cache line before only loading the data to cache once. Our experiments show that this method achieves around 99% occupation on average.

This method operates as follow: (1) The starting address of a memory block is assigned to the pointer; (2) We translates virtual address to physical address. Once the physical address is obtained, we can extract its *set* number; (3) After that, we select the target cache line among the lines (ways) in the set using the DC CISW instruction. The DC CISW instruction's operands are a *set* number and a *way* number, and thus, we can choose a specific cache line (way) in a set to clean and invalidate it. We typically start from the way 0 to the last way; (4) Lastly, we load the data to the cache line. The pointer is increased by the length of a cache line so that we can point to the next cache set of the way in the next round. If the *way* has been fully filled by data, we fetch data to the next way. Steps (1) – (4) are iterated until Prime is done.

We also conduct experiments with an alternative method shown in Algorithm 2. In this method, we clean and invalidate all cache lines of the L1-D cache before loading the data. Experiments show that this method achieves around 95% occupation on average.

4.3 Count Using Cache Refill Events

The Performance Monitor Unit (PMU) includes logic to gather various statistics on the operations of the processor and memory system during runtime. We use overlooked PMU features called "L1/L2 Cache Refill Event" to count how many cache lines have been updated. A cache refill event can be triggered by any access causing data to be fetched from outside the cache. Therefore, every cache miss can be counted by using the event.

After the secure world occupies some cache lines using the Prime method, it yields control to normal world, and Count function will execute. If a cache line is refilled while accessing the memory, the counter will increment. Therefore, this function gives us how many cache lines have been changed between Prime and Count.

4.3.1 Two Counting Modes. There are two counting modes we use in the experiments:

Line-counting mode. The smallest unit for counting a cache refill event is a line. For example, if the L1-D cache is a 128-set 4-way cache, we can check each of the 512 lines to count how many refill events occur. In this mode, the covert channel can transmit at most 9 bits ($log_2 512$) every time.

Set-counting mode. Another option is to count the cache refill events on only one way, so just 128 lines will be checked. A way can be chosen by using the DC CISW instruction. In this mode, the covert channel can transmit at most 7 bits ($log_2 128$) every time. However, we only need to Prime one way in this mode. Therefore, this mode can achieve higher bandwidth than the line-counting mode.

4.3.2 Defeating Data Prefetching. One of the challenges we encountered in implementing Count is the automatic data prefetcher [2, 4]. Data prefetching is a technique that fetches data into the cache earlier than the instruction that uses the data is executed. To do so, the prefetcher monitors data cache misses and learns an access pattern. However, a data prefetching does not trigger a refill event. So, the counter will not increment when a new cache line fill is caused by data prefetching.

There are several methods to prevent data prefetching. One way is to disable the prefetcher directly by changing the corresponding bit in the auxiliary control register. However, it is only safe to do so after the MMU is enabled, which does not fit in our attack model. Moreover, disabling the prefetcher will downgrade the performance of the system. Another way is to access memory locations in a random and unpredictable order, so it is difficult for the prefetcher to learn a pattern. However, this method increases the complexity of implementing covert channels.

We solve the problem by employing the instruction synchronization barrier (`ISB`). The `ISB` instruction flushes the pipeline of a core and the prefetcher buffer as well. It is normally used when the context or system registers are changed as well as after the branch predict maintenance operations.

4.4 A Simple Message Encoding Method

Even though Prime+Count introduces significantly less noise than Prime+Probe, noise is still inevitable due to the world switching and other factors. One way to correct the errors introduced by noise is to adopt error correction encoding methods, such as Reed-Solomon error correction [30]. However, those encoding methods significantly (1) increase the size of message, (2) are time consuming to perform, and (3) increase the size of the code base. Hence, adopting those methods could even further reduce the bandwidth of the covert channel and increase the chances of being discovered. A recent study also suggests that directly applying error-correcting codes does not work due to cache-based covert channel noise characteristics [25].

Fortunately, our empirical experiments show that the introduced noise in Prime+Count (error in number of cache refill events) is manageable. Therefore, we design a simple encoding method, which essentially ignores the least significant bits of the received data. We call this approach the *bucket method*.

The basic idea of the bucket method is to divide the numbers of cache refill events into several groups. Table 2 illustrates one example of using the bucket method when 2 bits of data are transferred from the secure world using a 7-bit channel (128 sets in set-counting mode). In this example, when the *Sender* wants to send message 2, it will try to occupy 70 cache lines, which may result in 85 cache refill events detected by the *Receiver*. The *Receiver* then uses the

Message to be Sent	# of Cache Sets should be Occupied by the *Sender*	# of Cache Events detected by the *Receiver*	Bucket Ranges set by the *Receiver*	Message decoded by the *Receiver*
0	10	23	0 - 43	0
1	40	60	44 - 71	1
2	70	85	72 - 99	2
3	100	111	100 - 128	3

Table 2: An example of the bucket method. We assume there are 128 sets and set-counting mode is used. The channel can transmit as most 7 bits ($log_2 128$) every time. In this example, only 2 bits are transmitted.

bucket method to decode the message back to 2. The range of a bucket should be decided empirically.

4.5 Cross-Core Covert Channels

We use the same PRIME+COUNT approaches as the single-core covert channel for cross-core covert channel except for the level of the cache refill event. Besides that, as shown in Algorithm 1 Line 11, the whole L1-D cache should be cleaned and invalidated after the PRIME. If the L1-D cache has data which was used to occupy the L2 cache after the PRIME, the remaining data in the L1-D cache will cause cache hits during COUNT even if the secure world the *Sender* loads all cache lines of the L2 cache. Cleaning and invalidating the L1-D cache using the `DC CISW` instruction does not affect the L2 cache.

Because the L2 cache is shared by many cores and the cache size is much bigger than the L1-D cache, in practice it is impossible to prevent other cores from changing the cache lines during the time of PRIME or after PRIME. Therefore, the noise caused by other cores makes line-counting mode infeasible for building cross-core covert channels. Consequently, we design a modified set-counting mode. The set-counting mode for the single-core environment counts cache misses of one way. For the cross-core covert channel, we check cache misses of all cache lines in a set spanning all ways.

5 IMPLEMENTATION

We implemented the PRIME+COUNT method and covert channels using PRIME+COUNT on the two devices as listed in Table 1. Also, we open source the prototype with the expectation that it will be utilzed and extened by security researchers[2].

The software implementation consists of a normal world module (the *Receiver*) and a secure world module (the *Sender*) to simulate the scenario that the *Sender* tries to smuggle sensitive data out to the normal world. Note that with a simple implementation twist the PRIME+COUNT technique and covert channels based on it can be used to send data from the normal world to the secure world as well.

In the single-core scenario implementation, the normal world module is a loadable kernel module (LKM) that can execute the `SMC` instruction directly. In the case of the Samsung Tizen TV, the *Sender* is a secure world application that does not have kernel privileges. To invoke the application, a new `SMC` handler is added to the kernel of the `secureOS`. Note that, in Samsung Tizen TV, only authenticated trusted applications can be loaded on the `secureOS`, and only Root TA can load LKM to the kernel of `secureOS`. Therefore, in practice a malicious kernel-level *Sender* needs to bypass Samsung's code vetting first. For the Hikey board implementation, we implemented the *Sender* in kernel by modifying the `tee_entry_fast` function in the `entry_fast.c` of the OP-TEE.

In the multi-core covert channel scenario, we implemented two kernel threads in the normal world and assigned each of them to a different physical core. One of the threads acting as the *Receiver* stays in the normal world. The other kernel thread executes `SMC` and invokes the *Sender* in the secure world. The *Sender* and the *Receiver* use L2 cache to communicate.

The normal world kernel module consists of 1,134 SLoC for both test environments. The secure world implementation on the Hikey board consist of 84 SLoC, whereas the secure world application on Samsung Tizen TV has 319 SLoC.

6 EVALUATION

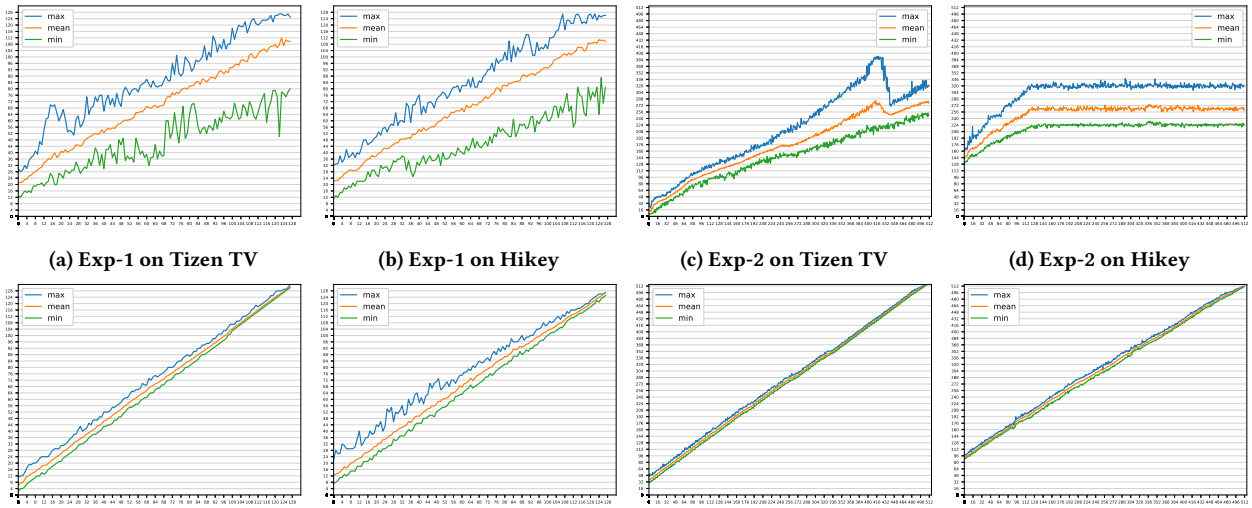
In this section, we report the evaluation results of cross-world PRIME+COUNT-based covert channels on the TrustZone architecture. In section 6.1, we evaluate how much noise our PRIME+COUNT method could reduce compared with previous approaches. Section 6.2 discusses how we choose bucket ranges in the experiments. Section 6.3 measures the bandwidth of covert channels under different conditions. In Section 6.4 shows images transferred using covert channels.

6.1 Effectiveness of PRIME+COUNT

6.1.1 Single-core Scenario. We designed four experiments to demonstrate the effectiveness of our PRIME+COUNT method under the single-core scenario. In each experiment, the *Sender* tries to load a specific number of lines/sets (x-axis), and the *Receiver* detects how many lines/sets changes (y-axis). The configurations of experiments are listed as follows: (1) Exp-1: PRIME with repeated loading 50 times, no instruction barrier, set-counting mode; (2) Exp-2: PRIME with repeated loading 50 times, no instruction barrier, line-counting mode; (3) Exp-3: Our PRIME method, with instruction barrier, set-counting mode; (4) Exp-4: Our PRIME method, with instruction barrier, line-counting mode.

We repeated the experiment on each device 1,000 times. Figure 2 shows the evaluation results. The x-axis represents the number of cache lines/sets loaded by the secure world *Sender*, whereas the y-axis represents how many L1 cache refill events were detected by the *Receiver*. The blue line indicates the maximum number of cache fill events detected, whereas the green line shows the minimum

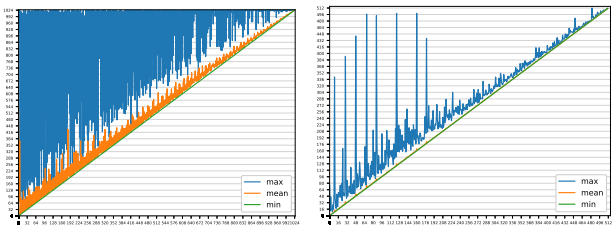
[2]https://github.com/Samsung/prime-count



(a) Exp-1 on Tizen TV (b) Exp-1 on Hikey (c) Exp-2 on Tizen TV (d) Exp-2 on Hikey

(e) Exp-3 on Tizen TV (f) Exp-3 on Hikey (g) Exp-4 on Tizen TV (h) Exp-4 on Hikey

Figure 2: We conducted multiple experiments on both devices to demonstrate the effectiveness of our PRIME method and instruction barrier under the single-core scenario. (1) Exp-1: PRIME with repeated loading, no instruction barrier, set-counting mode; (2) Exp-2: PRIME with repeated loading, no instruction barrier, line-counting mode; (3) Exp-3: Our PRIME method, with instruction barrier, set-counting mode; (4) Exp-4: Our PRIME method, with instruction barrier, line-counting mode. By comparing the first row and the second row, we can clearly see that the variance of noise is significantly reduced using our PRIME method with instruction barrier.



(a) Exp-5 on Tizen TV (b) Exp-5 on Hikey

Figure 3: Number of Loaded Cache Sets versus Detected L2 Refill Events under the Cross-core Scenario.

number of cache fill events detected. The orange line denotes the average over the 1,000 experiments.

From the first row of Figure 2, which is the previous PRIME approach on both devices, we can see those approaches are far from reliable and the gaps between the maximums and minimums are large. It is particularly interesting to see the number of cache refill events will even stay at around 256 on average no matter how many lines the *Sender* tries to load as shown in Figure 2-(d). We tried to look for explanations and failed to find any answers in any official specifications of Hikey or ARM documents.

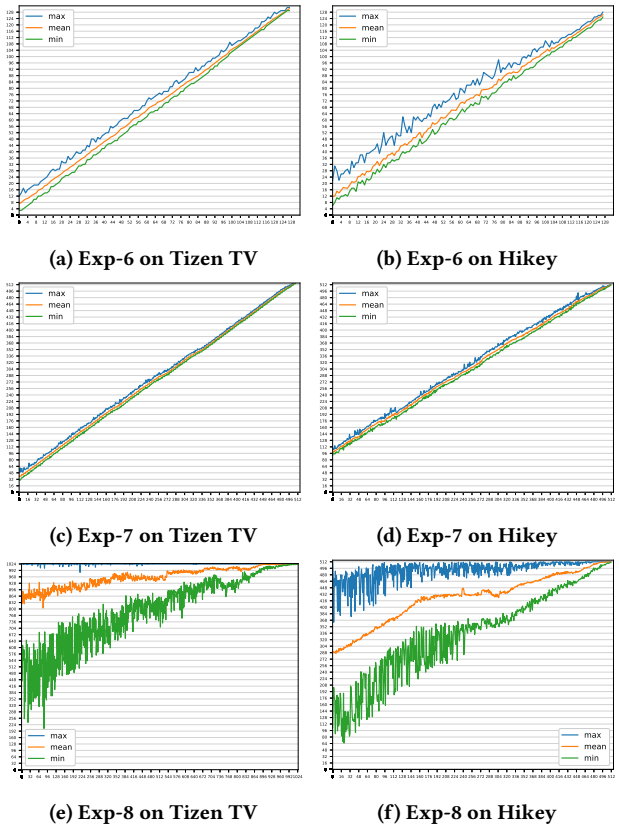
By comparing the first row (previous PRIME techniques) and the second row (our PRIME technique) of Figure 2, we can clearly see that the variance of noise is significantly reduced using our PRIME method with an instruction barrier.

6.1.2 Cross-core Scenario. We also conducted cross-core experiments on both devices using our PRIME method, with instruction barrier and set-counting mode (Exp-5). As Figure 3 shows, the noise under the cross-core scenario is much stronger than it is under the single-core scenario. Also, the results on Hikey is more stable than the results on Tizen TV. This is because there are several applications running on the Tizen system when we were conducting the experiments.

6.1.3 Under Extreme Conditions. We are interested in how our approach performs under extreme conditions. To this end, we ran a program in the normal world that creates many threads that exceed the number of cores on each board. The threads stay in an infinite loop in which they keep reading and writing data to memory after allocating a memory region that has the same size as the L2 cache.

We conducted multiple experiments with three different configurations: (1) Exp-6: the set-counting mode under the single-core scenario; (2) Exp-7: the line-counting mode under the single-core scenario; (3) Exp-8: the set-v counting mode under the cross-core scenario. Figure 4 suggests our approach performs well in the single-core scenario even under extreme conditions. However, the error rate is very high in the cross-core scenario.

6.1.4 Under a Real-world Condition. In addition, we also tested our approach in the cross-core scenario under a more realistic condition, where a YouTube application was running in the Samsung Tizen TV (Exp-9). As shown in Figure 5-(a), the noise was alleviated compared to Figure 4-(e) (Exp-8). However, Figure 5-(b) implies that the cross-core covert channel is difficult to utilize because there are many overlaps in the ranges of each bucket.



(a) Exp-6 on Tizen TV (b) Exp-6 on Hikey

(c) Exp-7 on Tizen TV (d) Exp-7 on Hikey

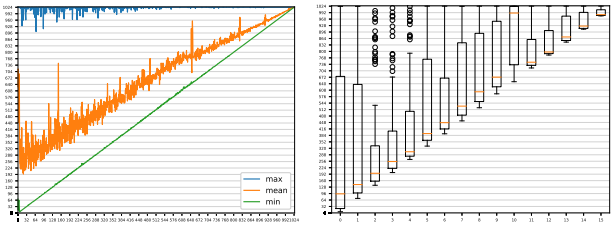
(e) Exp-8 on Tizen TV (f) Exp-8 on Hikey

Figure 4: Number of Loaded Cache Sets versus Detected Refill Events under Extreme Conditions. We conducted multiple experiments. (1) Exp-6: the set-counting mode under the single-core scenario; (2) Exp-7: the line-counting mode under the single-core scenario; (3) Exp-8: the set-counting mode under the cross-core scenario.

6.2 Choosing Bucket Ranges

Figure 6 shows the distributions of the number of cache refill events when we select 16 buckets. We assumed that the covert channel is employed to send 4 bits per time. The *Sender* tries to load a specific number of cache lines/sets (x-axis), and the *Receiver* detects how many events are occurred and decodes it to a message (y-axis).

The box and whisker diagram used in Figure 6 is to display the distribution of data. Data from the first to the third quartiles is in the box, and the red line inside the box represents the median. The bottom line and the top line represent the minimum and maximum value, respectively. The other small circles are outliers. As shown in Figure 6, it is difficult to find overlapped ranges in the line-counting mode after we applied our approaches. On the other hand, in the set-counting mode, the available numbers of the event are smaller than the line-counting mode, and thus, there are overlapping refill event numbers between buckets.



(a) Exp-9 on Tizen TV (b) Exp-9 on Tizen TV

Figure 5: Experiment under a realistic condition, where a YouTube application is running.

6.3 Capacity Measurement

For the capacity measurement, we evaluated how many bytes can be transferred per second using the channels. In particular, we designed 4 experiments: (1) Exp-10: the *Sender* tries to load all cache lines/sets (write all ones to the channel); (2) Exp-11: the *Sender* does not loads anything (write zero to the channel); (3) Exp-12: the *Sender* tries to load all cache lines/sets (write all ones to the channel) under extreme conditions; (4) Exp-13: the *Sender* does not loads anything (write zero to the channel) under extreme conditions; We ran all four experiments 500 times on both devices using different counting modes.

As shown in Table 3, the single-core set-counting mode of Exp-11 has the highest capacity and the cross-core of Exp-12 has the lowest capacity for both Hikey board and Samsung Tizen TV. The results may be surprising at the first glance since our experiments showed line-counting mode has lower noise and 2 more bits to use than set-counting mode. Further analysis reveals the reason behind this phenomenon is that the code of line-counting mode takes much longer time to run than its set-counting mode counterpart. This finding demonstrates the importance of efficient code execution to the covert-channel capacity.

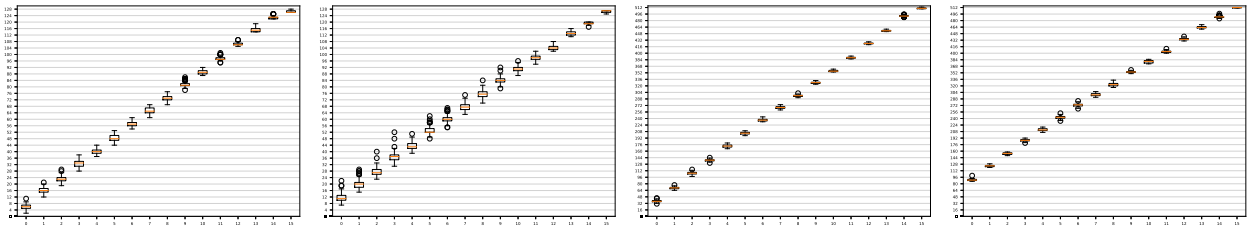
6.4 Image Transfer

We used the covert channels to transmit images from the secure world to the normal world under different conditions using both devices. Figure 7 shows the results of experiments on the Tizen TV. Column (a) shows the original images. The other images are all the ones we retrieved from the normal world using covert channels.

Overall, the quality and accuracy of the transferred images decrease from left to right; and even under extreme conditions (Figure 7 column (f)), the covert channel can still transmit data with some accuracy. The images illustrate that the covert channels we built using PRIME+COUNT are effective.

We especially can transfer data without noise in the single-core scenario using the line-counting mode as shown in Figure 7-(b). Because there is no overlapped region between the boxes in Figure 6-(c) and (d), we set each bucket to have enough range so that the receiver can decode correct message.

However, the cross-core covert channels have low accuracy particularly when YouTube was running and under extreme conditions as illustrated in Figure 7-(e) and (f), Under these conditions where we cannot avoid much noise, the number of cache refill events



(a) Exp-3 on Tizen TV (b) Exp-3 on Hikey (c) Exp-4 on Tizen TV (d) Exp-4 on Hikey

Figure 6: The chosen bucket ranges for different experiment configurations versus Detected Refill Events.

Attack Scenario	Test Device	Counting Mode	Exp-10	Exp-11	Exp-12	Exp-13
Single-core	Samsung Tizen TV (Cortex-A17)	Set-counting	10,330.97	27,408.13	4,868.28	12,971.03
		Line-counting	5,293.62	8,216.97	2,517.62	3,892.50
	Hikey Board (Cortex-A53)	Set-counting	10,273.43	15,646.21	3,812.29	6,201.89
		Line-counting	2,605.33	5,101.91	875.12	1,824.15
Cross-core	Samsung Tizen TV (Cortex-A17)	Set-counting	19.32	45.83	15.31	17.73
	Hikey Board (Cortex-A53)		52.14	95.04	22.33	26.49

Table 3: Capacities of Covert Channels (Byte/Second).

increases unexpectedly as the Figure 4-(e), (f) and Figure 5 show. Therefore, message sent by the *Sender* is likely to go other buckets (to higher numbers) because of severe noise.

7 DISCUSSION

7.1 Limitations of Prime+Count

First off, it is worth noting that covert channels made by Prime+Count could be detected by monitoring PMUs. To detect use of L1/L2 cache refill events, a defender can check the performance monitors event counter selection register (PMSELR) and the performance monitors selected event type register (PMXEVTYPER) [4].

In addition, Prime+Count is not as fine-grained as other cache attacks, including Prime+Probe and Flush+Flush, because it only cares about how many cache sets/lines have been updated. Adopting Prime+Count for spying a victim program and even extract cryptographic keys from another address space may be difficult if not impossible, because Prime+Count cannot answer which cache sets/lines have been used. However, due to the coarse-grained characteristic of Prime+Count it can reduce noise introduced by world switching, pseudo-random replacement policy, and other factors, which makes it a better choice to build cross-world covert channels.

7.2 Cross-world Covert Channels without Normal World Kernel Privileges

To loosen up the attack model and allow normal world applications to use the covert channels, we can adopt the Prime approach proposed in [21] that can be conducted in userland without using the `DC CISW` instruction. As mentioned in Section 3, we can also utilize the Linux *perf_event_open* system call to monitor "L1/L2 cache refill events" in userland to implement Count.

7.3 Limitations of Our Experiments

While we took great efforts to maintain our experiments' validity, we could not consider some factors that may have affected the bandwidth of the constructed covert channels. Specifically, SeCReT is not openly available (in fact, the authors were unwilling to share their code or system with us), therefore we were unable to run our experiments with SeCReT enabled. It is unclear how much SeCReT or similar solutions would impact the CPU load and even the number of accesses to the cache. We want to emphasize that the deployment of SeCReT or a strong monitor will not affect the feasibility of the proposed covert channels but only downgrade the bandwidth.

8 RELATED WORK

Cache Side Channel Attacks: Cache side channel attacks exploit the leakage of information caused by micro-architectural time differences between a cache hit and a cache miss [47]. They have been used to steal cryptographic keys in victim programs [22, 29, 40, 41, 45], trace the execution of programs [1, 7, 21], and extract other sensitive information [32, 35, 44, 46]. Even though covert channels can be built using various techniques [8, 34], cache-based covert channel received a lot of attention in recent years [36]. Xu et al. explored cross-VM L2 cache covert channels in Amazon EC2 [38]. Wu et al. designed a high-bandwidth and reliable data transmission cache-based covert channel in the cloud [37]. Maurice et al. characterized noise on cache covert channels and built a robust covert channel based on established techniques from wireless transmission protocols [25].

The Security of TrustZone: SeCReT showed that TrustZone itself cannot guarantee secure communication between normal and secure world [18]. Machiry et al. presented vulnerabilities that permit normal world user-level applications to read and write any memory location in the kernel by tricking secure world into performing the

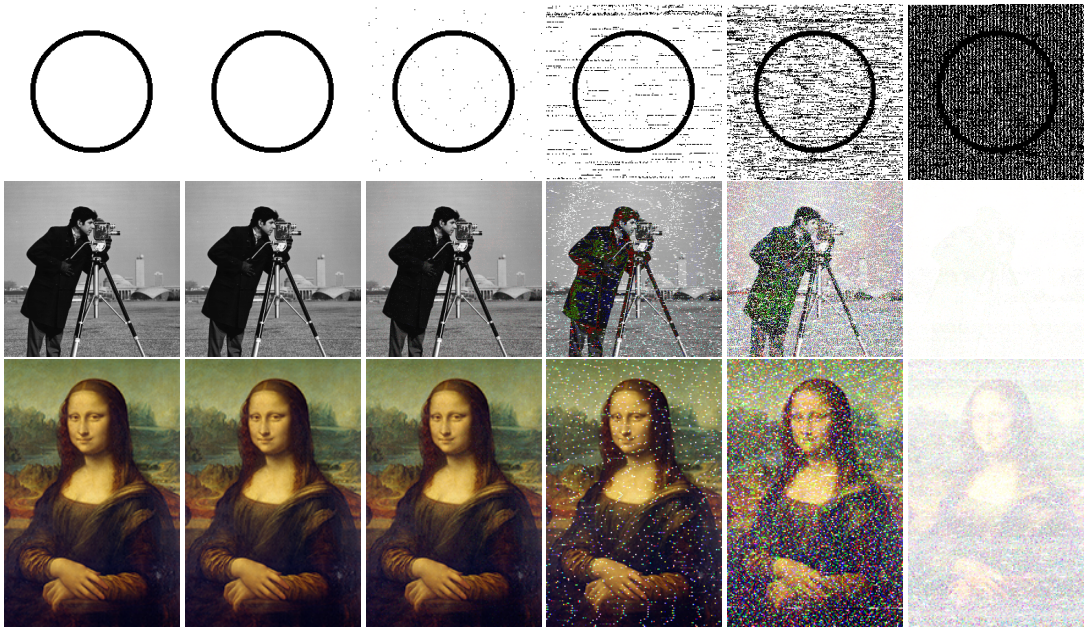


Figure 7: Transferred images using covert channels on Tizen TV. (a) Original images; (b) Single-core line-counting; (c) Single-core set-counting; (d) Cross-core; (e) Cross-core when YouTube is running; (f) Cross-core under extreme conditions.

operations on its behalf [23]. ARMageddon demonstrated how to use PRIME+PROBE to spy code executions on TrustZone [21]. TruSpy demonstrated that it is possible for a normal world attacker to steal a fine-grained secret from the secure world using timing-based cache side-channel [42]. In this paper, we presented the first attempt to build cross-world covert channels in the TrustZone architecture.

9 CONCLUSION

In this paper, we presented cross-world covert channel attacks on ARM TrustZone, which is designed to provide hardware-assisted isolation. We demonstrated that existing channel protection solutions, such as SeCReT, or even more powerful mechanisms, such as a strong monitor, can be bypassed. We discussed the reasons why previous attacks, including PRIME+PROBE and FLUSH+RELOAD, do not work for the cross-world scenario on ARM. And, we designed a low noise, no shared memory needed cache attack named PRIME+COUNT by leveraging overlooked PMU "L1/L2 cache refill events." Our experiments showed that PRIME+COUNT-based cross-world covert channels could achieve bandwidth as high as 27 KB/s under the single-core scenario and 95 B/s under the cross-core scenario.

ACKNOWLEGMENT

Many thanks to the anonymous referees for their valuable and helpful comments. We would also like to thank our shepherd, Fengwei Zhang.

This material is based upon work supported in part by Samsung Research, Samsung Electronics, the Center for Cybersecurity and Digital Forensics at Arizona State University, the National Science Foundation (NSF 1651661), the Defense Advanced Research Projects Agency (DARPA HR001118C0060), and the Global Research Laboratory Program through the National Research Foundation of Korea funded by the Ministry of Science and ICT under Grant NRF-2014K1A1A2043029.

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] 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)*. 256–273.
[2] ARM. 2012. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html. (2012).
[3] ARM. 2012. ARMv6-M Architecture Reference Manual. https://silver.arm.com/download/download.tm?pv=1102513. (2012).
[4] ARM. 2016. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.k/index.html. (2016).
[5] ARM. 2016. SMC CALLING CONVENTION System Software on ARM Platforms. http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf. (2016).
[6] ARM. 2017. ARM Trusted Firmware. https://github.com/ARM-software/arm-trusted-firmware. (2017).
[7] Billy Bob Brumley and Risto M Hakala. 2009. Cache-timing template attacks. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*. 667–684.
[8] Serdar Cabuk, Carla E Brodley, and Clay Shields. 2004. IP covert timing channels: design and detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. Washington, DC, 178–187.
[9] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. 2017. Downgrade Attack on TrustZone. *arXiv preprint arXiv:1707.05082* (2017).

[10] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* 49 (2016), 1162–1174.

[11] 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.

[12] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

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

[14] Berk Gülmezoğlu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. A faster and more realistic flush+ reload attack on AES. In *Proceedings of the International Workshop on Constructive Side-Channel Analysis and Secure Design*. 111–126.

[15] 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.

[16] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. 2015. Know thy neighbor: crypto library detection in cloud. *Proceedings on Privacy Enhancing Technologies* 2015, 1 (2015), 25–40.

[17] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2016. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Singapore, 85–96.

[18] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[19] laginimaineb. 2016. Exploit that extracts Qualcomm's KeyMaster keys using CVE-2015-6639. https://github.com/laginimaineb/ExtractKeyMaster. (2016).

[20] laginimaineb. 2016. Qualcomm TrustZone kernel privilege escalation using CVE-2016-2431. https://github.com/laginimaineb/cve-2016-2431. (2016).

[21] 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.

[22] 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.

[23] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[24] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: cross-cores cache covert channel. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 46–64.

[25] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[26] MITRE. 2013. CVE-2013-3051 Detail. https://nvd.nist.gov/vuln/detail/CVE-2013-3051. (2013).

[27] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. 2017. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*.

[28] OP-TEE. 2017. OP-TEE Trusted OS Documentation. https://www.op-tee.org/. (2017).

[29] 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)*. 1–20.

[30] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[31] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Systems* 37, 2 (2007), 99–122.

[32] 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.

[33] Dan Rosenberg. 2013. Unlock the Motorola Bootloader. http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html. (2013).

[34] Gaurav Shah, Andres Molina, Matt Blaze, et al. 2006. Keyboards and Covert Channels.. In *Proceedings of the 15th USENIX Security Symposium (Security)*. Vancouver, Canada, 59–75.

[35] 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.

[36] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Computer Security Applications Conference (ACSAC)*. 473–482.

[37] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA, 159–173.

[38] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 29–40.

[39] Yongcheol Yang, Jiyoung Moon, Kiuhae Jung, and Jeik Kim. 2018. Downloadable trusted applications on Tizen TV: TrustWare Extension: As a downloadable application framework. In *Proceedings of the 2018 IEEE International Conference on Consumer Electronics (ICCE)*. Las Vegas, NV.

[40] 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.

[41] 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.

[42] 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. https://eprint.iacr.org/2016/980.pdf. (2016).

[43] 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.

[44] 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.

[45] 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.

[46] 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, Arizona, 990–1003.

[47] YongBin Zhou and DengGuo Feng. 2005. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptology ePrint Archive* 2005 (2005), 388.