TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices

Ning Zhang*, Kun Sun†, Deborah Shands‡ Wenjing Lou*‡, Y. Thomas Hou*
*Virginia Polytechnic Institute and State University, VA
{ningzh, wjlou, thou}@vt.edu
†George Mason University, Fairfax, VA
{ksun3}@gmu.edu
‡National Science Foundation, Arlington, VA
{deborah.shands}@gmail.com

***Abstract*—As smart, embedded devices are increasingly integrated into our daily life, the security of these devices has become a major concern. The ARM processor family, which powers more than 60% of embedded devices, introduced TrustZone technology to offer security protection via an isolated execution environment called secure world. Caches in TrustZone-enabled processors are extended with a *non-secure (NS)* bit to indicate whether a cache line is used by the secure world or the normal world. This cache design improves system performance by eliminating the need to perform cache flush during world switches; however, it also enables cache contention between the two worlds.**

In this work, we present TruSpy, the first study of timing-based cache side-channel information leakage of TrustZone. Our proposed attack exploits the cache contention between normal world and secure world to recover secret information from secure world. Two attacks are proposed in TruSpy, namely, the normal world OS attack and the normal world Android app attack. In the OS-based attack, the attacker is able to access virtual-to-physical address translation and high precision timers. In the Android app-based attack, these tools are unavailable to the attacker, so we devise a novel method that uses the expected channel statistics to allocate memory for cache probing. We also show how an attacker might use the less accurate performance event interface as a timer.

Using the T-table based AES implementation in OpenSSL 1.0.1f as an example, we demonstrate that it is possible for a normal world attacker to steal a fine-grained secret from the secure world using a timing-based cache side-channel. We can recover the full AES encryption key via either the OS-based attack or the Android app-based attack. Since our zero permission TruSpy attack is based on the cache design in TrustZone enabled ARM processors, it poses a significant threat to a wide array of devices. To mitigate the newly discovered threat, we also propose both application-based and system-oriented countermeasures.

1. Introduction

With the continuous growth in network capabilities, more and more embedded devices are connected. Having a smart home is no longer a story from science fiction movies [1]. On the other hand, the sheer volume of cyber attacks nowadays has been unsettling. Security is now one of the major concerns in adopting these smart technologies [2].

ARM family processors have been deployed in more than 60% of the embedded devices [3]. To enhance the security of mobile systems, ARM introduced a security extension called *TrustZone* [4], which offers the ability to protect security sensitive tasks within an isolated execution environment. TrustZone has been adopted in a wide variety of commercial products [5], [6] and academic projects [7], [8], [9], [10] to enable secure processing. The protected environment is called *secure world*, and the normal environment is called *normal world* or *nonsecure world*. In order to provide isolation of resources between the two worlds, hardware components in TrustZone-enabled platforms are augmented with an additional *NonSecure (NS)* flag bit to indicate the security domain.

Processor cache is one of the basic components in modern memory architecture to bridge the gap between the fast processor operation and relatively slower memory access. To support memory isolation in the TrustZone architecture, both instruction and data cache lines are extended with the *NS* flag bit [4] to mark the security domain of these lines. Even though the secure cache lines are not accessible by the normal world, both worlds are equal when competing for the use of cache lines. In other words, when the processor is running in one world, it can evict the cache lines used by the other world due to cache contention. The goal of this cache design is to improve the system performance by maximizing the usable cache space and eliminating the need for cache flush during a world switch. We observe that though the contents of processor cache are protected by the hardware extension, the access pattern to these cache lines is not protected, leaving TrustZone vulnerable to cache side-channel attacks.

Unlike software exploitations that target vulnerabilities in the system, side-channel attacks target information leakage of a physical implementation due to various interactions with its execution environment. Side-channel information can be obtained from different types of physical features, such as power [11], electromagnetic wave [12], acoustic [13] and time [14], [15], [16]. Among these side-channels, the cache-based timing attack is one of the most active areas of research [15], [16], [17], [18], [19], [20], [21], [22], [23]. A cache timing side-channel exists because the memory access delay from the processor cache is significantly less than that from DRAM. Bernstein [23] recovered the AES key by timing the execution of cryptographic operations. Osvik et al. [18] proposed the general technique called *prime and probe*. It captures fine-grained cache access information of the victim by observing changes to known cache states due to the execution of a cryptographic algorithm. Recently, it has also been shown that the shared memory pages of OS libraries can be exploited by attackers to enable a powerful cache side-channel attack called *flush and reload* [17], [16], [21].

In this paper, we present TruSpy, a cache-based timing side-channel attack. To the best of our knowledge, this is the first study on cache timing-based information leakage of ARM TrustZone. TruSpy circumvents TrustZone protection to extract sensitive information in the secure world using less privileged normal world processes. Since the memory separation between the normal world and the secure world is strictly enforced by the TrustZone hardware, the more efficient flush and reload attack cannot be used. This is because memory sharing is one of the requirements for flush and reload attack. As a result, our proposed attack adopts the prime and probe technique [18] to exploit the cache contention between the normal world and the secure world.

There are two key requirements for applying the TruSpy attack to recover secrets from the secure world. First, the attacker must be able to fill the cache with memory contents that will cause cache contention with the victim. Second, the attacker must be able to detect changes in the cache state. More specifically, an attacker needs access to a high precision timer to distinguish data retrieval from different levels in the memory hierarchies. In TruSpy, we show how these conditions can be met under different levels of access privileges to the system resources.

Two attacks are presented in this paper, *the normal world OS attack* and *the normal world Android app attack*. In the first attack, the attacker has full control of the normal world operating system. It can obtain detailed virtual-to-physical address translation via the page table. The ability to obtain such translation is crucial in allocating memory for the prime and probe attack. Furthermore, the attacker can use the cycle counter in the performance unit as a high precision timer. The cycle counter can only be accessed in the privilege mode. In the second attack, we challenge ourselves further to ask if it is possible to launch TruSpy attack from a non-privileged Android app with no special permissions. For an app running in the user space, neither the virtual-to-physical address translation nor the cycle counter is available. To allocate the memory for prime and probe in the app, we propose *statistical matching*, in which memory page for probing is identified by comparing the channel statistics to the expected statistics of the cryptographic algorithm. Furthermore, we use a common kernel performance measure function via system call to replace the high precision timer that is only accessible in the OS kernel.

We implement our attack on a Freescale i.MX53 development board running CortexA-8 processor. We demonstrate the side-channel information leakage of TrustZone using a T-table-based AES implementation as an example, since side-channel attack on AES is widely used in the literature as example for fine-grained information extraction [15], [19], [18], [23], [24]. The attack from the normal world kernel can recover the full AES128 secret key using 3000 rounds of observed encryption in 2.5 seconds. Despite significant noise from the timer, the user space Android app can still recover the full AES128 secret key with 9000 rounds of observed encryption within 14 minutes.

In summary, we make the following contributions.

- We identify the side-channel information leakage from the design of dynamic cache allocation between the normal world and the secure world in TrustZone. The leakage is due to a fundamental design choice of the TrustZone-enabled cache architecture, which aims to improve system performance by allowing two worlds to share the same cache hardware.
- Though prime and probe is a well-known method on x86, we present the first detailed adoption of the technique on ARM processor. There are new challenges such as random cache replacement policy, inability to interrupt encryption, and lack of inclusive or exclusive cache.
- For the first time, we show that even an unprivileged Android app can launch side-channel attacks on TrustZone. We tackle two more challenges - finding virtual-to-physical mapping and lack of high precision timer. Without direct access to virtual-to-physical address translation, the Android app attack allocates memory by using statistical properties of the channel itself or correlating to kernel function with known addresses. Furthermore, the app is able to use the less accurate, common OS performance event function to replace the high precision performance register.
- We show the practicality of our attack by implementing and testing the proposed side-channel attacks on a hardware platform. The kernel space attack can extract a full 128 bit AES key within 3000 rounds of encryption, while the user space attack can extract the full AES key within 9000 rounds of encryption. We discuss potential mitigations against this new side-channel attack on secure world of TrustZone.

In the rest of the paper, we first present background information on ARM TrustZone, cache architecture, and an overview of cache side-channel attacks in section 2. The

threat model is presented in section 3. The design of TruSpy attack is presented in section 4. The details of applying the attack to recover AES secret key are described in section 5. We then present potential countermeasures in section 6. Lastly, a discussion is given in section 7, followed by the conclusion in section 8.

2. Background

In this section, we summarize the hardware protection offered by TrustZone, then give an overview of the memory architecture of TrustZone enabled ARM processors. Finally we provide an overview of cache side-channel attacks.

2.1. ARM TrustZone Architecture

TrustZone is a security extension to the ARM architecture with modifications to the processor, memory, and I/O devices [4]. TrustZone provides a system-wide isolated execution environment for secure workloads. Many of the recent ARM processors support this security extension [25], [26]. The traditional operating domain is called *normal world*, *non-secure world*, or *rich OS*, while the protected domain is called *secure world*. Resources in the secure world cannot be accessed by normal world processes. The processor security state is captured by the *NS* bit in the *security configuration register (SCR)*. When the bit is clear, the processor is in the secure world. When the bit is set, the processor is in the normal world. The SCR is only accessible while the processor is in the secure privileged mode. While the processor is in secure monitor mode, the security context is always secure regardless of the *NS* bit value.

2.2. Processor Memory Hierarchy

Figure 1 shows a typical memory hierarchy of modern computer systems. Programs run in virtual address space. When the processor needs to fetch a value in memory, it passes the virtual address to the memory management unit (MMU). The MMU first attempts to retrieve the virtual address to physical address translation from the translation lookaside buffer (TLB). If there is no hit, the MMU then parses the page table for the translation and places it inside the TLB. Parsing of the page table is often referred to as a page walk. Once the physical address is obtained, the MMU goes through the system memory hierarchy to retrieve the value at that address.

Modern processors often have multiple levels of caches to enable faster memory access. Higher levels of caches are faster and placed closer to the processor core. Because of the limited on-die space, processor cache is usually small in size compared to the physical DRAM. Modern caches are usually organized with N-way associative table. The basic unit of memory allocation in cache is called a line, or cache line. In most processors, a line is often 64 bytes. A cache line can be virtually indexed, virtually tagged (VIVT), virtually indexed, physically tagged (VIPT), physically indexed,

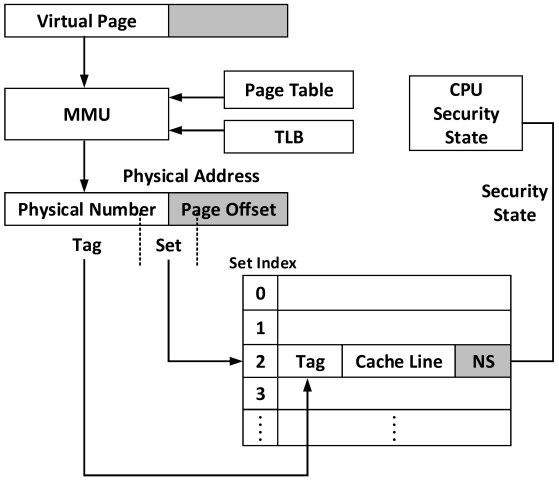

Figure 1. TrustZone Enabled Processor Memory Hierarchy

physically tagged (PIPT) or physically indexed, virtually tagged (PIVT). Many modern processors have a large TLB and adopt PIPT for all levels of data and unified cache. As a result, the physical memory address determines the set index inside the cache table. The ARM Cortex-A8 processor has two levels of caches. Its level one (L1) cache consists of two 32 KB 4-way 128-set caches, for caching instruction and data, respectively. The level two cache (L2) is a 256 KB 8-way 512-set unified cache. Both L1 data cache and L2 unified cache adopt PIPT.

With the addition of TrustZone, each cache line has been extended with an *NS* bit, which specifies the security state of the cache line [4]. The goal of this design to use the *NS* bit to distinguish cache lines of the normal world from those of the secure world, eliminating the need to flush cache during the world switch. The content of the caches, with regard to the security state, is dynamic. Any cache line can be evicted to make space for new data, regardless of its security state. In other words, it is possible for a secure cache line fill to evict a non-secure cache line, and vice versa. This cache design is the key enabler for launching side-channel attack against the protected execution in TrustZone.

2.3. Cache Side-Channel Attack

Traditional cryptanalysis focuses on the cryptographic algorithm and develops attacks exploiting theoretical weakness. On the other hand, side-channel attacks focus on information leaked through hardware or software features of a cryptographic implementation.

Side-channel attacks are well-studied in computer security [14], [11], [22]. With significant research on cache side-channel attacks [14], [11], [27], [28], [29], [30], [13], [23], [31], [32], [33], [16], [17], [34], [21], [18], [20], [15] and defenses [35], [36], [37], [38], [39], [40], [41], [42]. The concept of using side-channel information as a means to attack cryptographic schemes first appeared in a seminal paper by Kocher [14]. In [14], Kocher exploited differences

TABLE 1. COMPARISON OF TRUSPY TO OTHER ACCESS-BASED CACHE SIDE-CHANNEL ATTACKS ON ARM

	Attacker Privilege		Target Protection		Attack Type			
Attacks	App	Kernel	OS	TrustZone	Evict+Reload	Flush+Reload	Prime+Probe	Cache Storage
TruSpy	✓	✓		✓			✓	
Armageddon [19]	✓		✓		✓			
Alias-Driven [24]		✓		✓				✓
ROP-based [45]	✓		✓			✓		

in computation times to break implementations of RSA and discrete logarithm-based cryptographic algorithms. Besides time, other physical attributes such as electromagnetic emission [12], power consumption [11] or acoustic noise [13] have been investigated as viable sources for side-channel attacks. Bernstein [23] was the first one to show the existence of timing dependencies introduced by the data cache, which allows key recovery for the modern T-table implementation of AES [23]. There are three main categories of cache-based side-channel attacks: time driven [23], trace driven [43], and access driven [44], [31], [15], [19], [20]. The differences between them are the attackers' capabilities, with time driven attacks the least restrictive.

Osvik et al. [18] proposed two techniques for attackers to determine which cache set is accessed by the victim, namely, *evict and time* and *prime and probe*. In evict and time, the attacker modifies a known cache set and observes the changes in the execution time of the victim's cryptographic operation. In prime and probe, the attacker fills the cache with known states before the execution of the cryptographic operation and observes the changes in these cache states. Gullasch et al. [17] identified another powerful cache side-channel attack enabled by system memory deduplication. The attacker flushes the memory shared between the malicious process and the victim process, such as a crypto library, with kernel samepage merging (KSM) enabled. After the victim executes the cryptographic algorithm, the attacker measures the time to load the memory into a register to determine if the memory has been accessed by the victim process. This new method was later named by Yarom et al. as *flush and reload* in [16]. With the growing interests in Cloud computing, another line of research [22], [46], [15], [20] focuses on recovering secrets from neighboring virtual machines rather than processes on the local machine.

Most of the current research focuses on side-channel investigation of the Intel [47] x86 architecture processors, but little has addressed the ARM [48] platform. The Bernstein attack was first tested by Weiss et al. [32] in ARM processors. Besides Bernstein's timing attack, evict and time was used in [49], [19] to attack the T-table implementation of AES. Memory duplication was also used in [45] to launch a flush and reload attack to track the execution path of shared libraries. Compared to [45], [19], the protection of our target is different. Due to the memory protection by TrustZone, it is impossible to use the evict and reload [19] or flush and reload [45]. Instead, we have to use the less reliable prime and probe approach, which is further complicated by the random replacement policy of the processor cache. Though the feasibility of prime and probe was briefly discussed in [19], it provides limited details and assumes that the physical page number can be retrieved from *pagemap*. We assume that the attacker has no knowledge of the physical memory layout.

In addition to cache timing side-channel, a recent study demonstrated a new cache storage side-channel based on unexpected cache hit in cache incoherence [24]. The attack can recover secrets from the secure domain in the Cortex-A7 processor [24]. Compared to the storage-based side-channel that requires the root privilege to cause programming faults at the hardware level, we implement the first timing-based side-channel attack that exploits the high-performance design of the ARM system. Unlike the storage side-channel, our attack does not require kernel privilege and can be launched with a zero permission Android app. Moreover, the storage based side-channel relies on faults that can be easily removed by eliminating the mismatched cacheability attribute in future ARM designs. By contrast, the cache-timing side-channel we exploit is based on the basic design of TrustZone to fasten the context switching between two worlds. It is difficult to remove the side-channel without causing performance impact.

Table 1 compares our work with recent access-based cache side-channel attacks on ARM devices [24], [19], [45]. In summary, we are the first to demonstrate fine-grained side-channel information leakage from TrustZone in user space with no permission app.

3. Threat Model and Assumptions

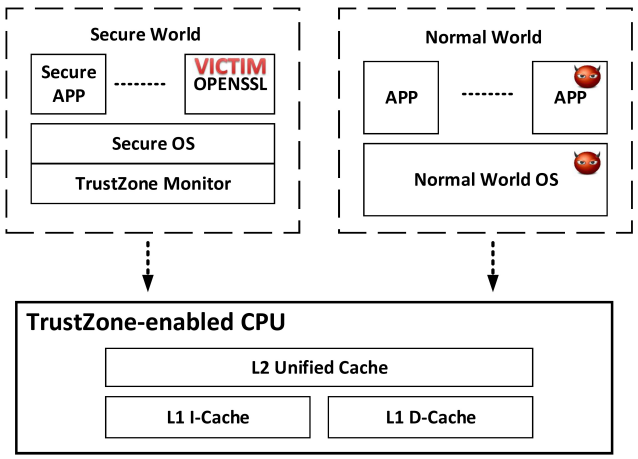

Figure 2. TruSpy Threat Model

TrustZone technology has been widely used in both academic projects [50], [9], [51], [10], [52], [53], [7] and commercial products [6], [5], [8]. As recommended in the ARM TrustZone whitepaper [4], it is often used to protect security sensitive workloads, such as cryptographic operations, in an isolated execution environment. In [53], [7], [54], [55], cryptographic libraries are protected in the secure world from a potentially compromised normal world OS running in the normal world. The threat model for our work is shown in Figure 2. We assume that a cryptographic library is implemented in the secure world and provides services to the OS in the normal world. Meanwhile, an attacker in the normal world can execute a spy process, which can be either a malicious Android app or the compromised normal world OS, targeting at the cryptographic module in the secure world. Though we demonstrate that it is possible to recover the full AES128 key using only the app level attack, the attacker that has compromised the normal world OS has access to more controls in the system and thus can recover the key in a shorter time.

Similar to previous works [15], [19], [33], [18], [23], we assume the attacker (i.e. the spying process) can trigger the encryption in the victim process. In our OS level attack, we assume that there are vulnerabilities in the OS that allow arbitrary code execution with kernel privilege. This assumption is common for launching attacks on hardware-enforced Trusted Execution Environments [56], [54]. For our app level attack, we make no assumption of vulnerabilities or hardware design flaws. Moreover, the Android app does not require any Android permission to run the attacking code. Therefore, the malicious code can be embedded into all types of applications and remains stealthy. Our attack can work in the user space as an unprivileged application. Thus, it works on non-rooted systems. Our attack does not assume specific Android functionality other than Java native invocation (JNI). Lastly, we make use of the performance event module in the kernel, which is included in most of the mainstream Android systems after Android 1.0.

4. TruSpy Attack

ARM TrustZone [4] aims to provide hardware-based system-wide security protection. Besides processor protection, TrustZone also provides isolation of memory and I/O devices. In the ARM TrustZone cache architecture, an *NS* flag is inserted into each cache line to indicate its security state (normal vs secure). When the processor is running in the normal world, the secure cache lines are not accessible. However, when there is cache contention, a non-secure cache line can evict a secure cache line and vice versa. This cache design improves the system performance by eliminating the need to perform cache flush during a world switch; however, the cache contention also leaks side channel information [14].

TruSpy attack exploits the cache contention between the normal world and the secure world as a cache timing side channel to extract sensitive information from the secure world. Though the flush and reload approach [16], [17], [45] has recently gained considerable attention due to its simplicity and efficiency, it cannot be applied here. Memory sharing between attacker and victim is the key enabler for flush and reload attack, but the memory protection of TrustZone prevents such shared memory. Instead, our attack follows the general technique of prime and probe [18] to learn the cache access pattern of the victim process.

There are two requirements for a successful TruSpy attack.

- First, the attacking process has to be able to fill in cache lines at individual cache sets that will cause cache contention with the victim process.
- Second, the attacker has to be able to detect changes in the cache state. This is often accomplished by measuring the time it takes to load a particular memory address into register using a high precision timer. Other methods include using cache performance counter and cache incoherence [24].

In the rest of the section, we present the details of how these two requirements are met in TruSpy. In 4.1, we first present the overall workflow of launching a TruSpy attack. We then present two TruSpy attacks with different assumptions on the attacker's capability. The first attack presented in 4.2 assumes that the adversary has full control of the operating system in the normal world. It has access to various OS-level controls such as virtual-to-physical address mapping and a high precision timer. We then relax the assumption that the normal world OS kernel is compromised and develop a more sophisticated attack that is capable of recovering the secrets in the secure world from a non-privileged Android app. The Android app attack presented in 5.4 runs in user space, and it does not require root privilege or special Android permissions.

4.1. TruSpy Attack Workflow

The TruSpy attack flow consists of five major steps, as shown in Figure 3. The first step is to identify the memory to use for cache priming. The key is to find the memory that will be filled in cache sets that are also used by the victim process in the secure world. This step is often accomplished by working out the mapping from virtual address to cache sets [18], [15].

The second step is to fill the cache. In this step, the spy process fills the cache with its own memory so that each cache line that can be used by the victim is filled with memory contents from the address space of the attacker. This step will allow the attacker to obtain a known cache state before handing the control flow to victim process to spy on.

The third step is to trigger the execution of the victim process in the secure world. When the victim process is running, cache lines that were previously occupied by the attackers are evicted to the DRAM. As a result, the cache configuration from the attacker's perspective has changed because of the execution of the victim process. Since this step is non-interruptible due to the protection of TrustZone,

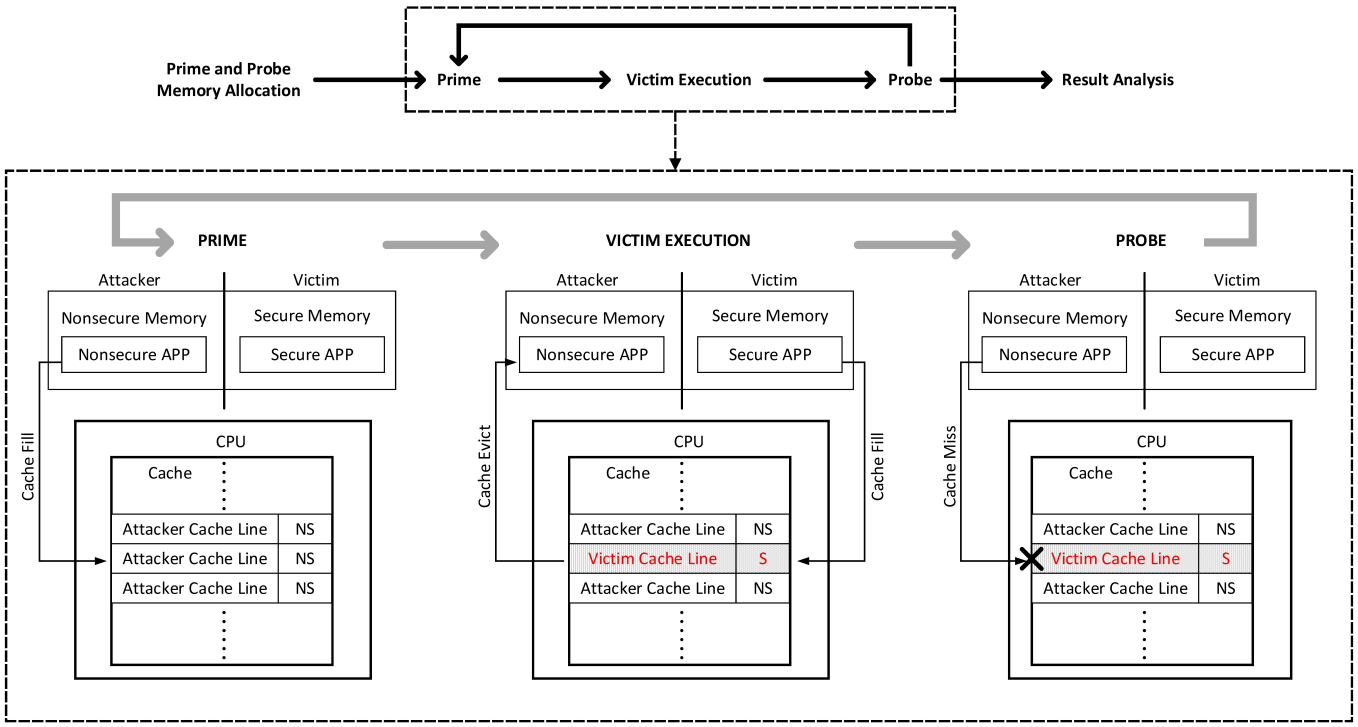

Figure 3. TruSpy Attack Flow

it is more challenging for this attack to succeed without fine-grained information on the victim process cache access.

The fourth step is to measure the change in cache configuration after the victim finishes its execution in the secure world. For each cache line that was previously primed in the second step, the time to execute memory load instruction *ldr* is measured. If the time it takes to load the memory into register is short, then cache set of which the memory is mapped to was not evicted by the victim process. In other words, the victim did not execute a particular path or did not make use of a particular data that is indexed into this cache set. Once the results are recorded for all the memory locations that were primed, the attack goes back to the second step and continues to collect more side-channel information.

The fifth step is the last step. The collected channel information is analyzed to recover secret information such as cryptographic keys within the secure domain.

4.2. Attacking from Normal World Kernel

Resources in the normal world are isolated from the secure world under the protection of ARM TrustZone. The memory separation between the two worlds is enforced by the hardware. As a result, there is no shared memory between secure world cryptographic module and the normal world attacking process. Therefore, the more efficient flush and reload approach [16], [17] is not applicable in this setting. Despite the isolation of memory between the two worlds, the cache system for both the secure world and the normal world is implemented on common hardware and allows resource contention between the two worlds. Therefore, our attack follows the general approach of prime and probe [18] to capture the side-channel information from this resource contention on the shared cache between the two worlds.

When the normal world OS of the embedded device is compromised, attackers have access to a variety of resources to facilitate the execution of the timing side-channel attack against security sensitive processes in the secure world. Among these resources, the capability to obtain virtual-to-physical address mapping as well as the access to an accurate timer is crucial in TruSpy.

4.2.1. Mitigating Cache Random Replacement. In modern ARM architecture, memory system generally has multiple levels. Cache is the top level memory system, and there are multiple levels of caches as described earlier in the background section. Many previous attacks [15], [16], [17] exploit the inclusive cache design in Intel processors. However, the cache in many ARM processors, such as Cortex-A8, is neither inclusive nor exclusive. This implies that the presence of a cache line in the top level cache does not necessarily indicate its presence or absence in the lower level caches. Furthermore, many cache controllers in ARMv7 family use random replacement policy. Therefore,

when there is cache contention, a cache line from the *n* cache way is chosen at random to fill the newly allocated memory. This cache random replacement policy in ARM processors adds additional noise when there are multiple levels of caches in the hierarchy. As a result, the cache architecture in ARM devices makes it a challenging task to accurately assess changes in cache state.

Even though it is easier to distinguish memory access from cache, lack of inclusiveness and random replacement makes probing the lower level cache a lot less desirable. In the OS level attack, due to the access to processor tick level timer through the performance counter registers, cache access to different levels can be reliably determined. Therefore, in TruSpy, the top level L1 cache is used to capture information leakage from the cache contention.

4.2.2. Allocating Memory for Prime and Probe. In order to gain fine-grained information on cache access of the victim process, an attacker must accurately fill its controlled memory into specific cache areas so that it will cause cache contention with the victim process. More specifically, the attacker needs to find memory that will be mapped to the same cache sets as the memory used by victim process. As described previously in the background section, modern cache controllers often use physical address for tagging and indexing so that tasks can switch without the need to purge cache contents.

To determine the cache set index that a memory address maps to, it is necessary to obtain the physical address of the memory. However, in modern computer architectures, processor executions use virtual computer addresses. Memory access is translated from the virtual address to the physical address by the MMU unit in the processor using the page table configured by the operating system. This address translation makes memory allocation that will cause cache contention with the victim process more challenging.

An attacker who controls the OS can perform a page table walk to figure out the physical address of any virtual address. In fact, if the memory is allocated from the kernel space, the physical address and the virtual address are offset by a constant value 0x10000000 in many systems. In the TruSpy attack, instead of using fragmented memory, we make use of the section mapping in the ARM kernel. Many modern processors come with cache for page translation besides instruction and data cache. This cache is called the translation lookaside buffer. Each page translation occupies one entry in the translation buffer. Larger mapping such as section paging is introduced by ARM to use the TLB more efficiently. This functionality is also used in the ARM Linux kernel to make large memory mappings.

Each memory section in ARMv7 has a size of 1 MB. The lower 20 bits of all virtual memory addresses in a mapped section are offsets in the section. Therefore, the lower 20 bits of all virtual memory match their physical address counterparts. The upper 12 bits are the section based address, and those bits are different between the physical address and virtual address. In Cortex-A8, each cache line has a length of 64 bytes. The L1 cache is divided into two 4-way 128-set caches. The level two cache is a unified 8-way 512-set cache of size 256 KB. The offset of memory section spans 20 bits, which covers more address space than L1 and L2 caches, therefore, we are able to use a single memory section to fill all the cache lines in the Cortex-A8 cache, eliminating the need to work out the physical address for individual memory.

4.2.3. Priming the Cache. Once the prime and probe memory is allocated, the attacker needs to fill in the cache. However, as shown in Figure 3, prime and probe forms a cycle that needs to be repeated many times. At the beginning of prime step, the cache could be filled with secure cache lines. Due to the security protection of TrustZone, secure cache lines are not affected by cache maintenance operations in the normal world [57]. Therefore, our first step is to repeatedly load memory that is indexed into targeted cache sets to drive out the secure cache lines using cache contention. In our implementation, we reload enough memory to refill the cache sets across all ways in all levels of cache hierarchy. More specifically, in order to make sure that all of the secure cache lines are evicted, we use eight distinct physical memory ranges that map back to the same L2 cache sets to perform the secure line eviction since there are eight cache ways in the Cortex-A8 L2 cache. By repeatedly loading them, all secure cache lines are evicted and then replaced with non-secure cache lines. With all the secure cache lines being driven out, we issue a *clean and invalidate* cache instruction to clean out the cache. The L1 cache is then filled with our prime memory.

4.2.4. Probing the Cache with Cycle Counter. After the execution of the victim's sensitive function in step three, the control flow is redirected back to the attacker. The changes in the processor cache states due to secure world execution have to be measured and recorded in this step. More specifically, the attacker needs to determine if the cache lines filled in the prime step are still in the cache. We extract side-channel information from cache contentions on the top level cache, L1 cache, in order to reduce the noise added by the random replacement policy of the cache controller. Therefore, a high precision timer is needed to distinguish cache hits on L1 cache or lower level memory. In x86 system, $rdtsc$ instruction can be used to provide sub-nanosecond resolution timestamps. However, on ARM processors, such a timer is not available. Instead, ARMv7 architecture provides a performance monitoring unit, which has a cycle count register (PMCCNTR). TruSpy makes use of this cycle count register to measure the time it takes to load memory into register to distinguish cache hits at different levels of cache hierarchy.

A code snippet to measure the cache set is shown in Listing 1. Instruction pipelining is a technique used in many modern processors to realize instruction-level parallelism. It enables higher processor throughput. On the other hand, it could influence the measurement of memory load instruction time. In order to measure only the memory load time, *isb* instruction (instruction barrier) is used to finish all

instructions in the pipeline before the cycle count register is read. Furthermore, in order to avoid other memory operation affecting the timer measurement, a *dmb* (data memory barrier) instruction is used to finish all the memory operations before the timer measurement. The *ldr* instruction is then used to load the memory address stored in $r3$ into $r0$. *dmb* instruction is used to force the completion of memory access before the cycle counter is read again. The difference between the two timer readings is then stored into memory address pointed in $r9$, which is an array used to store the time results of loading individual cache sets. The result array is pre-allocated before the execution of the prime and probe attack. We configure the cache attribute of the memory page to be no-write-allocate. Thus, when *str* instruction is used, the value in register goes directly to the memory without causing a cache fill. This ensures that execution of the measurement function does not change the cache state itself.

```
1 ...
  isb
3 dmb
  mrc   p15,0,r1,c9,c13,0
5 ldr   r0,[r3]
  dmb
7 mrc   p15,0,r2,c9,c13,0
  sub   r0,r2,r1
9 str   r0,[r9]
  ...
```

Listing 1. Probing Cache

Once the time is recorded, it can be compared to the known value of the platform to classify the level of the memory access. Memory access at different levels (L1, L2, DRAM) can be distinguished reliably using the cycle count register in the performance unit on Cortex-A8. Cache hit on L1 cache has an average of 90 clock ticks, while loading from L2 cache uses 107 clock ticks and loading from memory takes 311 clock ticks. Thus, the performance counter register, accessible only in privilege code, allows the attacker to not only reliably tell the difference between cache miss and cache hit, but also the cache level in the cache hit.

4.3. Attack from Android App

It is significantly more challenging to launch TruSpy attack from an Android app. Unlike x86, many architecture features such as high precision timer and cache flush instruction are not available to a user space application on ARM processors [48]. We will present our solution to these challenges in the following paragraphes.

4.3.1. Obtaining Prime and Probe Set. The first challenge is obtaining the memory that will cause cache contention with the victim process in the secure world. We refer to this memory as the *probing memory*, i.e. the memory used for probing the victim. When the OS kernel is compromised, virtual-to-physical address mapping is used to identify the memory for cache probing. However, since the address space of a user program is configured by the kernel, an application only has access to virtual memory addresses, and the virtual-to-physical address translation is not available to a user process. Lack of access to this translation poses a significant challenge, because most modern processor caches are physically indexed and physically tagged. Without the physical address of the memory, the attacker would not be able to target specific cache lines during the prime and probe attack.

All previous prime and probe based attacks [18], [20], [15] focus on resolving the translation from virtual address to physical address. Using this translation along with the cache indexing scheme from physical address to cache set, the attacker can identify memory that will map to a specific cache location known to be used by the victim. In [20], [15], the large offset within a huge page is used to gain insight of the mapping from virtual memory address to cache set. However, huge page support has not yet been incorporated in the mainstream Linux kernels for ARM processors. Alternatively, the unprotected Linux proc file system may be used to figure out the process memory address mapping [19]. However, this address mapping information is protected in many mainstream kernels [58] due to the severity of the rowhammering attack [59]. Our memory allocation strategy in TruSpy takes a different approach. Instead of extracting the virtual-to-physical address mapping from unprotected OS functions [19], TruSpy obtains the probing memory without address translation. We present two complementary methods for allocating the probing memory as follows.

Statistical Matching: With the statistical matching, TruSpy identifies the probing memory page by observing its correlation to the victim process. The intuition behind our allocation method is that, for the memory page that can cause contention with the victim, it is often possible to observe patterns of victim's footprint on the memory.

Using AES as an example, the memory of interest in the victim process is the T-table, which has a size of 4 KB. When a sensitive function, such as a cryptographic routine, executes, it will pollute some portion of the L1 cache with its access to T-table entries. Most modern systems use 4 KB memory page. Without loss of generality, let's assume the T-table is split into two different virtual 4 KB pages. The first half of the T-table will be mapped to higher addresses of one page, while the second half of the T-table will be mapped to lower addresses of another page. Furthermore, we know that the table access during the encryption will cause cache contentions with other processes that are mapped to the same cache area. The attacker should be able to observe the cache pollution pattern to determine if the page can be used for probing. More specifically, when a user page shows pollution in the higher addresses, we can conclude it correlates to the first part of the T-table. On the other hand, if the pollution is in the lower addresses of the page, then this page maps to the second part of the T-table. The heat map for cache pollution of two memory pages on our platform is shown in Figure 4. The color shows the number of times that the cached probing memory is evicted, therefore the darker the color, the more pollution there is. The cache pollution pattern of the two pages can be distinguished by observing

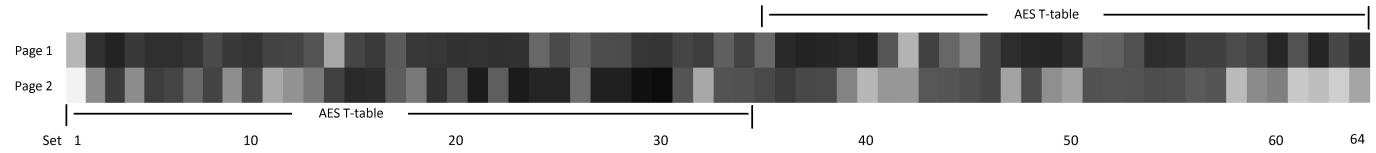

Figure 4. Cache Heat Map after AES Encryption

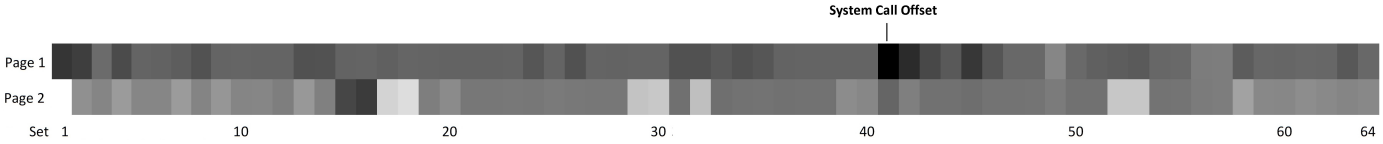

Figure 5. Cache Heat Map for Kernel Function Tracing

the color difference at the higher cache sets of the page. Page 1 shows significantly more pollution at higher offsets. Using this method, an attacker can reliably determine if a memory page correlates to the first part of the T-table or the second part of the T-table. Our implementation of the attack uses *mmap* interface to allocate a large trunk of memory (e.g., 2MB in our experiments) and use the aforementioned method to determine the relative position of the memory page with respect to the T-table by observing the cache pollution. The labeled allocated memory can then be used for prime and probe. Even though our illustrated example is on T-table based implementation of AES for the evaluation platform, the presented approach applies a wide range of software and platforms, as long as the victim process exhibits statistical properties.

Kernel Function Correlation: Even though statistical matching offers unique opportunities to allocate probing memory without resolving the virtual-to-physical memory translation, it can be limited when there are too few samples to be statistically significant or when there is no inherent statistical property that can be exploited. We propose a complementary method called *kernel function correlation* when statistical matching does not apply. The basic idea behind kernel function correlation is that it is possible to determine the cache set of the virtual user memory page by observing the caching pollution when a kernel function with known physical offset is invoked. The first step is to identify kernel functions that will map to different parts of the targeted cache. In the second step, for any given virtual memory page obtained in the attacker process, the cache pollution statistics is collected when each of these kernel functions is invoked. Kernel functions that are mapped to the same cache area with the virtual user page should cause the most cache line evictions. In the last step, the physical address offset of the kernel function can be used to determine the physical address offset as well as the cache set number of the user page.

Figure 5 shows the cache pollution levels of two memory pages when the same kernel function is invoked. Page 1 maps to the same portion of cache as the kernel function, while page 2 maps to a different part of the cache. We can see that they are clearly distinguishable. The cache pollution at the offset of the kernel function that is invoked can be clearly observed.

4.3.2. Probing the Cache with Performance Event System Call. As previously discussed, an accurate timer is required to distinguish memory access from L1 cache, L2 cache or DRAM memory. In x86 systems, *rdtsc* is accessible to both user space programs and the kernel. However, reading the performance monitoring cycle counter in ARM is a privileged operation that is only accessible to the kernel. For an Android app on a protected (non-rooted) mobile device, it is impossible to access cycle counter with the MCR instructions. However, since Linux kernel version 2.6, a performance event system call has been added to the kernel for user space applications to access the cycle counter.

To use the performance event interface provided by the kernel, the user space process first calls the *perf_event_open* function with a *perf_event_attr* struct as a parameter. The struct specifies the process id, CPU id and the type of performance event to monitor. Upon successful registration, the kernel returns a file descriptor for further control and communication. Using the *IOCTRL* function, the performance monitoring event can be reset and enabled. Now an app can use system calls to obtain a 64-bit hardware cycle counter from the performance monitoring unit in the ARM processor.

Though the performance event function in the kernel allows user space programs to access hardware timer using file descriptor operations, the handling of system call still introduces a significant amount of noise in the measurement. This is especially true when the timer is used to measure a single memory load operation, which is on the scale of microseconds. The relative scale of the noise can be observed using the L1 cache access time measurement as an example. The mean of L1 cache access time measured in kernel is 90.7 us and the standard deviation from a sample of 1000 measurements is only 3.91. On the other hand, the mean of L1 access measured with the performance event system call interface is 1745 us and the standard deviation is 1166.31, more than half of the mean. While L1 access can be clearly distinguished with memory access in the kernel,

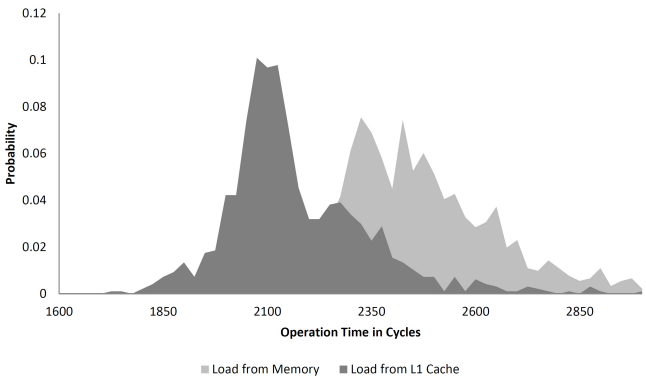

Figure 6. Memory LDR Operation Time using perf_event_read

distinguishing cache access from memory access using the performance event interface as a timer is not a trivial task. Differentiating access to the top level cache from access to the secondary level cache is even more challenging with this amount of noise. A probability distribution of memory load time is shown in Figure 6 for cache hit and cache miss. In a cache miss, memory content is filled into the caches from the DRAM, then loaded into the processor register. In a cache hit, the content is loaded into the register directly from the fast processor cache. Therefore, *ldr* instruction is significantly faster if it is a cache hit. However, as shown in Figure 6, the two distributions overlap significantly due to the added noise from the system call.

Due to the lack of inclusiveness or exclusiveness guarantee and the random replacement policy on cache in ARM processors, TruSpy resortes to probing the L1 cache. In order to use L1 cache for probing, it is necessary to distinguish access to L1 cache and access to L2 cache. The timing difference between access to different levels of cache is significantly smaller than that between cache and physical memory. This small difference in timing can be difficult to measure due to the system call noise depicted in Figure 6. Furthermore, there is no architecture support method to fill memory only into L2 cache lines from the user space. Though it is possible to use the write allocation trick [7] to write only to L2, *write_alloc* is not enabled by default when memory pages are mapped to the user space process, and the page caching attributes cannot be modified from the user space. It becomes impossible for an attacker without root privilege to fill only L2 cache lines to even measure the L2 access time.

To tackle this challenge, we make an assumption on the L2 access time distribution and use the value that will maximize the probability of correctly labeling samples to L1 cache access and L2 cache access in two steps.

First, we assume that the access time distribution of L2 is similar to the access time distribution of memory. We make this assumption because memory access from L2 will often cause cache eviction from L1. Due to lack of inclusiveness in the cache hierarchy, the cache line evicted from L1 is most likely not cached in L2. In order to make space for the newly evicted L1 cache line, a line in L2 has to be randomly selected for eviction out into the memory, and the evicted line from L1 is now stored in L2. This chained cache line eviction also applies when contents are loaded from memory. Though the exact strategy in cache miss handling varies in different processor implementations, almost all of them involve filling the cache line in one or more levels of cache. This cache fill is accompanied by cache eviction as described above. As a result, we use the probability distribution of memory access to estimate the probability distribution of L2 cache access. This estimation is not perfect, and it can always be improved with better models by incorporating details in the cache miss handling algorithm and the cache replacement algorithm for the specific processor model.

The next step is to estimate the population mean. We assume that despite of the noise in the measurement process, the population mean of access time from different levels of the memory hierarchy (L1, L2, DRAM) using perf_event_read should follow the general patterns measured by directly utilizing the performance counter. This measurement is hardware specific, and can be obtained by the attacker on a test system or from hardware datasheets. We shift the mean of the population of memory access time proportionally as those values measured using the raw performance counter to compensate the fact that the line fill is from L2 instead of DRAM.

With the estimated population of L2 cache access time, we are able to calculate an optimal cut-off value to maximize the probability of correctly asserting the cache access level. This optimal cut-off value is key to enabling fine-grained access-based timing side-channel given a noisy timer. Our approach to obtain the optimal cut-off value is not restricted to the performance event timer. If the performance event timer is not available, we can still apply the same technique to get an estimate using less accurate timers such as POSIX real-time clock or another thread that keeps an incrementing variable.

5. Extracting Fine-Grained Secret - Applying TruSpy to AES

To demonstrate the capability of TruSpy on extracting fine-grained secrets from the secure world, we apply TruSpy to recover the AES secret key protected by TrustZone. Though there are new hardware accelerators for AES [47], [48] that do not use T-tables in memory, we select AES as the target victim since the side-channel information leakage of AES is well-studied and has been used as a benchmark in other studies on side-channel information leakage of isolated containers [31], [18], [19], [15], [24].Thus, our experimental results, such as bits correctly recovered, can be compared with other related works. Furthermore, AES is representative for all table based cryptographic implementations. The same attack on the C implementation of AES applies to other table based implementations.

In the rest of the section, we will present the evaluation of TruSpy on AES encryption. The section is organized

as follows. The side-channel vulnerability of table based AES implementation is introduced in 5.1. The evaluation platform is then presented in 5.2. Lastly, the detail results and methods for the attack from kernel and from normal world app are presented in 5.3 and 5.4 respectively.

5.1. AES Side-Channel

Advanced Encryption Standard (AES) [60] is one of the most widely used symmetric key cryptographic algorithm. In this section, we show how TruSpy can be applied to recover the full AES key protected in the secure world of TrustZone. More specifically, our attack targets the software-based AES implementation in the OpenSSL 1.0.1f. The latest version of OpenSSL is currently 1.0.1h, which has the same AES implementation as 1.0.1f in our demonstration.

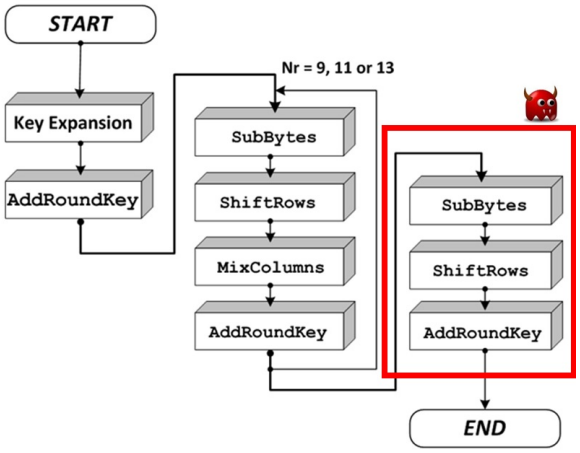

Figure 7. The Last Round of AES Algorithm

As shown in Figure 7, AES is built on top of four basic operations, *substitution bytes*, *shift rows*, *mix columns*, and *add round key*. When a block needs to be encrypted, the secret key is first expanded into N round keys. In the case of AES128, it is 10 rounds. If we denote the plaintext input block to be X, the round number be i, round key to be K_i, mix column to be $MC()$, substitution bytes be $SB()$, and lastly shift rows be $SR()$, then we can rewrite the process described in Figure 7 mathematically as,

$$X_i + 1 \begin{cases} X_i \otimes K_i & i = 0 \\ MC(SR(SB(X_i))) \otimes K_i & 0 < i < i_{max} \\ SR(SB(X_i))) \otimes K_i & i_{max} \end{cases}$$

To speed up the Galois field (GF) operations, modern software-based AES implementations, including the latest OpenSSL, use precomputed values stored in a large table. There are four tables in the OpenSSL, T_0 to T_4, each has 256 integers and occupies 1 KB memory. Therefore, the total size of all four tables is 4 KB. The entire table can often fit in the top level processor cache. With this technique, GF operations are translated into simple table lookups in cache and bit shifts, significantly improving the performance. Unfortunately, this table-based implementation also offers an opportunity to extract the secret key based on access patterns of the table.

While there are various ways to utilize this access pattern to extract keys [33], [18], [31], our attack focuses on the last round of the AES [33], [15]. In AES-128, it is possible to deduce the original encryption key with any one of the round keys, since the key schedule is invertible [33]. Let $C[j]$ denote the j^{th} byte of the ciphertext C after the encryption, and the last round can be written as

$$C[j] = T_l[X_{i_{max}}] \otimes K_{i_{max}}[j] \quad (1)$$

From Equation 1, we notice that the last round key can be recovered by taking the XOR of the T-table entry and the cipher text value. In our system model, cryptographic processing is executed in the secure world. A normal world process provides plaintext to the secure world and receives the ciphertext as result from the secure world. Therefore, the ciphertext is known to the attacker. The other variable in the equation is the T-table entry, which can be guessed from the cache access pattern obtained from the prime and probe.

Each cache line in Cortex-A8 has 64 bytes. Each entry in the T-table is an integer. Therefore, each cache line holds 16 T-table entries. During the TruSpy attack, when a cache line previously occupied by the attacker is evicted during the execution of AES by the victim, the attacker can deduce that 1 out of the 16 possible entries in the cache line was used in the encryption process, and thus producing 16 possible candidates for the j^{th} key byte. Ideally, the attacker needs to interrupt the encryption process and use only the cache profile changed in the last round to deduce the key. Unfortunately, since secure world is protected by the TrustZone and is not interruptible, a cache miss in the probing step does not necessarily indicate that the victim in the secure world used that particular T-table entry on the last round. Such uncertainty can be compensated with larger samples. Assuming AES is used in the CBC mode, the input blocks to the AES algorithm can then be considered random bits. Under random input, each T-table entry should have $(1 - Prob(\text{no access})) \approx 92\%$ probability of being used in each round of encryption. However, when the T table entry is used in the last around, cache probing would show it is always used. Therefore, by probing more samples, it is still possible to deduce the value of the key byte.

In our implementation, since the AES key has 16-byte length and each byte can be any of the 0 to 255 values, we set up a 256 by 16 two-dimensional array to store the statistics. After each round of prime and probe, for each of cache lines evicted by the encryption process, we increment the counter value at result$[j][k_i]$. In the end, we use the counter value at result$[j][k_i]$ as the probability of j^{th} key byte equal to k_i.

5.2. Evaluation Platform Configuration

We build a prototype of TruSpy on the FreeScale i.MX53 development board. It is equipped with a single ARM Cortex-A8 processor with 1GB DDR3 DRAM. ARM Cortex-A8 has two levels of cache. The top level one (L1) cache has two 4-way 128-set caches, one for data and one for instruction. The level two (L2) cache is an 8-way 512-set unified cache. The Android system is ported from Adeneo Embdedded [61], running Android 2.3.4 platform with a 2.6.33 Linux kernel. The security monitoring code is approximately 800 source lines of code (SLOC). To demonstrate the attack, we port the C reference AES functionality in OpenSSL 1.0.1f [62] into TrustZone. The AES C reference implementation is the default solution when no-hw and no-asm options are specified. Lastly, since the security monitor code size is small for minimizing the trust computing base (TCB), there is no file system support in the secure world. Once the AES code is loaded into secure memory along with the rest of the security monitor system, it stays at a fixed location in the physical memory. This implies that the four T-tables have a fixed physical memory address.

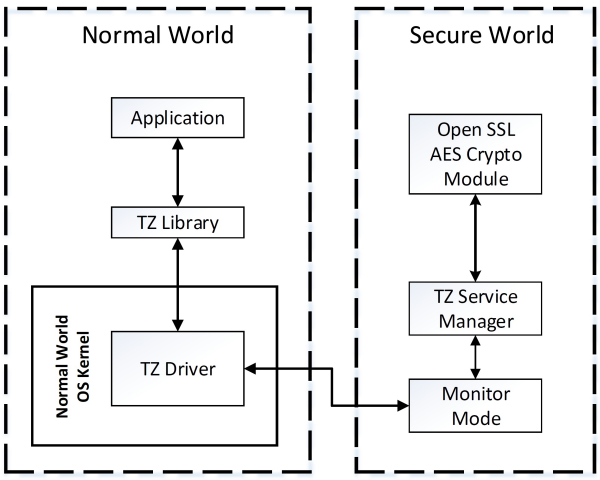

Figure 8. Platform Software Deployment

To use a TrustZone function from an Android app, we implement a full stack service framework based on the design published in the ARM TrustZone whitepaper [4]. More specifically, we write a simple secure execution driver that allows a user space app to interact with the cryptographic module in the secure world. The software architecture diagram is shown in Figure 8. The TZ library exposes three types of functions to the user application, *set_key*, *encrypt*, and *decrypt*. Upon receiving a request from the user process, the TZ library prepares the memory buffers and structures and then invokes the system call using the *swi* instruction. Next, the TZ driver in the kernel space copies the user buffer into kernel and switches into the security monitor via *smc* instruction. Once the AES crypto module receives the request from the TZ service manager, it processes the request. When the request is to encrypt, the crypto module operates on the normal world memory for reading in plaintext and writing out ciphertext. The code and data of the cryptographic module are stored within the secure world memory and thus protected from the normal world. Cache is enabled in all components, including stack, code, and data of all secure components.

In our attack demonstration, the encryption key is chosen at random by the TZ service manager in the secure world. Following the design philosophy of TrustZone, we assume that most of the I/O and content consumers reside in the normal world. Therefore, the AES crypto module in the secure world offers encryption and decryption service to processes in the normal world. Each user level process can trigger an encryption via the TZ support library. Therefore, the user-level process knows the start and the end of encryption as well as the resulting ciphertext.

5.3. TruSpy Attack from Normal World OS

We implement the OS-level attack as a kernel module. The prime and probe steps are implemented in assembly to avoid cache pollution during the probing process. We assume that the physical address of the AES T-table is known through reverse engineering. Memory section is used to map memory into cache sets. We then search for memory that can cause cache contention with the victim using the large offsets in memory section.

Once the memory to be used for prime and probe is identified, the attack repeatedly invokes the AES encryption function with random plaintext, and record cache changes after each encryption. After a fixed number of encryptions, it attempts to guess the key. The effectiveness of the attack is shown in Figure 9. The x-axis shows the number of encryptions observed, and the y-axis shows the number of key bytes correctly guessed. For AES128, the key length is 16 bytes (128 bits). Each data point is an average of 100 experiments. We can see from the graph that it takes roughly 3000 rounds of encryption for the attacker to correctly guess the entire key. It takes approximately 2.5 seconds to execute and analyze 3000 rounds of AES encryption on our embedded processor. Note that when one or two bytes are not guessed correctly, the correct key byte is often the second or third on the list. Therefore, if the result analysis is done on a different machine with more computing power, the number of encryptions required can be further reduced.

5.4. TruSpy Attack from Normal World App

It is significantly more challenging to attack processes in the secure world from a non-privileged Android app in the normal world. First, due to lack of access to virtual-to-physical address mapping, we have to use the statistical matching method introduced earlier in the paper to allocate the memory pages used to probe the T-table access. For the TruSpy attack on our evaluation platform, since the memory allocation of the table is pre-compiled in the data section of the binary and it remains fixed and contiguous within the secure domain, we can identify the offset of the table

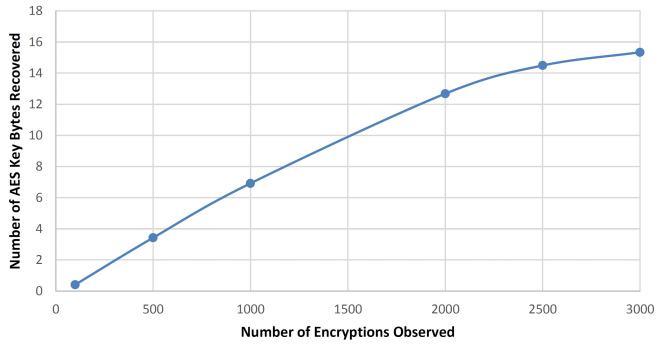

Figure 9. Attack from Normal World OS

through reverse engineering. We know that the T-table starts off at 0x1780, therefore the T-table is mapped to 0x0 to 0x780 and 0x1780 to 0x2000 in the L1 cache as shown in Figure 10. Given that each memory page has 0x1000 bytes, a memory page will either have cache pollution from 0x0 to 0x780 or from 0x780 to 0x1000. If we observe that the memory page has cache pollution at offset 0x780 to 0x1000, then we know that this page correlates to the first part of the T-table, and resides in the upper L1. On the other hand, if we observe cache pollution from offset 0x0 to 0x780, then we know it most likely correlates to second part of the T-table and maps to the lower part of L1. Ideally, we would want to use memory pages that are mapped to the same L2 cache area as the T-table for more accurate priming. Unfortunately, using statistical matching, we can only gain knowledge if the memory page is suitable to use for probing, which is most likely determined by its offset in the L1 cache. Given the noise level of perf_event_read system calls, random cache replacement in the cache hierarchy, and inability to directly write to L2 in user space, it is very difficult to identify the L2 mapping a memory page based on the timing side-channel itself.

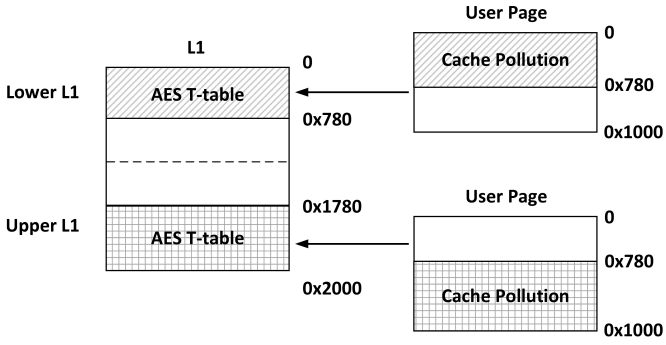

Figure 10. TruSpy Probing Memory Allocation

The second challenge is the lack of access to an accurate timer. In order to distinguish L1 cache access from memory access to other levels of memory, we need to pinpoint the tiny difference between L1 access time and L2 access time. However, there is no architectural support to write directly to L2. Therefore we use the estimated L2 access time as discussed in 4.3.2. Even though we can choose the optimal cut-off value to maximize the probability of correctly differentiating L1 access from L2 access, the inaccurate timer still causes a significant amount of noise in the probing process.

Despite the two aforementioned challenges, we are still able to steal the full AES key within 9000 rounds of encryption. The results of attack from Android app is shown in Figure 11. The x-axis shows the number of encryptions observed, and the y-axis shows the number of key bytes correctly guessed. It takes approximately 9000 encryptions to correctly guess all the key bytes. It takes approximately 14 minutes to run and analyze 9000 rounds of AES encryption on our test platform. Much of this time is used in loading a large amount of memory in order to evict the secure cache. Since user space does not have access to the translation from virtual-to-physical address to optimally evict secure cache lines, we use the naive approach of loading significant amount of memory to increase the probability of evicting all secure cache lines.

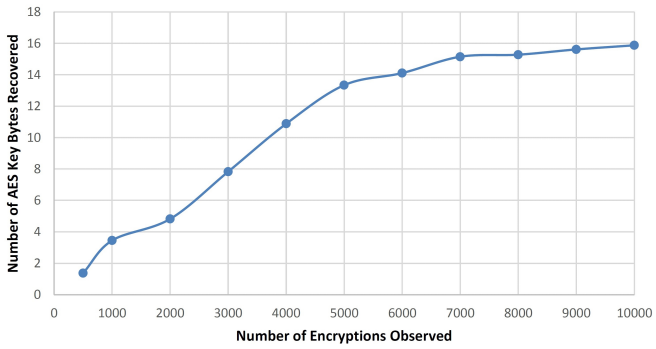

Figure 11. Attack from Normal World Application

6. Countermeasure

Since the first cache-based timing side-channel attack [14] proposed nearly two decades ago, much research has been done to advance the attack and defense on cache information side-channel. Similar to the previous work, the ultimate goal of this study is to further our understanding in side-channel information leakage in secure containers and improve future designs. Defense against side-channel attacks generally follows two directions. The first direction attempts to work on the implementation of the cryptographic operation itself [63], while the second direction focuses on hardening the system on which the cryptographic operations runs on to eliminate the timing side-channel [64], [37], [65], [36], [66].

6.1. Application Oriented Defense Approach

Among the general approaches to improve the software implementation of the cryptographic algorithm, the most straight forward approach is to focus on the cryptographic algorithm itself. Hardware-based cryptographic implementations such as AES-NI instruction [47] can be used to replace the software-based implementation and thus eliminate the side-channel information leaked from the execution of software. OpenSSL also adopts one mitigation to preload the AES T-table [62], so that the access to the table can no longer be detected by attackers, assuming that the execution of the cryptographic algorithm cannot be interrupted. On the other hand, Cock et al. [67] demonstrate that despite the constant time fix in OpenSSL 1.0.1e, it still exhibits a considerable side-channel on the ARM Exynos4412 processor.

There are also defense efforts to randomize the software control flow [63] so that there is no fixed relation between the execution path and the cache set. Without grasping such mapping information, it would be impossible for the attacker to deduce secrets from the cache profile. Similar techniques can be applied to randomize T-table entries. This way, TruSpy will not be able to aggregate the statistics to deduce the key.

6.2. System Oriented Defense Approach

This general direction focuses on breaking the two requirements of cache timing side-channel attacks, including TruSpy. First, an attacker has to be able to fill memory into cache to cause resource contention with the victim process, a line of research focuses on eliminating this resource contention. In [64], memory allocations are crafted by the kernel so that attacker memory will never be mapped to the same cache set as the victim memory. It can successfully thaw our Android app level attacker; however, the OS level attacker, who has full control of all non-secure memory, can still succeed. The defender can also preload the sensitive binary in the cache before execution [7] or flush sensitive secure cache to memory to eliminate the information on the table access. However, this approach has high performance impacts on TrustZone-enabled ARM platforms that are designed to share the cache between two worlds without the need to flush cache during world switching. Lastly, there are also secure cache hardware designs that can eliminate the cache timing side-channels with moderate performance penalty [68], [69].

The second requirement of creating a cache timing side-channel is the ability to detect changes in the cache states. Thus, the attacker needs a high precision timer to distinguish memory access from different levels of memory system. If we can constrain the access of the kernel performance event interface to only privileged processes, we can prevent the access to perf_event from Android apps in non-rooted mobile phones.

7. Discussion and Future Work

7.1. Applicability to Other Target Victims

In this work, we use AES as a demonstration to show that TruSpy is capable of extracting fine-grained information, such as secret keys, from the secure world of TrustZone. However, the applicability is by no means limited to AES. The same method that tracks T-table access in AES can be applied to other table based cryptographic implementation [15], [19], [20]. The methods presented in TruSpy can also be applied to trace the execution of sensitive input-dependent function in the secure world as demonstrated in [22], [19]. Lastly, when KASLR is implemented in the secure world, it is plausible to use cache side-channel to break it [70].

7.2. Applying TruSpy on Other Platforms

Though the principle of TruSpy is not restricted to single core processor, since our current implementation makes use of the top level L1 cache to reduce noise, some further study needs to be performed to extend our mechanism on multi-core processors. In many modern multi-core processors, each individual processor core has its own dedicated L1 cache. In order to continue using cache contention on L1 as the source of side-channel information, the attack needs to be scheduled to run on the same processor core right after the victim core, which could often be difficult to achieve. On the other hand, the last level cache is usually shared by all cores, and a number of previous works focus on using the last level cache as the source for side-channel information [15], [19], [22], [20]. When extending TruSpy to work on multi-core processors such as ARM Cortex-A9 [26], the attack will need to use the last level shared cache to apply the prime and probe technique. Fortunately, on some ARM processors such as CortexA-9, the cache controller enforces inclusiveness to make sure any cache line present in a higher level cache is also present in the lower level cache. However, there will be additional noise in the lower level cache due to the random cache replacement policy. It also is possible to exploit cache locking [25], [26] to prevent cache fill into specific cache ways and thus reduce noise in the channel.

8. Conclusions

In this work, we propose the first timing based cache side-channel attack capable of extracting secrets from the hardware-protected secure world in TrustZone. We develop two attacks, one in the kernel space and one in the user space of the normal world, respectively. When the normal world OS is compromised, it takes approximately 3000 rounds of observed AES encryption in 2.5 seconds to extract the full AES encryption key from the secure world. For a malicious Android app that does not have the root privilege, despite the lack of access to virtual-to-physical translation as well

as an accurate timer, our attack can correctly extract the full AES key with around 9000 rounds of observed encryption in a couple of minutes. Since our attack relies on one basic design of TrustZone enabled cache architecture and does not use any unique functionalities from a particular version of Android, it has impacts on a wide range of ARM processors running various Android versions.

Disclaimer

The opinions expressed in this article are the authors' own and do not reflect the view of the National Science Foundation or any agency of the U.S. government.

References

[1] "Google i/o 2016." https://events.google.com/io2016/. Accessed: 2016-07-27.

[2] G. Ho, D. Leung, P. Mishra, A. Hosseini, D. Song, and D. Wagner, "Smart locks: Lessons for securing commodity internet of things devices," 2016.

[3] "Arm holdings and qualcomm: The winners in mobile." http://www.forbes.com/sites/darcytravlos/2013/02/28/arm-holdings-and-qualcomm-the-winners-in-mobile/.

[4] "ARM Security Technology, Building a Secure System using TrustZone Technology," apr 2009.

[5] "Samsung knox." https://www.samsungknox.com/en.

[6] "Sierraware." http://www.sierraware.com/open-source-ARM-TrustZone.html.

[7] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case:cache-assisted secure execution," in *IEEE Symposium on Security and Privacy*, 2016.

[8] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 90–102, ACM, 2014.

[9] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for ARM," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pp. 558–569, 2014.

[10] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment," in *Proceedings of the Network and Distributed System Security Symposium*, NDSS'15, 2015.

[11] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology-CRYPTO'99*, pp. 388–397, Springer, 1999.

[12] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "Ecdsa key extraction from mobile devices via nonintrusive physical side channels." Cryptology ePrint Archive, Report 2016/230, 2016. http://eprint.iacr.org/.

[13] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology–CRYPTO 2014*, pp. 444–461, Springer, 2014.

[14] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology-CRYPTO'96*, pp. 104–113, Springer, 1996.

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes," in *2015 IEEE Symposium on Security and Privacy*, pp. 591–604, May 2015.

[16] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 719–732, 2014.

[17] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Security and Privacy (SP), 2011 IEEE Symposium on*, pp. 490–505, IEEE, 2011.

[18] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Topics in Cryptology–CT-RSA 2006*, pp. 1–20, Springer, 2006.

[19] M. Lipp, D. Gruss, R. Spreitzer, and S. Mangard, "Armageddon: Last-level cache attacks on mobile devices," *CoRR*, vol. abs/1511.04897, 2015.

[20] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, pp. 605–622, May 2015.

[21] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *21st ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 990–1003, ACM, 2014.

[22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "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*, pp. 199–212, ACM, 2009.

[23] D. J. Bernstein, "Cache-timing attacks on aes." https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[24] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *IEEE Symposium on Security and Privacy*, 2016.

[25] *ARM Cortex-A8 Processor Technical Reference Manual*, June 2012.

[26] *ARM Cortex-A9 Processor Technical Reference Manual*, June 2012.

[27] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology-EUROCRYPT'97*, pp. 37–51, Springer, 1997.

[28] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology-CRYPTO'97*, pp. 513–525, Springer, 1997.

[29] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh.," in *USENIX Security Symposium*, vol. 2001, 2001.

[30] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 54–65, ACM, 2014.

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

[32] M. Weiss, B. Heinz, and F. Stumpf, "A cache timing attack on aes in virtualization environments," in *Financial Cryptography and Data Security*, pp. 314–328, Springer, 2012.

[33] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *Research in Attacks, Intrusions and Defenses*, pp. 299–319, Springer, 2014.

[34] A. C, R. P. Giri, and B. Menezes, "Highly efficient algorithms for aes key retrieval in cache access attacks," in *IEEE European Symposium on Security and Privacy*, 2016.

[35] B. A. Braun, S. Jana, and D. Boneh, "Robust and efficient elimination of cache and timing side channels," *arXiv preprint arXiv:1506.00189*, 2015.

[36] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1595–1606, ACM, 2015.

[37] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 406–418, March 2016.

[38] Q. Xiao, M. K. Reiter, and Y. Zhang, "Mitigating storage side channels using statistical privacy mechanisms," in *22nd ACM SIGSAC Conference on Computer and Communications Security*, (New York, NY, USA), pp. 1582–1594, ACM, 2015.

[39] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 431–446, USENIX Association, Aug. 2015.

[40] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *17th ACM Conf. on Computer and Communications Security (CCS)*, pp. 297–307, October 2010.

[41] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *53rd Design Automation Conference (DAC)*, June 2016.

[42] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers, "Sharing mobile code securely with information flow control," in *IEEE Symp. on Security and Privacy*, pp. 191–205, May 2012.

[43] O. Acıiçmez and Ç. K. Koç, "Trace-driven cache attacks on aes (short paper)," in *Information and Communications Security*, pp. 112–121, Springer, 2006.

[44] C. Percival, "Cache missing for fun and profit," 2005.

[45] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented flush-reload side channels on arm and their implications for android security," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '16, 2016.

[46] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 305–316, ACM, 2012.

[47] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Sep 2013.

[48] *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*, Dec 2011.

[49] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned aes t-tables," in *Constructive Side-Channel Analysis and Secure Design*, pp. 200–214, Springer, 2013.

[50] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as practical and secure location verication tokens for payments," in *Proceedings of the Network and Distributed System Security Symposium*, NDSS'14, 2014.

[51] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, pp. 75–88.

[52] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pp. 21–30, ACM, 2008.

[53] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *ACM SIGARCH Computer Architecture News*, vol. 42, pp. 67–80, ACM, 2014.

[54] D. Rosenberg, "Reflections on trusting trustzone," 2014.

[55] "Drm fusion downloadable and embedded drm solutions for ios, android and linux." http://www.insidesecure.com/content/download/2682/18371/version/4/file/A4_FLYER_DRM_Fusion_Agent_Gene_250315_2P_HD.PDF. Accessed: 2016-04-27.

[56] R. Thomas, "Next generation mobile rootkits," *Black Hat Europe*, 2013.

[57] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *IEEE European Symposium on Security and Privacy*, 2016.

[58] "Linux archive - patch 3.14 00.79." http://lwn.net/Articles/637892/. Accessed: 2016-04-30.

[59] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," in *Black Hat conference, https://www. blackhat. com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Ker nel-Privileges. pdf*, 2015.

[60] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[61] "Reference bsps for freescale i.mx53 quick start board." http://www.adeneo-embedded.com/Products/Board-Support-Packages/Freescale-i.MX53-QSB. Accessed: 2015-04-30.

[62] E. A. Young, T. J. Hudson, and R. Engelschall, "Openssl: The open source toolkit for ssl/tls," 2011.

[63] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity.," in *NDSS*, 2015.

[64] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: system-level protection against cache-based side channel attacks in the cloud," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 189–204, 2012.

[65] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 297–307, ACM, 2010.

[66] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," *arXiv preprint arXiv:1603.05615*, 2016.

[67] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 570–581, ACM, 2014.

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

[69] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 494–505, ACM, 2007.

[70] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 191–205, IEEE, 2013.