TruSense: Information Leakage from TrustZone

Ning Zhang*, Kun Sun†, Deborah Shands‡ Wenjing Lou*‡, Y. Thomas Hou*
*Virginia Polytechnic Institute and State University, VA
†George Mason University, Fairfax, VA
‡National Science Foundation, Arlington, VA

***Abstract*—With the emergence of Internet of Things, mobile devices are generating more network traffic than ever. TrustZone is a hardware-enabled trusted execution environment for ARM processors. While TrustZone is effective in providing the much-needed memory isolation, we observe that it is possible to derive secret information from secure world using the cache contention, due to its high-performance cache sharing design.**

In this work, we propose TruSense to study the timing-based cache side-channel information leakage of TrustZone. TruSense can be launched from not only the normal world operating system but also a non-privileged user application. Without access to virtual-to-physical address mapping in user applications, 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 how a normal world attacker can steal fine-grained secret in the secure world. We also discuss possible mitigations for the information leakage.

I. Introduction

Mobile devices are becoming an integral part of our life, handling more and more sensitive data such as bank accounts, private medical data, and even presidential electoral votes. With the ongoing transformation of information access from bulky desktop to the always available and connected mobile device, more important services are running on a single mobile platform now. At the same time, it is becoming increasingly challenging to produce bug free software due to the complexity of modern program. One of the promising answers to this challenge is to use hardware-enable secure execution technology [1], [2]. It provides an isolated execution environment to protect security sensitive tasks. Applications in these containers are protected by a small trusted computing base (TCB) utilizing hardware support from the rest of the system. Thus, even if the highly complex software is compromised, sensitive information remains protected in these containers.

ARM family processors have been deployed in more than 95% of the smart phones [3]. *TrustZone* [2] on ARM processors offers the ability to protect security sensitive tasks within an isolated execution environment. TrustZone has been adopted in a wide variety of commercial products such as Samsung Galaxy series, LX Nexus series, HTC one series [4], [5] and academic projects [6], [7] to enable secure processing. The protected environment is called *secure world*, and the normal environment is called *normal world*. While secure containers such as TrustZone are effective in providing separation for computations, information leakage via side channel remains a challenge [8], [9].

Unlike software exploitations that target vulnerabilities in the system, side-channel attacks target information leakage of a physical implementation via its interactions with the execution environment. Side-channel information can be obtained from different types of physical features, such as power [10], electromagnetic wave [11], acoustic [12] and time [13], [14], [15]. Among these side channels, the cache-based timing attack is one of the most important areas of research [14], [15], [16], [8], [17], [18].

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 [2] to mark the security domain of these lines. 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.

Despite extensive studies [19], [20] on the side-channel leakage of the Intel SGX secure containers, the study on information leakage from TrustZone is still limited [8]. In this paper, we present our investigation on the severity of side-channel information leakage from TrustZone container to both the privileged normal world OS and unprivileged normal world applications. We show that TruSense can circumvent TrustZone protection to extract sensitive information in the secure world using less privileged normal world processes. OpenSSL is currently used by 4.3 million websites on the Internet, powering about 45% of the top 1 million servers [21]. To demonstrate the severity of the leakage, we apply TruSense to extract full AES encryption key from OpenSSL protected inside the TrustZone.

There are two key requirements for applying TruSense 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. We show how well these conditions can be met under different levels of access privileges to the system resources, and the differences in the resulting channels.

We study the side-channel leakage in two environments, *the normal world OS* and *the normal world user space application*. For the first environment, 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. For the second environment, we would like to answer the question of how much information can be extracted by a non-privileged user space application 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 target. Furthermore, we develop a method to find an optimized cut-off value to distinguish L1 cache access from L2 cache access by examining the sample population using a less accurate system performance event as a substitute for the high precision timer accessible only in the kernel.

Our study is conducted on an ARM CortexA-8 processor. The information leakage of TrustZone is demonstrated using a T-table-based AES implementation as a sample target, since side-channel attack on AES is widely used in the literature as example for fine-grained information extraction [14], [8], [16], [18], [9]. The experiments show that it is possible for the normal world kernel to recover the full AES128 secret key using 3000 rounds of observed encryption in 2.5 seconds. Despite additional challenges due to lack of privileged access, the user space application can still recover the full AES128 secret key with 9000 rounds of observed encryption within 14 minutes. The TruSense tool we use to evaluate the side channel is available at https://github.com/n1ngz/TruSpy.

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.
- We perform a detailed study on the severity of side channel information leakage of TrustZone. We show that even an unprivileged user space application can launch side-channel attacks on TrustZone. We tackle two challenges, namely, finding virtual-to-physical mapping and lacking of method to distinguish cache accesses at different levels. Without direct access to virtual-to-physical address translation, the user space application attack allocates memory by using statistical properties of the channel itself or correlating to kernel function with known addresses. Also, the TruSense is able to derive an optimized value to distinguish different levels of cache accesses by examining the population using a lower precision timer.
- We show the severity of the information leakage by implementing and testing the proposed side-channel attacks on a hardware platform. We also discuss potential mitigations against the side-channel information leakage on secure world of TrustZone.

II. Side-Channel Information Leakages

The study of information leakage from sensitive compartment dates back to the Orange Book [22] by the US Government. Information leakage channels are often generalized into either a *side channel* or a *covert channel*. A side channel refers to attacker obtaining sensitive information, such as secret encryption keys, by observing how the execution in the sensitive compartment interacts with its physical environment. Victims in side-channel attacks are trusted parties. On the other hand, a covert channel refers to the secret communication channel between the two colluding parties across security boundaries. Side-channel attacks are well-studied in computer security [13], [17]. With significant research on cache side-channel attacks [13], [12], [18], [23], [15], [24], [25], [16], [14] and defenses [26], [27]. The concept of using side-channel information as a means to attack cryptographic schemes first appeared in a seminal paper by Kocher [13]. In [13], Kocher exploited differences in computation times to break implementations of RSA and discrete logarithm-based cryptographic algorithms. Besides time, other physical attributes such as electromagnetic emission [11], power consumption [10] or acoustic noise [12] have been investigated as viable sources for side-channel attacks.

Microarchitectural Timing Side Channels: From the microarchitrectural perspective, leakage channels in software are often classified into storage channels and timing channels. Storage channels such as value in registers and return of system calls have been well studied [28]. However, there is little work to formally define the temporal behavior of different systems. Bernstein [18] was the first one to show the existence of timing dependencies introduced by the data cache for AES, which allows secret key recovery [18]. There are three main categories of cache-based side-channel attacks: time driven [18], trace driven [29], and access driven [14], [8], where the differences between them are the attackers' capabilities, with time driven attacks the least restrictive. Osvik et al. [16] 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. [24]

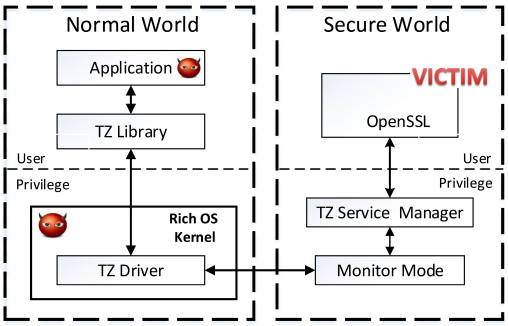

Fig. 1. TruSense Threat Model

identified another powerful cache side-channel attack enabled by shared memory. The attacker flushes the memory shared between the malicious process and the victim process, such as a crypto library. 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 [15].

Cache-based Timing Channels on ARM: Most of the current research on cache-based side-channel information leakage focuses on the Intel [30] architecture, and little has addressed the ARM [31] architecture [32], [8], [9]. Zhang et al. [32] proposed an attack on ARM devices using cache flush system call interface to perform flush and reload attack using the instruction cache. Lipp et al. [8] investigated the possibility of launching a wide range of side-channel attacks on ARM mobile devices. The primary focus is on applying the most powerful technique in x86, flush and reload, to ARM. Compared to [32], [8], our target is protected by TrustZone. We further demonstrates that the leakage enables attack from not only the normal world kernel but also normal world user application. 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 [9]. Compared to [9], TruSense does not require kernel privilege in the normal world, neither does it rely on programming faults at the hardware level.

To summarize, all the current studies [32], [8], [9] of cache side channel on ARM do not have a primary focus on leakage from the TrustZone. Furthermore, initial investigations on TrustZone attacks are all launched from the normal world OS kernel. In this work, we provide an in-depth study of the information leakage from TrustZone, by attackers not only in the kernel, but also in the user space of normal world.

III. Threat Model and Assumptions

As recommended by the ARM whitepaper [2], TrustZone is often used to protect security sensitive workloads, such as cryptographic operations, in an isolated execution environment [7], [6]. Cryptographic libraries are protected in the secure world from a potentially compromised normal world OS running in the normal world [6], [5]. The threat model for our work is shown in Figure 1. 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 user space application 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 just the application 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 [14], [8], [16], [18], 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 [5]. For our application level attack, we make no assumption of vulnerabilities in the operating system. Moreover, the user space application does not require any special permission to run the attack code. Therefore, the malicious code can be embedded into all types of applications and remains stealthy.

IV. TruSense - Sensing From the TrustZone Cache

ARM TrustZone [2] 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 [13].

TruSense 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 [15], [24], [32] 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 [16] to learn the cache access pattern of the victim process.

There are two requirements for a successful attack in TruSense. 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 [9].

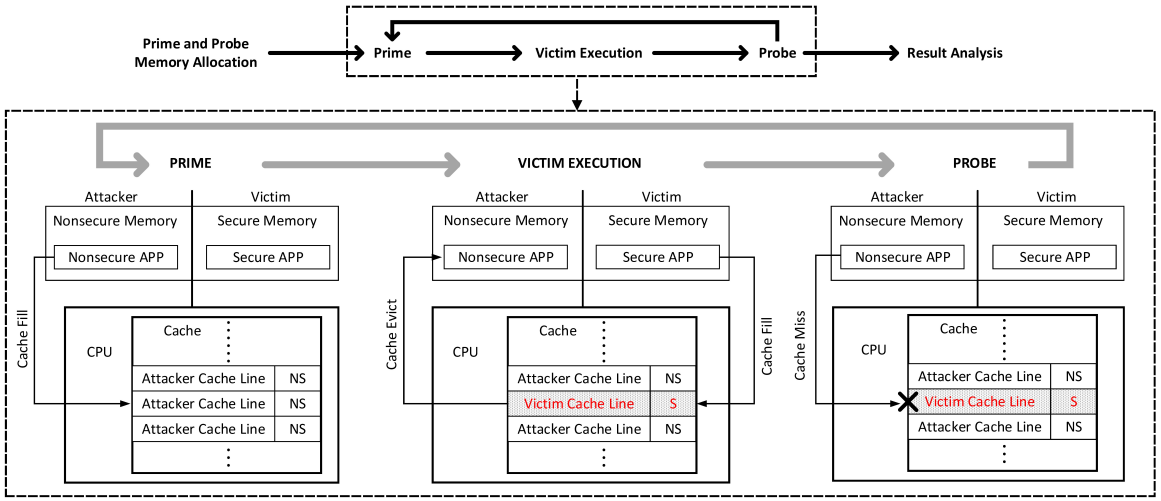

Fig. 2. TruSense Workflow

A. TruSense Workflow

The TruSense workflow consists of five major steps, as shown in Figure 2. 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 [16], [14], [25]. 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, 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 step two, the short execution time of memory load instruction indicates that the 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.

B. Attacking from Normal World Kernel

When the normal world OS is compromised, attackers have access to a variety of resources to execute the attack on TrustZone. Among these resources, the capability to obtain virtual-to-physical address mapping as well as the access to an accurate timer is crucial in TruSense.

1) 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. 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 more challenging. An attacker who controls the OS can perform a page table walk to figure out the physical address of any virtual address to obtain memory for priming.

2) Priming the Cache: Once the prime and probe memory is allocated, the attacker needs to fill in the cache. However, as shown in Figure 2, 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 [33]. Therefore, cache contention is used to push all the secure lines back to memory. Memory is repeatedly loaded in the normal world then invalidated using *clean and invalidate* cache instruction.

3) 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. We use the cycle count register (PMCCNTR) in ARMv7 performance monitoring unit as a high precision timer to distinguish cache hits at different levels of cache hierarchy. Data memory barrier and instruction barrier are also used to obtain accurate memory operaiton timing. From our experiment, 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) Countering Attack from Normal World Kernel: When the normal world kernel is compromised, the mitigation can only be deployed in the secure world. There are several approaches to counter the information leakage. The most straight forward approach is to eliminate the source of the side channel. Disabling caching in secure world or flushing the cache each time there is a world switch should eliminate the cache side channel, the performance impact can however be prohibitive. It is also possible to randomize the memory layout of the table used by the cryptographic function to increase the difficulty for prime and probe memory allocation by the attacker. Lastly, it is also possible to use hardware accelerator to perform cryptographic functions. However, this does not eliminate side-channel attack on other applications in secure world.

C. Attacking from User Space Application

It is significantly more challenging to launch TruSense attack from a user space application. 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 [31]. We will present our solution to these challenges in the following paragraphs.

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 [16], [14] 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 [25], [14], 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 [8]. However, this address mapping information is protected in many mainstream kernels due to the severity of the row hammering attack [34]. Our memory allocation strategy in TruSense takes a different approach. Instead of extracting the virtual-to-physical address mapping from unprotected OS functions [8], TruSense 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, TruSense 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 3. 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 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

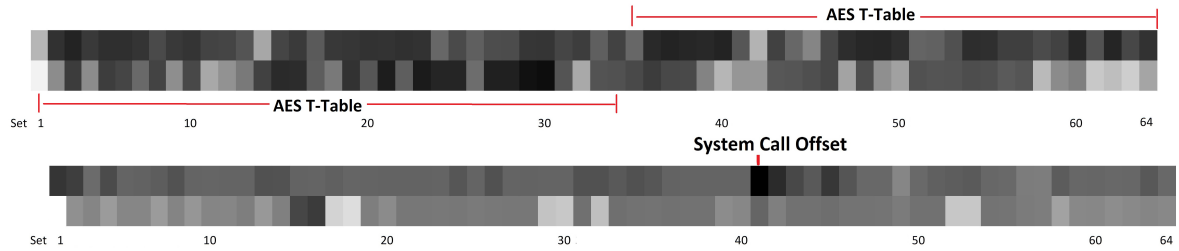

Fig. 3. Cache Heat Map for AES Encryption (top) and Kernel Function Tracing (bottom)

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. As shown in the first step, the kernel function symbol need to be known to the attacker. This can often be accomplished by reverse engineering with different tools and access to a sample system of the target with the identical system software. Figure 3 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.

2) Probing the Cache: As previously discussed, an accurate timer is required to distinguish memory access from L1 cache, L2 cache or DRAM memory. However, reading the performance monitoring cycle counter in ARM is a privileged operation that is only accessible to the kernel. For a user space application 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 application can use system calls to obtain a 64-bit hardware cycle counter from the performance monitoring unit in the ARM processor.

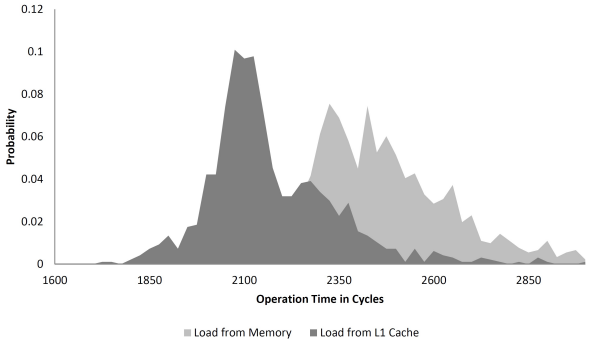

Fig. 4. Memory LDR Time Measured with Perf_event_read

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. While L1 access can be clearly distinguished with memory access in the kernel, 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 4 for cache hit and cache miss. *ldr* instruction should significantly faster on cache hits. However, Figure 4 shows that 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, TruSense resorts 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 4. Furthermore, there is no architecture support method to fill memory only into L2 cache lines from the user space. 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 often 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 to make room for the evicted L1 line. 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.

3) Countering Attack from Normal World Application: Besides the described countermeasures for normal world kernel, additional mitigation can be placed in the rich OS to prevent TruSense. From the memory perspective, it is possible to prevent the application from obtaining memory to cause contention. The memory allocation algorithm can be modified to only hand out pages that are mapped to non-sensitive regions of cache. It is also possible to flush cache before returning the TrustZone call back to the application. From the timer perspective, it is also possible for the rich OS to prevent access to performance interface. Unfortunately, there are often other alternative timing sources available [8].

V. Experiment Validation

A. Validation Target - AES Key Extraction

To demonstrate the capability of TruSense on extracting fine-grained secrets from the secure world, we apply TruSense to recover the AES secret key protected by TrustZone. Though there are new hardware accelerators for AES [30], [31] that do not use T-tables in memory, we select AES as the target victim because 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 [16], [8], [14], [9]. Thus, our experimental results, such as correctly recovered bits, can be compared with other related works. Furthermore, the same attack on the C implementation of AES applies to other table-based implementations. Briefly, if we denote the plaintext input block to be X, the round number to be i, round key to be K_i, mix column to be $MC()$, substitution bytes to be $SB()$, and lastly shift rows to be $SR()$, then we can rewrite the process 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. Therefore, the last round can be described as

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

If we know the ciphertex and T-table entry, then it is possible to obtain the key byte. The detail description of vulnerability of AES C implementation can be found in [35].

We build a prototype of TruSense on the FreeScale i.MX53 development board. The system is ported from Adeneo Embedded [36], running 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 [37] into TrustZone. AES code is loaded into secure memory along with the rest of the security monitor system, therefore it stays at a fixed location in the physical memory, therefore all the T-tables have fixed physical memory addresses. To use a TrustZone function

from a user space application, we implement a full stack service framework based on the design published in the ARM TrustZone whitepaper [2]. More specifically, we implemented a kernel driver to allow user space applications to interact with the cryptographic module in the secure world.

B. Key Recovery from Normal World OS

We implement the OS-level attack as a kernel module. The feasibility of launching prime and probe attack has been previously suggested [8], but the attack was not detailed. Our evaluation on the AES key recovery from the normal world OS aims to show the detailed channel in terms of time and rounds of encryption to recover the secret key from ARM TrustZone. 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.

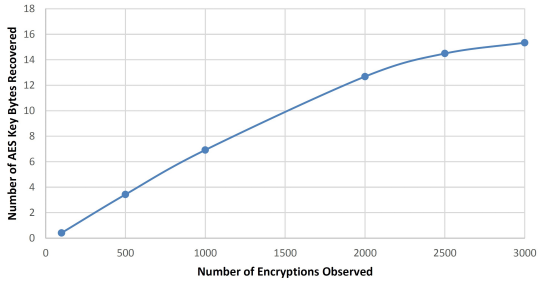

Fig. 5. Attack from Normal World OS

The effectiveness of the attack is shown in Figure 5. The x-axis shows the number of encryption 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 encryption required can be further reduced.

C. Key Recovery from Normal World User App

Normal world attack is evaluated in two aspect, the ability to obtain prime and probe memory without access physical memory mapping and the ability to recover key using the obtained memory.

1) Memory Acquisition - Statistical Matching: Due to lack of access to virtual-to-physical address mapping, we have to use the new statistical matching method to allocate the memory pages for probing the T-table access. Statistical matching in our prototype platform was illustrated in Figure. 3. However, the general feasibility of using statistical matching method to identify locality of arbitrary memory page within the processor cache should also be verified. Figure 6 shows the success rate of correctly identifying the page locality (offset) in the cache on our experimental platform. The x-axis shows the number of samples, and the y axis shows the correct percentage. With less than six rounds of encryption, it is possible to apply statistical matching to correctly identify if the 4KB memory page obtained resides in upper portion or lower portion of the L1 cache.

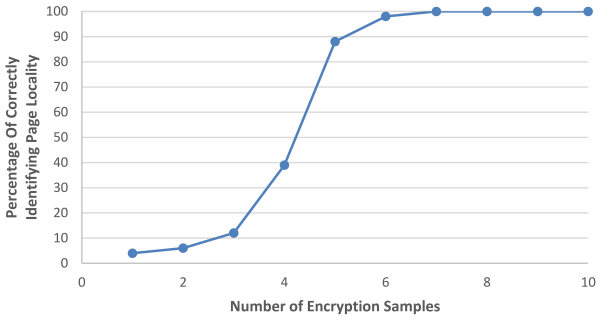

Fig. 6. Success Rate for Page Locality in Cache Identification

In this experiment, statistical matching is used to identify page locality in cache. This is possible, because we make the same assumption as previous works [14] that we can obtain the cache set offset of the memory area of the target. More specifically, we know the cache offset of the AES T-table in our experiment. However, if the library is loaded into the process with ASLR support in the trusted container [38], the offset will be unknown, because the target address is randomized. In such scenario, we test the possibility of using statistical matching to not only identify memory page cache locality, but also to identify the cache sets used by the target. We find that it is impossible to simply increase the number of samples to find memory that will cause contention with the target. We run the channel measurement function for more than five hours with around 200,000 rounds of encryption and fail to find any correlation. This is due to the cache hits introduced by making the library and system calls to read the performance event timer. To resolve this issue, we estimate the noise by recording the cache evictions in making a single system call, sys_read. The true cache contention is then estimated by subtracting off the background noise. After removing the noise, we are able to identify the relative target location in memory page showing high cache contentions with high confidence using 20000 rounds of encryption. We believe the rounds of encryption can be further reduced by exploiting more advance statistical methods, and we will investigate it as future work.

2) Key Recovery: 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 user space application is shown in Figure 7. The x-axis shows the number of encryption observed, and the y-axis shows the number of key bytes correctly guessed. It takes

approximately 9000 encryption 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. Most of the 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.

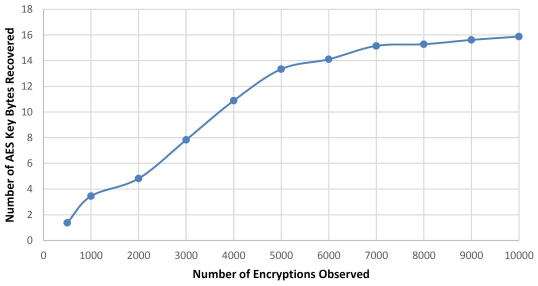

Fig. 7. Attack from Normal World Application

VI. Conclusion

In this work, we study cache-based timing side-channel information leakage in TrustZone, the trusted execution environment of ARM processors. Besides studying the information leakage from the normal world kernel perspective, we explore the possibility of launching side-channel attack on targets in TrustZone from a non-privileged user space application. TruSense can successfully extract the full AES encryption key from secure world with 3000 rounds of encryption in less than three seconds using the OS environment. Using new techniques including statistical matching and cache time loading estimation, the key recovery by user space application takes 9000 rounds and less than fourteen minutes. Though we only demonstrate the recovery of cryptographic keys from secure world, side-channel information leakage is widely applicable to many other libraries and applications. We hope that our finding in this paper provides a measurement of severity in secure container side-channel information leakage in ARM, and can fuel further research in the area of information leakage protection in hardware-assisted secure execution environments.

Acknowledgments

This work was supported in part by US National Science Foundation under grants CNS-1446478, CNS-1405747, and CNS-1443889. Dr. Kun Sun's work is supported by U.S. Office of Naval Research under grants N00014-16-1-3214 and N00014-16-1-3216. Dr. Deborah Shandss and Dr. Wenjing Lou's work was supported by (while they were serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution.," in *HASP, ISCA*, p. 10, 2013.
[2] "Building a Secure System using TrustZone Technology," apr 2009.
[3] "Arm markets." https://www.arm.com/markets/mobile.
[4] Samsung, "Samsung knox." https://www.samsungknox.com/en, 2015.
[5] D. Rosenberg, "Reflections on trusting trustzone," 2014.
[6] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *IEEE S&P*, 2016.
[7] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for ARM," in *ACM CCS*, pp. 558–569, 2014.
[8] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Sec*, 2016.
[9] R. Guanciale et al, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *IEEE S&P*, 2016.
[10] P. Kocher et al., "Differential power analysis," in *CRYPTO'99*.
[11] 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.
[12] D. Genkin, A. Shamir, and E. Tromer, "Rsa key extraction via low-bandwidth acoustic cryptanalysis," in *CRYPTO 2014*.
[13] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO'96*.
[14] 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 *IEEE S&P*, 2015.
[15] Y. Yarom and K. Falkner, "Flush + reload: a high resolution, low noise, l3 cache side-channel attack," in *USENIX Sec*, 2014.
[16] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *CT-RSA*, 2006.
[17] T. Ristenpart et al., "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM CCS*, 2009.
[18] D. J. Bernstein, "Cache-timing attacks on aes." https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.
[19] Y. Xu et al., "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE S&P*, 2015.
[20] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *NDSS*, 2017.
[21] "Openssl usage statistics." https://goo.gl/PK6PU9.
[22] D. C. Latham, "Department of defense trusted computer system evaluation criteria," *Department of Defense*, 1986.
[23] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, 2010.
[24] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *IEEE S&P*, 2011.
[25] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE S&P*, 2015.
[26] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *ACM CCS*, 2010.
[27] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *USENIX Sec*, 2015.
[28] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, 2016.
[29] O. Acıiçmez and Ç. K. Koç, "Trace-driven cache attacks on aes (short paper)," in *ICS*, 2006.
[30] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2013.
[31] *ARM Architecture Reference Manual ARMv7*, 2011.
[32] X. Zhang et al., "Return-oriented flush-reload side channels on arm and their implications for android security," in *ACM CCS*, 2016.
[33] N. Zhang et al., "Cachekit: Evading memory introspection using cache incoherence," in *IEEE EuroS&P*, 2016.
[34] G. Kroah-Hartman, "Linux archive - patch 3.14 00.79." http://lwn.net/Articles/637892/.
[35] N. Zhang et al., "Truspy: Cache side-channel information leakage from the secure world on arm devices," *IACR Crypto ePrint Archive*, 2016.
[36] witekio.com, "Reference bsps for freescale i.mx53 quick start board."
[37] E. A. Young, T. J. Hudson, and R. Engelschall, "Openssl: The open source toolkit for ssl/tls," 2011.
[38] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs," in *NDSS*, 2017.