SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors

Shengye Wan, Jianhua Sun
College of William and Mary
Williamsburg, VA
{swan, jsun01}.email.wm.edu

Kun Sun
George Mason University
Fairfax, VA
ksun3@gmu.edu

Ning Zhang
Washington University
St. Louis, MO
zhang.ning@wustl.edu

Qi Li
Tsinghua University
Beijing, China
qi.li@sz.tsinghua.edu.cn

***Abstract*—On ARM processors with TrustZone security extension, asynchronous introspection mechanisms have been developed in the secure world to detect security policy violations in the normal world. These mechanisms provide security protection via passively checking the normal world snapshot. However, since previous secure world checking solutions require to suspend the entire rich OS, asynchronous introspection has not been widely adopted in the real world.**

Given a multi-core ARM system that can execute the two worlds simultaneously on different cores, secure world introspection can check the rich OS without suspension. However, we identify a new normal-world evasion attack that can defeat the asynchronous introspection by removing the attacking traces in parallel from one core when the security checking is performing on another core. We perform a systematic study on this attack and present its efficiency against existing asynchronous introspection mechanisms. As the countermeasure, we propose a secure and trustworthy asynchronous introspection mechanism called SATIN, which can efficiently detect the evasion attacks by increasing the attackers' evasion time cost and decreasing the defender's execution time under a safe limit. We implement a prototype on an ARM development board and the experimental results show that SATIN can effectively prevent evasion attacks on multi-core systems with a minor system overhead.

***Index Terms*—Asynchronous Introspection, Evasion Attack, Trusted Execution Environment**

I. Introduction

Introspection mechanisms have been developed and deployed in a high privileged execution environment to prevent or detect security policy violations in a low privileged execution environment on the host machine [20]. In general, introspection mechanisms can be classified into two categories: *synchronous introspection* for attack prevention [7], [11], [12], [15], [16], [36], [37] and *asynchronous introspection* for attack detection [8], [14], [33], [37], [43], [48]. ARM TrustZone technology is a system-wide security mechanism to provide hardware-level isolation between two execution worlds that share the CPU in a time-sliced fashion, where the secure world has a higher privilege to access the system resources of the normal world such as memory, CPU registers, and peripherals, but not vice versa. To enhance the security of mobile devices, a number of TrustZone-assisted introspection mechanisms have been developed and deployed on millions of mobile devices [7], [11], [12], [37], [43].

Synchronous introspection mechanisms focus on intercepting and mediating security sensitive operations inline by the high privileged execution environment to prevent security policy violations in the low privileged execution environment. For instance, synchronous mechanisms have been developed in the virtual machine manager to ensure memory page protection in virtual machines [15], [16], [36]. Similarly, Samsung's KNOX Real-time Kernel Protection (RKP) mechanism [7], [37] relies on ARM TrustZone technique to intercept certain privileged system functions in the normal world and screen them through the secure world for inspection and approval before being executed.

However, synchronous introspection mechanisms face two main challenges. First, it has to hook up to all security sensitive locations that are potentially exploitable to attackers. Though it is possible to build up a near-complete list based on recently discovered policy violations, it is hard to ensure the completeness of such list. Second, certain implementation bugs, such as write-what-where, allows an attacker to launch *data attacks* bypassing the function checkpoints setup for the synchronous introspection [26], [35]. Once an attacker discovers any vulnerability of synchronous introspection, she can deploy a persistent rootkit to maintain the root access to the normal world OS (rich OS), steal data or mislead user behaviors without being detected by synchronous introspection.

Asynchronous introspection mechanisms can effectively detect those persistent rootkits via analyzing attacking traces of security policy violations from a snapshot of memory along with CPU state information that is periodically or randomly acquired from the low privileged execution environment (e.g. the normal world). Besides simply checking the integrity of the invariant kernel code, a number of proof of concept approaches have been developed to provide a more fine-grained security checking on dynamic kernel data structures after filling the semantic gaps [8], [14], [33], [48]. Unlike the synchronous introspection that requires to intercept all read/write transactions on the target, asynchronous introspection conduct the introspection based on the snapshot of the target, which makes it more effective to introspect the target completely and therefore detect a persistent attack.

One major limitation on applying asynchronous introspection mechanism in practice is that the introspection process may introduce a large system overhead. Particularly, on single core ARM processors, whenever the secure world is performing the security checking, the entire rich OS will be

suspended during the memory acquisition and online memory analysis process. Due to this poor usage experience on mobile devices, TrustZone-based asynchronous introspection has not been widely deployed or enabled.

Modern multi-core ARM processors creates new opportunities to deploy a practical asynchronous introspection based on TrustZone without pausing the rich OS. Specifically, the ARM multi-core architecture allows each core to enter its secure world independently, so the rich OS and the secure OS can run in parallel [9], [23], [28]. It is now feasible to make one core or all cores taking turns to perform the asynchronous introspection tasks while leaving other cores to continue the normal world's operations. For example, Samsung KNOX includes a Periodic Kernel Measurement (PKM) mechanism in the secure world to perform periodic asynchronous introspection on a specific core [37].

In this paper, we reveal a new type of evasion attack that can defeat the asynchronous introspection on multi-core systems by removing the attacking traces concurrently from one core while the security checking is executing on another. Evasion attacks target at defeating asynchronous introspection by predicting precisely the time of next security check and thus removing all attacking evidence to avoid detection [37], [48]. However, on multi-core mobile devices that can run both normal world and secure world concurrently, besides removing the attacking traces before security check, an attacker can also hide its attacking trace right after the start of introspection but before it has the opportunity to examine any malicious bytes. We name this type of evasion attacks as TZ-Evader.

There are two main challenges to be solved when designing a TZ-Evader attack. First, the malicious code running in the normal world needs to know if the asynchronous introspection is running on any core's secure world; however, the ARM TrustZone architecture protects the secure world running information from being accessed by the normal world. To solve this challenge, we propose to utilize the CPU core's availability as the side channel information to decide if the introspection is running on any core. We develop a user-level prober to stealthily probe the current state of each core. Second, when one core enters the secure world and begins to run the inspection, the malicious normal world needs to detect the core's state changes at an earliest time in order to maximize its evasion capability. To solve this challenge, we propose a kernel-level prober that can accurately monitor the running state changes of all cores. There are two implementation options for deploying the kernel-level prober, either by intercepting the timer interrupt to inject the prober in the rich OS or by manipulating the real-time scheduler of the Linux kernel to add the prober as a high priority process.

We implement a proof-of-concept TZ-Evader attack by integrating the kernel-level prober with traditional persistent rootkit on the ARM Juno r1 development board [5]. We evaluate its effectiveness against the state-of-the-art asynchronous introspection mechanisms, and the experimental results show the new TZ-Evader attack can accurately detect the running of asynchronous introspection and thus conduct a successful evasion attack.

With a deep understanding of the TZ-Evader attack, we propose a secure and trustworthy asynchronous introspection solution called SATIN in the secure world to defeat the TZ-Evader attack. The basic idea is to minimize the running time of each introspection and maximize the probing delay of TZ-Evader at the same time. We propose a number of techniques including *random wake-up time*, *random introspection area*, and *random CPU affinity* to ensure that the asynchronous introspection is always completed before TZ-Evader can hide any attacking traces. We implement a prototype of SATIN on the ARM Juno r1 development board and the experimental results show that it can effectively detect the TZ-Evader attacks with a minor system overhead.

In summary, we make the following contributions.

1) we discover a new evasion attack called TZ-Evader against asynchronous inspection on multi-core ARM processors. The attack utilizes the side channel information to infer if any core is running in the secure world and then begins to clean the attacking traces simultaneously on other cores that run in the normal world.
2) We develop a high-accurate probing technique called KProber for the normal world to fast probe the running state of all cores. Based on KProber, we implement a proof-of-concept TZ-Evader, which can defeat existing TrustZone-Based asynchronous introspection mechanisms.
3) We propose a secure and trustworthy asynchronous introspection mechanism called SATIN to protect mobile devices against TZ-Evader. It wins the race condition over the attacker by minimizing the running time of each introspection round and maximizing the probing delay of TZ-Evader.

II. Background

A. ARMv8-A Security Model

The ARMv8-A architecture is the latest 64/32-bit ARM architecture, which supports execution instructions with 64-bit registers and remains backward compatible with the 32-bit ARMv7 architecture. In the AArch64 security model, each instruction is executed at one of the four Exception Levels: *EL0*, *EL1*, *EL2*, and *EL3*, as shown in Figure 1. EL3 is the highest privilege level that only contains a Secure Monitor for controlling the context switch between the secure world and the normal world. In the normal world, the user applications run at EL0, the guest OSes run at EL1, and the hypervisor runs at EL2. In the secure world, the secure applications run in the S-EL0 level, and the secure OS runs in the S-EL1 level. There is no S-EL2 level, so the secure world does not support a hypervisor layer. The asynchronous introspection module can be implemented at either S-EL1 or S-EL0 level.

B. Preemptive/Non-preemptive Secure Mode

The ARM interrupt management framework is responsible for configuring the interrupt routing behavior [3]. There are

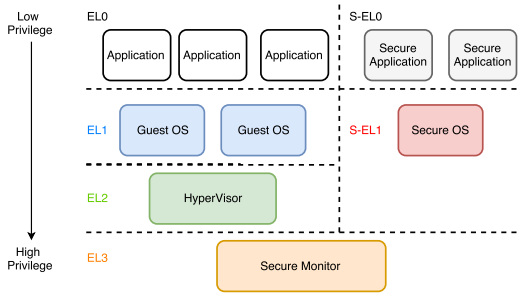

Fig. 1: ARMv8-A Security Model [1]

two generic requirements. First, it should be guaranteed to route secure interrupts to be handled by the secure world, even when the current execution is in the normal world. Thus, it protects secure interrupts against potential intervention from non-secure software. Second, it should be able to route the non-secure interrupts to the normal world when current execution is in the secure world. When the non-secure interrupt is configured to be routed to EL3, the secure monitor in EL3 can save the state of software in secure world before handing the interrupt to non-secure software. In this case, the secure world is *preemptive*. When the non-secure interrupt is configured to be routed to the S-EL1 or S-EL0, the secure software can either hand the interrupt to the non-secure software in a preemptive mode, or ignore the interrupt until its running task completes in a *non-preemptive* secure mode. OP-TEE OS [28] is an open-source secure operating system that supports preemptive secure world.

III. Evasion Attacks on Multi-Core Processors

A. Assumptions and Threat Model

We assume the secure world can be trusted and all the introspection components in the secure world are secure from attacks in the normal world. The asynchronous introspection can run randomly on any core at any time, and it cannot be intercepted by the normal world. We assume the asynchronous introspection does not suspend the rich OS on all cores; otherwise, it will face the same poor user experience problem as that on single-core processors. We assume the rich OS can be compromised and the attacker can bypass the existing synchronous introspection mechanisms to gain root privilege [26], [35] (see discussion in section VII-A). We assume the attack is an Advanced Persistent Threat (APT), which aims to maintain its presence on the target and makes various effort to remain undetected. For example, a key-logger may collect all user inputs on the keyboard by intercepting a system interrupt, while the hijacking is detectable to the introspection. In this case, whenever the introspection is running, the key-logger should stop the attack and clean its attack trace to camouflage its existence; Meanwhile, for all the other time, it remains in the attacking phase.

B. New Attack Surface

On multi-core ARM processors, attackers may defeat the existing asynchronous introspection by satisfying two requirements. First, the malicious code in the normal world can detect if one core is entering the secure world. Second, before the core in the secure world can access the attacking traces, the malicious code running on other cores can remove the attacking traces.

1) Probing CPU Core's Running State: Since the normal world cannot directly access any secure world information, we propose to utilize the availability of the shared CPU cores as a side channel information to infer the running state of each core. The main idea is that after the secure world holds one core to perform the introspection, the normal world cannot use that core to run any process. A user-level prober process can be used to conduct this probing task. To trace when the normal world loses the control on a CPU core, the prober process assigns each core with a child-thread, which keeps reporting back the corresponding core's availability. Since the rich OS kernel may migrate one thread task to other cores, especially when one core is paused, we fix the CPU affinity of each thread. Thus, when one core enters the secure world, the attached thread will be paused and cannot be migrated to other cores by the OS scheduler. When one thread is paused, the prober process can detect that the corresponding core enters the secure world.

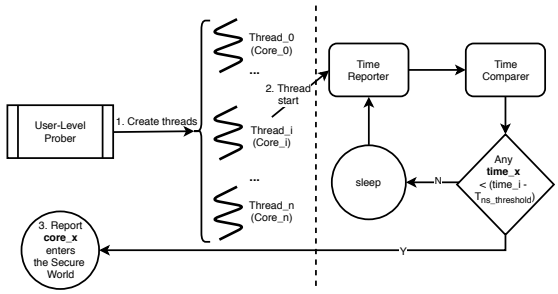

Fig. 2: User-level Multi-thread Prober

Figure 2 shows the multi-thread design of the user-level prober. For a device with n cores, we start a process with n threads, and each thread's CPU affinity is fixed to its corresponding core. Each thread has two components: *Time Reporter* and *Time Comparer*. On core i, the Time Reporter obtains the latest time $time\_i$ from a shared timer among all CPU cores and then reports the time into a buffer that is readable to all threads. After that, the Time Comparer compares core i's $time\_i$ with all other cores' latest reported times.

Since each thread reports its latest time independently, even if we can start the Time Reporters on all cores simultaneously, there exists a time difference when reading those reported time buffers and comparing their values. Meanwhile, since the kernel scheduler manages to provide the "fairness" to all threads, even though we cannot control all threads in a completely synchronized manner, each thread can be executed within a threshold, and the time differences between any two threads have an upper limit. We define this upper limit as $T_{ns\_threshold}$.

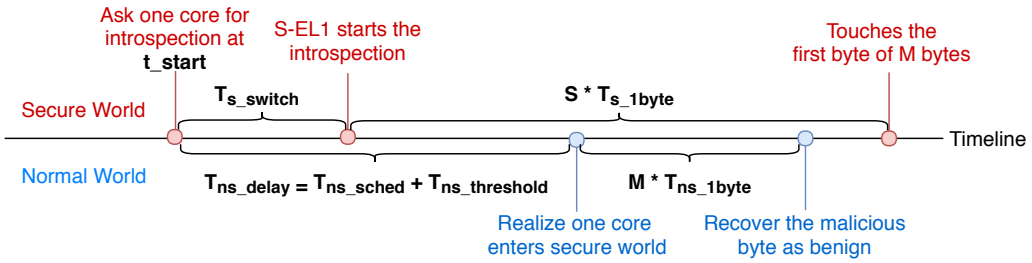

Fig. 3: Race Condition Between Two Worlds on Multi-Core System

If thread i finds any core x's latest $time\_x$ is earlier than its reporting time and the latency is larger than the threshold, namely, $time\_i > (time\_x + T_{ns\_threshold})$, then this thread reports that $core\_x$ switches to the secure world. To remain stealthy, we can afford to have the threads sleep for a time interval. We assume the threads can be scheduled with the average frequency f_{ns_sched}, so between any two consecutive rounds of Time Comparer, there is a delay $T_{ns\_sched} = 1/f_{ns\_sched}$. Finally, the prober needs time $T_{ns\_delay} = T_{ns\_sched} + T_{ns\_threshold}$ to detect that one core enters the secure world.

We implement and evaluate this user-level prober on the ARM Juno development board (see Section IV-A). When running one round of probing every 8 seconds, we found the T_{ns_delay} is smaller than 5.97×10^{-3} s while the average time for one core to conduct a kernel integrity check is 8.04×10^{-2} s. We can see the prober is capable of detecting the running of a typical TrustZone-based kernel integrity checking. Since each step of the prober requires no modification with OS kernel privilege, it is stealthy and useful to check if the secure world is ever entered.

2) Race Condition on Multi-core Processors: One state-of-the-art defense mechanism against evasion attacks is to trigger the security checking randomly. However, on multi-core systems, we observe that as long as the normal world is not frozen on all cores, there exist a race condition between the normal world to remove the attacking traces and the secure world to detect the attacking traces. We assume the introspection will check N bytes of critical memory and the normal world needs to clean M bytes of attacking traces.

We formalize the race condition between the evasion attacks and the asynchronous introspection. We suppose the secure world requests to start the asynchronous introspection at the time moment t_start. To switch one core to the secure world for conducting introspection, the ARM processor needs to save the normal world context of this core and then switch to the secure world. We define this context switching period as T_{s_switch}. Thus, the moment that the secure world starts the integrity checking can be as early as $t\_start + T_{s\_switch}$. We define the time for the secure world to inspect one byte as T_{s_1byte}. Since the secure world assumes all N bytes may be vulnerable while it does not know exactly what bytes are modified by the attacker, it may check S bytes before it touches any malicious byte where $0 \leq S \leq N-M$. Therefore, the introspection cannot find any malicious bytes before the time moment $t\_start + T_{s\_switch} + S * T_{s\_1byte}$.

From the attacker's point of view, since she wants to attack the system with the longest duration (e.g., to collect inputted password), she would only stop the attack and hide the attacking trace if she believes there is an introspection. If it can predict the t_start, it can easily hide before the introspection starts. When the t_start is unknown or unpredictable, the attacker need to detect it by probing the secure world's state. We define the probing delay for the attacker to notice one core entering the secure world as T_{ns_delay}. Assume the time for the attacker to recover one malicious byte is T_{ns_1byte}. The total recovering time for the attacker is $T_{ns\_recover} = M * T_{ns\_1byte}$. Then after the moment $t\_start + T_{ns\_delay} + T_{ns\_recover}$, the secure world cannot detect the malicious modification. The timing relationship between the two worlds is shown in Figure 3. Finally, for the attacker that is able to escape from the introspection, the condition 1 should be satisfied :

$$(T_{s\_switch} + S * T_{s\_1byte}) > (T_{ns\_delay} + T_{ns\_recover}) \quad (1)$$

We can see that the key element for evasion attacks to win the race condition is to minimize the probing delay T_{ns_delay}. Though the user-level prober shows the capability of understanding the secure world state stealthily, its probing stability and accuracy cannot be guaranteed. Particularly, when one core is scheduled with several threads that have the same or higher schedule priority than the probing thread, the prober's T_{ns_sched} is increased. To improve the probing accuracy, we should minimize the detection delay. In the next section, we present a new evasion attack called TZ-Evader based on a more accurate kernel-level prober.

C. TZ-Evader

TZ-Evader includes two components: a kernel-level attack module that conducts the persistent attack on the OS and removes the attacking trace accordingly; a fine-grained kernel-level prober module that can accurately and reliably monitor the running state of each core. The kernel-level prober also contains two components Time Reporter and Time Comparer, the same as the user-level prober. After obtaining the root privilege in rich OS, the prober module is loaded. If the prober can receive the time reported from all CPU cores, TZ-Evader believes there is no introspection in the secure world and keeps the attack active on the rich OS. Once the prober module

reports that one core may be switched to the secure world, TZ-Evader begins to remove its attacking trace. We provide two implementation options for the kernel-level prober. The first option KProber-I is based on intercepting the timer interrupt to inject the prober in the rich OS,and the second option KProber-II works by manipulating the real-time scheduler of the Linux kernel to add the prober as a high priority process to be scheduled.

1) KProber-I: On ARM processors, each core has its own timers to generate time interrupts. The Time Reporter and Time Comparer are injected into the normal world timer interrupt handler, so as to ensure the prober being executed with the same frequency as the timer interrupts. After this hijacking, for any incoming timer interrupt to core i, the interrupt handler updates the $time\_i$ into its corresponding $buffer\_i$ and compares it with other $n-1$ cores' time reports before resuming the normal timer interrupt handler. Linux kernel is typically configured as the $CONFIG\_NO\_HZ\_IDLE$ mode, which means when the core is not in the IDLE state, the per-core timer raises the timer interrupt for scheduling-clock ticks periodically with the frequency of HZ. For most versions of the Linux kernel, $100 \leq HZ \leq 1000$ [13]. To avoid any core entering the idle mode, KProber-I keeps running a user-level multi-threads program on each core. KProber-I can guarantee to work with a frequency no less than HZ on any core, no matter how many tasks are running on that CPU core. Though this implementation option can achieve the highest time accuracy from the rich OS perspective, it requires to modify the timer interrupt handler, which may introduce extra attacking trace for the defender to detect. In section III-C2, we present another implementation without modifying any kernel static area. Moreover, since there are many potentially unknown mechanisms to manipulate the handler, the defender has to scan the entire kernel for detecting all potential preparation traces, which gives KProber-I a larger chance to be recovered as we evaluated in Section IV-C.

2) KProber-II: This prober utilizes the Linux's real-time (RT) scheduler to ensure a reliable execution of Time Reporter and Time Comparer. According to the Linux kernel design, RT scheduler has higher scheduling priority than the default Linux CFS scheduler, which is responsible for scheduling most of Linux application threads. Meanwhile, RT scheduler can be used to schedule tasks with higher priority. Therefore, by setting the prober with the highest priority of RT scheduler, KProber-II can protect the reliable execution of Time Reporter and Time Comparer from being affected by either CFS-scheduled threads or low priority RT-scheduled threads.

Theoretically speaking, the timer-interrupt based prober is more stable than the RT scheduler based prober, since the frequency of the RT scheduler relies on the timer interrupt. However, injecting a prober into the interrupt handler demands more engineering efforts than simply increasing the priority of the attacking threads using the real-time scheduler. We present more implementation details in Section IV-A1.

IV. TZ-Evader Implementation and Evaluation

A. TZ-Evader Implementation

We develop a prototype of TZ-Evader on ARM Juno r1 development board [4], which is featured with the ARM big.LITTLE technology that consists of a 4-core Cortex-A53 "LITTLE" processor for maximum power efficiency and a 2-core Cortex-A57 "big" processor to achieve maximum computation performance. The secure monitor running in EL3 is provided by ARM trusted firmware (ARM-TF), and the secure world OS running in S-EL1 is modified based on the Test Secure Payload (TSP) of ARM-TF [3]. We modify the secure timer interrupt handler in the TSP to perform the integrity check over the normal world. The normal world runs OpenEmbedded LAMP OS with kernel version lsk-4.4-armlt in EL1, which is downloaded using the script from Juno Wiki of ARM Community [5].

1) Kernel-Level Prober Implementation: We deploy two types of KProber to probe a specific core or a randomly chosen core. To probe a specific core's running state, we fix one thread of Time Reporter on the targeted core and fix another thread containing Time Reporter and Time Comparer on another core. To probe a random CPU core, we assign each core with one thread that contains Time Reporter and Time Comparer.

To implement the timer interrupt based KProber-I, one key technical issue is to hijack the time interrupt handler. In ARMv8-A architecture, the address of the original timer interrupt address is saved in the *IRQ Exception Vector*, which can be located in the AArch64 Exception Vector Table [2]. The table's starting address is saved in the Vector Based Address Registers $VBAR\_ELi(1 \leq i \leq 3)$. After locating the timer interrupt, we modify its corresponding table entry to redirect it to our hijacking code.

For the real-time scheduler based KProber-II, we use the function $pthread\_setschedparam()$ to schedule the targeted threads with the real-time scheduler. We use the rt-scheduler $SCHED\_FIFO$ with the priority parameter $sched\_get\_priority\_max(SCHED\_FIFO)$ for all KProber-II's threads. After investigating the relationship between thread sleeping and CPU utilization, we set the sleep time $T_{sleep} = 2 \times 10^{-4}$ s and we assume the $T_{ns\_sched} = T_{sleep}$. In the following experiments, we implement Time Reporter with KProber-I and Time Comparer with KProber-II to demonstrate that both techniques can achieve reliable probing results.

2) Sample Kernel-Level Attack: To facilitate the evaluation of TZ-Evader, we implement a kernel-level attack that can hijack the GETTID system call. Successful system hijacking requires modifying an entry of the system call table, and this attack modifies one 8-bytes address of the system call table. Since the system call table is defined as text kernel data, TrustZone-based introspection can detect the GETTID system call is hijacked if the introspection scans and detects any of these 8 bytes is modified. Note there are many other kernel level attacking vectors, we just use GETTID hijacking attack as an example to study the evasion attacks.

B. Attack Evaluation

1) Introspection Time Delay: We first evaluate the time delay of the introspection. As we mentioned in the Equation 1, TrustZone-based asynchronous introspection suffers two major delays: T_{s_switch} and $s * T_{s\_1byte}$. To evaluate T_{s_switch}, we execute the context switching function of Test Secure Payload Dispatcher 50 times on one A53 core and one A57 core. The result shows for a secure timer interrupt raised at t_start, the time for the dispatcher to pause the normal world and jump to the related timer interrupt on the A53 core or A57 core are similar, ranging from 2.38×10^{-6} s to 3.60×10^{-6} s.

Then we evaluate T_{s_1byte} regarding two different introspection techniques. Traditional hardware-assisted asynchronous kernel introspection takes a snapshot of the kernel [47], [48] and then analyzes the memory copy. Since this copy remains inaccessible by the attacker, the analysis steps after taking the snapshot are not vulnerable to the TOCTTOU attack. Meanwhile, since the secure world and the normal world share the system hardware, TrustZone-based introspection can directly read the normal world OS' kernel from the secure world. After reading the kernel data, it can hash the data and compare the hash value to a pre-calculated authorized value. In our experiment, we measure the time for the secure world to take the snapshot and hash the kernel data. We use *djb2* [31] as the hash function. Each measurement is repeated 50 times. Table I shows that directly hashing the kernel's memory is more efficient than capturing and hashing the snapshot. In addition, it consumes less memory than the snapshot approach. Therefore, directly hashing the memory is better than taking snapshot when the asynchronous introspection targets at the static kernel area. We also find that it takes less time to conduct the introspection on the A57 core than the A53 core, since A57 core is more powerful than the A53 core.

TABLE I: Secure World Introspection Time

Core-Time	Hash 1-Byte	Snapshot 1-byte
A53-Average	1.07×10^{-8} s	1.08×10^{-8} s
A53-Max	1.14×10^{-8} s	1.57×10^{-8} s
A53-Min	9.23×10^{-9} s	9.24×10^{-9} s
A57-Average	6.71×10^{-9} s	6.75×10^{-9} s
A57-Max	7.50×10^{-9} s	7.83×10^{-9} s
A57-Min	6.67×10^{-9} s	6.67×10^{-9} s

2) Attack Time Delay: We evaluate normal world attack time delay in two aspects, where $T_{ns\_recover}$ is introduced by the the kernel-level attack module, and $T_{ns\_threshold}$ is introduced by the prober module. We repeat the measurement of the recovery time $T_{ns\_recover}$ 50 times on one A53 core and one A57 core. For the A53 core, the average recovering time is 5.80×10^{-3} s. For the A57 core, the average recovering time is 4.96×10^{-3} s.

Then we present the prober's time delay $T_{ns\_threshold}$ when KProber is probing all cores simultaneously. As the prober execution involves all available cores, we present the prober's time delay $T_{ns\_threshold}$ regardless of core types. To observe the variation of the threshold, we execute the KProber with different probing periods. For each probing period, we choose the largest difference calculated by the Time Comparer as the threshold, and we repeat the measurement 50 times. We present the average threshold, maximum threshold, and minimum threshold of the 50 rounds for each time period in Table II.

TABLE II: Probing Threshold on Multi-Core

Probing Period	Average	Max	Min
8 s	2.61×10^{-4} s	7.76×10^{-4} s	1.07×10^{-4} s
16 s	3.54×10^{-4} s	1.38×10^{-3} s	1.31×10^{-4} s
30 s	4.21×10^{-4} s	8.99×10^{-4} s	2.59×10^{-4} s
120 s	5.26×10^{-4} s	9.49×10^{-4} s	3.18×10^{-4} s
300 s	6.61×10^{-4} s	1.77×10^{-3} s	4.18×10^{-4} s

Based on the experiment results, we find that the average threshold becomes larger along with a longer probing period and the maximum threshold is around 1.8×10^{-3} s. To further understand the variation of the threshold, we investigate the reported time of each thread and identify that, in some rare cases, Time Comparer on $core\_i$ may get the $time\_x$ of the $core\_x$ with an abnormal large delay, which is up to 1.3×10^{-3} s. This cross-core reading delay leads to the large threshold. Meanwhile, a longer probing period increases the occurrence of those rare cases, so the average threshold increases too.

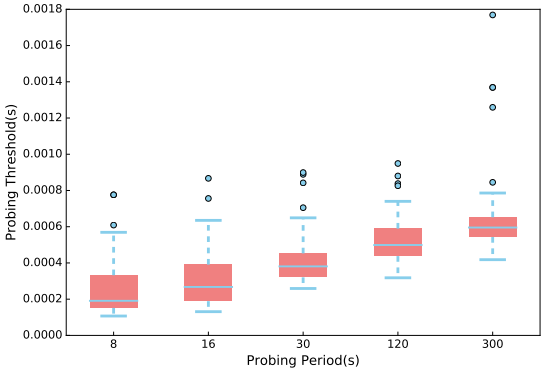

Fig. 4: KProber Probing Threshold Stability

To present the stability of KProber, we show the variation of the thresholds with different probing periods in Figure 4. We can see that even though the KProber's average probing threshold increases with the probing period, the upper whiskers of the thresholds only go up slightly, and only few extreme large outliers are introduced for probing period 300 s, which go over 1×10^{-3} s.

Finally, we also observe that setting introspection with a fixed CPU affinity is easier to be probed than using all cores randomly. As we mentioned in Section IV-A1, we also evaluate the case to use KProber for figuring out one single core's availability. We conduct the experiment with the same setting as presented above. According to our experiment result, the average thresholds to probe the single core only equal to $\sim 1/4$ of the presented threshold for probing all cores, for all five probing periods we evaluated. This means the more cores KProber needs to probe, the larger probing threshold it suffers.

C. Race Condition Analysis

Based on the experiment results, we have following observations on the race condition between the normal world and the secure world:

1) The switch delays T_{s_switch} for the A53 core and A57 core are similar.
2) The secure world can use more powerful CPU core (i.e., A57 core) for introspection to achieve a shorter T_{s_1byte}.
3) The secure world is easier to be probed if it's CPU affinity is predictable to the normal world.
4) The timing bottleneck of TZ-Evader is the time period for recovering its attacking trace $T_{ns\_recover}$.

According to Equation 1, we know the attack can finish its cleaning task after the secure world inspects S memory bytes, where:

$$S > \frac{T_{ns\_sched} + T_{ns\_threshold} + T_{ns\_recover} - T_{s\_switch}}{T_{s\_1byte}} \quad (2)$$

Now we consider the worst case for the TZ-Evader: the introspection starts on one A57 core while the TZ-Evader uses on one A53 core to remove its attacking trace. In Section IV-B1, we have $T_{s\_switch} \leq 3.60 \times 10^{-6}$ s. Also, the secure world can inspect the kernel data with the maximum speed $T_{s\_1byte} = 6.67 \times 10^{-9}$ s. The attacker recovers its attacking trace as the lowest efficiency $T_{ns\_recover} = 6.13 \times 10^{-3}$ s, and $T_{ns\_sched} = 2 \times 10^{-4}$ s. In Section IV-B2, we have for the worst case of TZ-Evader, $T_{ns\_threshold} \leq 1.80 \times 10^{-3}$ s. By applying the experimental data into the above Equation 2, we have $S \leq 1218351$ *bytes*. Supposing the attack bytes appears randomly in the kernel and the introspection checks the entire OS kernel whose size is 11916240 *bytes*, nearly $1 - \frac{1218351}{11916240} \approx 90\%$ of the kernel space is not protected by existing TrustZone-based asynchronous introspection. Therefore, as long as both kernel attack module and prober module do not locate in the 10% scanned area, the system suffers TZ-Evader attack.

V. SATIN: Secure Asynchronous Introspection

We propose a secure and trustworthy TrustZone-based asynchronous introspection mechanism named SATIN that can practically inspect rich OS and effectively defeat TZ-Evader attacks. We first discuss two major challenges on developing the secure asynchronous introspection and then present the detailed design of the proposed mechanism. Figure 5 shows the architecture of SATIN, which is implemented in the secure world. It consists of two major components, where the integrity checking module performs the integrity checking on rich OS using a divide-and-conquer method to control the checking time of each round and the self activation module is responsible for waking up the secure world with the help of a secure timer and ensuring that the entrance cannot be either predicted or quickly probed by the normal world.

A. Asynchronous Introspection Challenges

We identify two main challenges on developing secure asynchronous introspection mechanisms against TZ-Evader.

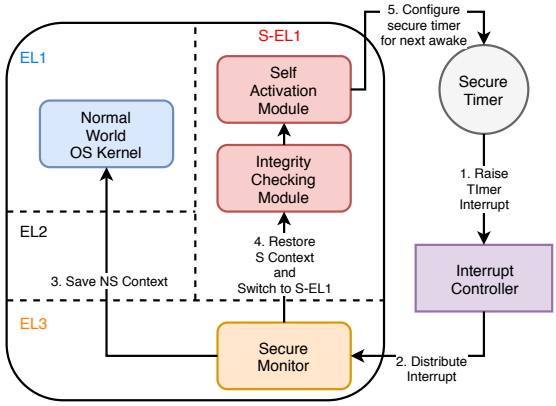

Fig. 5: SATIN Architecture

1) Challenge 1: Performance vs. Detection Accuracy: On single core processors, the TrustZone-based introspection introduces unacceptable overhead as the introspection has to suspend the rich OS during the security checking process. The pausing issue leads to poor user experience. For example, a music song played in the normal world becomes incoherent when the CPU enters the secure world. Similarly, on the multi-core processors, we cannot frequently freeze all cores even for a small period of time. For instance, Azab et al. [8] propose an introspection mechanism that requires to freeze all CPU cores on the server every 8 or 16 seconds.It becomes one of the biggest obstacles for asynchronous introspection mechanisms to be widely adopted and deployed, though a better detection accuracy can be achieved by freezing all cores. Meanwhile, even a multi-core platform can execute the introspection and other tasks simultaneously, the execution of the secure world still possesses some shared resources such as the CPU core. It is not well studied how much overhead is introduced by running the introspection in parallel to the rich OS.

2) Challenge 2: Evasion Attack: It is a challenge to defend against evasion attack when designing asynchronous introspection mechanisms [20]. If the attacker can escape ahead of the introspection by predicting or probing the execution of the asynchronous introspection, then the introspection result cannot be trusted [43], [45]. On single core processors, random checking is an effective scheme to defeat evasion attacks. However, on multi-core ARM processors, TZ-Evader can even escape from the random checking on any random core. It is critical to develop a secure asynchronous introspection mechanism to defeat the new evasion attacks.

B. Integrity Checking Module

To improve the detection rate, we propose to reduce the introspection time for each round by dividing the entire OS kernel into smaller areas and taking turns to check one area in each round. Therefore, it can guarantee to finish one round of security checking right before the malicious normal world can probe it but have not chance to remove any attacking traces. The integrity checking module prepares the hash value of each small area's benign status during booting stage. Then for each

round of wake-up, it scans one small area and compares the hash value with the pre-calculated benign one. If the integrity checking module finds any abnormal small area, it can raise an alarm to the server side or the device user. To ensure that the malware cannot remove its traces before we finish checking on one small area, the size of each small area should be smaller than $(T_{ns\_delay} + T_{ns\_recover} - T_{s\_switch})/T_{s\_1byte}$ bytes. We develop a pseudo-random method to select the next small area for introspection. Suppose the set $set_{area} = \bigcup_{i=0}^{m-1} area\_i$ contains all the areas of the OS kernel, and each $area\_i$ satisfies the above size condition. When the secure world starts one round of the introspection, the module randomly picks one area $area\_x$ from set_{area} and then applies $set_{area} = set_{area} - area\_x$. If $set_{area} == NULL$, then SATIN resets $set_{area} = \{area\_0,\ ...,\ area\_(m-1)\}$.

Also, the integrity checking module needs to guarantee its execution is not interrupted by other non-secure parties. According to the latest ARM interrupt routing model [3], the normal world interrupt signal is possible to interrupt the execution of secure world. To prevent the normal world from using interrupts to interfere in the introspection process, SATIN needs to block all interrupts during each round of the introspection. We propose to deploy the integrity checking module into the secure timer interrupt handler, and then set EL3 to not trap the normal world interrupt by configuring $SCR\_EL3.IRQ = 0$. With this setting, even if the normal world interrupt signal can arrives the secure world, their priorities are lower than the integrity checking module.

C. Self Activation Module

The self activation module is designed to make sure the secure world can invoke the integrity checking module anytime without the help of the normal world. It relies on a secure timer that the normal world cannot access. Each TrustZone-enabled core has an individual secure timer that can only be read or written with the secure world privilege. During the booting time, the self activation module is invoked once on each core to write the next awake time into the secure timer register.

After the trusted booting process, when the timer condition meets, the timer raises an interrupt for the secure world and the secure monitor switches the core from the normal world to the secure world to handle this interrupt. By configuring the secure timer, we can activate the secure world without involving the normal world. Thus, we can prevent the normal world from disturbing the invocation of introspection. When one core enters the secure world via the secure timer interrupt, SATIN first performs the introspection on one small area and then sets the awake time for the next round of introspection. The self activation module decides the next awake time by $time\_x$, which is set to a base period time t_p (e.g., 8s, 16s, etc.) plus a random deviation t_d (e.g., a random time from $-t_p$ to t_p). By applying the random deviation with the next awake time, the interval between two consecutive rounds of introspection is among $[0, 2 * t_p]$, which means at any moment the introspection could start to scan and the attacker has to keep probing all cores. In addition, the random deviation can

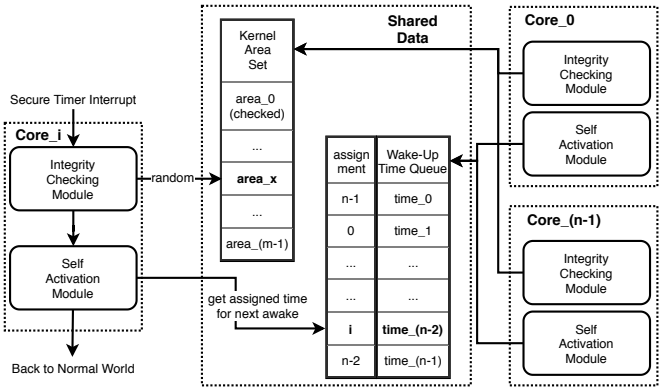

Fig. 6: Multi-Core Collaboration of SATIN

effectively minimize the exposure of any recognized patterns on the invocation of the asynchronous introspection to the normal OS. We set $t_p = T_{goal}/m$, where T_{goal} is the time period to guarantee that all the target areas can be scanned at least once.

D. Multi-Core Collaboration

To increase the checking accuracy, we propose to choose a random core for conducting the introspection task. This design choice is based on the observation that if only one core is used for asynchronous introspection, the malicious normal world can achieve a better probing accuracy than that when all cores are randomly chosen to conduct introspection, as mentioned in Section IV-B2.

Figure 6 illustrates the collaborative introspection of SATIN on the multi-core architecture. When any core i wakes up for the introspection, it randomly takes one kernel area from the shared Kernel Area Set set_{area} and inspects this area. Later, other cores are not going to inspect this area repeatedly since core i removes the area it chooses from the set. If there is no more area available, the set is refilled with all areas again. Next, core i obtains the next wake-up time from a wake-up time queue and configures it's secure timer accordingly, where the wake-up time queue is responsible to coordinate all cores that wake up in a random sequence.

Coordinating all cores to wake up in a random sequence is also a challenged task. ARMv8-A architecture does not provide a solution for one core to directly read or write the timer of another core. In this case, an intuitive design is to use the cross-core interrupt to notify all cores on serving the introspection in turn. ARMv8-A allows one core to generate a secure interrupt to forcibly switching another core into the secure world, so after core i finishes one round of introspection, it can switch another core j into secure world and then core j sets the secure timer for the next round introspection. However, the switch of core j can also be probed by the normal world so this method may leak the wake up sequence to the normal world, which can defeat the benefits from randomly waking up cores.

To protect the wake-up pattern from the normal world, SATIN does not apply the cross-core interrupt mechanism, and

instead coordinates all cores via the secure memory. SATIN stores the wake-up time of each core in the wake-up time queue and requires each wake-up core to check the queue to get next wake up time. For the devices with n cores, the wake-up time queue contains n wake-up time slots. Each time value is determined by the rule discussed in V-C. The initial wake-up sequence of all CPU cores is set during the trusted booting process. Then when any core enters the self activation module, it picks the next time based on a random assignment. Once all core extract their assigned slots from the queue, the self activation module refreshes the queue with n newly generated time values and newly generated random assignment. By flushing the wake-up time queue and re-assigning the time slots to each core, we can use all cores for kernel introspection while prevent the normal world from knowing the awake core or awake time for the next round of introspection. Finally, all cores can coordinate to make sure each core inspects a random memory area at a random wake-up time.

VI. SATIN Implementation and Evaluation

A. SATIN Implementation

To verify the effectiveness of SATIN on defeating TZ-Evader, we develop a prototype of SATIN on the ARM Juno r1 development board with the same configuration as mentioned in Section IV-A. We modify the Test Secure Payload in S-EL1 to implement the major modules of SATIN.

1) Self Activation Module: When the self activation module is called on one core, the module stops the secure timer by updating the register $CNTPS\_CTL\_EL1$. Then it takes the assigned wake-up time from the Wake-Up Time Queue and writes the value to comparer register $CNTPS\_CVAL\_EL1$. After that, the module restarts the secure timer for that core. For any core in the processor, when the shared physical timer counter register $CNTPCT\_EL0$ becomes equal to or greater than the per-core register $CNTPS\_CVAL\_EL1$, the core raises a secure timer interrupt.

2) Integrity Checking Module: As we calculated in Section IV-C, for each area of the checking module, its size must be smaller than 1218351 bytes. Also, we implement the integrity checking module to guarantee that each section of the normal world OS's $System.map$ only belongs to one area for introspection. Thus, we divide the normal world's kernel into 19 areas according to the System.map. Among these areas, the largest one contains 876616 bytes and the smallest one contains 431360 bytes. During the booting time, SATIN hashes these 19 areas and then saves these hash values into an authorized hash table stored in the secure world.

B. SATIN Evaluation

We evaluate the performance of SATIN in two aspects, namely, the effectiveness on defeating TZ-Evader and the triggered extra system overhead.

1) Defeating TZ-Evader: SATIN can effectively and securely defeat TZ-Evader attacks. First, the introspection module is deployed in the secure world, so we can protect the introspection module from being compromised by malware in the normal world. We assume the hardware-assisted TrustZone technique can be trusted to protect the secure world. Second, it can prevent malware from removing its traces *before* the invocation of each introspection, since the normal world cannot accurately predict or intercept the invocation of introspection operations. Third, it can detect malware that uses race condition to remove its traces *during* the introspection. Because we divide the entire large introspection area into smaller areas, we can finish the introspection of one small area even before the malware detects the entrance of one core into the secure world and then begins to remove the attacking trace. In addition, it is user-friendly. The introspection does not require to fully freeze the rich OS in the normal world. On multi-core processors, since not all cores are forced to enter the secure world at the same time, the rich OS can continue to run on the remaining cores when one core conducts the introspection on one core.

In our introspection mechanism, every m rounds of the introspection can guarantee scanning the entire OS kernel once and the average time between two rounds is t_p. Within the time period $m * (t_p) + \sum_{i=0}^{m-1} size_{area\_i} * T_{s\_1byte}$, it can successfully catch the malicious memory bytes within the checked areas. In our experiment, the entire time is approximately $152\,\mathrm{s}$.

To validate the detection results, we execute TZ-Evader in the normal world while running SATIN simultaneously in the secure world. We set the probing thresholds of KProber as $1.8 \times 10^{-3}\,\mathrm{s}$. TZ-Evader maliciously modifies one system call handler which resides in the $area\_14$ of the integrity checking module. SATIN conducts 190 rounds of introspection to examine the entire kernel 10 times. KProber can faithfully report all 190 rounds of introspection without any false negative or false positive. Among these rounds, SATIN checks $area\_14$ 10 times and correctly detects the hijacked handler all the time. The average time between two consecutive checks for $area\_14$ is $141s$. In the meanwhile, TZ-Evader attempts to attack during these 10 checks but all the recovery efforts fail since the memory cleaning occurs later than the introspection.

2) SATIN Overhead: We use UnixBench [41] to evaluate the performance overhead on normal world operations when enabling our TrustZone-based asynchronous introspection. Figure 7 shows the normalized performance degradation when we use the self activation module to wake up the secure world across all cores of the device compared to the case where the self activation module is not enabled.

Since our experiment platform consists of 6 cores (i.e., 4 A53 cores and 2 A57 cores), we measure the overhead using two sets of experiments: executing each benchmark program once (1-task) and invoking 6 copies of the same benchmark simultaneously (6-task). In general, activating the introspection incurs 0.711% and 0.848% performance degradation in the 1-task and 6-task cases, respectively. This is reasonable as there is an increasing chance for SATIN to interrupt the normal world when more cores are utilized simultaneously. We also notice that the two tasks `file copy 256B` and `context switching` experience the largest overhead: 3.556% and

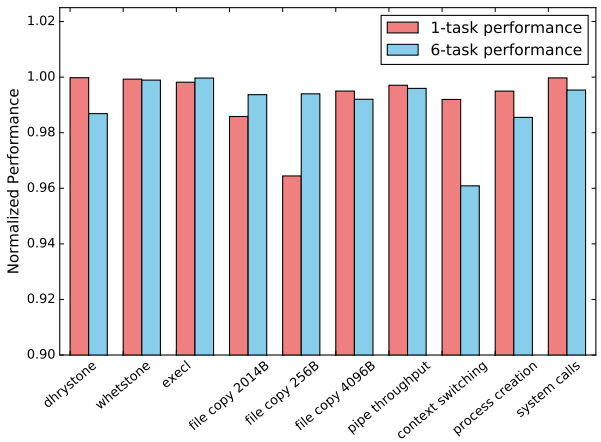

Fig. 7: SATIN Overhead

3.912%. The reason is that the test program happens to stay right at the random-selected core for the secure world more times than other cases. We believe this level of performance overhead is acceptable especially when the normal world is not suspended for even one nanosecond on multi-core systems.

C. SATIN Security Analysis

1) SATIN's Security: SATIN's security is based on the feature that its control flow cannot be affected by the attackers, which can be realized by utilizing TrustZone. Also, even considering recent TrustZone-related side-channel attacks [52], since they are focused on attacking TrustZones confidentiality instead of control flow, SATIN is secure.

2) SATIN Capability: SATIN is capable of detecting advanced persistent attacks that leave attacking traces during an extended period, even if they apply TZ-Evader to hide their traces. Similarly, SATIN can detect evasion attacks utilizing other side channels. Finally, SATIN can reduce attack efficiency and maximize the chances to detect APT attacks.

VII. Discussion

A. TZ-Evader under Synchronous Introspection Protection

To accurately probe the secure world state, TZ-Evader provides two implementation options, namely, (i) using the RT scheduler and (ii) modifying the address of the IRQ exception vector. Both options need to get the root privilege in the normal world, which may have been protected by certain synchronous introspection solutions. However, due to the incomplete hooking locations and implementation bugs [20], it is difficult, if not impossible, for the synchronous introspection to ensure that the malware cannot obtain the root privilege in the rich OS. For example, real attacks [26] have been discovered to gain the root privilege even if the normal OS kernel is being protected by real deployed synchronous introspection mechanism KNOX-RKP [7].

After getting the root privilege, the attacker can freely manipulate the RT scheduler, which has not been hooked or protected by any existing synchronous introspection mechanisms. Alternatively, with the root privilege, the attacker can modify the address of IRQ exception vector by launching the following data attack that cannot be prevented by the synchronous introspection [20]. The synchronous introspection method in both [7] and [17] set the vector table as non-writable, so any writing attempt to the exception vector will trigger a page fault and thus be trapped into the synchronous introspection. However, after getting the root privilege, the attack can utilize a write-what-where vulnerability [26] to change the Access Permissions (AP) bits of the related page table entry from non-writable to writable. After that, the attacker can freely modify the vector table without triggering the corresponding synchronous introspection.

B. TZ-Evader Limitation

In TZ-Evader, the probing threshold $T_{ns\_threshold}$ may vary on different ARM processors, and the attacker has to evaluate this value before attacking a targeted ARM processor. If the attacker has a device which has the same processor configuration as the attacking target, then the attacker can disable the secure world to get the $T_{ns\_threshold}$ value quickly and accurately. However, if the attacker is not able to evaluate the threshold on a fully controlled device, then $T_{ns\_threshold}$ needs to be learned from the victim directly. The attacker needs to run multi-threads Time Reporter and Time Comparer for a relatively long time (e.g., one hour) to study how the threshold varies. For each time the secure application is running, the attacker can observe the time difference among all cores. With the long-term study, the attacker can determine $T_{ns\_threshold}$ for the target device and then start the TZ-Evader.

C. Necessity of Asynchronous Introspection

Synchronous introspection protection is an effective defending mechanism; however, due to the high-performance overhead, incomplete hooking locations, and implementation bugs, it may still be circumvented [20], [26]. With a small execution overhead, asynchronous introspection provides one more layer of secure protection in addition to synchronous introspection. For example, Samsung TIMA deploys a synchronous introspection mechanism called Real-time Kernel Protection (RKP) in the hypervisor to protect virtual machines in the normal world and deploys an asynchronous introspection mechanism called Periodical Kernel Measurement (PKM) in TrustZone to protect the hypervisor [37]. Moreover, one main usability limitation of asynchronous introspection (i.e., rich OS suspension) on the single-core processors can be resolved on the multi-core processors by assigning one core on security checking and continuing normal operations of the normal world OS on other cores.

D. Portability of SATIN

SATIN architecture has three requirements, namely, multi-core processors, a high-privileged operating mode, and a secure timer. Since most modern processors support multi-core architecture, it is reasonable to allocate one core for performing the asynchronous introspection on most processors.

Therefore, besides ARM TrustZone, SATIN can be ported on other TEE architectures that can provide a secure timer and high-privilege for introspection.

VIII. Related Work

A. Asynchronous Introspection

Asynchronous introspection mechanisms [14], [15], [18], [22], [24], [27], [33], [34], [36], [48] have been popularly deployed to protect OS kernel integrity. OSck [18] executes a verifier process alongside the target kernel and periodically scans the memory to identify any policy violation. SigGraph [27] proposes to use the graph-based signature to scan the kernel data structure instance and detect the rootkits that are capable of manipulating the data structures. Specialized security tools have been constructed for running on a trusted virtual machine (VM) to detect any security violation on a target VM [15], [16], [36].

Zhang et al. [53] first propose the concept of using an isolated device as the integrity monitor. Then, Copilot [33] utilizes a PCI add-in card to periodically verify the hash checksum of the kernel static data. Later, several system management mode (SMM) based introspection mechanisms have been proposed [8], [24], [47], [48], where HyperCheck [48] and SPECTRE [47] employ the SMM to outsource the snapshot of the kernel to a remote server and conduct the introspection on the server side. HyperSentry [8] performs the kernel measurement locally by periodically triggering the host's SMM via an out-of-band channel. Among SMM-based security mechanisms, multi-core platforms are only briefly mentioned in [8] on freezing all cores during the SMM-based measurement task. The authors of HyperCheck [48] mention that it could be extended on multi-core processors; however, there is no detailed design about it. Several introspection mechanisms are proposed based on other hardware components which can check the kernel transparently [14], [42]. Ether [14] proposed an Intel-VT [19] based kernel analyzer to analyze the software within the virtual machine. LO-PHI [42] transparently examines the kernel memory snapshots without exposing any software-based artifacts by using additional hardware sensors and actuators.

B. Synchronous Introspection

A number of synchronous introspection mechanisms [7], [17], [25], [30], [32], [40], [46] have been proposed to work on different architectures too. On ARM processors, SPROBES [17] and TZ-RKP [7] are two TrustZone-based synchronous introspection mechanisms proposed recently. SPROBES [17] injects special code into the security-sensitive kernel handlers so it can dynamically check these handlers in the secure world and provide the real-time protection for the normal world. TZ-RKP [7] achieves a similar security goal but focuses on monitoring the data integrity and optimizing the rich OS's performance. Besides utilizing existing hardware-features of the ARM processor, customized hardware has been developed to snoop the memory bus and monitor the security-related writes to the kernel area [25], [29], [30].

C. Hardware-Assisted TEE

Nowadays more and more mechanisms have been proposed to provide a hardware-assisted trusted execution environment (TEE) on various hardware architectures [49]. Based on the ARM TrustZone technology, several works [21], [50]–[52] are proposed to investigate and enhance the security of the TrustZone secure world. Meanwhile, TrustZone has been utilized to enhance the security of applications running in the normal world against a malicious rich OS [12], [38], [44]. Santos et al. [38] propose to run the security-sensitive piece of the normal world .NET apps within the secure world. TrustICE [44] provides the solution to allocate the isolated environment for any normal world application, and Cho et al. [12] extend this idea for isolating both normal world application and the hypervisor. Besides the ARM hardware architecture, SICE [9] introduced the SMM-based isolated environment for x86 multi-core platforms. SICE can provide the remote attestation for the user to verify the integrity of the kernel within its isolated environment. Based on SMM, it is plausible to port our secure asynchronous introspection on X86 multi-core processors. Several works about the recent Intel hardware feature SGX [6], [10], [39] are also capable of measuring the integrity of the kernel running in the SGX enclave and providing remote attestation. However, since SGX enclaves are scheduled by the host OS, the SGX technique cannot be used to perform asynchronous introspection against the host OS.

IX. Conclusion

In this paper, we propose a trustworthy and practical TrustZone-based asynchronous introspection mechanism for ARM multi-core platform. We first show that on multi-core systems, even if the secure world uses a random core to inspect the rich OS kernel at random time point as previous asynchronous introspection solutions do, the malware in the normal world can still escape from the security checking by utilizing the race condition between the detector running on one core and the malicious evader running on other cores at the same time. We identify this new type of evasion attack as TZ-Evader and conduct a systematic study on it. We develop a proof-of-concept TZ-Evader attack that uses an accurate kernel-level prober to defeat the existing asynchronous introspection. Finally, we develop a secure TrustZone-based asynchronous introspection mechanism called SATIN on multi-core ARM processors to defeat the TZ-Evader attacks. We implement a prototype of SATIN on ARM Juno r1 development board and the experimental results show that SATIN can effectively prevent evasion attacks on multi-core systems with a minor overhead.

X. Acknowledgments

This work is partially supported by the U.S. ONR grant N00014-16-1-3214, ONR grant N00014-16-1-3216, ONR grant N00014-18-2893, NSFC grant 61572278, and NSFC grant U1736209.

REFERENCES

[1] ARM, "Programmer's guide for armv8-a," 2015, http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf.

[2] ——, "Arm exception table," 2018, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/CHDEEDDC.html.

[3] ——, "Arm trusted firmware," 2018, https://github.com/ARM-software/arm-trusted-firmware.

[4] ——, "Juno Arm Development Platform," 2018, https://developer.arm.com/products/system-design/development-boards/juno-development-board.

[5] ARM Community, "Arm linaro instruction," 2018, https://community.arm.com/dev-platforms/w/docs/303/juno.

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx." in *OSDI*, vol. 16, 2016, pp. 689–703.

[7] 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*. ACM, 2014, pp. 90–102.

[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[9] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388.

[10] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[11] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi, "Regulating arm trustzone devices in restricted spaces," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16, 2016, pp. 413–425.

[12] Y. Cho, J.-B. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices." in *USENIX Annual Technical Conference*, 2016, pp. 565–578.

[13] Corbet, "How fast should hz be?" 2005, https://lwn.net/Articles/145973/.

[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[15] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[16] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 586–600.

[17] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014.

[18] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 279–290.

[19] Intel, "Intel virtualization technology," 2018, https://www.intel.com/content/www/us/en/data-center/new-center-of-possibility.html.

[20] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 605–620.

[21] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment." in *NDSS*, 2015.

[22] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[23] M. A. Kinsy, S. Khadka, M. Isakov, and A. Farrukh, "Hermes: Secure heterogeneous multicore architecture design," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, May 2017, pp. 14–20.

[24] K. Leach, C. Spensky, W. Weimer, and F. Zhang, "Towards transparent introspection," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 248–259.

[25] H. Lee, H. Moon, I. Heo, D. Jang, J. Jang, K. Kim, Y. Paek, and B. Kang, "Ki-mon arm: A hardware-assisted event-triggered monitoring platform for mutable kernel object," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[26] Lev Aronsky, "Knoxout-bypassing samsung knox," 2016, http://media.wix.com/ugd/4e84e6_668d564cc447434a9a8fda3c13a63f6a.pdf.

[27] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures." in *Ndss*, 2011.

[28] Linaro, "Optee secure os," 2018, https://github.com/OP-TEE/optee_os.

[29] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, *Cpu transparent protection of os kernel and hypervisor integrity with programmable dram*. ACM, 2013.

[30] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 28–37.

[31] Ozan (oz) Yigit, "Hash functions," 2018, http://www.cse.yorku.ca/~oz/hash.html.

[32] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[33] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX Security Symposium*. San Diego, USA, 2004, pp. 179–194.

[34] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115.

[35] Project Zero, "Lifting the (hyper) visor: Bypassing samsung's real-time kernel protection," 2017, https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html.

[36] A. Saberi, Y. Fu, and Z. Lin, "Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[37] Samsung Electronics Co. Ltd., "White paper: An overview of the samsung knox platform," https://kp-cdn.samsungknox.com/6ee7dbf222f5eabeafea9d15e3986f09.pdf.

[38] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 67–80, 2014.

[39] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 335–350.

[41] B. Smith, R. Grehan, T. Yager, and D. Niemi, "Byte-unixbench: A unix benchmark suite," 2011.

[42] C. Spensky, H. Hu, and K. Leach, "Lo-phi: Low-observable physical host instrumentation for malware analysis." in *NDSS*, 2016.

[43] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *In Proc. European Symposium on Research in Computer Security*, 2014.

[44] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 367–378.

[45] J. Wang, K. Sun, and A. Stavrou, "A dependability analysis of hardware-assisted polling integrity checking systems," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 2012, pp. 1–12.

[46] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[47] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

[48] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assisted integrity monitor," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 332–344, 2014.

[49] F. Zhang and H. Zhang, "Sok: A study of using hardware-assisted isolated execution environments for security," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016, p. 3.

[50] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 337–352.

[51] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 72–90.

[52] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices." *IACR Cryptology ePrint Archive*, vol. 2016, p. 980, 2016.

[53] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 239–242.