Work in Progress: Exploring Schedule-Based Side-Channels in TrustZone-Enabled Real-Time Systems

Mohamed Anis Aguida and Monowar Hasan
School of Computing, Wichita State University, Wichita, KS, USA
Email: mxaguida@shockers.wichita.edu, monowar.hasan@wichita.edu

***Abstract*—Our research demonstrates the existence of side-channel information leaks in TrustZone-enabled real-time systems. Our algorithm can infer the critical tasks' arrival times and pinpoint when the system switches between regular and secure execution modes. By precisely obtaining such timing information, an adversary could infer the task execution patterns inside the secure system — thus putting the system's safety, security, and integrity at risk. Considering that secure enclaves such as TrustZone are used for executing security-critical functionalities, our findings will help designers be aware of side-channel vulnerabilities and assist them in designing better, leakage-proof systems.**

I. Introduction

Real-time systems find use in many safety-critical applications such as avionics, automobiles, self-driving cars, manufacturing and control systems, healthcare, to name but a few. In the past, security was often not a design priority for such systems due to *(a)* limited resources (e.g., processor, memory, energy), *(b)* use of propitiatory protocols, hardware (i.e., such systems were considered "air-gaped"), and *(c)* popular beliefs (i.e., *how does someone get in? what do they do once they intrude?*). However, with the increased complexity, the use of off-the-shelf hardware/software stacks, and the proliferation of emerging "connected" and IoT-specific applications challenged those decade-long conceptions. The sophisticated, real-world attacks on real-time systems (e.g., control systems [1], automobiles [2], airplanes [3], medical devices [4], to name a few) show that security threats are real and we need layered defense mechanisms to protect those systems. One way to ensure tamper-proof execution of embedded applications is to leverage Trusted Execution Environments (TEEs), such as ARM TrustZone [5] available on modern off-the-shelf processors. TrustZone allows the execution of security-critical codes inside a "trusted enclave" by using a hardware-software co-design approach. While the vanilla TrustZone was not designed for real-time applications, recently researchers show how TrustZone can be retrofitted for real-time schedulers. Although TrustZone can provide means for secure executions, without careful design/analysis such systems can also "leak" critical information [6]. An adversary can leverage such *side-channel information* and drive the systems in an undesirable way. In this work, *we investigate the problem of information leakage in TrustZone-enabled real-time microcontrollers*. In particular, we study the *existence of schedule-based side-channels* real-time TrustZone systems running on Cortex-M processors. The goal of our research is to identify whether an adversary can *(a) infer when the system transitions into a secure mode*, for instance, to execute security-critical codes; and *(b)* whether can *predict execution patterns* of the tasks that are running inside the secure enclave – all, without having sufficient privilege (i.e., an unprotected, non-secure, low-priority task running on the userspace). Knowing such information is vital since it can have detrimental aftereffects. For instance, if an adversary can know when a security-critical code is running (say inside the TrustZone secure enclave) or when a system transition is to the secure mode, the attacker can prevent the code from being executed, infer other side-channel information (e.g., cache usage) [7] or perform denial-of-service attacks that may prevent the system to switch into secure execution mode — hence deviates the main objective of placing a secure enclave in the system. To the best of our knowledge, there exists no prior work that examines schedule-based information. Our preliminary study shows that it is feasible to infer when the system switches to a secure mode from regular execution mode (and vice-versa) and the arrival times of the security-critical tasks. From an adversary's point of view, this reveals the secrets inside the TrustZone, which enables the attacker to make an accurate, targeted attack; for instance, taking control of a critical system component [7]. We will present the details of our algorithm in later sections. Now, we start with the background on ARM TrustZone before we present the system and threat model.

II. Background, Models & Assumptions

A. Background — ARM TrustZone

ARM TrustZone is a set of hardware security improvements for Arm application processors (Cortex-A) and Cortex-M family of Arm microcontrollers [8]. TrustZone takes a system-wide, system-on-chip (SoC) and hardware/software-based approach to ensure security [5]. The TrustZone technology is based on the concept of "secure world" and "non-secure (normal) world" protection zones. The processor's software can run in either a secure or a non-secure state. In this work, we investigate side channels on a Cortex-M-based TrustZone enabled real-time system. On Cortex-M microcontrollers, the bridge between two worlds is handled by a set of mechanisms built into the core logic. Both worlds are hardware segregated, with non-secure software unable to access secure world resources directly. This strong hardware-enforced separation of worlds opens up new possibilities for application and data security. Critical programs can reside inside the secure world without relying on the regular operating system (called rich OS) for protection. This is achieved by restricting the rich OS to operate inside the normal world only [8].

B. System Model

We consider a uniprocessor, fixed-priority, preemptive real-time system, running on a Cortex-M-based TrustZone platform. The system is divided into two modes (viz., *secure world (S)*, and *non-secure world (NS)*) and consists of n real-time tasks $\Gamma = \{\tau_1, ...\tau_N\}$. We denote the set of tasks that run in the secure world and non-secure world as Γ_S and Γ_{NS}, respectively. We assume that the secure tasks are periodic, and the non-secure tasks can be either periodic or sporadic. Each task τ_i is characterized by $(p_i, d_i, e_i, a_i, pri_i, s)$ where p_i is the period, d_i is the relative deadline, e_i is the worst-case execution time (WCET), a_i is the initial task offset (i.e., the arrival time) and pri_i is the priority. and s is the state of the task whether it is secure or non-secure (i.e., *S*: for secure, *NS*: for non-secure). We consider an implicit deadline system (i.e., $d_i = p_i$), and each task has a distinct period. We use the same symbol τ_i to represent a task's job for simplicity of notation. We further assume that the taskset is "schedulable" by a fixed-priority, preemptive real-time scheduler and scheduled using the Rate Monotonic (RM) scheduling algorithm [9]. Let $hp(\tau_i)$ denote the set of tasks that have higher priorities than that of τ_i and $lp(\tau_i)$ denote the set of tasks that have lower priorities than τ_i. Let Ω define all secure tasks periods where $\Omega = \{p_i; i \in S\}$. We define an "execution interval" of a task to be an interval of time $[a, b)$ during which the task runs continuously. If τ_i is preempted, then the execution will be partitioned into multiple execution intervals, each of which has a length less than e_i.

C. Threat Model

We assume that the attacker can gain access to a user space of a non-secure task that we refer to as *observer task* τ_o, and is targeting some secure tasks τ_{t_i} called *victim tasks*, where $\tau_o \in \Gamma_{NS}$ and $\tau_{t_i} \in \Gamma_S$. We assume that the attacker knows all the tasks periods and their execution times. The attacker may have access to the system timer. Note that gaining access to the victim task or making the attack after inferring the switching points between secure and non-secure worlds are out of our scope. In this work, we focus on *(a)* modeling the timing side-channel attack on Cortex-M based TrustZone-enabled systems to infer the exact time of the switching between the two worlds and *(b)* getting the arrival times of the secure tasks inside the secure enclave.

III. Schedule-based Information Leakage

Recall from the earlier discussion that non-secure software is unable to access secure world resources directly by design. In this work we show that it is feasible to infer schedule and task execution patterns by using the information observed from a non-privileged, non-secure (i.e., normal world) task $\tau_i \in \Gamma_{NS}$. We infer the arrival times of the secure tasks

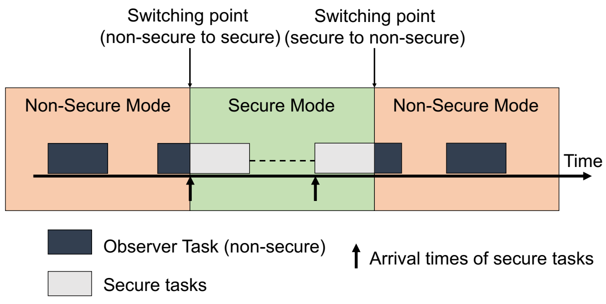
Switching point (non-secure to secure)
Switching point (secure to non-secure)
Non-Secure Mode
Secure Mode
Non-Secure Mode
Time
Observer Task (non-secure)
Secure tasks
Arrival times of secure tasks

Fig. 1. High-level schematic of our proposed approach.

running inside the secure enclave, as well as the switching points between the two execution modes (i.e., from non-secure to secure, and from secure to non-secure), as shown in Figure 1. For instance, if we precisely know the switching point between the two worlds, this information can lead to a malicious/unprotected task (say controlled by the attacker) stopping the execution of the targeted secure task. Hence, such timing and schedule information inference in the secure world would lead to information leakage, resulting in undesirable manipulation by the adversary. Our research aims to design an analytical framework to infer the arrival times of *all* the tasks executed in the secure world, by using a userspace non-privileged, non-secure task (Section III-A). Our proposed techniques can also infer the context switching time points between the two worlds. We now introduce our proposed algorithm. Figure 2 presents the workflow of our scheme. Our inference algorithm is a five-step process performed as follows:

1) *Monitoring execution intervals:* We first construct the execution intervals of the observer task τ_o.
2) *Analyzing execution intervals:* We arrange the constructed execution intervals in a timeline that will help us in the following (inference) phase.
3) *Inferring secure tasks' arrival times:* We then predict the arrival time of the highest priority secure task. Once we infer its future arrivals, we then move to the next highest priority secure task. We repeat the above steps until we iterate all the secure tasks.
4) *Constructing the schedule:* We organize the constructed execution intervals of the secure tasks to predict the schedule.
5) *Infer the switching points:* Finally, we infer the context switching points between the two worlds.

A. Overview

The key idea of our inference algorithm is to leverage scheduler characteristics for inferring the timing information. In particular, we leverage the fact that *lower priority tasks can only execute when a higher priority task is not running*. Thus, when inferring the arrival time of each secure task (i.e., tasks that are running inside the enclave), we first track the *execution of an observer task* (i.e., non-secure task). We then

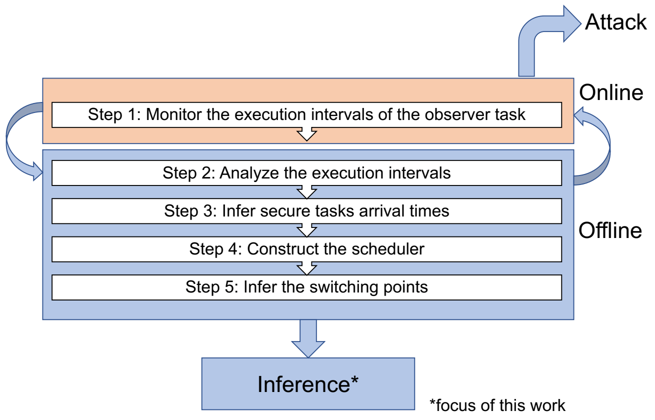
Attack
Online
Step 1: Monitor the execution intervals of the observer task
Step 2: Analyze the execution intervals
Step 3: Infer secure tasks arrival times
Offline
Step 4: Construct the scheduler
Step 5: Infer the switching points
Inference*
*focus of this work

Fig. 2. High-level workflow of our inference process.

Algorithm 1 Inferring Context Switch Points

```
1:  INITIALIZATION
2:  $\Omega = \{p_i; i \in S\}$
3:  $H = lcm(p_i; i \in T)$
4:  $ExecIterv \leftarrow \{\}$
5:  BEGIN
6:  Construct the execution intervals of a non-secure task (observer
    task)
7:  $ExecIterv \leftarrow \{IntervalsOF(\tau_o)\}$
8:  while $\Omega \neq NONE$ do
9:      $S \leftarrow \min\{\Omega\}$
10:     $\tau \leftarrow \{\tau_i; \tau_i \in \Gamma\}$ where $Period(\tau) == S$
11:     Predict the arrival time, and infer the future arrival times of $\tau$
        (see Section III-C-III-D)
12:     $ExecInterv \leftarrow ExecInterv + IntervalsOF(\tau)$
13:     $\Omega \leftarrow \Omega - S$
14: end while
15: Infer the switching points between secure and non-secure worlds
16: END
```

predict the execution of a secure task. This process works due to the fact that a secure task can not execute when the observer task is executing by definition.

Algorithm 1 formally presents our idea. First, we initialize the variables: *(a)* Ω denotes all secure tasks' periods, *(b)* the *hyperperiod* H is the minimum interval of time after which the schedule repeats itself, and *(c)* $ExecInterv$ is the variable that contains the execution intervals of the tasks that we already investigated (i.e., observer task, secure tasks). We monitor the observer task τ_o and construct its execution intervals (*line 6*), this step is done at runtime (i.e., online). Note that, to ensure that the task completes before its deadline, the inference operation must take minimal time. The function $IntervalsOF$ constructs the execution intervals. We iterate on all the secure tasks and construct their execution intervals by predicting their arrival times and inferring their future arrival times. This process starts by selecting the lowest period among all secure tasks periods *(lines 9-10)*. We then use a timeline divided into windows that match the period of the secure task (see Section III-C for details). Hence we can predict the secure task arrival time *(line 11)* and infer future arrival times. Once we construct the execution intervals of the secure task, we add it to the *ExecInterv* variable to be used for the next secure task *(line 12)*. Finally, we can construct the task schedule inside the enclave. As a result, we can also infer the switching points between the secure and non-secure worlds *(line 15)*.

B. Monitoring the Execution Intervals

As mentioned earlier, the first step of our inference process is to monitor the observer task's execution intervals. Using the system timer, we implement a function in the observer task that keeps track of its execution intervals. For a stealthy attack, we need to ensure that the observer task is executing in a timely manner in addition to the function of constructing the execution intervals without exceeding its deadline. We note that this is the only step that is executed online; all the remaining steps can be performed offline (i.e., on the attacker system).

While constructing the execution interval of the observer task, the task could be preempted by other higher-priority tasks. Hence, it is not straightforward to construct the exact execution interval since we do not know the exact point for the preemption. To overcome the preemption problem, we can check the periods of tasks that have a priority higher than the observer task $hp(\tau_o)$. For this, we calculate the modulo to each tick time in the execution interval. Algorithm 2 formally presents the calculation of preemption points to accurately construct the execution intervals. Let us define Φ to represent the tasks that have higher priority than the observer task τ_o. Further, let Ψ define the tasks that preempt the observer task. We start our inference by *(a)* getting the begin time, *(b)* executing the observer logic, and then *(c)* getting the end time *(lines 5-7)*. We initialize the $TickCount$ variable by the begin time. We then iterate the interval $[BeginTime, EndTime]$ by incrementing $TickCount$ by one. Note that we iterate over all the tasks defined by Φ *(lines 8 to 11)*. In each iteration, we calculate the modulo of the $TickCount$ with the period of a task $\tau_j \in \Phi$. If the modulo equal to zero, which implies that the task is arrived in the $TickCount$ point. We add the task to Ψ along with the tick time point as $\Psi \leftarrow \{\tau_j, TickCount\}$ *(lines 12-13)*. At the end, we return Ψ *(line 20)*. Once we find the arrival time of all the tasks that appear in this execution, we can precisely construct the execution interval of the observer task.

C. Analyzing the Execution Intervals

Once we constructed the execution intervals of the observer task, the next step is to analyze these intervals in a way that we can extract a piece of useful information from them. For this, we use a *timeline divided into windows* that match the period of the secure tasks. To be specific, we arrange the observer task execution intervals into a timeline. Each window of the timeline equals the period of the secure task (i.e., the task under analysis to infer its execution intervals). This representation allows us to observe how the observer task is affected by the secure task. A way to graphically represent this timeline is using the "*schedule ladder diagram*" [7]. As the length of the window is equal to the period of the secure task,

Algorithm 2 Infer Observer Task Preemption Points

```
 1: INITIALIZATION
 2: $\Phi = \{hp(\tau_o)\}$
 3: $\Psi \leftarrow \{\}$
 4: BEGIN
 5: $BeginTime \leftarrow getTime()$
 6: ObserverTask logic execution
 7: $EndTime \leftarrow getTime()$
 8: $TickCount \leftarrow BeginTime$
 9: $j \leftarrow 0$
10: while $TickCount \leq EndTime$ do
11:     while $j \leq lengthOf(\Phi)$ do
12:         if $TickCount \mod p_j = 0$ then
13:             $\Psi \leftarrow \{\tau_j, TickCount\}$
14:         end if
15:         $j++$
16:     end while
17:     $TickCount++$
18:     $j \leftarrow 0$
19: end while
20: Retrurn $\Psi$
21: END
```

we can assure that each time window contains an instance of the secure task (recall that secure tasks are periodic). Using the fact both observer and the secure task cannot execute at the same time, we can use some experimental techniques to predict the arrival time of the secure task. Note: our experiments are ongoing, and we omit the details due to space constraints.

D. Inferring Secure Tasks Arrival Times

After constructing the intervals of the observer task, we can now predict the arrival time of the secure task. As mentioned in the previous section, we organize the observer executions in a timeline, then we take the first secure task which has the highest priority task among all secure tasks τ_s where $hp_{\Gamma_S}(\tau_s) = None$ (i.e., the lowest period). This condition will *(i) accurately predict the first arrival time* (and, hence, future arrivals), and *(ii) ensure that there is no other secure task is executing at that time*. We use a combination of experimental and analytical techniques to predict the arrival time of τ_s and then its future arrival time. This latter secure task execution interval is added (as well as the observer task execution intervals) in a new timeline with a window length equal to the period of the next highest priority secure task. We perform the same operation described above for all the secure tasks.

E. Constructing the Schedule and Inferring the switching points

To construct the schedule, we need to examine all the secure tasks with the same steps that we mentioned earlier (Sections III-B-III-D). We note that these steps do not affect the performance of the system since they can be performed offline. Only the first step (i.e., monitoring the observer task's executions) will be performed online for a single hyperperiod.

After constructing the schedule, we can pinpoint the arrival times of the secure tasks. This may allow an attacker to make more targeted attacks, for instance, to block any *secure task* at the right time. Another information we can infer is the time switching points between the secure and non-secure worlds. Such information could help the attackers to carry out further attacks such as interrupting the system or overriding the behavior of a specific task.

IV. Related Work & Conclusion

Researchers demonstrate side-channel attacks in various contexts [10], [10]–[12]. However, they *(i)* are not designed for TrustZone-based systems and *(ii)* do not consider real-time properties. Assessing the security of the general-purpose ARM TrustZone against side-channel attacks has also been studied [6] Chen et al. [7] propose a scheduler side-channel attack (called ScheduLeak) on a vanilla real-time system. In contrast, our research shows that TrustZone-enabled real-time systems are vulnerable to schedule-based inference attacks. We introduce a novel timing side-channel attack that targets ARM TrustZone-based real-time microcontrollers.

We are extending our preliminary findings with further conceptual analysis and real-world demonstrations. As proof of concept, we implemented our algorithm on an NXP LPC55S69 platform running a TrustZone-enabled real-time operating system (Amazon FreeRTOS). Knowing that TrustZone technologies (or TEEs in general) act as a "shield" for ensuring security, we believe that our findings will provide valuable hints for the real-time engineers on how to conceal schedule-based information leaks.

References

[1] D. U. Case, "Analysis of the cyber attack on the ukrainian power grid," *Electricity Information Sharing and Analysis Center (E-ISAC)*, vol. 388, 2016.

[2] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, *et al.*, "Experimental security analysis of a modern automobile," in *The Ethics of Information Technologies*, pp. 119–134, Routledge, 2020.

[3] H. Teso, "Aircraft hacking: Practical aero series," in *4th Hack in the Box Security Conference in Europe*, 2013.

[4] S. S. Clark and K. Fu, "Recent results in computer security for medical devices," in *International Conference on Wireless Mobile Communication and Healthcare*, pp. 111–118, Springer, 2011.

[5] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[6] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 1097–1105, IEEE, 2018.

[7] C.-Y. Chen, S. Mohan, R. Pellizzoni, R. B. Bobba, and N. Kiyavash, "A novel side-channel in real-time schedulers," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 90–102, IEEE, 2019.

[8] A. Holdings, "Building a secure system using trust-zone technology," 2005.

[9] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[10] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs," *Journal of Cryptographic Engineering*, vol. 5, no. 2, pp. 95–112, 2015.

[11] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*, pp. 1–20, Springer, 2006.

[12] J.-S. Coron, "Resistance against differential power analysis for elliptic curve cryptosystems," in *International workshop on cryptographic hardware and embedded systems*, pp. 292–302, Springer, 1999.