Teetime: Timing-Based Approach for Determining Secure World Behavior in ARM TrustZone

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Jason Ashley
Spring, 2021

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Introduction

Trusted Execution Environments (TEEs) separate security-sensitive operations from those that are not. This is in an attempt to secure the sensitive operations through isolation. If such isolation is successful, this can provide confidentiality, integrity, and even more availability. One of the most popular basis for a TEE is ARM's TrustZone. Born out of the idea that even the operating system some piece of hardware is running cannot be trusted to keep application data secure, this TEE instead allows another operating system to run parallel to the main one, where calls for secure data are handed over to trusted applications in the secure operating system. These two operating systems are said to live in different "worlds," secure and normal, with an (ideally) minimal interface between them. ARM processors, with their hundreds of millions, if not billions, of deployments in phones, remote sensors, and other consumer electronics, are often open to attacks, so an additional layer of security provided by a TEE would aid in its securing these systems. However, ARM's TrustZone and its various implementations do not necessarily guarantee the separation between worlds that they are designed to promote. Several flaws in the the various implementations of TrustZone have been found in many different steps of the development and design process.

Architecture

Unlike approaches to Trusted Execution Environments that use separate physical hardware, ARM TrustZone relies on processor time being separated between the normal and secure worlds[1]. Physical cores are shared between these two worlds as two virtual cores. The worlds are switched as necessary, often by calls from the normal world to secure through the Secure Monitor. Each world ideally runs independent of the other, with the Secure Monitor relaying requests from one world to the other, in the most basic approaches for sharing data.

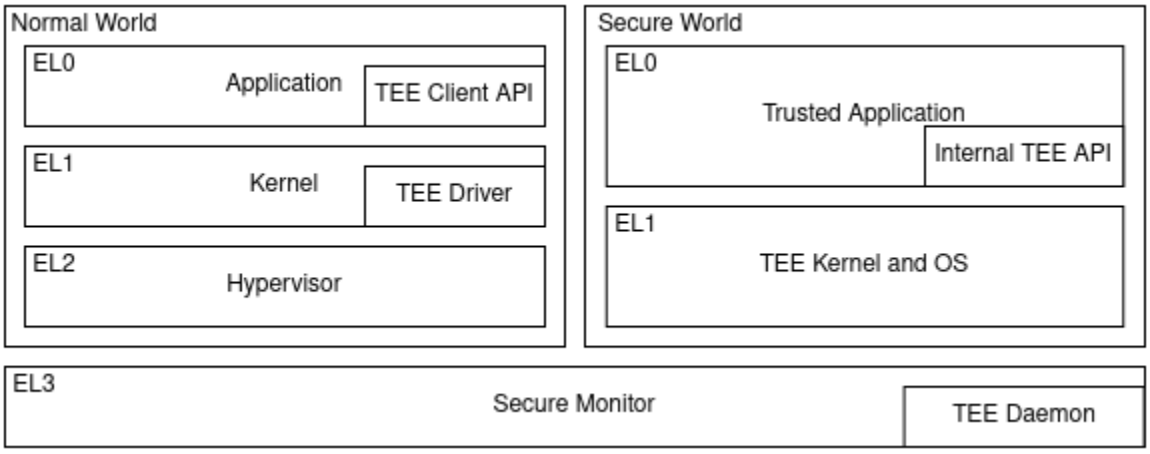

Figure 1: ARM TrustZone architecture overview

ARM has a notion of “Exception Levels” (ELs) that indicate what system registers are available for access. In recent ARM cores, the levels are defined from EL0 to EL3, where EL0 is least privilege[2]. User applications are expected to run in EL0. The operating system itself is expected to run at one higher level, EL1, to gain access to more system registers for memory management and exception handling. If the OS running in the normal world itself is running on a hypervisor, which would be useful for allowing multiple operating systems to be run in normal world, the hypervisor has access to EL2, which provides it more direct access to registers that might be useful for virtual machine management, or essentially being able to run two operating systems in parallel. EL3, finally, is dedicated for “Monitor Mode,” which is where TrustZone’s Secure Monitor sits. EL0-EL2 are separated into secure and non-secure classifications, highlighting whether some privilege is used in the secure world or not.

At the “highest level” of privileges, the Secure Monitor has access to every register on the system. This watches for calls to the secure world from the normal world and will allow a switch into the secure world to handle a call. However, as this monitor does have the ability to read and control the entire processor state, it is vital that this system is protected quite well. If vulnerabilities are found in this, it could be possible to overwrite arbitrary memory in either world, modify the processor’s execution state in some manner, or modify hardware configurations.

In the secure world, Trusted Applications, though they run at EL0, have access to the wider TEE API, which gives it the ability to access critical information only available through TEE. The TEE Kernel and OS will run at EL1, but there is not expected to be a hypervisor between this kernel and the Secure Monitor. These operating systems and kernels should have permissions for memory, devices, storage, or any other peripherals that may have been blocked from the normal world due to their security-critical roles.

TrustZone is designed to ensure several capabilities. These include isolated execution between worlds, secure storage of secure world data, device identification, device authentication, device attestation, and platform integrity[3].

Previous Research

There are a range of previous attacks that have been attempted and performed on various TrustZone implementations. In a study of previous attacks, they are often separated into three major categories: architectural issues, implementation issues, and hardware issues. While attacks could be further broken down based on which TrustZone implementation they are for, some of the vulnerabilities are quite common between implementations. Particularly, due to the wide range of activities expected of a TrustZone TEE now, it is possible they could have a large API. If a problem is found in any of these API calls, this has the potential to compromise the whole system. Further, some of the expected behavior for TEEs leads to them violating their promise of isolation, such as for sharing memory to make data transfers between worlds more rapid. These issues may be solved with software patches, in most cases, though some stem from the architecture of the system allowing for such operations.

However, there are classes of problems that may not be solved with patches, including hardware side-channels. In some studies, for instance, devices where an ARM processor is coupled with an FPGA were studied. The FPGA was found to be capable of compromising the secure world in a few different manners using the provided extensions[4]. In more common hardware, faults are injected to gain critical information about operations in the secure world, such as through the CLKSCREW attack or through methods using Rowhammer[5], [6]. Information is also leaked through cache, as highlighted by several different attacks which used a variation of a Prime+Probe and the contention of cache lines between the normal and secure world to retrieve data from the secure world[7]. In some cases, the attacker would be able to distinguish valid keys from invalid keys based on data collected (such as in ARMageddon), but some attacks have been successful in retrieving 128-bit AES keys[1]. Further, Prime+Count techniques have been used to create covert channels between worlds[8]. Yet another approach for this discussion consists of using the branch predictor, priming it with a set of branches, then allowing the secure world to operate[9]. The attacker regains control after the secure world and checks for mispredictions, allowing them to guess at branches taken, which has resulted in a 256-bit key being recovered. Even the power consumption has been used to help extract sensitive information, meaning even exposing power consumption information is a risk to this secure environments, as shown by an attack on Intel's SGX[10].

There have been a case of side-channel vulnerabilities caused by software behavior in TrustZone TEEs, too, not just hardware. The biggest case of this would be the LibTomCrypt library that leaked details about exponents used based on execution time in OP-TEE[1], [11]. Larger exponents required more calculation time. Based on the length of time in the secure world working on this exponent calculation, it was possible to roughly guess the prime number used. By modifying this operation to make it take a more constant time, this problem was mitigated. However, other functions in the TEE may exhibit non-constant time operation, still.

The protections available in TrustZone TEEs against vulnerabilities are often also quite lacking. Many do not support or have limited versions of common protections found in modern operating systems, such as ASLR, stack cookies, or guard pages, making attacks more likely to succeed[1]. In fact, QSEE, Qualcomm's TrustZone implementation, only has ASLR for some bits of the address space, making it possible to guess at addresses quickly, while many others lack ASLR entirely in and out of kernel space[12]. Due to the wide attack surface in modern TrustZone TEEs, expanding far beyond their original intentions, further, more extreme approaches have been suggested, such as switching to memory safe languages to avoid buffer-based attacks, including many isolated environments to separate trusted applications from each other, and taking more measures to secure channels across worlds[3].

Primary Concern

One possible concern not apparently addressed with current TrustZone implementations is the different runtimes of different operations. While calls to the TEE are scheduled by the Linux kernel before the switch into the secure world occurs, the time in the secure world may depend on how long it

takes for the trusted application to return. This can vary based on the type of operations the TEE is performing. For instance, consider two functions, one that fetches details from memory while the other fetches a key from secure storage and decrypts a message, then places it into a buffer. Depending on implementation and hardware, the second operation could take significantly longer than the first. Now consider two applications in this situation running in the normal world, one malicious and one that calls the TEE. If the malicious application is constantly timing and looking for differences in the time between schedules using high precision timers, it is possible it will find that when the TEE is called, it has a higher delay before returning. This would allow the malicious application to determine when the TEE has been entered. However, if the TrustZone implementation is known and sufficient data has been collected on the runtime of certain trusted applications that may be in use, it is possible this difference in timing over time could be statistically analyzed to determine the exact type of call made to the TEE.

This would be useful for profiling applications that are known to access the TrustZone TEE when it is otherwise impossible to get details about the running application. Such information could be useful for targeting other attacks, such as advanced phishing attacks that only launch on a request for a password or fingerprint in a banking application on Android, or a more targeted attack that is designed explicitly for a particular trusted application. In combination with other potential side-channel attacks, this could also be useful for more specific targeting when secure accesses occur in embedded devices, allowing for more finely tuned attacks. This type of vulnerability could also be considered breaking the isolated execution guarantee of TrustZone, allowing the attacker to gain information about the execution of a program accessing the TEE, especially if any variance of parameters to a trusted application could also be detected.

Set Up

OP-TEE, an open source TrustZone implementation, will be used for testing this possible attack and gleaning some information about the actions performed in the secure world from the normal world[13].

QEMU was first used to test the build process and attempt to test for this potential problem without hardware. Fortunately, within an environment setup with GCC 9.3, this was built according to the provided directions for QEMU for ARMv7 processors. Unfortunately, this setup did not provide reasonably consistent access to the high precision timers, making it difficult to precisely gauge the timing spent on other processes or in the TEE.

Attempts were originally made to run OP-TEE on hardware on a Pine A64. This single board computer by Pine 64 uses an Allwinner A64 SOC. A very simple Linux kernel and shell was built in an Ubuntu 20.04 LTS virtual machine. The build process was originally attempted in a QEMU-based VM, but this apparently crashed the VM. The process was repeated in a VirtualBox-based VM. No crashing was observed during this build process. However, there was no simple build config developed for this particular chip, so the following command was used to make the OP-TEE OS project:

```
make CFG_ARM64_CORE=y \
        CFG_TEE_LOGLEVEL=4 \
        CFG_TEE_CORE_LOG_LEVEL=4 \
        CROSS_COMPILE32="ccache arm-none-eabi-" \
        CROSS_COMPILE64="ccache aarch64-linux-gnu-" \
        DEBUG=1 \
        PLATFORM=sunxi-sun50i_a64
```

This built the operating system for the secure world, but the client, Linux kernel, and bootloader still needed to be built. Fortunately, the client did not need additional configuration, and U-Boot just needed a disk to write to at a 8KB offset.

The Linux kernel for the Pine A64 lacked a GPU driver and needed a modified DTS (Device Tree Source) to specify that OP-TEE would be enabled. The kernel configuration options can be configured for the GPU driver, and adding the following block to the relevant DTS for the Pine A64 in the kernel source will make allow it to support OP-TEE:

```
firmware {
    optee {
        compatible = "linaro,optee-tz";
        method = "smc";
    };
};
```

After this, the SD card must be partitioned appropriately and U-Boot configured. While there are several configurations online, the one utilized was the following:

```
fatload mmc 0 0x46000000 Image
fatload mmc 0 0x49000000 sun50i-a64-pine64-plus.dtb
setenv bootargs console=ttyS0,115200 console=tty1 earlyprintk root=/dev/mmcblk0p2
    rootwait panic=10
booti 0x46000000 – 0x49000000
```

Next, the root filesystem had to be created. Several options for this exist, such as debootstrap or buildroot.

Unfortunately, once these steps were completed, the Pine A64 would fail to boot, getting stuck in a bootloop on U-Boot. While this could be caused but the particular configuration for U-Boot, it

could also have been many of the other steps or an incompatibility between some parts of the tooling. This would be worth exploring in the future as the Pine A64 is a relatively inexpensive and complete development platform for such a project as this.

After this failed to get running, a Raspberry Pi 3 was used instead to load OP-TEE. Fortunately, OP-TEE provides a complete build script for all parts of the Raspberry Pi. This includes the TEE client, OS, Linux kernel, bootloader, and rootfs. Unfortunately, the rootfs built for the system is quite minimal and the Raspberry Pi does not have secure memory or peripheral pathways. This leads to several limitations. The first (and biggest) is that the Pi cannot actually be trusted to protect its memory or connected devices across worlds. As the intention is not to test this, this limitation can be ignored. However, if future work does explore this area, a different hardware platform would be advisable.

The rootfs for this install script is setup by buildroot. While buildroot is great in a lot of cases, it is a very minimal Linux setup not fit for development, moreso for deployment. Thus, other applications will either have to be added into the buildroot configuration. Some basic text editors were added, but gcc and other compilers are not officially supported. However, it appears the relevant libraries are added to the rootfs image, but the executable is not, despite being successfully cross-compiled. This might just be an issue with the specifications for this version of buildroot or some misconfigured options. Custom executables may be easily added to the image through specifying a configuration for the program, a buildroot makefile that patches in necessary arguments for cross-compiling, and adding the necessary lines for that file to the configuration (this is specified in the buildroot documentation, though was kept quite simple for this). TEE applications will need to be added from within the OP-TEE sources. For custom applications that call the TEE, it will be necessary to at least add a client application. For the demonstration, two trusted applications (TA) are added to the TEE, as well as a client for normal world calls to these TAs. One TA, called “Short,” runs a simple addition problem with parameters passed by the user. The second, “Long,” runs a series of multiplications in the TA. Realistically, the difference in execution times of these applications will be similar, but “Long” should indeed take a detectable amount of time longer. This replicates two TAs with presumably constant running times running a short operation, but would also be difficult to differentiate. Further, the example TAs within OP-TEE may be used to show more extreme situations.

The next step was finding some approach for timing the operation. There were two major possibilities. One was timing on the device and saving and storing the data there. The advantage is it could be run by itself and could allow for rapid iteration of tests. However, there are resource limitations that could be encountered, plus this method will introduce more timing difficulties due to the necessary writing to memory or storage. An alternative method is timing on some other device and sending a particular message repeatedly to the device to indicate when timings occur. If using a relatively fast device for both sending and receiving, times should still be distinguishable. However, scheduling behavior on either device could skew the results. To limit this, the former method will be used. The Raspberry Pi will be connected with UART for simple communications using and FTDI USB adapter connected appropriately to the Pi’s UX and TX pins on the GPIO. For this demonstration, as the Raspberry Pi will be provided with an SD card containing a large storage space, its memory will be

used to store the collected data, later being exported over that same UART connection after the test has been executed and run to completion. Use of the UART buffer will be limited throughout the duration of the test so that it will not impact the test.

Thus, there is now a piece of hardware running OP-TEE, a minimal Linux environment, and custom code for timing. Three programs were then run, one that calls the a short-running trusted application 20 times, another that runs a longer-running trusted application 20 times, then one that just yields processor time. At the same time, of each of these programs, the timing application was writing into a file buffer. After some period of time, a script managing these programs and these instances shut down the timing application. These three programs were run individually with the timing application and a seperate file saved with the timing data between schedules. In this manner, it was possible to determine the impact on time between schedules for some normal world application and the possibility of determining what TA was in use based on the collected data.

Results

The three situations were tested, each running 20 times (or, in the case of the yielding application, set to yield the approximate for the approximate amount of time as the max of the two trusted applications' running times. Before the TA was executed, approximately one second of timing data was recorded. Following the TA exiting, one more second of data was collected. The following behavior was observed amongst the short, long, and no trusted application setups:

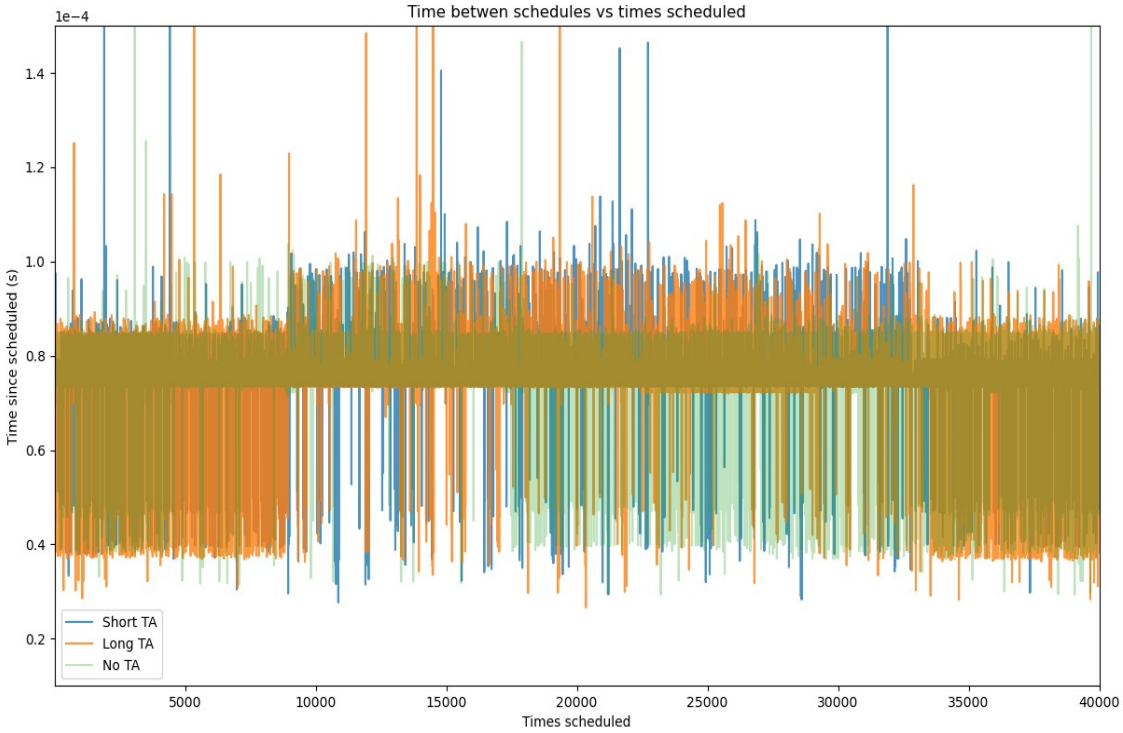

Figure 2: Time between schedules vs times scheduled with "short" and "long" TAs

Using these custom trusted applications, the times between schedule are lowest when no trusted application is in use. There is a noticeable increase in times between schedules when the trusted applications are in use, which means use of the TEE could be detected in these cases with a well-tuned program. As for detecting the particular trusted application in use, the time at the beginning and end was first removed. Based on those data points, there was an average difference of 6.00% between the runtimes of these two similar trusted applications across all of the runs (with the long TA's average always exceeding the short TA's), which is significant and could be detectable. From this collected data, the following details could be collected from the runs of the trusted applications (beginning and end waits removed):

Table 1: Timing data for "long" and "short" TAs

Metric	Short	Long
Timing points collected	22628	24028
Mean time between schedules (s)	7.496161273643274e-05	7.496522086732146e-05
Median time between schedules (s)	7.4063e-05	7.4115e-05
Total running time of 20 runs (s)	1.6962313729999876	1.8012643269999984
Max time between schedules (s)	0.000196875	0.000524271
Min time between schedules (s)	2.177e-05	2.8073e-05

During one set of runs, the behavior of the these situations appeared as the following:

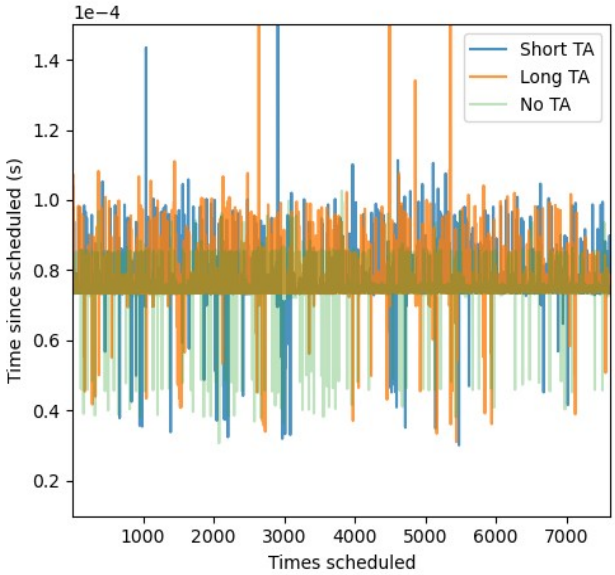

Figure 3: Time since schedule vs times scheduled with waits removed

While the two TAs did not have very different behaviors, the case without a TA clearly had a much lower time between schedules more consistently.

However, comparing the short TA and an example one provided by OP-TEE that performs AES encryption and decryption shows a more extreme situation, where the TAs have very different behaviors:

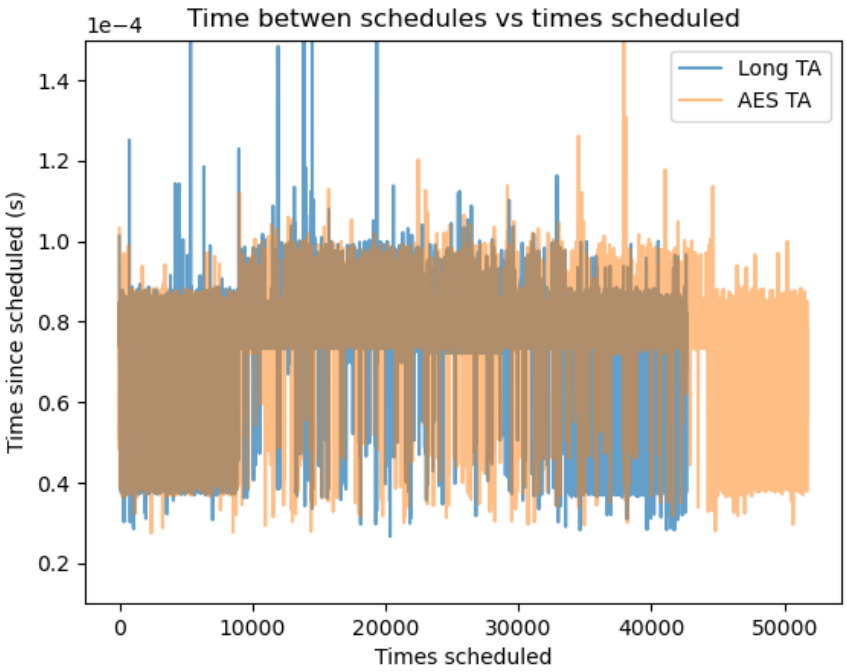

Figure 4: Time between schedules vs times scheduled w/ AES example

Between these two, there was a 19.65% difference in running time, with the AES example TA requiring much more time than the simple “long” TA. In fact, the “long” TA’s data collection had already stopped in totality while the AES was still in the TA (leading to the uneven stopping points on this graph). The example does perform a chain of actions, but is not unlike some applications that might be deployed that accept a set of parameters to perform a longer running task.

Future Work

Continuing this line of research, it would be interesting to see what information could be gleaned by measuring execution time of different functions. If this method indicates it is possible to reliably determine current behavior of a TrustZone TEE, it would be interesting to first develop a proof of concept on an Android phone, running a timing service in the background in some manner, then triggering an event to hijack a login or convice a user to enter information into a malicious prompt when a banking application or similar makes a request to the TEE. Beyond this, it could be possible to detetermine behavior within particular trusted applications, beyond determining which trusted applications or functions are in use. For instance, could this be used to guess at different aspects of the parameters in use such as key length, message length, or required execution order within the TEE. For

reducing the leakage of this information, future efforts could work towards finding methods of randomizing execution times within the TEE or ensuring applications run with constant or similar runtimes could mitigate this danger, though this would require exploration into how this could be implemented so as to not allow differentiation between applications written by different groups, such as by modifying the behavior of the TEE OS so that applications would not have to make this consideration.

Conclusion

Throughout this project, ARM's TrustZone, a basis for Trusted Execution Environments (TEEs) on ARM processors, was explored to learn about its architecture, known vulnerabilities, and development process. Throughout the several different major types of vulnerabilities in TrustZone were explored, especially those relating to side-channel attacks. This led to a question as to whether the timing differences between functions themselves could be used to profile the current operations on the hardware. OP-TEE was used throughout for experimentation, eventually being run on a Raspberry Pi 3B for its processing and hardware capabilities, though it did not have perfectly secure I/O paths. Ideally, this has shown yet another potential avenue for side-channel attacks, granted with more limited usefulness in retrieving exact data, but much more useful for profiling a device and its usage and targeting other attacks. With fine-tuning through statistical analysis or machine learning, it is possible guessing could be quite accurate, possibly utilizing execution patterns to further guess at parameters and behavior. As such, future development within TrustZone TEEs should consider this problem and work towards more constant time approaches for the time being.

References

[1] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems,” in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 1416–1432. doi: 10.1109/SP40000.2020.00061.

[2] “AArch64 Exception model.” https://developer.arm.com/documentation/102412/0100/Privilege-and-Exception-levels (accessed May 03, 2021).

[3] N. Koutroumpouchos, C. Ntantogian, and C. Xenakis, “Building Trust for Smart Connected Devices: The Challenges and Pitfalls of TrustZone,” *Sensors*, vol. 21, no. 2, Jan. 2021, doi: 10.3390/s21020520.

[4] E. M. Benhani, L. Bossuet, and A. Aubert, “The Security of ARM TrustZone in a FPGA-Based SoC,” *IEEE Trans. Comput.*, vol. 68, no. 8, pp. 1238–1248, Aug. 2019, doi: 10.1109/TC.2019.2900235.

[5] A. Tang, S. Sethumadhavan, and S. Stolfo, “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management,” 2017, pp. 1057–1074. Accessed: May 03, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[6] GreHack, Grenoble, France. *Attack ARM TrustZone using Rowhammer*, (Dec. 11, 2017). Accessed: May 03, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=FiPFBWxHVmI

[7] 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*, Honolulu, HI, Apr. 2018, pp. 1097–1105. doi: 10.1109/INFOCOM.2018.8486293.

[8] H. Cho *et al.*, “Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, San Juan PR USA, Dec. 2018, pp. 441–452. doi: 10.1145/3274694.3274704.

[9] K. Ryan, “Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm’s TrustZone,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom, Nov. 2019, pp. 181–194. doi: 10.1145/3319535.3354197.

[10] M. Lipp *et al.*, “PLATYPUS: Software-based Power Side-Channel Attacks on x86,” *2021 IEEE Symp. Secur. Priv. SP*, 2021.

[11] “Security Advisories.,” *Linaro*. https://www.op-tee.org/security-advisories/ (accessed May 03, 2021).

[12] J. Guilbon, “Introduction to Trusted Execution Environment: ARM’s TrustZone,” *Quarkslabs*, Jun. 19, 2018. https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html (accessed May 03, 2021).

[13] “Open Portable Trusted Execution Environment,” *Linaro*. https://www.op-tee.org/ (accessed May 03, 2021).