KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

User attestation for the PinePhone

Oberon Swings

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige software

Promotor:
Prof. dr. ir. F. Piessens

Evaluatoren:
Prof. dr. D. Hughes
Ir. P. Totis

Begeleiders:
Dr. J. Mühlberg
Ir. F. Alder
Ir. S. Pouyanrad

Academiejaar 2021 – 2022

Voorwoord

TODO

Oberon Swings

Inhoudsopgave

Abstract

TODO

Samenvatting

TODO

Lijst van figuren

List of Abbreviations

Abbreviations

IoT	Internet of Things
TEE	Trusted Execution Environment
I/O	Input and Output
OS	Operating System

Hoofdstuk 1

Introduction

Smartphone functionality. Smartphones have become an essential part of our daily lives and everyone is assumed to have one of their own. These devices can be spotted everywhere, people use them at home, on the bus or even at work. In most cases these phones are only occasionally used for text messaging or calling but very often for reading e-mails, surfing on the web or even for services like e-banking. Because of this wide range of functionality some people may even replace their personal computer by a smartphone entirely. The success of smartphones lies in their ease of use and always being accessible, people just carry them in their pocket. Besides the user having access to their phone all the time, a smartphone also has or could have access to the internet all the time. It is known that the internet is the gate to lots of services which are perceived as necessities lately.

IoT security. Devices that make lots of connections on the go while also utilizing online services for which sensitive data is required may introduce security vulnerabilities. The fact that people are using their smartphones for services like online banking or even consulting health related reports implies that some sensitive data must be stored on these devices or at least present while they are interacting with it. While this data should be protected very well it is present on an Internet of Things (IoT) device for which security solutions and standards present today are not adequate in terms of protection against the existing threats. The hardware similarities between smartphones and IoT devices are more prominent than one might expect, this is because the architecture of smartphones stems from that of IoT devices. The main issue here is that IoT devices are designed for performance, they only have a small number of tasks but these need to be executed as fast or as energy efficient as possible. This also applies to smartphones because while the functionality of a smartphone is close to that of a personal computer the hardware is not. This weak link in the hardware gives rise to multiple different attack strategies that adversaries can utilize to steal sensitive data from smartphone users. Lately improvements have been made in this area by extending the processors of these devices with features that make it possible to setup a Trusted Execution Environment (TEE) on them. A TEE can increase the security of an IoT device, this is often achieved by utilizing core

security services for critical operations. Examples of these critical operations are cryptographic operations, storing data in secure memory or accessing Input and Output (I/O) through trusted paths.

Our contributions. This thesis is based on the reproduction of existing work namely [1], the solution of this paper is replicated as closely as possible. The proposed system attempts to increase the security of a smartphone by having the TEE attest the code of the user and Operating System (OS) space. To allow for a direct comparison between the performance achieved in the paper and our implementation some experiments in the paper are redone. After this comparison more elaborate experiments are executed to give a better view on the trade offs between performance and increased security. To allow others to easily reproduce or review the work that has been done all code and experiment setups are made available in open source. The performance is of course only a small aspect of the analysis of the solution, to do the security analysis, this work (and the solution of the initial paper) are compared to similar work. In this comparison it is discussed whether the solution is the most effective out of the existing ones, which other alternatives may achieve more in terms of defended attacks or achieved security guarantees. Finally it is evaluated which type of solution is the most promising as the direction for future work based on the comparison with the similar alternatives.

Outline. In the next chapter more background information about among other things Remote Attestation and ARM TrustZone will be given. In the third chapter the methods to solve the problem will be explained. In chapter four and five the implementation of the attestation program are elaborated upon and the outcome of the experiments will be made clear respectively. The sixth chapter will conclude this thesis informing the reader about related work and future improvements. The final chapter will give a discussion on the presented work.

Hoofdstuk 2

Background

2.1 Remote Attestation

Goals

Supplying evidence about a target to a verifier is the most important goal for remote attestation. [2] defines attestation as follows:

> "Attestation is the activity of making a claim to an appraiser about the properties of a target by supplying evidence which supports that claim."

Making a claim in this context refers to stating whether the target is in a secure state or not, this is often done implicitly. The appraiser can be seen as the verifier, it receives the evidence (and the claim) and will decide based on those whether the claim is valid and the target is still trusted. Remote attestation is achieved when the verifier is a remote service provider which is accessed through a network.

Requirements

The target must adhere to certain constraints to provide the necessary abilities for correct attestation. First of all a trusted base is needed that can enforce an isolation mechanism to avoid it being tampered with. This isolation will make sure that the attestation can be executed even when the target is unreliable. Another important aspect of the attestation is the ability to measure useful aspects of the system, this means that there needs to be structure to these measurements to be able to understand them. When these measurements are requested, they need to be executed in a trusted manner and the results need to be sent to the verifier securely.

The verifier needs comprehensive and fresh information about the target to be able to correctly attest it. Based on this information the verifier makes decisions on the reliability and trustworthiness of the target. These decisions are referred to as attestations, it should be possible to draw conclusions from multiple attestations or make predictions based on them. Besides a complete set of information about the

target the verifier also needs proof of the trustworthiness of this information because it is used as evidence for the decision making process.

Techniques

Integrity Measurement Architecture implements attestation by measuring the code of programs before running them on the target. The verifier can check whether the program's code has been modified based on these measurements [3] [4] [5] [6]. According to [7] and many others Integrity Measurement Architecture (IMA) is inflexible and static because updates for programs are hard to take into account. The solution is also very limited because there are a variety of ways a program can misbehave without the code base being tampered with.

Attestation on Program Execution is an important step in the right direction, it measures the dynamic behavior of the program. [8] propose to observe the system calls the program makes to verify whether it adheres to the permitted control flow. Although it is a big improvement there are still weaknesses like the granularity of a system call might not give enough details and the behavior of a system is much more than the system calls alone. Many similar solutions that focus on the dynamic behavior of code have been proposed all with their weaknesses and shortcomings [9] [10] [11] [12].

Combined strategies are being proposed more and more often due to them providing protection against a wider variety of vulnerabilities. Model-based Behaviour Attestation was proposed by [13] which is again mentioned by the same researchers in their analysis about existing techniques [7]. The platform is expected to enforce a certain security model and the attestation will verify whether the platform behaves accordingly. The behavior of the platform is monitored by a variety of techniques like IMA or Property Based Attestation (PBA) for a 'full picture' approach. PBA is focused on properties that the platform possesses, it is still very hard to map configurations of the platform to certain properties but it does provide lots of flexibility in terms of attestation. [14] have also combined a variety of approaches to attest the trust of IoT devices and implemented multiple modules that attest certain platform properties based on different measurements.

2.2 Trusted Execution Environment

Definitions

A definition of a Trusted Execution Environment is given by [15]:

> "Trusted Execution Environment (TEE) is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code,

> data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties. The content of TEE is not static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting back door security flaws are not possible."

The requirements to achieve a secure and trusted execution environment are largely accomplished by the separation kernel. This kernel simulates a distributed system which divides the system into strongly isolated partitions with different security levels. For instance data in one partition cannot be leaked by shared resources because they are sanitized and cannot be read or modified by other partitions. A partition also needs to give explicit permission before others are able to communicate with it and a security breach in a partition cannot impact any other partitions.

Trust is a very important aspect of a TEE, there are multiple types of trust with different origins. Static trust is measured only once, before deployment in most cases and assumed to never change during the lifetime of the device. Dynamic trust on the other hand is based on the state of the system and this state changes continuously. In this latter case the trust needs to be measured periodically to have an up to date view on it. To be able to do these measurements a trusted entity is required, this is because trust cannot be created but needs to be transferred from the Root of Trust (RoT) to the component that is being measured. To reach this final target, a chain will be created which links intermediate components by having the next one build upon the trust of the previous component and this is how a Chain of Trust (CoT) is constructed.

Building blocks

The Root of Trust is the starting point for the secure boot process which assures that only code with certain properties is given control. For instance during the boot process checking the integrity of the succeeding component before loading it and giving it control will construct a Chain of Trust. This CoT is necessary because that is the basis of the trust of the separation kernel. This RoT is often implemented using some hardware component that is trusted [16] [17] [18] [19].

The separation kernel is a very important component of the architecture because it is responsible for the secure scheduling and information flow control. The secure scheduling makes sure that the TEE doesn't affect the rich OS too much to allow the latter to remain responsive and meet real-time requirements. Information flow control is tightly coupled with the inter-environment communication, this is an interface which allows communication between TEE and the rest of the system. To avoid security risks from this communication it needs to adhere to the following guidelines: reliable isolation, minimum overhead and protection of the communication structures.

As shown in these papers [20] [21] [22] [23] security risks in implementations of these components do exist and they need to be dealt with to make sure the TEE operates as it is expected to.

The TEE is where Trusted Applications (TA) run, it also has a trusted kernel which is kept as minimalistic as possible. The minimalism of the kernel is to avoid software bugs which could introduce security vulnerabilities. The trusted kernel provides services like secure memory and trusted I/O which are important features to allow the system to be used in a secure manner. Secure memory ensures confidentiality, integrity and freshness of stored data. The trusted I/O protects authenticity and confidentiality of communication between TEE and peripherals.

2.3 ARM TrustZone

Core principles

ARM TrustZone [24] implements the TEE on the processor level which means it runs below the hypervisor or OS [25]. This approach divides the system into two main partitions namely the Normal World (NW) and the Secure World (SW). The processor executes either in the NW or in the SW, these environments are completely isolated in terms of hardware and the SW is more privileged to make sure the NW doesn't behave in a malicious way. The partition between these worlds gives rise to new and better security solutions for applications running on these types of System on Chips (SoC) [26] [27] [28] [29]. The SW can provide services like data storage, I/O and virtualization all with hardened security guarantees because of these hardware features.

Implementation

The Normal and Secure World are the two main environments in which the processor will be executing code. The Normal World shelters the rich OS (like Linux), it is mainly due to the size of these operating systems that they cannot be trusted to run in the secure world. The risk of there being implementation bugs that introduce security risks is too high. Also user level applications run in the NW, for peripherals for instance they rely on the rich OS and depending on the service the rich OS relies on the Secure World, some services can also be requested from the user application directly to the SW. The Secure World is where the trusted kernel runs, the implementation of this component is kept very minimal and needs to be designed and implemented very securely to avoid vulnerabilities.

The NS-bit is the 33rd bit (in a 32-bit architecture) that flows through the entire pipeline and can be read from the Secure Configuration Register (SCR) to identify the world in which the operation is being executed. The processor has a third state which is the monitor state, this is necessary to preserve and sanitize the processor state when making transitions between NW and SW states. The new privileged

instruction Secure Monitor Call (SMC) allows both worlds to request a world switch and the monitor state will make sure this is handled correctly. The only other way of getting into the monitor state is with exceptions or interrupts from the Secure World.

TZ Address Space Controller (TZASC) can be used to configure specific memory regions as secure or non-secure, such that applications running in the secure world can access memory regions associated with the normal world, but not the other way around. Making these partitions is also performed by the TZASC which is made available through a programming interface only available from within the secure world. A similar approach is taken for off-chip ROM and SRAM, this is implemented using the TrustZone Memory Adapter (TZMA). Whether these components are available or not and how fine grained the memory can be partitioned depends on the SoC because they are optional and configurable.

TZ Protection Controller (TZPC) is in the first place used to restrict certain peripherals from worlds, for instance to only allow the secure world to access them. It also extends the Generic Interrupt Controller (GIC) with support for prioritized interrupts from secure and non-secure sources. This prioritization is important to avoid Denial of Service (DoS) attacks on the secure world.

2.4 PinePhone

ARM TrustZone is used in the System on Chip (SoC) of the PinePhone which is a popular solution amongst smartphone suppliers, Samsung KNOX [30] and Android KeyStore [31] are two examples of this. It is evident that this solution has a lot of potential and the industry believes in it's potential but often the technical details are not disclosed which gives the academic world little opportunities to build upon this technology [25].

Open source smartphone is the best way to describe the PinePhone, this is a powerful tool for researchers and developers to learn how to use the ARM TrustZone framework. The PinePhone is a pioneer in this aspect because their hardware developments are all open to inspect and they are take into account the development ideas of their community [32]. Not only the hardware that is used is made 'open source' but the main operating system is Linux [33] which enables the user to control every nook and cranny of the hardware.

2.5 OP-TEE

OP-TEE stands for Open Portable TEE [34], it is an open source implementation of the API's that are exposed to Trusted Applications (TA) and that communicate with the TEE. It was designed with ARM TrustZone in mind but is applicable to other realizations of TEEs as well. The main design principles applied when creating

OP-TEE were isolation, small footprint and portability. While the two first principles seem logical and have to do a lot with the security of the final product the portability is not straight forward but a nice feature to allow a very diverse community to have a related framework which encourages collaborations.

2.6 Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes [1]

System overview

The threat model assumes that attackers have physical access to the IoT device and are able to launch a wide variety of attacks. The attackers are assumed to be able to tamper with the images of the secure and normal world (including that of the OS) before the device is booted up. Another assumed attack is one where the adversary injects malware into the normal world during runtime and tamper with the normal world applications. Only the security of the text section of a program is considered but on the other hand it is assumed that this code is on ROM that is protected from modification. Lastly the secure world and remote attestation server are assumed to be trustworthy and secure.

The solution that is proposed uses a hybrid booting method to ensure the load-time integrity and remote attestation to ensure the runtime integrity of the system. Secure boot is used to load the kernel of the secure world, this provides strong guarantees that the secure world starts in a secure and known state. The normal world is booted with what is called trusted boot which uses attestation to provide proof of the integrity of the image that is being started. Before the control is given to the rich OS it's image is measured and after it has started it should send this to the remote attestation server to verify the measurement. The remote attestation service is implemented in the secure world. The memory pages of the rich OS are periodically measured, encrypted by an attestation key and sent to the remote attestation server for verification.

Hybrid booting

Secure boot starts with a Root of Trust (RoT), which in this case is achieved by using the OCROM and eFuse of the IoT device. The first-stage bootloader is encrypted with a private key and stored on the OCROM to verify the integrity during the boot phase. The hash of the public key is stored in the eFuse to verify it's integrity during the boot phase. The images of the second-stage bootloader and secure kernel are measured and signed before deploying the device. These measurements and signatures are stored in the flash memory while the hash of the public key of the secondary bootloader is stored in the eFuse. The hash of the public key of the trusted kernel is stored in the secondary bootloader to achieve an incremental chain.

The secure boot phase starts with the first-stage bootloader which locates the second-stage bootloader, the public key and its signature. Secondly the first-stage bootloader calculates the hash of the public key and verifies the integrity of the public key. After successful verification it uses the public key to obtain the measurement result for the second-stage bootloader. Finally the first-stage bootloader calculates the hash of the second-stage bootloader and verifies its correctness. The second-stage bootloader does this entire process for the secure kernel to complete the secure boot phase.

Trusted boot is setup by producing a hash chain, the image of the rich OS is hashed first and concatenated with the image of the file system and hashed again. This final hash value is stored in the remote attestation server, during run-time this hash value will need to be sent to the remote attestation server in a secure way. To achieve secrecy a symmetric key is used, storing this in the IoT device is not trivial so this is solved with the following method. The Cryptographic Acceleration and Assurance Module (CAAM) is used to execute cryptographic functions in a secure environment. This module is used to generate a 256-bit blob key, this key is used to encrypt the attestation key. A MAC is calculated from the attestation key to ensure its integrity. The blobkey is itself encrypted with a Blob Key Encryption Key (BKEK) which is derived from the master key (MK) by the CAAM. The MK is stored in Secure Non-Volatile Storage (SNVS) which is assumed to be secure by default.

The trusted boot phase starts with the NW attestation client application establishing a TLS connection with the remote attestation server and requesting a nonce. The measurement Trusted Application (TA) restores the attestation key from the Blob using the CAAM. The TA measures the rich OS and filesystem images, appends the nonce to the final hash value and encrypts this combination with the attestation key. The encrypted text is put in shared memory to allow the client application to send it to the remote attestation server. On the remote attestation server the cyphertext is decrypted and verified to check for integrity violations and replay attacks.

Page-based attestation

The idea is to measure the code segments of the programs in the normal world on the IoT device, it is assumed that this code base does not change in the lifetime of the device. The secure world is trusted but the normal world is still vulnerable to attackers, that is why attesting the code in the normal world would increase the security for the applications running in the normal world. The measurement is done on pages of 4KB at a time so programs will have multiple tuples of the form *{process-name, page-hash}* which will later be used to verify the integrity of the process. The first measurement is done before deploying the IoT device and the results are stored on the remote attestation server to be able to compare the future measurements with.

Process integrity measurement starts with the measurement Trusted Application which resides in the Secure World, it requests the memory address of the initial process. The client application translates the virtual address of the *init_proc* into a physical address which is later translated to the virtual address in the secure world memory address space. With this address the measurement TA iterates over all processes and measures their code pages. This measuring method uses the *task_struct* which has a doubly-linked-list structure so it enables the TA to find all processes.

Process integrity attestation uses the measurement of the process integrity measurement stage. First the IoT device requests a nonce from the remote attestation server with which a TLS connection is established. The measurement TA encrypts the measurement results concatenated with the nonce using the attestation key. The measurements of the processes are encrypted individually meaning that a set of cyphertexts is sent to the remote attestation server. The remote attestation server decrypts the measurements of the processes and checks whether integrity violations can be found, this means new software or old software that has been modified.

Evaluation

The effectiveness of the secure boot process is measured by whether the secure boot phase is able to detect any violations against the integrity of the images, the signatures or the public key which it does correctly. For the trusted boot the focus lies on whether the remote attestation server is able to identify an abnormal system status, this is the case because NW programs can still be executed but the remote attestation server will verify the system state. The process integrity attestation is tested by tampering with existing programs and inserting additional programs, both these cases are also picked up on by the attestation.

Performance of the boot procedures is measured by comparing the mean time of 30 iterations with secure and trusted boot and 30 iterations without it. The secure boot adds little overhead on the second-stage bootloader while the trusted boot almost doubles the time it takes for the secure kernel to boot. The main reason why the secure kernel takes this long is because the image of the filesystem and rich OS is rather large and takes some time to measure. The overhead of the measurement TA and the attestation CA is measured by calling rich OS services while these modules are running and with them disabled. The overhead these modules introduce is between -0.55% and $+0.67\%$.

Security analysis is executed on the hybrid booting approach and the page-based process attestation. In the booting method a Chain of Trust (CoT) is constructed from the Root of Trust (RoT) residing in the eFuse and OCROM. A successful secure boot ensures that the secure world can be seen as the secure base from which the normal world can be booted. If the normal world image is tampered with the remote attestation server will pick up on this threat. The execution of the measurement TA and the results it generates are both secure because of the isolation in the secure

world. The results pass through the normal world but they are encrypted at this stage, the encryption key is also securely stored in the secure world giving the normal world no opportunity to get hold of the information. The main drawback of this approach is that the method relies on the rich OS to access the paging structure and process management kernel objects.

Hoofdstuk 3

Method

3.1 Detailed Problem

Smartphones often store, use and transmit sensitive data of their owners. Transmission will often happen on a 4G network because smartphones are frequently used on the go. This form of mobile computing implies that lots of people in the near surroundings of the smartphone can pick up on these signals. The security of these transmission methods like 4G are already extensively analyzed [35] [36] [37] but there is still room for improvements because vulnerabilities like a paging storm attack [38] or session hijacking [39] are still possible. 4G is of course not the only communication channel available on a smartphone, Bluetooth is a very popular option to connect devices to a Personal Area Network (PAN). While very popular there are still many security risks and possible attacks against it like reflection attack [40], cross technology pivoting attack (forcing a different protocol to be used) [41], Bluejacking, Bluesmacking and Bluesnarfing [42] and finally entropy downgrade attacks which make brute force attacks on the keys possible [43]. Certain security risks related to data transmission do not have a direct impact on the user but could have future implications as [44] show, they found a way to identify the smartphone OS based on smartphone traffic which could give attackers insight in what vulnerabilities to use. Although very relevant, the transmission stage is not the only option for data to leak, it still needs to be protected during storage and when it is used at execution time.

Adversaries have many possibilities when it comes to stealing sensitive data from smartphone users while the data is being used during execution or stored on the device. [45] for instance proposed AlphaLogger which infers the letters being typed on a soft keyboard based on the vibrations and [46] we're able to achieve identity theft through data cloning of auto login credentials. Besides custom attacks, there are plenty of well known risks as well. [47] have executed a review of various malicious software threats and mitigations, [48] researched software attacks that take advantage of hardware resources to conduct fault injection or sidechannel analysis and [49] reviewed the possible threats that can be introduced by smartphone providers altering the Android OS to add their signature flavor. These attacks are possible due to the

fact that the design of smartphones is based on that of IoT devices, which means that lots of focus lies on the performance of the final product. This performance is often hard to achieve because the resources want to be kept to a minimum to lower the price or keep the device small. Security is still seen as a performance killer, which is very unfortunate because it should have an essential role in the design and implementation of a system. The security of a system like smartphones is even more crucial because lots of people are unaware of the possible threats they face.

3.2 System Model

The system model is a smartphone that is owned by a user but for which multiple software providers offer programs. These providers don't necessarily trust eachother but they do want guarantees that the execution of their software will not be interfered with by software of other providers. The user of the device is also it's owner, they have full control over what software should be able to be installed and run on the platform. This is very different from large smartphone companies where the company has a signature key which is kept secret from the user of the smartphone. Programs that are not signed with this signature key will be rejected for installation on the device. The smartphone in this case is a PinePhone which is equipped with 4 ARM Cortex A53 Cores that are TrustZone enabled. Lots of smartphones have ARM processors as System on Chip (SoC) which means that ARM TrustZone is the ideal candidate for the implementation. It provides additional hardware security features which make sure that the attack surface of the smartphone becomes very narrow and well defined. While providing additional security it only has a minimal impact on performance due to it being implemented on hardware [50] [51]. ARM TrustZone makes sure that there is a Trusted Execution Environment (TEE) available with capabilities like secure memory, trusted I/O and many others. The Trusted kernel is responsible for making partitions of the memory only accessible to the secure world which should then be able to provide secure data storage services to the normal world applications. Secure data storage is necessary to make sure that data from one application cannot be read or modified by another one. Trusted I/O paths on the other hand allow the user application to request I/O features from the Secure World (SW) instead of the rich OS. Because these connection go through the SW the rich OS is not able to inspect or modify the data that is transmitted to the I/O peripheral, this is important in cases where an adversary has gained control over the rich OS. For this TEE to work correctly a trusted kernel is required, for this the implementation of OP-TEE is used which provides the interfaces to communicate with the TEE and call Trusted Applications (TA) from the normal world. The normal world runs a Mobian Linux distribution as kernel which is specifically written for a smartphone device. The system can be put together using the source of the Mobian distribution [52], OP-TEE [53], U-boot [54] and the ARM Trusted Firmware [55]. Unfortunately this was not realized in this work so the QEMU emulator [56] was used to test the code and run the experiments. QEMU allows to run OP-TEE on a desktop while emulating the TEE, because OP-TEE can run on the PinePhone

the code and experiments should be reproducible on the PinePhone if correctly configured with OP-TEE.

3.3 Attacker Model

Physical access brings along lots of risk because the adversary has a variety of possible attacks they could launch from this position. The TEE can, when combined with some specific hardware make these attacks ineffective or a lot harder to execute. For instance reading from the hardware memory becomes ineffective because everything on there that is sensitive is encrypted by the TEE, only the TEE can decrypt this information with the help of a Trusted Platform Module (TPM). Tampering with memory can be made less effective by using secure boot, this ensures that the TEE is started up in a secure and known state from which a trusted base can be ensured. With this trusted base attestation could be run on the memory to check whether inconsistent memory pages can be found before they receive control in case of them being code pages. Another hardware attack is one where a physical back door is exploited, this is assumed to be impossible because the processor has been designed for security purposes and thus no back doors should be available. The main advantage defenders have in terms of hardware attacks is that they are often very hard, reading the physical memory is doable but this should be dealt with by the TEE combined with a TPM. There are still lots of hardware attacks for which TEE's are vulnerable, manipulation of RAM and eFuse bypassing secure boot [57], micro architectural structures leaking information [58] and electro magnetic analysis of side channels [59]. These attacks are very advanced and would be really hard to execute but they do exist and not many defense mechanisms are present to protect against them.

OS/Firmware attacks have the ability to compromise all user level applications because the OS is the 'trusted' layer on which these user level applications rely. This is the main area where ARM TrustZone and other TEE implementations make a very big difference in security. ARM TrustZone for instance is implemented below the rich OS which means that the trusted kernel has more privileges than the rich OS, the rich OS has of course more functionality but to achieve this functionality it will in some cases have to rely on the trusted kernel. A TEE increases the security when an OS attack has succeeded by shielding the user level applications from this OS, this is done by allowing the user level application to request services like trusted I/O and secure memory from the TEE itself without interference of the OS. Examples of this are a container using ARM TrustZone [51], securing camera and location peripherals [60] and checking whether the OS executes system services correctly [61]. It does not make claims about making OS attacks harder because the vulnerabilities in the rich OS are still there, the normal world could be attested which would allow the detection of an OS attack but the TEE doesn't have special protection mechanisms to avoid it from happening. This is not surprising of course, OS attacks are often

a consequence of software bugs in the implementation that give rise to security vulnerabilities which are very hard (if not impossible) to avoid.

Software attacks try to tamper with the control flow of certain program executions or get hold of sensitive data through malicious code. Most of these attacks can be defended against by a TEE implementation or with the help of a TEE. Applications can be isolated from each other making it a lot more difficult for a malicious application to tamper with the execution or data of another one. This isolation can be enforced using virtualization or specialized Trusted Applications (TA), they make sure the application can only be interacted with using a very well defined interface. The virtualization needs to be implemented correctly to make sure no data is leaked in between the partitions, in the case of the TA it can use the TEE functionality to store its data securely. Software attack defense mechanisms based on ARM TrustZone come in many forms, isolate application and secure communication [62], control flow integrity scheme [63] and reverse engineering protection [64].

3.4 Solution

As discussed in the attacker model, the TEE (ARM TrustZone in this case) will provide a certain level of security on its own by providing secure and trusted services to user applications. These services can be utilized to achieve secret encryption, trusted I/O paths and secure data storage [25]. These services rely on the assumption that the TEE itself is secure and trusted, this can be achieved through secure boot [65]. Secure boot ensures that the device starts in a known secure state, to achieve this a Root of Trust (RoT) is needed from which a Chain of Trust (CoT) is constructed. The RoT is often implemented by using a Trusted Platform Module (TPM) along with hardware memory specifically designed for secure storage like an eFuse for instance. The CoT ensures that only code that is verified will be able to execute and get control over the device during boot time. When the device is successfully started up using secure boot it can be confidently assumed that the trusted kernel and TEE will work as intended. This of course is only the beginning, a TEE provides a framework which needs to be used to implement secure solutions and defense mechanisms against known attack strategies.

An important application that is built upon the TEE framework is one where the integrity of the control flow, code and data is guaranteed. Achieving this level of security is rather hard, weakening this constraint in the sense of making sure violations to this integrity are detected is achievable. Attestation can be used to check the integrity of an application or running system depending on what properties are looked at to measure the reliability of the target. Remote attestation seems like a weird solution in the context of a smartphone, but the user (which in our case is seen as the owner of the device) could be alerted about the attestation results. This attestation process can run within the TEE on the device and because of this will be tamper proof against software and OS attacks. Notifying the user will also need to

be done in a secure manner, for this the trusted I/O paths that ARM TrustZone provides can be used.

Hoofdstuk 4

Implementation

The goal for the implementation is to guarantee that the executable memory pages that are loaded into the memory are not tampered with. This integrity check is achieved by measuring the code pages of the processes in advance, for instance during the installation of the particular code. This measurement can be done in a variety of ways, in this work hashing was chosen because of it's common use and it being well understood by the community. After these initial measurements have been taken they need to be stored in a secure place, the secure memory of the PinePhone was chosen because it can guarantee the integrity and confidentiality of the data. Last but not least these measurements need to be retaken during execution time and compared with the initial measurements to verify whether the running code has been tampered with. These run time checks need to happen periodically to guarantee no modifications are introduced, or that the changes are observed before they can damage the system too much.

4.1 NW application

The Normal World (NW) application begins with looking up the processes that are currently running by examining the */proc* directory. In this directory there is a collection of directories with numbers as names, these numbers represent the process identifiers (pid). The pid is used within Linux to allow the operating system to differentiate between the different processes and manage them. It is of course necessary to attest all these processes, however the explanation will focus on just one for now to keep things clear. Take for instance the directory */proc/1*, this one will always exist due to the fact that pid 1 is associated with *proc_init* which is the initial process the Linux operating system forks. Within this directory there are two important files *pagemap* and *maps*. The *maps* file gives an overview on the different memory regions associated with the process. Besides the virtual address range other information is also present for instance whether the pages are executable or writable and where their symbolic origin is in the file structure. The *pagemap* file on the other hand is necessary to do the translation from virtual address to physical address.

The translation of the virtual addresses that are found in the *proc/pid/maps* file into physical addresses is based on a solution from [66]. The solution investigates the *proc/pid/pagemap* file and generates a *pagemap_entry* from it. This *pagemap_entry* is used to structure the information associated with one entry in the *maps* file. Based on this *pagemap_entry* the physical address can be derived from the virtual address found in the *maps* file. Do note, the *maps* file provides a contiguous virtual memory region, it is not guaranteed that the physical memory will also be contiguous so the first virtual address of every page is translated into a physical address. Pages are 4kB large so lots of virtual addresses will have to be translated into physical ones. After the physical addresses are obtained they are put together into a list and a memory reference of this list is sent to the secure world for further investigation.

4.2 Attestation PTA

The attestation Pseudo Trusted Application (PTA) receives a memory reference with inside the buffer all the physical memory addresses of the pages that need to be attested. Before the Secure World (SW) is able to access these memory pages they need to be mapped into its virtual memory space. The mapping is done using the *core_mmu_add_mapping* function from the OP-TEE kernel, when this mapping is successful the virtual address can be obtained using *phys_to_virt*. The function *phys_to_virt* returns the virtual address in the PTA where the memory can be accessed. With the TA having access to the memory pages, the actual measurements can start. The hashing algorithm used is SHA-256, this can easily be substituted by another algorithm provided in the OP-TEE library if necessary. The hashing algorithms are provided by the OP-TEE framework and easily usably from within the PTA.

In the initialization phase the hash digests are stored in the secure memory of the device. The files written from the PTA to secure memory are only accessible by this PTA and are protected against everything in the Normal World. In this file the hashes are sorted based on which *pagemap_entry* they come from and the page number within this region, this allows to later uniquely identify the hash value with which will need to be compared. After the initial hash values have been stored the initialization phase is complete. This implies that the security of the stored measurement values is guaranteed under the assumption that the secure world does indeed protect the secure memory against access (read and write) from outside the Secure World.

During the attestation phase, all the same steps will be taken as the initialization phase has taken up to this point except for storing the calculated hash values. Instead during this phase the newly calculated hash values will be compared with the hash values that are stored in the protected file where the initialization phase has written the measurement results. For the comparison the hash value and the initial value are put into 4 *uint64_t* data types each, the first one from the hash is then compared

with the first one of the initial value, the second one with the second one and so on, using the standard comparison functionality. The number of comparisons that fails is saved and printed in the debugging output stream of the secure world. Ideally the user should be notified about these faults and possibly which processes may experience an impact from this based on whether the process uses the code for which the attestation has failed. Based on the information the user receives from the attestation they can decide what action to take as the owner of the device.

4.3 Improvements and extensions

The first and probably most important improvement has to do with the dependency on the Rich OS. Looking up the virtual addresses and translating them into physical addresses needs to be done from within the NW OS at the moment, this is because the data structures that contain this information are owned by the NW OS. On top of that this functionality is only possible with root privilege because the necessary files are not accessible otherwise. If possible this should be improved because an attacker in control of the rich OS could alter these data structures to hide the changed processes from the attestation PTA. The adversary could for instance keep an unchanged process hierarchy loaded in memory while the device is actually running on a different malicious process hierarchy. As long as it cannot be guaranteed that the addresses the secure world receives are the addresses of the actual processes that are running, the attestation can be bypassed.

Secondly, the attestation currently attests the executable pages present in RAM which is sufficient but for the initialization phase all the possible executable pages need to be measured to have a reference value to compare with. To achieve that the entire code base of the process is measured during the initialization phase the binary files of the modules can be looked up. If it is possible to access these rich OS files from within the secure world they could also be attested as if they were loaded in memory. By measuring from these files the initialization phase is not bound by the executable memory pages present in RAM anymore and can attest all pages. Having an initial value for all code pages is important because measuring a memory page for the first time while the device is already deployed, there is no guarantee that the memory page hasn't been tampered with already so the initial value does not provide strong additional security guarantees.

A significant extension to allow this proof of concept to be turned into a complete solution is to notify the user using trusted I/O. When informing the user about the results of the attestation it is important to keep in mind the security of the channel on which this is achieved. ARM TrustZone provides trusted I/O paths which can be used for this goal. Trusted I/O could be used to inform the user of the problem (which program has been tampered with for instance) or it could also take on a more coarse grained approach like an LED light signal that some piece of software has failed the attestation which is easier but less useful. What details can be derived

from this communication is not necessarily of the greatest importance, it is important how this communication works. To make sure the NW cannot interfere with the communication from the attestation PTA to the user it should be implemented using the provided APIs of ARM Trusted Firmware-A (TFA) [55]. Secure I/O paths is an extensively researched topic in the field of Trusted Execution Environments, for instance to protect user data from compromised browsers [27] or to have a general trusted I/O path between the user and trusted services [67]. Using Secure I/O is necessary to build a fully functional solution based on the proof of concept presented in this chapter. The Secure I/O was not included in this work because no new functionality would be showcased and it would divert too much from the main focus of the implementation namely the attestation of the NW processes.

The second major extension focuses on the added security guarantees the attestation process provides. To achieve great security guarantees it is necessary to do more than just code attestation, lots of software attacks are based on the used data structures and don't impact the text section of programs. As discussed in the background on attestation there are a variety of methods to achieve this and a great example of an extensive attestation method is [14]. These forms of attestation are of course more complicated to execute and take a great amount of implementation effort. Due to the limited time and experience the attestation for this work was kept relatively simple to show that this form of attestation is achievable on the PinePhone. When designing a final product it is of course encouraged to use more advanced techniques to increase the security guarantees of the code execution on the device.

Lastly it was found out that in a recent framework update of OP-TEE an attestation PTA was added to the kernel [68]. The big difference between that PTA and the implementation described above is that it attests the secure world processes (TA's) while the implementation presented here is focused on attesting the processes of the Normal World. Even though there are certain differences, it is definitely possible to change the implementation 'slightly' to make it fit into this provided attestation PTA and reuse some of it's already present functionality. Before this could ever happen the first weakness of this implementation should probably be tackled namely the fact that this method still relies on the rich OS.

Hoofdstuk 5

Experiments

The main purpose of the experiments executed for this work is to provide a comparison with the paper [1] discussed in the background section about remote attestation using ARM TrustZone. To be able to compare the results of this work and that of the paper the experiments need to be made as similar as possible which will be clearly visible. Secondly for the security evaluation the solution of this work will be evaluated and the differences with the security evaluation of the paper will be highlighted.

5.1 Performance

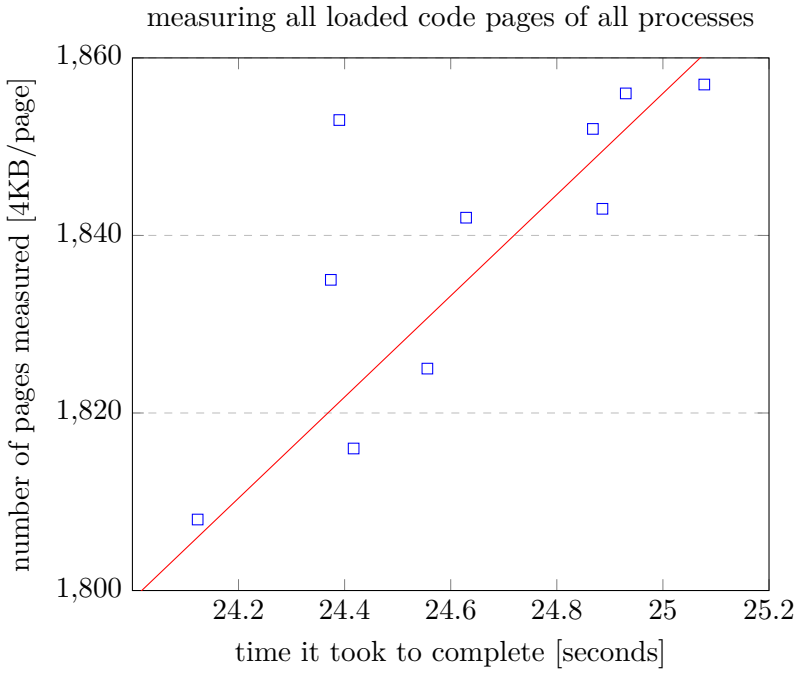
The trusted boot experiment in the paper is based on the attestation of the normal world before giving it control. In the paper they talk about 107 MB of file system image that is being measured in 1.276 seconds so this could be a valuable starting point to compare their performance with the performance achieved in this thesis. Because the implementation of this work is focused on measuring code pages there will be a difference in what is measured. While it can be somewhat assumed that the memory space of the file system in the paper is contiguous or at least the mapping can be found rather easily, the code pages in the implementation of this work are spread across multiple processes which need to be considered one by one. Looking up the right memory pages to measure them incurs additional overhead which needs to be taken into account, that is why we opted for three different experiments that relate to this trusted boot experiment from the paper. One experiment will measure all the code pages of all the processes in the simulation, the second one will measure all code pages of one process and the last experiment will measure one contiguous memory region of code pages from one process.

The overhead of running the attestation module is measured in the paper by executing system services from the Linux kernel and running this experiment with and without the attestation module being active. The results in the paper of this experiment were overhead between -0.55% and $+0.67\%$ on different system calls. These results don't give much insight in the matter in our opinion because some key

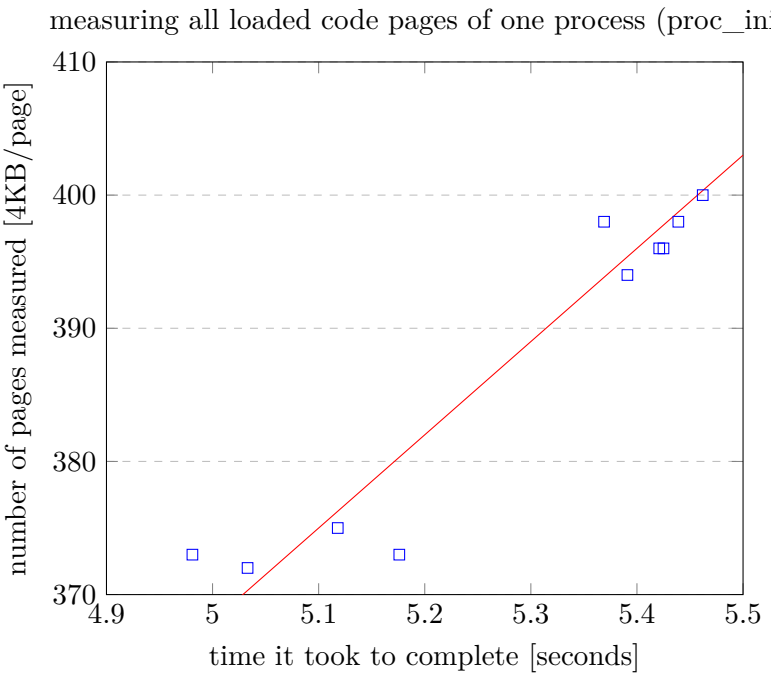
information is missing to correctly assess the results. First of all it is not mentioned how often the attestation is running while these system calls are made. Secondly it can be assumed that the attestation does not get priority over the system calls but this is also not explicitly mentioned. Last but not least, the range of -0.55% and $+0.67\%$ doesn't really tell much, this could just be standard deviation. For this experiment it was not mentioned how often it was executed which gives the impression that this is a one time measurement. They are however explicit about calling each service 1000 times, with intervals of 250 ms between each call and they do this with 7 different system services, which as they also include adds up to almost 30 minutes.

5.2 Performance Evaluation

The total amount of loaded code pages in RAM is around 1850, the time it takes to measure those twice (once for initialization and once for attestation) is about 25 seconds. A plot is shown which provides the experiment being executed 10 times, also a linear regression line can be found to provide the general correlation between the amount of pages measured and the time it takes.

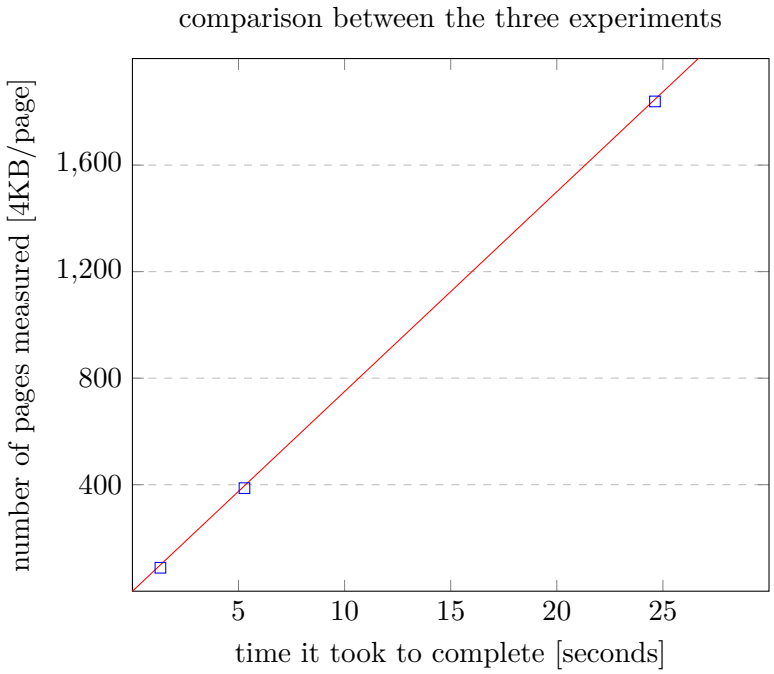


To provide insight in how long it takes to measure one process (twice) we executed the last experiment on just the *init_proc* process.



In the first two experiments the amount of pages that are loaded into RAM fluctuated a bit, this probably has to do with pages being swapped in and out of memory. In this last experiment only one executable memory region of the *init_proc* is measured and the amount of pages stays constant for all 10 iterations. The amount of pages measured is 88 and the average execution time is 1.319 seconds, the maximum execution time is 1.353 and the minimum 1.299.

Last but not least the three experiments are plotted in one graph to show how they relate to each other. This plot clearly shows that the amount of time the measurements take is proportional to the amount of pages that are measured. No surprises here because hashing the memory pages is the most compute intensive task of the program but it is good to compare between the different experiments nonetheless. This allows us to compare our achieved performance to that of the paper because we can extrapolate the results to the necessary amount that was measured in the paper.



As can be seen in the three executed experiments, the performance of the implementation of this work is far from that of the paper. Certain aspects are important to highlight which give a partial explanation about the cause of these deviations. First of all the implementation is not written nor fine tuned for performance, it is merely written as a proof of concept. This is due to the restricted amount of time and the lack of experience with making software with high performance. Secondly the memory regions that are attested in this work are only the executable memory pages of a process. These pages need to be identified using multiple files which generally takes more time than in the case that there is one large chunk of memory that is attested. Lastly these experiments are executed on a Qemu emulation of OP-TEE on a general purpose laptop (Lenovo Thinkpad P50) while the experiments in the paper are executed using a hardware prototype.

While it is hard to make any conclusions based on these pessimistic results, it is useful to elaborate about the balance between performance and added security. A good balance between performance overhead and security assurance depends on the use case. In case of a smart phone we believe even with these results that attesting the executable pages that are loaded in RAM memory every 30 minutes has not too much impact on the user experience. Depending on the amount of newly loaded memory pages when an application is started it could even be considered to attest these pages before starting to execute them. This is a lot more sensitive towards the user experience because this happens while the user is waiting for the application to open, while on the other hand running the attestation in the background on the already loaded pages in RAM should have minimal impact on the user. These

considerations do not take into account the energy consumption of running the attestation, because we didn't have the means for these kinds of experiments.

5.3 Security Properties

The attestation method presented in this work focuses on measuring the code pages of processes loaded in RAM. A code page can only gain control or be executed when it is loaded in RAM first. Keeping this prerequisite in mind it should suffice to only attest those pages to ensure no code page that has been tampered with gets to be executed. This last statement of course needs to be loosened a bit because the attestation runs periodically so a code page may have been executed before the changes are noticed by the attestation PTA.

Integrity of the measurement execution is of utmost importance when it comes to attestation. In remote attestation the critical code runs on a hardened server which guarantees that it is infeasible to tamper with the execution control flow. In the case of the PinePhone it is not remote attestation we present but user attestation, the critical code which compares the measurements runs in the Secure World. The Trusted Execution Environment provided by ARM TrustZone does guarantee that the Normal World cannot influence the execution of the Secure World in any way as long as the device was started up successfully using secure boot.

Secure storage of results is very important to have a chance at making remote attestation work. If an adversary were to be able to tamper with these values they could make every attestation attempt fail due to the changed initial value. In case a bad hashing algorithm is chosen and the attacker is able to read the initial hash digest they could also try to forge a collision attack where they tamper with a page in such a way that it's hash digest doesn't change but the code in the page does something the attacker desires. To make sure the reference values are not tampered with they need to be stored in secure memory which should not be accessible from outside the secure world. In the secure world these values should only be written during the initialization phase and afterwards only read. This statement can again be loosened a bit in case there are software updates or additional software to be installed on the device. The initialization phase could be executed again for those code pages and updating or adding the measurement results to the secure storage.

5.4 Security Evaluation

Security guarantees that can be made are the integrity of the measurement execution control flow and the integrity and confidentiality of the results that are being stored. These are achieved due to secure boot enabling the trusted execution environment of ARM TrustZone. These are key assumptions in the field of remote attestation and thus are also very important in this case of user attestation. Of course when discussing the security guarantees a certain solution offers it is not only

about the execution and storage of that application itself but also what additional security guarantees it provides to the system overall. The most important guarantee that an attestation solution tries to provide is being able to detect modifications or unexpected possibly malicious changes to the aspects of the system it attests. In our case the code pages of processes are being attested, we only measure the ones that are present in RAM because only then they are able to do harm to the system. There is still an unsolved problem as also stated in the paper that the rich OS needs to provide the physical addresses of the code pages. This means that malware capable of self hiding or transient root kits are still possible threats that could stay unnoticed to this solution. If an adversary has taken control of the OS there are countless ways in which the attacker could deny the attestation PTA the necessary addresses which at this moment would not result in an attestation alert to the user.

OS/firmware attacks are present in the attacker model of the paper on which this work is based. We believe that with this method the code pages of all processes including those of the rich OS can be attested and this will enable the user to be notified about any tampering with this code base. This does not mean however that tampering with these code pages has become harder and besides tampering with code there are lots of different methods to perform an OS/firmware attack. We believe that this work can be extended upon to thoroughly attest the Normal World (user applications and rich OS) but in it's current state it only allows detection of a small portion of the possible OS/firmware attacks. Even with the problem that the rich OS provides the physical memory addresses solved this incompleteness persists.

The paper also seems to claim that software attacks are protected against or detectable in the case of attestation. Again in this case we do not think this is a valid statement because the integrity of the code is only a small portion of the attack surface. In this attestation method the data structures are not checked which are the main target for very well known attacks that have been around for decades like buffer overflows and return oriented programming. Extensions to this solution need to be realized to detect software attacks, OS/firmware attacks likewise.

Hoofdstuk 6

Discussion

6.1 Related work

The paper on remote attestation, secure and trusted boot on IoT Nodes [1] is the paper on which the implementation and experiments of this thesis are based and is extensively explained in the background section. To provide a better understanding of the effectiveness of the solution it is compared to three related works. Before this comparison these three papers are briefly explained while still providing in depth explanations on the parts that are most relevant for the comparison. The main focus of this comparison will be on the added security guarantees and the assumptions that have been made with respect to the remote attestation. After these comparisons trade offs, weaknesses and possible improvements will be discussed in the future work.

SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE

The solution described in the paper about SecTEE [69] aims to implement a framework using ARM TrustZone to achieve similar security guarantees as a hardware-based secure enclave architecture. The writers provide the following contributions with their paper. First of all SecTEE the new secure enclave architecture which achieves 'the hightest level of security' for ARM platforms using ARM TrustZone. Secondly a locking mechanism is introduced which makes sure enclave pages cannot be accessed while the enclave is running to prevent cross-core side channel attacks. The TEE OS is also extended to provide functionality like identification, remote attestation and sealing sensitive data. Lastly an implementation of SecTEE based on OP-TEE is provided along with experiments showing the performance.

Their wording of 'the highest level of security' means that it should be resistant to privileged host software attacks, board-level physical attacks, page fault based side-channel attacks and cache based side-channel attacks. The hardware attacks on which is focused are cold boot attacks [70], bus monitoring attacks [71] and Direct

Memory Access (DMA) attacks. Attacks against the internal state of the SoC are not considered because those are assumed to be very sophisticated and require expensive equipment. For these goals to be achieved certain requirements are listed. For one thing a Device Sealing Key (DSK) needs to be present, this is a symmetric key only known by the device itself and used to protect secrets related to the device. As follows a Device Root Key (DRK) which is an asymmetric key pair and is necessary to identify and authenticate the device. Lastly the Manufacturer's Public Key needs to be hard coded on the device to make sure it is able to verify the signature on software updates from the manufacturer.

An important aspect of the SecTEE architecture is the method that is applied for memory protection. SecTEE protects enclaves from physical attacks by using a similar approach as the OP-TEE pager. It is an on-demand paging system which runs the entire TEE system on On Chip Memory (OCM). Whenever a page leaves this OCM it is encrypted to ensure confidentiality and integrity of the data while it is stored on the DRAM. Another key feature of SecTEE is it's side-channel resistance. Side-channel attacks from the secure world are avoided by using a page coloring mechanism. Different enclaves can never share the same cache set which ensures that one enclave will never be able to evict cache lines of another one. Of course side-channel attacks can also come from the Normal World (NW) especially because the NW and Secure World (SW) share the same cache in the ARM TrustZone architecture. In the NW there are of course certain limitations to what is possible with the cache lines due to privileges, the prime and probe method is still relevant in this case though. SecTEE cleans and invalidates all cache levels when the CPU switches from the SW to the NW. This however is not sufficient because cross-core side-channel attacks can launch the attack while the CPU is still executing in the SW. To handle this problem the cache set of the enclave is locked which guarantees that the cache of the enclave cannot be probed or manipulated by the NW. Some final aspects that deserve some highlighting are enclave identification, measurement and remote attestation in SecTEE. Enclaves are published along with the public key of the author and the signed integrity value of the image. With this information the enclave can be verified and identified before it is run. During run-time the enclave keeps track of important aspects like enclave ID, integrity and a flag indicating whether it is privileged. For the remote attestation a specific Quoting Enclave is created, this is a privileged enclave which is able to make the system calls related to the management of attestation keys, other enclaves are not allowed to do this. These keys are used to sign the report data of the attestation together with the run-time measurement to provide proof to the verifier that it comes from the correct enclave.

First of all the number of lines of code are measured because these implementations extend the TEE kernel which is the Trusted Computing Base (TCB) and this needs to be kept minimal because bugs could easily introduce security vulnerabilities. Next the overhead of the trusted computing features is identified and discussed. This overhead is acceptable in case (Eliptic Curve Cryptography) ECC with keys of 256

bits are used, in case RSA keys are used of 2048 bits certain system calls take too long to be usable. The performance of certain enclaves was measured and the xtest benchmark of OP-TEE were executed. SecTEE was about 4 times slower than OP-TEE on the benchmark and 40 times slower than OP-TEE on the enclave execution. It is argued that the memory protection mechanism is the cause of the greatest performance overhead in SecTEE. Finally to test the effectiveness of the side channel defense the effectiveness of the attack on plain OP-TEE is compared to the case of SecTEE. In the case of a NW attack there is a clear difference between prime and probe timings in OP-TEE while both memory accesses take an equal amount of time in case of SecTEE which makes it impossible for an attacker to learn anything about the cache behavior of the victim enclave. For the SW attack an AES key could be recovered after 256,000 encryptions in OP-TEE while in SecTEE the attacker could not learn any information about the used key.

TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone

TrustShadow [61] is designed to protect existing applications from untrusted Operating Systems (OS) without the need to modify the applications themselves. This is achieved by using TrustShadow as a lightweight runtime system that shields the sensitive information of the application from the OS. This system does not provide system services itself but requests them from the OS, it does however guarantee security through encryption, context switching and verifying return values. To ensure that the OS cannot get hold of the encryption keys, sensitive data or interfere with the security verification TrustShadow runs in the Secure World of the ARM TrustZone processor.

The solution described in the paper about TrustShadow is mainly focused on protecting applications from an untrusted OS. This means that the attacker is able to launch Direct Memory Access (DMA) attacks, modify interrupted process states, change program execution control flow, hijack system services and control communication channels [72]. Denial of Service (DoS) attacks and side-channel attacks such as timing and power analysis are not taken into account in the solution.

To make sure the rich OS is not able to tamper with the memory of the applications and the runtime system the memory of the different stakeholders is isolated from each other. TrustShadow utilizes the Trust Zone Address Space Controller (TZASC) to make three partitions in memory, one unprotected for the applications, one protected for the applications and one for the runtime system itself. The virtual memory space is equally distributed between the OS and the runtime system TrustShadow, and the runtime system also maps the physical address space holding the OS to efficiently locate shared memory. Another aspect that needs to be taken into account is that the OS is used to handle exceptions. When an exception is thrown the state of the application is stored and the state for the OS is restored. The runtime system then reproduces the exception in this OS state to allow the OS to handle it without

having to get sensitive details about the application. There are two exceptions that the runtime system implemented in the secure world and those are the random number generation and floating point operations. These two exceptions are security critical because the used cryptography relies on the security of these operations. Similar to exceptions are system calls, the OS needs some information about the application requesting the system call but TrustShadow sanitizes this information to make sure only the necessary data is received by the OS. After receiving the result from the system call the runtime system also checks this result on correctness before forwarding it to the application. Another example of sensitive information about an application that an untrusted OS should not have access to is the memory mapping from virtual to physical address space. This information is stored in the page tables, TrustShadow makes use of the OS page fault handler to update and retrieve the page table entries that are stored in the secure world. This ensures that the rich OS does not require the information because it is controlled by the runtime system. After the memory location of a page has been found this can be loaded for execution, before this loading is finished however the page is checked on it's integrity. TrustShadow verifies the executable pages of an application and the executable pages of the shared libraries based on a manifest file related to the application.

TrustShadow was evaluated with a variety of measuring methods. Microbenchmarks using LMBench, file operations using Sysbench, HTTP and HTTPS request handling using Nginx and finally the performance overhead on data analytic applications. The performance overhead caused by the runtime system was negligible across all tests. In terms of security the OS still has a lot of control, it could for instance invoke a system call with the original *execve* instead of the new *tz_execve* which largely bypasses TrustShadow. Another concern is the possibility to manipulate or forge manifest files which could allow an attacker to tamper with the code of an application without the runtime system being able to detect it.

TZ-MRAS: A Remote Attestation Scheme for the Mobile Terminal Based on ARM TrustZone

In the paper [73] a TrustZone based Mobile Remote Attestation Scheme (TZ-MRAS) is proposed. There are three main challenges the writers tackle, the first one is building a root of trust for measuring, storage and reporting of the attestation results. The second challenge is to execute binary-based attestation on hardware constrained devices while defending against Time Of Check to Time Of Use (TOC-TOU) attacks. The last challenge consists of the performance optimization of storing measurement logs in a hash tree which has considerable overhead during the construction and updating phases.

The assumptions of the TrustZone model are followed meaning that the processor is trusted and a chain of trust is achieved to trusted applications. The secure world is the trusted computing base (TCB) and it provides features to attest the integrity of the normal world. The attacker on the other hand is assumed to have kernel level

access meaning they can tamper with the REE kernel and applications in any way they desire. DoS and side-channel attacks are not considered in this paper.

To tackle the first challenge related to the root of trust, the functionality of a TPM chip is provided by the TEE of TrustZone. These functionalities are implemented in TA's running in the secure world to ensure isolation from the normal world. The measurement module keeps track of REE and dynamically detects integrity changes in the kernel but can also update the Secure Measurement Log (SML). Secondly the TOC-TOU attacks are protected against by updating the SML with a dynamic update mechanism, namely ProbeIMA. Probes are added to all operations that are able to modify the code pages of the kernel meaning that any attempt to execute any of these operations will be detected. Lastly the structure of the SML is optimized by storing the root node in a hardware protected register and the rest of the hash tree in secure memory. On top of the storage also the design is related to the construction, insertion and revocation algorithms avoid too frequent construction and update operations.

The performance overhead in terms of execution time difference is only touched on shortly but it is less than 10% in the worst case so seems to be very acceptable. Besides the performance overhead the security analysis is backed up with experiments that show the solution works and is able to detect the attacks used. The simulated attacks are a rootkit that hijacks the kernel function of the system call table and a TOC-TOU attacks that modifies the code pages and restores them again in between integrity checks. Lastly the depth of the hash tree is measured for a varying amount of updated nodes, this is mainly to provide proof that the construction and updates on those trees have a limited impact on performance.

6.2 Comparison of Approaches

Effectiveness

All solutions focus on measuring the binaries of the executable files to verify the integrity of the runtime system. This is achieved by having a TA in the secure world which does the measurements, stores the results and encrypts them before sending it to the remote attestation server. Running the attestation shields the attestation process from being tampered with by the normal world as they all claim but it does not guarantee that the execution control flow of the attested applications cannot be tampered with. SecTEE seems to keep track of more data for the attestation process but nothing is said about the integrity of data structures or other critical parts of the system.

Except for the paper on SecTEE all solutions have a very similar goal and their solutions are also very comparable. All three other solutions (TrustShadow, TZ-MRAS and the one described in this paper) mainly focus on the attestation of

normal world processes but these are not very well shielded or isolated from the rich OS. SecTEE uses remote attestation as one of the many tools to protect the applications or at least identify violations thereof. Due to the effort the writers of SecTEE put into the memory protection scheme, the page coloring mechanism and the way enclaves need to be implemented it achieves better security guarantees for the enclaves. An enclave has to be implemented as a TA, which in and of itself introduces additional security guarantees like it being infeasible for the rich OS to tamper with the execution control flow of the enclave.

None of the solutions mentioned give much attention to the openness of the secured system which is one of the main aspects at the beginning of this thesis. The solution in this thesis differs from [1] in the sense that the initial measurement results are stored in the secure memory of the secure world instead of at a remote attestation server. Storing these values on the device itself allow for the solution to be extended to allow the user to attest the device. This should provide the same security guarantees because all solutions described here rely on the secure memory of ARM TrustZone. On top of that it avoids the trusted third party that needs to be convinced the integrity of the software running on the device is not violated. The only party that needs to be assured that the integrity of the system is preserved is the user who can be notified through trusted I/O. If software providers also need this guarantee the attestation solution could shut down applications automatically for which the attestation fails instead of giving the user the choice.

Assumptions

All solutions assume an IoT device or smartphone device meaning the solutions need to be adapted to devices with limited hardware specifications. Based on this assumption all compared solutions chose for ARM SoC with TrustZone capabilities. In this context the secure world of ARM TrustZone along with its TA's are all assumed to be trusted and secure and isolated from the rich OS. In the case of SecTEE the additional assumption is made that the enclaves (isolated applications) are written as Trusted Applications which may be feasible for an open platform but when these enclaves need to be signed by a manufacturer's private key it achieves an entirely different goal. On the other hand the other solutions claim to attest the normal world and provide similar protection against the rich OS which is unfeasible if only the code pages of an application are measured for integrity violations.

SecTEE claims to protect against certain side-channel attacks and hardware attacks like DMA and cold boot attack. These claims seem to be met due to the secure world being instantiated in a trusted manner with secure boot and the protected code running in the secure world. The papers about TrustShadow and TZ-MRAS acknowledge that side-channel attacks are not dealt with but DMA attacks are prevented by protecting the address space mapping and encrypting the stored data. The reproduced solution does not give specific details about side-channel attacks or DMA attacks it only mentions hardware, software and OS attacks. These claims

seem excessive because only the attestation TA itself is protected against these kinds of attacks, due to the TEE and trusted boot. The applications in the normal world are still vulnerable to many attacks of which only a few (changing the executable code of the application) are detected by the solution. Last but not least OS attacks cannot be prevented using this method, an attack can in some cases (depending on what is modified) be detected with remote attestation but none of the solutions make an effort to make attacking the OS harder because it is in most cases not considered when discussing the applications running in the normal world.

6.3 Future Work

Trade offs

One important sensitivity point is to decide whether every page in an executable memory region gets its own hash value or all the pages in this memory region get one combined hash value. The fine-grained option allows to only measure the code pages that are currently loaded into the RAM memory while otherwise these unused pages also need to be loaded to measure them. The downside to this is of course that more initial values need to be stored to allow the comparison during the attestation to take place.

Another trade off that could be put forth is the hashing algorithm that is used to measure the memory pages. Of course the hashing algorithm needs to adhere to certain standards to avoid collisions,... but a Hash-based Message Authentication Code (HMAC) algorithm that uses a key during the hashing process is definitely excessive. The hashing mainly needs to be fast because lots of memory pages need to be attested and this needs to happen relatively often to guarantee improved security. The hash digests do not need to be secure in the sense that they are merely used to detect whether the hashed information has changed or not.

Weaknesses

In the proposed solution there is still a strong dependency upon the rich OS. This dependency mainly comes from the files that are used to determine the physical addresses of pages related to the processes that are measured. These files are part of the data structures the rich OS owns, a solution to this problem could be to move the responsibility of the memory mapping into the secure world. The downside to this is that it increases the TCB because the secure world is often seen as the Trusted Computing Base. This can be achieved in a relatively light weight fashion as demonstrated in the TrustShadow paper where the page table is stored inside the secure world and the secure world handles page faults that are generated in the rich OS.

Partial attestation introduces a fake sense of security because not all possible attacks are checked. We have mentioned that the execution control flow of an

application can be tampered with through the manipulation of data structures. There are lots of ways similar attacks can be achieved to make applications behave differently than intended. Some of these attacks are protected against better than others in the last few solutions like checking the results of a system call executed by the rich (untrusted) OS. [14] describes a variety of possible aspects for which integrity checks are useful to increase the level of trust one can have in an application. This is probably harder to achieve when trying to protect against a rich OS like Linux compared to the Contiki OS that was used but it looks like a promising direction for future work.

Hoofdstuk 7

Conclusion

Solution overview

Reproduction of the solution provided in the paper is the starting point of this thesis.

Extensions on this reproduced solution are necessary because the original solution does not achieve all goals.

Shortcomings

Attacker possibilities that are not accounted for are still present, the security measures taken in the solution are not adequate for the wide variety of attacks that it claimed to protect against.

Desired guarantees the solution should be able to provide are not entirely met.

Positives

Code for this thesis is made available as open source code to make it easier to reproduce the experiments and continue work on this research topic by other researchers.

Thorough comparison has been executed on the provided solution and solutions of related work to give an overview of what direction is most promising to achieve the security goals.

Bijlagen

Bibliografie

[1] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for arm trustzone-based iot nodes," *Journal of systems architecture*, vol. 119, p. 102240, 2021.

[2] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International journal of information security*, vol. 10, no. 2, pp. 63–81, 2011.

[3] A. Yu and D. Feng, "Bbacima: A trustworthy integrity measurement architecture through behavior-based tpm access control," *Wuhan University journal of natural sciences*, vol. 13, no. 5, pp. 513–518, 2008.

[4] T. Jaeger, R. Sailer, and U. Shankar, "Prima: policy-reduced integrity measurement architecture," in *Proceedings of the eleventh ACM symposium on access control models and technologies*, SACMAT '06, pp. 19–28, ACM, 2006.

[5] J. Duan, G. Cai, K. Xu, and S. Guo, "Integrity measurement based on tee virtualization architecture," in *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, pp. 370–376, IEEE, 2020.

[6] M. Kucab, P. Borylo, and P. Cholda, "Remote attestation and integrity measurements with intel sgx for virtual machines," *Computers & security*, vol. 106, p. 102300, 2021.

[7] M. Alam, T. Ali, S. Khan, S. Khan, M. Ali, M. Nauman, A. Hayat, M. Khurram Khan, and K. Alghathbar, "Analysis of existing remote attestation techniques," *Security and communication networks*, vol. 5, no. 9, pp. 1062–1082, 2012.

[8] L. Gu, X. Ding, R. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM workshop on scalable trusted computing*, STC '08, pp. 11–20, ACM, 2008.

[9] Y. Qin, J. Liu, S. Zhao, D. Feng, and W. Feng, "Ripte: Runtime integrity protection based on trusted execution for iot device," *Security and communication networks*, vol. 2020, 2020.

[10] T. Ali, R. Ismail, S. Musa, M. Nauman, and S. Khan, "Design and implementation of an attestation protocol for measured dynamic behavior," *The Journal of supercomputing*, vol. 74, no. 11, pp. 5746–5773, 2017.

[11] B. Stelte, R. Koch, and M. Ullmann, "Towards integrity measurement in virtualized environments - a hypervisor based sensory integrity measurement architecture (sima)," in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, pp. 106–112, IEEE, 2010.

[12] H. Ba, H. Zhou, J. Ren, and Z. Wang, "Runtime measurement architecture for bytecode integrity in jvm-based cloud," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, vol. 2017-, pp. 262–263, IEEE, 2017.

[13] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert, "Model-based behavioral attestation," in *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, SACMAT '08, pp. 175–184, ACM, 2008.

[14] J. T. Mühlberg, J. Noorman, and F. Piessens, "Lightweight and flexible trust assessment modules for the internet of things," in *Computer Security – ESORICS 2015*, vol. 9326 of *Lecture Notes in Computer Science*, pp. 503–520, Cham: Springer International Publishing, 2016.

[15] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, IEEE, 2015.

[16] T. MATSUMOTO, M. IKEDA, M. NAGATA, and Y. UEMURA, "Secure cryptographic unit as root-of-trust for iot era," *IEICE transactions on electronics*, vol. E104.C, no. 7, pp. 262–271, 2021.

[17] G. Fotiadis, J. Moreira, T. Giannetsos, L. Chen, P. B. Rø nne, M. D. Ryan, and P. Y. A. Ryan, "Root-of-trust abstractions for symbolic analysis: Application to attestation protocols," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 13075 of *Lecture Notes in Computer Science*, pp. 163–184, Cham: Springer International Publishing, 2021.

[18] S. Kinney, *Trusted platform module basics: using TPM in embedded systems*. Mbedded technology series, Amsterdam ; Boston: Elsevier Newnes, 2006.

[19] S. Zhao, J. Lin, W. Li, and B. Qi, "Research on root of trust for embedded devices based on on-chip memory," in *2021 International Conference on Computer Engineering and Application (ICCEA)*, (Piscataway), pp. 501–505, IEEE, 2021.

[20] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, pp. 1741–1758, ACM, 2019.

[21] P. Guo, Y. Yan, C. Zhu, and J. Wang, "Research on arm trustzone and understanding the security vulnerability in its cache architecture," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12382 of *Lecture Notes in Computer Science*, pp. 200–213, Cham: Springer International Publishing, 2021.

[22] F. Khalid and A. Masood, "Hardware-assisted isolation technologies: Security architecture and vulnerability analysis," in *1st Annual International Conference on Cyber Warfare and Security, ICCWS 2020 - Proceedings*, pp. 1–8, IEEE, 2020.

[23] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Krügel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *NDSS*, ndss symposium, 2017.

[24] ARM Ltd., "TrustZone trustzone for cortex-a," 2022. Last accessed on 26th of April 2022.

[25] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys*, vol. 51, no. 6, pp. 1–36, 2019.

[26] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *MobiSys 2018 - Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pp. 1–13, ACM, 2018.

[27] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. GarciaJr, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh, "Fidelius: Protecting user secrets from compromised browsers," 2018.

[28] R. Chang, L. Jiang, W. Chen, Y. Xiang, Y. Cheng, and A. Alelaiwi, "Mipe: a practical memory integrity protection method in a trusted execution environment," *Cluster computing*, vol. 20, no. 2, pp. 1075–1087, 2017.

[29] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on operating systems principles*, SOSP '17, pp. 287–305, ACM, 2017.

[30] Samsung, "KNOX stay connected, protected, and productive," 2022. Last accessed on 26th of April 2022.

[31] Google, "Android keystore system," 2022. Last accessed on 26th of April 2022.

[32] PINE64, "PINE64 pinephone," 2021. Last accessed on 26th of April 2022.

[33] PINE64, "PinePhone software releases," 2022. Last accessed on 26th of April 2022.

[34] OP-TEE, "OP-TEE about op-tee," 2022. Last accessed on 26th of April 2022.

[35] M. A. Ferrag, L. Maglaras, A. Argyriou, D. Kosmanos, and H. Janicke, "Security for 4g and 5g cellular networks: A survey of existing authentication and privacy-preserving schemes," *Journal of network and computer applications*, vol. 101, pp. 55–82, 2018.

[36] R. M. Zaki and H. B. A. Wahab, "4g network security algorithms: Overview," *International journal of interactive mobile technologies*, vol. 15, no. 16, pp. 127–143, 2021.

[37] J. K. Fadhil, G. K. Zrar, and M. H. S, "Analysis of encryption algorithms proposed for data security in 4g and 5g generations," in *ITM web of conferences*, vol. 42, p. 01004, EDP Sciences, 2022.

[38] K. Fang and G. Yan, "Paging storm attacks against 4g/lte networks from regional android botnets: rationale, practicality, and implications," in *Proceedings of the 13th ACM Conference on security and privacy in wireless and mobile networks*, WiSec '20, pp. 295–305, ACM, 2020.

[39] Y.-H. Lu, C.-Y. Li, Y.-Y. Li, S. Hsiao, T. Xie, G.-H. Tu, and W.-X. Chen, "Ghost calls from operational 4g call systems: Ims vulnerability, call dos attack, and countermeasure," in *Proceedings of the 26th Annual International Conference on mobile computing and networking*, MobiCom '20, pp. 1–14, ACM, 2020.

[40] T. Claverie and J. L. Esteves, "Bluemirror: Reflections on bluetooth pairing and provisioning protocols," in *2021 IEEE Security and Privacy Workshops (SPW)*, pp. 339–351, IEEE, 2021.

[41] R. Cayre, G. Marconato, F. Galtier, M. Kaâniche, V. Nicomette, and G. Auriol, "Cross-protocol attacks: Weaponizing a smartphone by diverting its bluetooth controller," in *WiSec 2021 - Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 386–388, 2021.

[42] N. Patel, H. Wimmer, and C. M. Rebman, "Investigating bluetooth vulnerabilities to defend from attacks," in *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, (Piscataway), pp. 549–554, IEEE, 2021.

[43] D. Antonioli, N. Tippenhauer, and K. Rasmussen, "Key negotiation downgrade attacks on bluetooth and bluetooth low energy," *ACM transactions on privacy and security*, vol. 23, no. 3, pp. 1–28, 2020.

[44] Y. Zhu, N. Ruffing, J. Gurary, Y. Guan, and R. Bettati, "Towards smartphone operating system identification," *IEEE transactions on dependable and secure computing*, vol. 18, no. 1, pp. 411–425, 2021.

[45] A. R. Javed, M. O. Beg, M. Asim, T. Baker, and A. H. Al-Bayatti, "Alphalogger: detecting motion-based side-channel attack using smartphone keystrokes," *Journal of ambient intelligence and humanized computing*, 2020.

[46] W. Song, M. Jiang, H. Yan, Y. Xiang, Y. Chen, Y. Luo, K. He, and G. Peng, "Android data-clone attack via operating system customization," *IEEE access*, vol. 8, pp. 199733–199746, 2020.

[47] R. Setyawan, A. A. Rahayu, K. F. Nur Annisa, and A. Amiruddin, "A brief review of attacks and mitigations on smartphone infrastructure," *IOP Conference Series: Materials Science and Engineering*, vol. 852, no. 1, p. 12141, 2020.

[48] J. GRAVELLIER, "Remote hardware attacks on connected devices," 2021.

[49] S. Kumar, L. J. Kittur, and A. R. Pais, "Attacks on android-based smartphones and impact of vendor customization on android os security," in *Information Systems Security*, vol. 12553 of *Lecture Notes in Computer Science*, pp. 241–252, Cham: Springer International Publishing, 2020.

[50] J. Amacher and V. Schiavoni, "On the performance of arm trustzone: (practical experience report)," in *Lecture Notes in Computer Science*, vol. 11534 of *Distributed Applications and Interoperable Systems*, pp. 133–151, Springer International Publishing, 2019.

[51] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "Tz-container: protecting container from untrusted os with arm trustzone," *Science China. Information sciences*, vol. 64, no. 9, 2021.

[52] Linux community, "Mobian sunxi64-linux," 2022. Last accessed on 29th of April 2022.

[53] OP-TEE, "OP-TEE optee_os, optee_client," 2022. Last accessed on 29th of April 2022.

[54] DENX Software Engineering, "Pine64 u-boot," 2022. Last accessed on 29th of April 2022.

[55] ARM-Software, "ARM Software arm trusted firmware," 2022. Last accessed on 29th of April 2022.

[56] OP-TEE, "OP-TEE qemu v8," 2022. Last accessed on 29th of April 2022.

[57] M. Gross, N. Jacob, A. Zankl, and G. Sigl, "Breaking trustzone memory isolation and secure boot through malicious hardware on a modern fpga-soc," *Journal of cryptographic engineering*, 2021.

[58] 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*, CCS '19, pp. 181–194, ACM, 2019.

[59] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay, "How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip," in *Information Security Theory and Practice*, vol. 10741 of *Lecture Notes in Computer Science*, (Cham), pp. 93–109, Springer International Publishing, 2018.

[60] A. S. Salman and W. K. Du, "Securing mobile systems gps and camera functions using trustzone framework," in *Intelligent Computing*, vol. 285 of *Lecture Notes in Networks and Systems*, pp. 868–884, Cham: Springer International Publishing, 2021.

[61] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *MobiSys 2017 - Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pp. 488–501, ACM, 2017.

[62] D. Zhang and S. You, "iflask: Isolate flask security system from dangerous execution environment by using arm trustzone," *Future generation computer systems*, vol. 109, pp. 531–537, 2020.

[63] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m," *International journal of parallel programming*, vol. 49, no. 2, pp. 216–236, 2020.

[64] R. Ben Yehuda and N. J. Zaidenberg, "Protection against reverse engineering in arm," *International journal of information security*, vol. 19, no. 1, pp. 39–51, 2019.

[65] H. Jiang, R. Chang, L. Ren, and W. Dong, "Implementing a arm-based secure boot scheme for the isolated execution environment," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, vol. 2018-, pp. 336–340, IEEE, 2017.

[66] cirosantilli, "common_userland.h," 2022. Last accessed on 16th of May 2022.

[67] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems, APSYS 2014*, APSys '14, pp. 1–7, ACM, 2014.

[68] D. Harbin, "Trusted Firmware OP TEE v3.17.0 release," 2022. Last accessed on 16th of May 2022.

[69] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, pp. 1723–1740, ACM, 2019.

[70] M. Huber, B. Taubmann, S. Wessel, H. P. Reiser, and G. Sigl, "A flexible framework for mobile device forensics based on cold boot attacks," *EURASIP Journal on Multimedia and Information Security*, vol. 2016, no. 1, pp. 1–13, 2016.

[71] ncc group, "Tpm genie: Interposer attacks against the trusted platform module serial bus," 2018. Last accessed on 25th of May 2022.

[72] C. Qiang, W. Liu, L. Wang, and R. Yu, "Controlled channel attack detection based on hardware virtualization," in *Algorithms and Architectures for Parallel Processing*, vol. 11334 of *Lecture Notes in Computer Science*, pp. 406–420, Cham: Springer International Publishing, 2018.

[73] Z. Wang, Y. Zhuang, and Z. Yan, "Tz-mras: A remote attestation scheme for the mobile terminal based on arm trustzone," *Security and communication networks*, vol. 2020, 2020.