Literature Study

Oberon Swings

October 18, 2021

Contents

Chapter 1

InitialPapers

1.1 Demystifying Arm TrustZone: A Comprehensive Survey

1.1.1 Introduction

Beginning

Manufacturers of TrustZone-enabled SoCs (System on Chip) were somewhat reluctant to disclose technical details or any architectural-related details. Research involving TrustZone has been further slowed down by the limited availability of development platforms in which all of TrustZone's capabilities were unlocked. For instance, certain boards were natively programmed to boot the processor directly into the normal world, thereby preventing system developers from deploying code inside the secure world.

Lately

A dynamic open source community has matured, working toward the definition of common API specifications to promote interoperability across TrustZone-based solutions. The research community, in particular, has been extremely active in exploring new ways to leverage TrustZone as a key-enabler technology for enforcing Trusted Execution Environments (TEEs) and hardware-assisted virtualization. TrustZone can provide fundamental primitives for securing sensitive data while benefiting from its wide deployment across a large number of mobile and low-end devices.

1.1.2 TrustZone for Application Processors

Overview

TrustZone for application processors refers to the hardware-based security built into SoCs to provide a foundation for improved system security for

Cortex-A processors. The most important architectural change at the processor level consists in the introduction of two protection domains designated by the name of worlds: the secure world and the normal world. The world where the processor currently executes is determined by the value of a new 33rd processor bit, also known as the Non-Secure (NS) bit. The value of this bit can be read from the Secure Configuration Register (SCR) register, and it is propagated throughout the system down to the memory and peripheral buses.

Hardware Isolation

To reinforce hardware isolation between worlds, the processor has banked versions of the special registers, as well as some system registers. In the normal world, the security-critical system registers and processor core bits are either totally hidden or conditioned by a set of access permissions supervised by the secure world software.

Memory

The memory infrastructure has also been extended with TrustZone security features, in particular with the introduction of the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA). The 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. A similar memory partitioning functionality is implemented by the TZMA, but targeting off-chip ROM or SRAM.

Interrupts

TrustZone technology allows for system devices to be restricted to secure or normal worlds. This is achieved with the introduction of a TrustZone Protection Controller (TZPC). The TrustZone architecture extends the Generic Interrupt Controller (GIC) with support for prioritized secure and non-secure sources. Interrupt prioritization is important to prevent denial-of-service (DoS) attacks by non-secure software, since it enables secure interrupts to be handled with higher priority than the non-secure interrupts.

TrustZone for Microcontrollers

In microcontroller applications, low power consumption, real-time processing, deterministic behavior and low interrupt latency are mainstream requirements, which lead TrustZone for Armv8-M to be designed from the ground up instead of being reused from Cortex-A processors.

1.1.3 Trusted Execution Environments

TEE-Enabling Technology

Modern computer systems tend to depend on large trusted computing bases (TCBs). Typically, a TCB comprises complex software such as the OS kernel, privileged services, and libraries. Systems featuring a bloated TCB tend to be more vulnerable to attacks than small-sized TCB systems. To address the TCB bloating problem, TrustZone has appeared as a fundamental hardware mechanism that enables to provide a TEE in which critical applications can execute securely featuring a TCB several orders of magnitude smaller than the rich OS. More specifically, a TEE consists of an isolated environment in which trusted applications can execute without the interference of the local (untrusted) OS.

TEE Kernel & Service

Essentially, the secure world provides a restricted execution environment where the TEE can reside. Whenever the system boots, the processor enters the secure world to give to any privileged firmware the chance to set up its internal data structures, configure the interrupt controller of the entire system, and set up protections for secure memory regions and peripherals. Upon the completion of these operations, the processor switches worlds and yields control to the bootloader of the rich OS. Depending on the trusted program that runs in the secure world, we distinguish two types of TEE architectures: TEE kernel or TEE service. The trusted kernel is responsible for: managing memory of the secure word, enforcing memory protection for each TEE, handling communication between TEE and the OS, and providing an API to TEE applications. However, TEE services implement a specific function and do not require any low-level OS logic to manage their own memory and cross-world communication. To prevent mutual interference, only one TEE service can be deployed on the device. This is a disadvantage when compared to TEE kernels, which allow multiple applications to run in independent TEE instances. However, a downside of TEE kernels is that they normally depend on larger TCBs when compared to systems where a single TEE service is deployed.

Trusted Kernel

On TrustZone-enabled platforms, the runtime support for sustaining the lifecycle of such applications is typically provided by a privileged trusted kernel, which runs in the secure world. The communication between the rich OS and the trusted kernel requires context switch between worlds. Given that rich OS and trusted kernels are not necessarily developed by the same manufacturer and nevertheless need to interoperate with each other, a lot of TEE standardization efforts have been made.

Trusted Service

An important class of trusted service solutions aims to provide secure storage and access to sensitive files in the presence of a potentially compromised local OS. Another relevant category of trusted services aims to provide secure authentication and cryptographic functions. Among the most challenging requirements of building trusted services, we find the need to provide secure I/O channels to the user interface. The difficulty lies in that the UI is supported by device drivers of the rich OS, which are both untrusted and difficult to implement with a small code footprint. To address this problem, instead of implementing the required drivers from scratch, some systems allow the secure world domain to reuse untrusted drivers implemented inside the rich OS.

TEE Systems for the Cloud

While some of these solutions are designed to operate on a standalone basis, other systems have been conceived to be tightly coupled with a cloud backend. By relying on the TEE, an attacker that manages to compromise the local OS will not be able to recover the keys and the content of the files. Before provisioning the keys into the TEE, it allows the cloud endpoint to remotely attest the client's software thereby ensuring that keys are properly allocated into the TEE rather than to the untrusted domain controlled by the client's OS. Beyond relying on client-side TEE, researchers have proposed new applications of TrustZone-assisted TEE on the cloud backend itself.

Userspace TEE

One class of TEE hardware allows for securing user space (ring 3) programs without the need to trust in privileged OS code running at ring 0 or below. Intel SGX allows for the creation of memory regions named enclaves, which are protected from hardware and software access. Most notably, SGX implements hardware-enabled memory encryption.

OS TEE

Another class of TEE hardware aims to implement secure execution environments at the OS level (ring 0). In AEGIS, part of the OS is split and runs inside a protected environment established by the processor.

Hypervisor TEE

Whenever a TEE hardware provides mechanisms to instantiate a TEE stack based on a trusted hypervisor, we say that it operates at ring -1. An example is Bastion, a security architecture that relies on both a modified processor and a trusted hypervisor to provide confidentiality and integrity protection for security-sensitive software.

Processor TEE

Certain TEE hardware technologies implemented by the processor can operate below the hypervisor level in ring -2. Arm TrustZone technology, can be highlighted as one of its most representative examples. In fact, the virtualization extensions to the Armv8 architecture allow for the deployment of an untrusted hypervisor in the normal world. Enabled by TrustZone, an independent trusted TEE stack can then reside inside the secure world.

Coprocessor TEE

Last, we mention a class of TEE hardware that relies on independent coprocessors; hence, we say they allow for the implementation of ring -3 TEE software stacks. The most prevalent of such technologies is the Trusted Platform Module (TPM). The TPM consists of a coprocessor, which is typically located on the motherboard. Its primary purpose is to serve for bootstrapping trust on the local platform: it is responsible for storing the software measurements computed during the trusted boot process of the system, and for securely storing cryptographic keys for remote attestation and data sealing operations.

Discussion

Although the reduction of TCB can help eliminate the presence of potential code vulnerabilities, that, by itself, cannot ensure its correctness. The latest efforts to overcome this challenge have leveraged software verification techniques to formally prove the correctness of privileged code residing within the secure world.

1.1.4 Trustzone-Assisted Virtualization

Overview

TrustZone technology, although implemented for security purposes, enables a specialized, hardware-assisted, form of system virtualization. With a virtual hardware support for dual world execution, as well as other TrustZone features like memory segmentation, it is possible to provide time and spatial isolation between execution environments. Basically, the non-secure software runs inside a VM whose resources are completely managed and controlled by a hypervisor running in the secure world. TrustZone-assisted virtualization is not particularly considered full-virtualization neither paravirtualization, because, although guest OSes can run without modifications on the non-secure world side, they need to co-operate regarding the memory map and address space they are using.

Single Guest

The single-guest configuration is the simpler system architecture of a TrustZone-assisted virtualization solution. The guest OS executes under the non-secure perimeter, and the hypervisor runs in the monitor mode. The hypervisor has a privileged view of the entire system, while the guest OS has limited access to system resources. The TCB of the system is confined to the code running on the secure world side, which means it just depends on the hypervisor size. Memory, devices, and interrupts assigned to the guest OS are configured as non-secure resources and they are directly managed by the guest OS, while the remaining secure resources are under strict supervision of the hypervisor. The guest OS manages its own MMU and cache lines.

1.1.5 Security Issues and Vulnerabilities

Overview

TrustZone provides several security primitives that developers can leverage to implement trust- worthy systems. A simplified but realistic multi-core prototype of the Arm TrustZone technology has been verified and proved to be secure from a hardware standpoint [30]; however, the poor us- age of TEEs coupled with some microarchitectural misconceptions have opened several security issues and vulnerabilities. While the former is a consequence of the lack of robust TEE runtime im- plementations, which results in failing to provide secure containers to applications, the latter is a direct consequence of architectural decisions or the existence of implementation-defined parts on TrustZone specification. Examples of specific microarchitectural attack vectors encompass hard- ware exceptions (SMC, IRQ, FIQ), caches, and power management modules.

TEE Vulnerabilities

although the security design of TEEs might be correct, i.e. secure architecture and perfect and robust isolation, the code running inside the TEE may contain vulnerabilities that can be exploited by attackers to corrupt the TEE and compromise the trust state of the entire system. Like Ning et al. [72], we agree that the current state of the art TEE research still lacks frameworks to verify and/or analyze the secure code, properly defense mechanisms within the trusted environments, methods for monitoring and detecting compromised TEEs, and resilient plans to recover and rejuvenate from attacks.

Hardware Vulnerabilities

A number of hardware-related vulnerabilities has also been uncovered over the last few years. The reported vulnerabilities affect different hardware parts of the platform, in particular, the compo- nents that constitute the platform's root of trust, caches, power management mechanisms, and FPGAs. Root of

trust. While Intel and AMD specify the TPM as the root of trust for their systems, Trust- Zone, per se, does not specify where keys for authentication and decryption shall be stored.

1.1.6 Future Directions

IoT Devices

The problem is that securing IoT devices can be a quandary, with hardware requirements and cost limitations pushing different design directions. To address this problem, Arm decided to span TrustZone to the new generation of microcontrollers, by making security practical at scale and across the entire value chain. With TrustZone built-in on the tiniest of things, Arm is easing the economics of security, reducing risk, cost, and the complexity of implementing robust security measures. While virtualization in embedded systems started by primarily being deployed on high-end devices, the increasing adoption of virtualization technology also starts finding some applicability on low-end hardware, but with several performance limitations due to the lack of hardware support on such devices. To fill this gap, Arm has recently included virtualization extensions in the new generation Cortex-R processors. The Cortex-R52 is the first processor from the Cortex-R family introducing hardware support for virtualization. Hardware virtualization support on real-time processor series is slightly different from the one existing on application processors, due to the need of copying with hard real-time capabilities.

1.2 Fidelius: Protecting User Secrets from Compromised Browsers

1.2.1 Introduction

Fidelius

Generally speak- ing, once malware infects the user's machine, it can effectively steal all user data entered into the browser. Modern browsers have responded with a variety of defenses aimed at ensuring browser integrity. However, once the machine is compromised, there is little that the browser can do to protect user data from a key logger.
Fidelius, that helps web sites ensure that user data entered into the browser cannot be stolen by end-user malware, no matter how deeply the malware is embedded into the system. When using Fidelius, users can safely enter data into the browser without fear of it being stolen by mal- 1ware, provided that the hardware enclave we use satisfies the security requirements.

Contributions

- The design of Fidelius, a system for protecting user secrets entered into a browser in a fully-compromised environment.
- A simple interface for web developers to enable Fi- delius's security features.
- The first open design and implementation of a trusted path enabling a hardware enclave to interact with I/O devices such as a display and a keyboard from a fully compromised machine.
- A browser component that enables a hardware enclave to interact with protected DOM elements while keep- ing the enclave component small.
- An open-source implementation and evaluation of Fi- delius for practical use cases.

1.2.2 Background

Thread Model

We assume that our attacker has the power to ex- amine and modify unprotected memory, communication with peripherals/network devices, and communication be- tween the trusted and untrusted components of the system. Moreover, it can maliciously interrupt the execution of an enclave. Note that an OS-level attacker can always launch an indefinite denial of service attack against an enclave, but such an attack does not compromise privacy.

Architecture

The goal of Fidelius is to establish a trusted path between a user and the remote server behind a web application. To achieve this goal, Fidelius relies on two core components: a trusted user I/O path and a web enclave. In practice, this involve subsystems for a secure keyboard, a secure video display, a browser component to interact with a hardware enclave, and the enclave itself.

1.2.3 Design

Trusted IO Path

The trusted user I/O path consists of a keyboard and display with a trusted dongle placed between them and the computer running Fidelius. Each device consists of trusted and untrusted modes. The untrusted modes op- erate exactly the same as in an unmodified system. The trusted keyboard mode, when activated, sends a constant stream of encrypted keystrokes to the enclave. The en- clave decrypts and updates the state of the relevant trusted input field. The trusted and untrusted display modes are active in parallel, and the trusted mode consists of a series of overlays sent encrypted from the

enclave to the dis- play. Overlays include rendered DOM subtrees (includ- ing, if any, the protected user inputs) placed over the un- trusted display output as well as a dedicated portion of the screen inaccessible to untrusted content.

Web Enclaves

A web enclave is essentially a hardware enclave running a minimalistic, trusted browser engine bound to a single web origin. A browser using a web enclave delegates the management and rendering of portions of a DOM tree and the execution of client-side scripts, e.g. JavaScript and Web Assembly, to the enclave. In addition, the web enclave can send and receive encrypted messages to and from trusted devices and the origin server. Finally, the web enclave provides client-side script APIs to access the DOM subtree, secure storage, and secure HTTP commu- nication.

Trusted IO Setup

In order to securely communicate, the web enclave and peripherals (or the dongles connected to them) must have a shared key. One option is to operate in a threat model with an initial trusted phase where we assume the computer is not yet compromised. Pre-shared keys are exchanged when the user configures the computer for the first time. Devices store the key in an internal memory, and the enclave seals the shared keys for future retrieval. The key can be accessed only by the enclave directly and not by user-provided JavaScript running inside it.

IO Secured Communication

The process of switching between trusted and untrusted modes presents an interesting security challenge. An authentication procedure between the enclave and the trusted devices can ensure that only the enclave initiates switches between trusted and untrusted modes, but this ignores the larger problem that the enclave must rely on the untrusted OS to inform it when an event has happened that necessitates switching modes. Avoiding that necessity would require moving a prohibitively large fraction of the browser and UI into an enclave.

Security Analysis

- Enclave Ommission Attack
- Enclave Missuse Attack
- Page Tampering Attack
- Redirection Attack
- Storage Tampering Attack

- Mode Switching Attack
- Replay Attack
- Input Manipulation Attack
- Timing Attack
- Multi Enclave Attack

1.2.4 Evaluation

Performance

We evaluate Fidelius in order to determine whether the overheads introduced by the trusted I/O path and web enclave are acceptable for common use cases and find that Fidelius outperforms display latency on some recent commercial devices by as much as 2.8x and prior work by 13.3x. Moreover, communication between the browser and enclave introduces a delay of less than 40ms to page load time for a login page.

Trusted Code Base

The trusted code base for Fidelius consists of 8,450 lines of C++ code, of which about 3200 are libraries for handling form rendering and another 3800 are our enclave port of tiny-js. This does not include native code running outside the enclave or in the browser extension because our security guarantees hold even if an attacker could compromise those untrusted components of the system.

1.3 POSTER: An Open-Source Framework for Developing Heterogeneous Distributed Enclave Applications

1.3.1 Introduction

Enclaves

Trusted Execution Environments (TEEs) allow an application to execute in a hardware-protected environment called enclave. Enclaves are isolated and protected from the rest of the system, ensuring strong confidentiality and integrity guarantees. Cryptographic primitives and cryptographic keys, which are unique per enclave and which can only be used by that enclave, enable secure communication and remote attestation.

1.3.2 Authentic Execution

Authentic Execution

We developed the concept of authentic execution to address the problem of securely executing distributed applications on a shared infrastructure and to also minimize the application's runtime TCB. Authentic execution provides a notion of security that we summarize as "if the application produces a physical output event, then there must have happened a sequence of physical input events such that that sequence, when processed by the application, produces that output event." which is roughly equivalent to the concept of robust safety. This guarantee relies on standard TEE security properties (i.e., strong software isolation and software attestation) but also on a notion ofsecure I/O where physical I/O channels can be connected to an enclave such that the application enclaves maintain exclusive access over I/O peripherals.

Objectives

We consider an open system as the basis for our framework. In this open system, software is deployed dynamically and multiple stake-holders may run applications on the same infrastructure, including on the light-weight IoT and Edge hardware. Thus, we consider scenarios where arbitrary new code can be loaded at run time and we consider powerful attackers that can manipulate all the software on the infrastructure (unless that software is isolated in enclaves),can manipulate network traffic, but cannot break crypto. Attacks against the hardware are out of scope.

1.4 SANA: Secure and Scalable Aggregate Network Attestation

1.4.1 Introduction

Overview

Embedded devices are often security and privacy critical, because they possibly have consequences in the physical world. One common attack is to modify or replace a device's firmware, as part of a larger attack scenario. Safe and secure operation of a device needs to be ensured to prevent this type of attack. It is important to guarantee its software integrity, e.g., via remote software attestation. Remote software attestation is an interactive protocol that allows a prover to prove its software integrity to a remote verifier. This is usually achieved by signing integrity-protected memory regions. Attestation of individual smart devices is a well established research area. However, to date there is a lack of viable approaches to securely scale device attestation to a very large number of devices:

Goals

1. scalability i.e., it efficiently verifies the integrity of a large collection of devices by means of a novel signature scheme, which allows aggregation of attestation proofs.
2. public verifiability, i.e., the produced aggregate signature can be verified by any one knowing the (aggregate) public key.
3. enabled untrusted aggregation, i.e., compromise (including physical tampering) of aggregating nodes does not affect the integrity of the attestation process.

Approach

- A novel signature scheme, called Optimistic Aggregate Signature (OAS) allows the aggregation of signatures on different attestation responses, while having a verification overhead that is constant in the size of the network.
- SANA is the first collective attestation scheme for networks of embedded devices that supports high dynamicity and adheres to common assumptions made in single-prover attestation. SANA leverages OAS over aggregation trees to provide highly scalable attestation of large device populations, in a single round-trip.
- The performance of SANA is analyzed on three state-of-the-art security architectures for low-end embedded devices (e.g., SMART, TrustLite, and TyTAN), for networks of up to 1, 000, 000 devices, in order to demonstrate its scalability.

1.4.2 Background & Related Work

Collective Attestation

SEDA, made a first step towards a collective attestation, i.e., the scalable attestation of large groups of interconnected devices. The main focus of SEDA is efficiency and applicability to low-end devices, rather than security in the presence of a strong adversary. Our proposed collective attestation protocol SANA overcomes the limitations of SEDA by

- Requiring minimal trust anchor in hardware only for the attested devices.
- Allowing aggregation to be performed by largely untrusted nodes, which are only required for availability.
- Limiting the effect of successful attacks on the hardware of an attested device to the device itself, i.e., it will not affect the attestation of other devices.

Similarly, Denial-of-Service attacks on one device in SANA will not affect other devices. Finally, SANA informs the verifier with ids as well as software configurations of the devices that failed attestation.

Aggregate Multi-Signatures

A new signature scheme Optimistic Aggregate Signature (OAS) is proposed, that

- Allows signatures on distinct messages to be aggregated.
- Provides a signature verification algorithm that is constant in the number of signers.

The communication overhead of the scheme is linear in the number of different messages, while the computational overhead is linear in the number signers who signed a different message than the default one. However, this number is assumed limited. Finally, we present a pairings-based construction of OAS, and combine it with aggregation trees, providing unlimited scalability.

1.4.3 Design

System Model

SANA is a protocol on network (G) between the following logical entities: prover (P), aggregator (A), owner (O), and verifier (V). A prover composes a proof of integrity of its software configuration, which is sent to a remote verifier. Provers can have different software and hardware configuration. However, we expect the majority of them to have a good software configuration (Good provers). An aggregator has the purpose of relaying messages between entities, and collecting and aggregating attestation responses from provers, or other aggregators. The entity O represents the network owner or operator, responsible for the deployment, as well as the maintenance, of every prover Prover in G. A physical device in G can embed the functionalities of every logical component described above, or a combination of them.

Protocol

Each prover is initialized with the cryptographic material needed to execute SANA collective. The initialization is performed in a secure environment, and preferably, but not necessarily, by O. At a given time, a verifier V, which possesses an appropriate attestation token generated by O, may attest G. Note that, if V and O are two distinct entities, the token is securely exchanged offline. In order to attest the network, V chooses an aggregator and sends it an attestation request containing an attestationtoken. The request is flooded in the network forming a logical aggregation tree, that has provers as leaf nodes, and aggregators as intermediate nodes. Leaf nodes of the aggregation tree, i.e., provers create their attestation response and send it to their parent nodes.

Aggregators, i.e., non-leaf nodes, in turn, aggregate the attestation responses received from their child nodes, and forward the result to their parents. Finally, the aggregated report is received and verified by V.

Requirements

- Unforgeability and Freshness: If the attestation hardware of a prover is unchanged and a correct verifier was able to validate the aggregate attestation result including a given prover, then the claimed integrity measurement reflects an actual software configuration of this prover at a time during this protocol run.
- Completeness: If the attestation hardware of provers is unchanged and a correct verifier was able to validate the aggregate attestation result for a given set of provers, then all provers actually reported their software configuration in the given protocol run.
- Scalablity: The protocol allows a verifier to attest a large network of devices. The communication and computational complexity for prover and verifier must be at most logarithmic in the number of devices in the collection.
- Public Verifiability: In a public key domain, the collective attestation evidence collected by a verifier can be verified by any party. In this case, the Unforgeability requirement only proves the state of the prover within the time window between generation of the challenge by the owner, and the receipt of the evidence from the verifier.
- Privacy Preservation: Verification does not require detailed knowledge of the configuration of G (e.g., its topology).
- Heterogeneity: The protocol is applicable to networks with heterogeneous devices. The scheme can use any integrity measurement mechanism used by devices in G.
- Availability: If all participants are honest and the network is available then the protocol produces collective attestation evidence.
- Limiting DoS: It should not be possible to run a global DoS attack over the whole network through one device.

Assumptions

It is assumed that all provers in G correctly implement the minimal hardware features required for secure remote attestation. A potential implementation of P could have: a Read Only Memory (ROM) that stores the protocol code and the related cryptographic key(s), and a simple Memory Protection Unit (MPU), that restricts access to cryptographic key(s) to protocol code only, and ensures secrecy of the key(s) through non-interruptible, and clean

execution of the protocol code. The assumption is made that the owner O to be trusted. Finally, all cryptographic schemes used in our protocol are assumed to be secure.

Implementation

SANA was implemented on TyTAN as isolated tasks, which are protected via secure boot. Further, the MPU was configured such that only SANA's tasks can access the protocols secret data. For example, according to rule #2 in the MPU table, the OAS secret key is only read accessible to createResponse(). TyTAN is a security architectures for embedded systems, that is based on TrustLite. TyTAN provides hardware-assisted isolation of system components with real-time execution. Isolation is fundamental to protect critical components against unintended access by other potentially malicious components. In TyTAN, a Memory Protection Unit restricts access to data, to the task that owns this data. Moreover, both authenticity and confidentiality of the tasks' code and data are based on secure boot.

1.4.4 Results & Evaluation

Communication Overhead

A token T with z good configurations consists of 20z + 58 bytes, and a challenge Ch of 20z + 78 bytes. A response α has size 32 + 32w + 20m bytes, where m is the number of distinct bad software configurations and w is the number of distinct OAS public keys of bad provers. The communication overhead of the each aggregator is, sending at most 32 + (20z + 78)g + 32w + 20m bytes and receiving at most 20z + 78 + 32g + 32w + 20m bytes where g is the number of neighbours. Finally, every prover P sends 84 bytes and receives 20z + 78 bytes.

Memory Cost

Each Prover in G stores the ids and values of s counters, its OAS secret key, its identity certificate, and the public key O. The storage overhead for every P is estimated as 10s + 228 bytes, where s is the number of counters used by O.

Performance

The aggregation tree approach allows provers, and aggregators on the same depth of the tree, to perform their computations in parallel. However, the OAS signature aggregation at depth d depends on the signature creation computations at depth d + 1. Consequently, the overall run-time of the SANA depends on the depth ($d = f(n + a) \in O(log(n + a))$) of the aggregation tree generated for the graph of the network, the number of neighbors of each aggregator, and the number of bad provers. The run-time t of SANA is

estimated as

$$t =\leq \left[110d + \sum_{i=0}^{d} (32w + 20m)\right] \times t_{tx} + \sum_{i=0}^{d} p \times t_{agg} + d \times (t_{ver} + t_{hash}) + t_{sign} \tag{1.1}$$

1.5 Sancus 2.0: A Low-Cost Security Architecture for IoT Devices

1.5.1 Introduction

Contributions

- We propose Sancus, a security architecture for resource-constrained, extensible networked embedded systems, that can provide strong isolation guarantees, remote attestation, secure communication, secure linking, and confidential software deployment with a minimal (hardware) TCB.
- We implement the hardware required for Sancus as an extension of a mainstream micro-processor, and we show that the cost of these hardware changes (in terms of performance, area, and power) is small.
- We implement a C compiler that targets Sancus-enabled devices. Building software modules for Sancus can be done using simple annotations with standard C code, showing that the cost in terms of software development is low as well.
- We implement a Contiki-based (untrusted) software stack to automate the deployment process of Sancus modules.
- We report on our experience with implementing a variety of applications on Sancus, and evaluate Sancus in terms of performance, hardware cost, and security.

1.5.2 Background

System Model

A single infrastructure provider (IP), owns and administers a set of microprocessor-based systems that we refer to as nodes (N). A variety of third-party software providers (SP) are interested in using the infrastructure provided by IP. They do so by deploying software modules (SM) on the nodes administered by IP.

System Requirements

Any system that supports extensibility by several software providers must implement measures to make sure that the different modules cannot interfere with each other in undesired ways. For high- to mid-end systems, this problem is relatively well-understood, and good solutions exist. Two important classes of solutions are the use of virtual memory, and the use of a memory-safe virtual machine.

System goals

The problem we address in this paper is the design, implementation and evaluation of a novel security architecture for low-end systems that is inexpensive and does not rely on any trusted software layer. The TCB on the networked device is only the hardware. More precisely, a software provider needs to trust only the hardware of the infrastructure and his own modules; he does not need to trust any infrastructural or third-party software on the nodes.

Attacker Model

We consider attackers with two powerful capabilities. First, we assume attackers can manipulate all the software on the nodes. Second, we assume attackers can control the communication network that is used by software providers and nodes to communicate with each other. Note that the security of the communication channel between IP and software providers is out of scope. With respect to the cryptographic capabilities of the attacker, attackers cannot break cryptographic primitives, but they can perform protocol-level attacks. Finally, attacks against the hardware of individual nodes are out of scope.

Security Properties

- Software module isolation
- Remote attestation
- Secure communication
- Secure linking
- Confidential deployment
- Hardware breach confinement

1.5.3 Design

Main Challenge

The main design challenge is to realize the desired security properties without trusting any software on the nodes, and under the constraint that nodes are

low-end, resource-constrained devices. An important first design choice that follows from the resource-constrained nature of nodes is that we limit cryptographic techniques to symmetric key, in particular authenticated encryption. While public key cryptography would simplify key management, the cost of implementing it in hardware is too high.

Cryptographic Primitives

Throughout the design of Sancus, we assume the existence of three cryptographic primitives. First, a classical cryptographic hash function is used to compute digests of data. Second, a key derivation function is used to derive a cryptographic key from a master key and some diversification data. Third, an authenticated encryption with associated data primitive is used to simultaneously provide confidentiality, integrity, and authenticity guarantees on data. Such a primitive consists of two functions: one for encryption and one for decryption.

Key Management

We handle key management without relying on public-key cryptography. IP is a trusted authority for key management. All keys are generated and/or known by IP. There are three types of keys in our design:

- Node master keys K_N shared between node N and IP.
- Software provider keys $K_{N,SP}$ shared between a provider SP and a node N.
- Software module keys $K_{N,SP,SM}$ shared between a node N and a provider SP, and the hardware of N makes sure that only SM can use this key.

The software provider keys $K_{N,SP}$ and software module keys $K_{N,SP,SM}$ are derived using a key derivation function as discussed in the overview section.

Memory Access Control

The memory access control logic in the processor enforces that (1) the data section of a module is only accessible while code in the text section of that module is being executed, (2) the text section can only be executed by jumping to a well-defined entry point, and (3) the text section cannot be written and can only be read while code in that section is being executed. Finally, it remains to decide how to handle memory access violations. We opt for the simple design of resetting the processor and clearing memory on every reset. This has the advantage of clearly being secure for the security properties we aim for. However an important disadvantage is that it may have a negative impact on availability of the node.

Remote Attestation & Safe Communication

These instructions can be used to provide confidentiality, integrity, and authenticity guarantees of data exchanged between modules and their providers. The ciphertext plus the corresponding tag can be sent using the untrusted operating system over an untrusted network. If the tag verifies correctly (using $K_{N,SP,SM}$) upon receipt by the provider SP, he can be sure that the decrypted plaintext indeed comes from SM running on N on behalf of SP as the node's hardware makes sure only this specific module can use the module key $K_{N,SP,SM}$. The reasoning is equivalent for data sent to the module. To implement remote attestation, we only need to add a freshness guarantee (i.e., protect against replay attacks). Provider SP sends a fresh nonce No to the node N , and the module SM returns the MAC of this nonce using the key $K_{N,SP,SM}$, computed with the encrypt instruction. This gives the SP assurance that the correct module is running on that node at this point in time.

Processor Isolation Implementation

For isolation, the processor needs access to the layout of every software module that is currently protected. Since the access rights need to be checked on every instruction, accessing these values should be as fast as possible. For this reason, we have decided to store the layout information in special registers inside the processor. Apart from hardware circuit blocks that enforce the access rights, we also added a single hardware circuit to control the MAL circuit instantiations. It implements four tasks: (1) combine the violation signals from every MAL instantiation into a single signal; (2) keep track of the value of the current and previous program counter; (3) keep track of the currently and previously executing SM; and (4) when the protect instruction is called, select a free MAL instantiation to store the layout of the new software module and assign it a unique ID.

Cryptographic Implementation

We have chosen to build these cryptographic primitives on the SpongeWrap authenticated encryption construction using spongent as the underlying sponge function. Since keyed sponge functions are shown to be pseudorandom functions, we can reuse SpongeWrap to calculate MACs, and consequently for key derivation. Since the security of SpongeWrap relies on the soundness of the sponge function it uses, it can also be used as a hashing function.

Compiler Entrypoint Implementation

Since the hardware supports a single entry point per module only, the compiler implements multiple logical entry points on top of the single physical entry point by means of a jump table. The compiler assigns every logical entry point a unique ID. When calling a logical entry point, its ID is placed in a

register before jumping to the physical entry point of the module. The code at the physical entry point then jumps to the correct function based on the ID passed in the register.

1.6 SeCloak: ARM Trustzone-based Mobile Peripheral Control

1.6.1 Introduction

Overview

SeCloack uses ARM Trustzone to provide reliable on-off controls for peripherals even when the platform software is compromised. This is important to protect users against databreaches because lots of personal smart devices contain more and more sensitive data. It is important to note that these devices also contain highly sophisticated hardware security in their architecture. These security features are often used for financial transactions.

Goals

SeCloak would allow users to reliably turn off certain peripherals like sensors or radios using hardware mechanisms. The design goals are

1. Allow users to securely and verifiably control peripherals on their device
2. Maintain system usability and stability
3. Minimize the trusted code base
4. Not change existing software including apps, frameworks or OS kernels

1.6.2 Background & Related Work

Virtualization

One approach is to run the platform OS as a guest within a virtual machine, leaving a hypervisor in control of peripheral devices. These systems are designed for isolating individual applications from the OS, but don't provide a mechanism to reliably control generic peripherals. These approaches seek to protect the integrity and confidentiality of I/O data paths while maintaining full functionality, which comes at the expense of a large TCB.

Hardware Security

Beyond virtualization, modern architectures offer trusted hardware components e.g., Intel SGX and ARM TrustZone, which can be used to isolate software components from an untrusted platform OS. By default, TrustZone supports a single isolated execution environment (the secure world), with a

secure boot process that can be used to verify the bootloader and the secure-world kernel. TrustZone hardware protections are used to protect sensitive memory regions, it doesn't address peripheral access control.

ARM TrustZone

TrustZone is the basis for commercial products that implement support for isolated execution of secure applications and for secure IO. Trust Leases allow applications to request (with user approval) leases to place the device in a restricted mode until some terminal condition is met. Their threat model assumes that the platform OS is trusted and correct. Viola enables custom, per-peripheral notifications whenever the I/O peripheral device is being used. SeCloak and Viola are complementary, the user could use SeCloak to disable devices and rely on Viola to notify them when specific enabled devices are in use.

1.6.3 Design Overview

NS-bit

TrustZone security model posits that the CPU can be in a "Non-Secure" (NS) mode or "Secure" mode, and all memory and bus accesses are tagged with the CPU mode using an extra "NS-bit" bus line. Different peripherals, including parts of RAM, IO devices and interrupts, can be configured to be available only in Secure mode. In our design, Linux runs as the NS-kernel, and SeCloak (with its secure kernel) controls the secure modes of the CPU.

Security Configuration Register

The ARM Security Configuration Register (SCR) contains the "NS" bit that describes whether the core executing code is in the non-secure or secure mode. It is this setting of the SCR that enables SeCloak to trap and emulate instructions issued by the Linux kernel. The SCR also contains similar configuration bits for interrupts, which SeCloak uses to listen for user input and support secure system reboot.

TrustZone Address Space Controller

ARM provides a TrustZone Address Space Controller (TZASC) that can partition portions of RAM such that they are available only to secure mode accesses, enabling isolation of the s-kernel memory from Linux.

Central Security Unit

A TrustZone compatible component, the Central Security Unit (CSU), that extends the secure/non-secure access distinction to peripherals. The CSU can be used to enable secure-only access for different peripherals (which SeCloak

uses), and also for programming access to various bus DMA masters (e.g., the GPU). SeCloak uses a purpose-built kernel (the s-kernel) that runs in TrustZone secure mode. The s-kernel programs each of these components (the SCR, the TZASC, the CSU) as required, both at initialization and runtime to enable SeCloak.

Threat Model

The threat model assumes that the device hardware is not malicious, and that the state of the hardware (number and type of IO devices, their physical addresses and buses, interrupts, etc.) is encapsulated in a "device tree" file that is signed by a trusted source. Beyond the hardware, we assume that the boot ROM and boot-loader are trusted and correctly loads the s-kernel. The s-kernel is also trusted and assumed correct. All other software in the system may be faulty or malicious. This includes any app the user may run, any framework layer (such as Android), any kernel modules, and the NS-kernel itself.

Secure Kernel

SeCloak's secure kernel is called the s-kernel. A signed device tree describes available hardware and protections to the s-kernel. Modern devices are equipped with secure, tamper-proof, non-volatile storage, into which device manufacturers embed (hashes of) public keys. The devices contain a one-time programmable Boot ROM that has access to these keys, which are "fused" onto the hardware.

Device Tree

The device tree structure describes the hardware devices present in a given system and how they are interconnected, with each node representing an individual device. It is important to note that the device tree, by our assumptions, must be an accurate and complete description of the hardware.

Booting

Upon boot, the s-kernel initializes hardware defaults prior to launching the NS-kernel. Specific steps include setting control and security registers to appropriate defaults, and setting memory protections such that the NS-kernel cannot overwrite the s-kernel's state. The s-kernel initializes its internal data structures by initializing the system MMU with virtual memory page table mappings for various regions, including regions for non-secure RAM, s-kernel heap, and for MMIO devices. The s-kernel also starts the non-boot CPUs, and initializes per-core threads and their contexts. Faults and calls from the NS-kernel transition the CPU into a monitor mode, and the s-kernel initializes the secure monitor with its stack pointer and call vector. Finally, the s-kernel opens and parses the device tree.

Non-Secure Kernel

The current version of the app is simple, allowing users to set ON/OFF preferences for the devices on our prototype board. Along with individual devices, the app allows users to choose different operating modes (e.g., Airplane, Stealth) and also provides the state of groups of peripherals (e.g., all networking devices.).

1.6.4 Results & Evaluation

Overhead

The download and upload performance shows that there is no visible impact of interception and emulation on WiFi transfers, despite an appreciable increase in execution time for individual load and store instructions. This is because the WiFi driver and controller, like all modern bulk data transfer devices, uses DMA to transfer packets. Once the controller firmware is loaded, and the DMA tables configured, each packet transfer (which can be many thousand bytes) requires very few (tens) MMIO instructions to initiate the DMA. We believe this result indicates that SeCloak can be used, even for high performance peripherals, without significant impact on user-perceived performance.

1.7 SEDA: Scalable Embedded Device Attestation

1.7.1 Introduction

Overview

We design SEDA, Scalable Em- bedded Device Attestation, which is, to the best of our knowledge, the first attestation scheme for large-scale swarms. SEDA repre- sents the first step in a new line of research on multi-device attesta- tion. Although SEDA adheres to the common assumption – made in most (single-prover) attestation techniques – of ruling out phys- ical attacks on devices, we discuss mitigation techniques for such attacks in Section 9.

Contributions

- First Swarm Attestation Scheme
- Security Model & Analysis
- Two Working Prototypes
- Performance Analysis

1.7.2 Swarm Attestation

Requirements

- Support the ability to remotely verify integrity of the swarm (S) as a whole.
- Be more efficient than individually attesting each device (D) in S.
- Not require the verifier (VRF) to know the detailed configuration of S.
- Support multiple parallel or overlapping attestation protocol instances.
- Be independent of the underlying integrity measurement mechanism used by devices in S.

Adversary Model

As common in the attestation literature [16, 24, 47, 48] we consider software-only attacks. This means that, although the adversary, denoted as ADV, can manipulate the soft- ware of (i.e., compromise) any device D in S, it cannot physically tamper with any device. However, ADV can eavesdrop on, and ma- nipulate, all messages between devices, as well as between devices and VRF. Furthermore, we rule out denial-of-service (DoS) at- tacks since ADV typically aims to remain stealthy and undetected while falsifying the attestation result for VRF.

Protocol Description

SEDA has two phases: (1) an off-line phase whereby devices are introduced into the swarm, and (2) an on-line phase performing actual attestation. The off-line phase is executed only once and consists of device initialization and device registration. The on-line phase is executed repeatedly for every attestation request from a verifier VRF.

Swarm Attestation

VRF starts attestation of S by sending an attestation request attest (containing a random challenge) to D 1 . VRF can randomly chose any device in S as D 1 or depending on its location or preference. Recall that VRF might be remote, or within direct communication range of one or more swarm devices. Eventually, VRF receives an attestation report from D 1 . VRF outputs a bit b = 1 indicating that attestation of S was successful, or b = 0 otherwise. VRF starts the protocol by sending a nonce N to D 1 . It, in turn, generates a new q and runs attdev with all its neighbors, which recursively run attdev with their neighbors. Note that N prevents replay attacks on communication between VRF and D 1 while the purpose of q is to identify the protocol instance and to build the spanning tree. Even- tually, D 1 receives the accumulated attestation reports of all other devices in S.

Chapter 2

ARMTrustZone

2.1 Trusted Execution Environment: What it is, and What it is not

2.1.1 Introduction

Trusted Computing was defined to help systems to achieve secure computation, privacy and data protection. Originally, trusted computing relies on a separate hardware module that offers a functional interface for platform security. A TEE is a secure, integrity-protected processing environment, consisting of memory and storage capabilities.

2.1.2 Trusted Execution Environment

The separation kernel is a foundation component of the TEE. It is the element that assures the property of isolated execution. The separation kernel, is a security kernel used to simulate a distributed system. Basically, it divides the system into several partitions, and guarantees a strong isolation between them, except for the interface for inter-partition communication. The SKPP defines separation kernel as "hardware and/or firmware and/or software mechanisms whose primary function is to establish, isolate and control information flow between those partitions.".

Policies The security requirements are composed of four main security policies:

- Data (spatial) separation. Data within one partition cannot be read or modified by other partitions
- Sanitization (temporal separation). Shared resources cannot be used to leak information into other partitions

- Control of information flow. Communication between partitions cannot occur unless explicitly permitted
- Fault isolation. Security breach in one partition cannot spread to other partitions

Definition 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 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 backdoor security flaws are not possible.

Trust In static trust, the trustworthiness of a system is measured only once and before its deployment. Dynamic trust is based on the state of the running system, and thus it varies accordingly. A system continuously changes its "trust status". In dynamic trust, the trustworthiness of a system is constantly measured throughout its lifecycle.

Root of Trust This definition requires a trusted entity called Root of Trust (RoT) to provide trustworthy evidence regarding the state of a system. The role of RoT is divided into two parts. First is the trusted measurement and second is the function that computes the trust score. The trustworthiness of the system, namely the generated score, depends on the reliability of the trust measurement. If a malicious entity can influence the trust measurement, then the generated score of trustworthiness is of no value. Therefore, RoT is necessarily a tamper-resistant hardware module.

Secure Execution Environment Secure Execution Environment (SEE) is a prerequisite for TEE, but it does not consider trust aspects. SEE is a processing environment that guarantees the following properties:

- Authenticity
- Integrity
- Confidentiality

In contrast to TEE, the design of SEE does not involve RoT to assert the integrity and authenticity of the loaded code. Moreover, it does not define secure mechanisms to update its applications and confidential data. In fact, in our definition, TEE is an open SEE that guarantees trust.

2.1.3 TEE Building Blocks

Secure Boot Secure Boot assures that only code of a certain property can be loaded. If a modification is detected, the bootstrap process is interrupted. An example implementation of secure boot is to verify the integrity of a succeeding component according to a given reference value.

Inter-Environment Communication Inter-Environment Communication defines an interface allowing TEE to communicate with the rest of the system. Each communication mechanism should satisfy three key attributes:

- Reliability (memory/time isolation)
- Minimum overhead
- Protection of communication structures

Secure Storage Secure Storage is storage where confidentiality, integrity and freshness of stored data are guaranteed, and where only authorized entities can access the data. It is based on three components:

- Integrity-protected secret key only accessible by the TEE
- Cryptographic mechanisms
- Data rollback protection mechanisms

Trusted I/O Trusted I/O Path protects authenticity, and optionally confidentiality, of communication between TEE and peripherals. Thus, input and output data are protected from being sniffed or tampered with by malicious applications. To be more precise, trusted I/O path protects against four classes of attacks: screen-capture attack, key logging attack, overlaying attack, and phishing attack. Trusted path to user-interface devices enables broader functionality within TEE. It allows a human user to directly interact with applications running inside TEE.

2.1.4 ARM TrustZone based TEE

ARM TrustZone technology can be seen as a special kind of virtualization with hardware support for memory, I/O and interrupt virtualization. This virtualization enables ARM core to provide an abstraction of two virtual cores (VCPUs): secure VCPU and non-secure VCPU. The monitor is seen as a minimal hypervisor whose main role is the control of information flow between the two virtual cores.

2.2 Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms

2.2.1 Introduction

The TCG specifications for the Trusted Platform Module (TPM) and the accompanying Trusted Software Stack (TSS) are focused on PC-style platforms. When attempting to implement TCG-compatible Trusted Computing systems on mobile and embedded devices a number of issues arise. Typical embedded and mobile platforms differ to PC-platforms with respect to their booting process and BIOS. For mobile phone platforms, a single TPM on the platform might be insufficient due to ownership issues. The TCG has published a set of specifications to address the different requirements of mobile and embedded devices, here two mobile versions to the TPM are defined. These requirements differ in terms of the supported command set and ownership rules:

- Remotely Owned (MRTM)
- Locally-owned (MLTM)

2.2.2 Prototype Mobile Trusted Platform Design

TCG's Mobile Reference Architecture decomposes the platform into isolated trusted engines owned by different entities. Those engines have an associated MTM and interfaces to communicate with other engines. The foundation for Access Control can be provided by SELinux and enforced by the TrustZone hardware features.

- Secure-world peripherals can not be accessed by any non-secure world software.
- Non-secure world software is not able to access secure-world memory without authorisation by secure-world.

Software Components The secure world partition and its operating system kernel are in ultimate control of the whole platform. For the implementation, an adapted version of the Linux kernel has been chosen as basis for the secure world operating system. This secure world Linux kernel contains a number of TrustZone specific extensions, most notably it provides a special user-space interface, allowing regular secure world user-space processes to act as "hypervisor" for the non-secure world partition.

Secure Boot Loader A secure boot process is of ultimate importance to mobile trusted computing because it provides the basis for establishing trust

on mobile and embedded devices. The existence of MRTMs implies a requirement for some kind of secure boot process. On mobile devices where hardware MTMs are available, a secure boot process can be implemented by relying on their capabilities. Platforms which only contain software MTMs cannot take advantage of having MTMs as hardware roots of trust. The modified u-boot for instance, is capable of measuring the Linux kernel image, its initial ramdisk and the kernel command line. Before control is handed over to the operating system kernel, these measurement values are compared to a Reference Integrity Metric (RIM) certificiate and attached to the kernel image. The boot process only continues if the kernel's RIM certificate can be successfully validated. The measurement values obtained by u-boot are handed over to the secure-world Linux kernel and are available after the Linux kernel hands over control to the userspace init process.

Software-based Mobile Trusted Modules The platform contains a software Mobile Local-Owner Trusted Module (MLTM) and a software Mobile Remote-Owner Trusted Module. The MLTM is started as a child process of the VM supervisor process, taking advantage of shared memory communication mechanisms. Since the supervisor process is in full control of the initial non-secure world executable image, secure boot can be implemented easily for the non-secure world partition, using the software MTMs available in secure world. Similarly to the VM supervisor process, the Trusted Engine process can take advantage of any software isolation mechanism available inside the secure-world.

Trusted Engines The TCG Mobile Reference Architecture is based on "Trusted Engines". These are described as isolated computation environments which typically have their own private MRTMs or MLTMs. Inter-engine communication is only possible by means of well-defined interfaces exposed by the individual engines. Trusted Engines are organised hierarchically with respect to their inter-engine communication interfaces.

TrustZone VM Supervisor The TrustZone VM supervisor is the fundamental component needed to support a non-secure world partition. This application utilises the TrustZone user-space interface exposed by the secure-world Linux kernel to act as hypervisor for the non-secure world. Any secure monitor calls invoked by the non-secure world guest VM are routed to the user-space VM supervisor after minimal processing inside the secure-world Linux kernel. Minimising the amount of secure-world in-kernel processing, reduces the secure-world privileged mode attack surface visible to a potential adversary coming from the non-secure world partition. To guarantee the isolation between the VM supervisor main process and its worker subprocesses, any privilege separation and access control mechanisms found inside the secure-world Linux kernel can be leveraged. This especially includes,

but is not limited to, mandatory access control enforced by SELinux domains or system call usage restrictions as supported by seccomp.

2.2.3 Virtualisation with ARM TrustZone

The framework features implementation of supervisors for non-secure world guests as ordinary secure-world user-space processes. Interaction with the guests is accomplished by using the user-space interface exposed by the framework. Only a small set of critical hypercalls has to be implemented within the secure-world host kernel.

Secure and non-secure world partitioning Depending on the implementation of the ARM TrustZone, the partitioning of peripherals and memory between secure and non-secure world can be hard-wired in silicon or can be reconfigurable by means of special platform dependent mechanisms. Platforms with dynamic secure/non-secure world repartitioning offer a high degree of flexibility with respect to handling non-secure world guests. Depending on the current software configuration of a non-secure world guest as reported by its associated MTM, hardware resources could be made accessible to that guest selectively. An interesting application of dynamic secure/non-secure world repartitioning could be the concurrent execution of multiple strongly isolated non-secure world virtual machines.

Interrupts two special TrustZone relevant properties of IRQ- and FIQ-type interrupts can be configured by secure-world privileged executives:

- Non-secure world access permissions to the global interrupt disable bit for FIQ-type interrupts can be configured to dissalow non-secure world modifications.
- The destination for handling IRQ- and FIQ-type interrupts can be either set to the currently executing worlds' regular vectors or to special Secure Monitor Mode vectors.

the virtualisation framework is not limited to a single non-secure world executive. As a consequence, IRQ-type interrupts can be targeted towards distinct, isolated non-secure world compartments running in parallel. In order to avoid unintended cross-compartment interference among the non-secure world compartments, interrupt handling must be under total control of the secure-world kernel. A non-secure world compartment must not be able to mask or disable any interrupt sources which are allocated to other non-secure world compartments running in parallel. The compartments are provided with a virtual interrupt controller managed by the secure-world kernel. From the non-secure world software point of view, this virtual interrupt controller consists of:

- global virtual interrupt controller status flags

- per-source virtual interrupt pending and mask flags
- secure monitor calls to interact with the secure-world part of the virtual interrupt controller

The lack of automatic notification of the secure world kernel is not a problem when disabling virtual interrupts. Actually the secure world part of the virtual interrupt controller has to check the "disabled" state of a virtual interrupt, when the corresponding hardware interrupt source actually fires. If the virtual interrupt target is disabled at this time, the secure world part masks the hardware interrupt source and records the virtual interrupt for deferred delivery.

Userspace supervisor interface A minimal set of operations required to implement a secure-world userspace supervisor process for the non-secure world guest compartment is the following:

- Opening the TrustZone VM interface
- Creating a guest VM
- Configuring guest VM resources
- Making guest VM memory accessible to the supervisor process
- Switch to non-secure world and handle requests from non-secure world
- Terminating the guest VM and release its resources

Chapter 3

SecureBoot

3.1 Implementing a ARM-based Secure Boot Scheme for the Isolated Execution Environment

3.1.1 Introduction

All the secure isolation based applications are based on the assumption of a secure execution environment. However, the isolation environment created by TrustZone is not a complete secure environment. It is possible that the device is attacked during start-up process. If the attacker modifies the system image in the external memory and obtains system privileges, the security of applications in the attacked system will be unknown.

3.1.2 Design and Implementation

The execution environment of TrustZone is an isolated execution environment rather than trusted. In order to build a truly trusted execution environment, secure boot is adopted through structuring root of struct and trust chain which ensures the device boot is secure.

Boot sequence

1. Device Power On
2. BootROM
3. First Stage BootLoader (FSBL)
4. U-boot
5. OP-TEE Kernel

6. Linux Kernel
7. System Running

Building Root of Trust The BootROM performs RAS authentication, AES decryption and HMAC authentication in sequence to check FSBL for tampering or being attacked. The keys for these algorithms during secure boot are saved in eFuse arrays on the board. After device is power-on or reset, the on-chip BootROM begins to perform CRC for its own integrity, it loads the FSBL into memory and compares the SHA signature of the boot image with the stored hash value. If all authentications have passed, the control will be turned over to the decrypted FSBL.

The trust chain The device could build a trust chain through validation of each stage, and it ensures that the execution environment is trusted. At each boot stage, the image of the next stage should be verified. This way the entire trust chain is being built, if verification of every stage is succesfull the device is booted up securely.

3.2 Implementing a ARM-based Secure Boot Scheme for the Isolated Execution Environment

3.2.1 Introduction

All the secure isolation based applications are based on the assumption of a secure execution environment. However, the isolation environment created by TrustZone is not a complete secure environment. It is possible that the device is attacked during start-up process. If the attacker modifies the system image in the external memory and obtains system privileges, the security of applications in the attacked system will be unknown.

3.2.2 Design and Implementation

The execution environment of TrustZone is an isolated execution environment rather than trusted. In order to build a truly trusted execution environment, secure boot is adopted through structuring root of struct and trust chain which ensures the device boot is secure.

Boot sequence

1. Device Power On
2. BootROM
3. First Stage BootLoader (FSBL)

4. U-boot
5. OP-TEE Kernel
6. Linux Kernel
7. System Running

Building Root of Trust The BootROM performs RAS authentication, AES decryption and HMAC authentication in sequence to check FSBL for tampering or being attacked. The keys for these algorithms during secure boot are saved in eFuse arrays on the board. After device is power-on or reset, the on-chip BootROM begins to perform CRC for its own integrity, it loads the FSBL into memory and compares the SHA signature of the boot image with the stored hash value. If all authentications have passed, the control will be turned over to the decrypted FSBL.

The trust chain The device could build a trust chain through validation of each stage, and it ensures that the execution environment is trusted. At each boot stage, the image of the next stage should be verified. This way the entire trust chain is being built, if verification of every stage is succesfull the device is booted up securely.

Chapter 4

OP-TEE

4.1 Open-TEE - An Open Virtual Trusted Execution Environment

4.1.1 Introduction

The following contributions are made:

- Design and implementation of a virtual TEE called Open-TEE which conforms to GlobalPlatform Specifications.
- It is shown that Open-TEE is efficient, hardware-independent and allows a developer to carry out much of the development life cycle of standard-compliant TEE applications using popular application development environments they already use.

4.1.2 Background

One initiative in TEE standardization has been undertaken by GlobalPlatform (GP), which is a cross industry, non-profit association which identifies, develops and publishes specifications that promote the secure and interoperable deployment and management of multiple applications on secure chip technology. These standardization efforts in GlobalPlatform could resolve the issue of inter-operable TEEs. However, they do not remove the obstacle in gaining access to the requisite hardware nor does simplify the task of developing and testing TAs.

4.1.3 OPen-TEE

The motivations are

- Enable developer access to TEE functionality

- Provide a fast and efficient prototyping environment
- Promote research into TEE Services
- Promote community involvement

The requirements are

- Compliance: Our framework should comply with GP's main interfaces, the Client and Core APIs.
- Hardware-independence: As a software based solution the framework should not be dependent on a particular TEE hardware environment, neither should the development system itself.
- Reasonable Performance: The framework must minimize the on-disk footprint and the memory consumption required to run it, the start-up and restart times should also be reasonable.
- Ease-of-use: The solution should be easily deployed and configured.

Architecture The architecture is split up in the following parts

- Base: Open-TEE is designed to function as a daemon process in user space. It starts executing Base, a process that encapsulates the TEE functionality as a whole. Base is responsible for loading the configuration and preparing the common parts of the system. Once initialized the Base will fork to create two independent but related processes. One process becomes Manager and the other, Launcher which serves as a prototype for TAs.
- Manager: Manager can be visualized as Open-TEE's "operating system". Its main responsibilities are: managing connections between applications, monitoring TA state, providing secure storage for a TA and controlling shared memory regions for the connected applications.
- Launcher: The sole purpose of Launcher is to create new TA processes efficiently. When it is first created, Launcher will load a shared library implementing the TEE Core API and will wait for further commands from Manager. Manager will signal Launcher when there is a need to launch a new TA. Upon receiving the signal, Launcher will clone itself. The clone will then load the shared library corresponding to the requested TA.
- Trusted Application Processes: The architecture of the TA processes is inspired by the multi-process architecture. Each process has been divided into two threads. The first handles Inter-Process Communication (IPC) and the second is the working thread, referred to respectively as the IO and TA Logic threads.

4.1.4 Evaluation

By following the GP standard and not emulating any specific TEE hardware, Open-TEE is independent of TEE hardware. TAs developed with Open-TEE can be compiled to any target TEE hardware architecture. We have verified that a non-trivial TA developed using Open-TEE has been succesfully compiled and run on a hardware TEE based on ARM TrustZone.

Chapter 5

Documentation

5.1 OP-TEE

5.1.1 Introduction

About OP-TEE

OP-TEE is a Trusted Execution Environment (TEE) designed as companion to a non-secure Linux kernel. OP-TEE implements TEE Internal Core API which is the API exposed to Trusted Applications and the TEE Client API, which is the API describing how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications. The non-secure OS is referred to as the Rich Execution Environment (REE) in TEE specifications. It is typically a Linux OS flavor as a GNU/Linux distribution or the AOSP. OP-TEE is designed primarily to rely on the Arm TrustZone technology as the underlying hardware isolation mechanism. However, it has been structured to be compatible with any isolation technology suitable for the TEE concept and goals, such as running as a virtual machine or on a dedicated CPU.

Goals

- Isolation - the TEE provides isolation from the non-secure OS and protects the loaded Trusted Applications (TAs) from each other using underlying hardware support.
- Small footprint - the TEE should remain small enough to reside in a reasonable amount of on-chip memory as found on Arm based systems.
- Portability - the TEE aims at being easily pluggable to different architectures and available hardware and has to support various setups such as multiple client OSes or multiple TEEs.

5.1.2 Interrupt Handling

Cases

Here all the cases are listed where OP-TEE OS is involved in world context switches. Optee_os executes in the secure world. World switch is done by the core's secure monitor level/mode, referred below as the Monitor.

1. When the normal world invokes the secure world, the normal world executes a SMC instruction. The SMC exception is always trapped by the Monitor. If the related service targets the trusted OS, the Monitor will switch to OP-TEE OS world execution. When the secure world returns to the normal world, OP-TEE OS executes a SMC that is caught by the Monitor which switches back to the normal world.

2. When a secure interrupt is signaled by the Arm GIC, it shall reach the OP-TEE OS interrupt exception vector. If the secure world is executing, OP-TEE OS will handle interrupt straight from its exception vector. If the normal world is executing when the secure interrupt raises, the Monitor vector must handle the exception and invoke OP-TEE OS to serve the interrupt.

3. When a non-secure interrupt is signaled by the Arm GIC, it shall reach the normal world interrupt exception vector. If the normal world is executing, it will handle straight the exception from its exception vector. If the secure world is executing when the non-secure interrupt raises, OP-TEE OS will temporarily return back to normal world via the Monitor to let normal world serve the interrupt.

Core Exceptions

All SMC exceptions are trapped in the Monitor vector. IRQ/FIQ exceptions can be trapped either in the Monitor vector or in the state vector of the executing world. When the normal world is executing, the system is configured to route:

- secure interrupts to the Monitor that will forward to OP-TEE OS
- non-secure interrupts to the executing world exception vector.

When the secure world is executing, the system is configured to route:

- secure and non-secure interrupts to the executing OP-TEE OS exception vector. OP-TEE OS shall forward the non-secure interrupts to the normal world.

Scheduling

OP-TEE yielding services are carried through standard SMC. Execution of these services can be interrupted by foreign interrupts. To suspend and restore

the service execution, optee_os assigns a trusted thread at yielding SMC entry. The trusted thread terminates when optee_os returns to the normal world with a service completion status. A trusted thread execution can be interrupted by multiple events

- Native Interrupt: In this case the native interrupt is handled by the interrupt exception handlers and once served, optee_os returns to the execution trusted thread.
- Foreign Interrupt: In this case, optee_os suspends the trusted thread and invokes the normal world through the Monitor (optee_os so-called RPC services). The trusted threads will resume only once normal world invokes the optee_os with the RPC service status.
- A trusted thread execution can lead optee_os to invoke a service in normal world: access a file, get the REE current time, etc. The trusted thread is first suspended then resumed during remote service execution.

Optee_os does not implement any thread scheduling. Each trusted thread is expected to track a service that is invoked from the normal world and should return to it with an execution status.

The OP-TEE Linux driver is designed so that the Linux thread invoking OP-TEE gets assigned a trusted thread on TEE side. The execution of the trusted thread is tied to the execution of the caller Linux thread which is under the Linux kernel scheduling decision. This means trusted threads are scheduled by the Linux kernel.

5.1.3 Notifications

There are two kinds of notifications that secure world can use to make normal world aware of some event.

- Synchronous notifications
- Asynchronous notifications

Secure world can wait for a notification to arrive. This allows the calling thread to sleep instead of spinning when waiting for something. This happens for instance when a thread waits for a mutex to become available. Synchronous notifications are limited by depending on RPC for delivery, this is only usable from a normal thread context. Secure interrupt handler or other atomic context cannot use synchronous notifications due to this.

Asynchrononous notifications uses a platform specific way of triggering a non-secure interrupt. This is done in a way suitable for a secure interrupt handler or another atomic context. This is useful when using a top half and bottom half kind of design in a device driver. The top half is done in the secure interrupt handler which then triggers normal world to make a yielding call into secure world to do the bottom half processing.

5.1.4 Memory Objects

A memory object, MOBJ, describes a piece of memory. The interface provided is mostly abstract when it comes to using the MOBJ to populate translation tables etc. There are different kinds of MOBJs describing:

- Physically contiguous memory
- Virtual memory
- Physically contiguous memory allocated from a pool
- Paged memory
- Secure copy paged shared memory

5.1.5 Memory Management Unit

Translation Tables

OP-TEE supports two translation table formats:

- Short-descriptor translation table format, available on ARMv7-A and ARMv8-A AArch32.
- Long-descriptor translation format, available on ARMv7-A with LPAE and ARMv8-A.

ARMv8-A AArch64 must use the long-descriptor translation format only. Translation table format is a static build time configuration option. The design around the translation table handling has been centered around these factors:

- Share translation tables between CPUs when possible to save memory and simplify paging.
- Support non-global CPU specific mappings to allow executing different TAs in parallel.

5.1.6 Pager

OP-TEE currently requires ¿256 KB RAM for OP-TEE kernel memory. This is not a problem if OP-TEE uses TrustZone protected DDR, but for security reasons OP-TEE may need to use TrustZone protected SRAM instead. The amount of available SRAM varies between platforms, from just a few KB up to over 512 KB. Platforms with just a few KB of SRAM cannot be expected to be able to run a complete TEE solution in SRAM. But those with 128 to 256 KB of SRAM can be expected to have a capable TEE solution in SRAM. The pager provides a solution to this by demand paging parts of OP-TEE using virtual memory.

Secure Memory

TrustZone protected SRAM is generally considered more secure than TrustZone protected DRAM as there is usually more attack vectors on DRAM. The attack vectors are hardware dependent and can be different for different platforms.

Paging Shared Secured Memory

Shared secure memory is achieved by letting several tee_pager_area using the same backing fobj. When a tee_pager_area is allocated and assigned a fobj it's also added to a list for tee_pager_areas using this fobj. This helps when a physical page is released. When a fault occurs first a matching tee_pager_area is located. Then tee_pager_pmem_head is searched to see if a physical page already holds the page of the fobj needed. If so the pagetable is updated to map the physical page at the appropriate location. If no physical page was holding the page a new physical page is allocated, initialized and finally mapped. In order to make as few updates to mappings as possible changes to less restricted, no access to read-only or read-only to read-write, is done only if the virtual address was used when the page fault occurred. Changes in the other direction have to be done in all translation tables used to map the physical page.

5.1.7 Stacks

- Secure monitor stack (128 bytes), bound to the CPU. Only available if OP-TEE is compiled with a secure monitor always the case if the target is Armv7-A but never for Armv8-A.
- Temp stack (small 1KB), bound to the CPU. Used when transitioning from one state to another. Interrupts are always disabled when using this stack, aborts are fatal when using the temp stack.
- Abort stack (medium 2KB), bound to the CPU. Used when trapping a data or pre-fetch abort. Aborts from user space are never fatal the TA is only killed. Aborts from kernel mode are used by the pager to do the demand paging, if pager is disabled all kernel mode aborts are fatal.
- Thread stack (large 8KB), not bound to the CPU instead used by the current thread/task. Interrupts are usually enabled when using this stack.

5.1.8 Shared Memory

Shared Memory is a block of memory that is shared between the non-secure and the secure world. It is used to transfer data between both worlds. The shared memory is allocated and managed by the non-secure world, i.e. the Linux OP-TEE driver. Secure world only considers the individual shared

buffers, not their pool. Shared memory buffer references manipulated must fit inside one of the shared memory areas known from the OP-TEE core.
OP-TEE supports two kinds of shared memory areas: an area for contiguous buffers and an area for noncontiguous buffers. At least one has to be enabled.

5.1.9 Thread Handling

OP-TEE core uses a couple of threads to be able to support running jobs in parallel. There are handlers for different purposes. In thread.c you will find a function called thread_init_primary(...) which assigns init_handlers (functions) that should be called when OP-TEE core receives standard or fast calls, FIQ and PSCI calls. There are default handlers for these services, but the platform can decide if they want to implement their own platform specific handlers instead.

Synchronization Primitives

- Spin-lock
- Mutex
- Condvar

5.1.10 Secure Boot

There are no additional specific build options for the verification of OP-TEE. If the authentication framework is enabled and specified the BL32 build option when building TF-A, the BL32 related certificates will be created automatically by the cert_create tool, and then these certificates will be verified during booting up.

Authentication Framework & Chain of Trust

The Trusted Firmware-A (TF-A) framework fulfills the following requirements:

- Possiblity for a platform port to specify the Chain of Trust in terms of certificate hierarchy and the mechanisms used to verify a particular certificate.
- The framework should distinguish between:
  - The mechanism used to encode and transport information.
  - The mechanism used to verify the transported information.

The framework has been designed following a modular approach.

Authentication Modes

The AM supports the following authentication methods:

- Hash
- Digital Signature

The platform may specify these methods in the Chain of Trust (CoT) in case it decides to define a custom CoT instead of reusing a predefined one. If a data image uses multiple methods, then all the methods must be a part of the same CoT. The number and type of parameters are method specific. These parameters should be obtained from the parent image using the IPM.

Specifying a Chain of Trust

A CoT can be described as a set of image descriptors linked together in a particular order. The order dictates the sequence in which they must be verified. Each image has a set of properties which allow the AM to verify it.

5.1.11 Secure Storage

Secure Storage in OP-TEE is implemented according to what has been defined in GlobalPlatform's TEE Internal Core API. This specification mandates that it should be possible to store general-purpose data and key material that guarantees confidentiality and integrity of the data stored and the atomicity of the operations that modifies the storage. There are currently two secure storage implementations in OP-TEE:

- Relying on the normal world (REE) file system. It is enabled at compile time by CFG_REE_FS=y.
- Making use of the Replay Protected Memory Block (RPMB) partition of an eMMC device, and is enabled by setting CFG_RPMB_FS=y.

They can be used simultaneously.

Key Manager

Key manager is a component in TEE file system, and is responsible for handling data encryption and decryption and also management of the sensitive key materials. There are three types of keys used by the key manager: the Secure Storage Key (SSK), the TA Storage Key (TSK) and the File Encryption Key (FEK).

Hash Tree

The hash tree is responsible for handling data encryption and decryption of a secure storage file. The hash tree is implemented as a binary tree where each node in the tree protects its two child nodes and a data block. The meta data is stored in a header which also protects the top node. All fields (header, nodes, and blocks) are duplicated with two versions, 0 and 1, to ensure atomic updates.

5.1.12 Trusted Application

Pseudo Trusted Application

A Pseudo Trusted Application is not a Trusted Application. A Pseudo TA is not a specific entity, it is an interface exposed by the OP-TEE Core to its outer world: to secure client Trusted Applications and to non-secure client entities. These are implemented directly to the OP-TEE core tree and are built along with and statically built into the OP-TEE core blob.

User Mode Trusted Applications

User Mode Trusted Applications are loaded (mapped into memory) by OP-TEE core in the Secure World when something in Rich Execution Environment (REE) wants to talk to that particular application UUID. They run at a lower CPU privilege level than OP-TEE core code. In that respect, they are quite similar to regular applications running in the REE, except that they execute in Secure World.
Trusted Application benefit from the GlobalPlatform TEE Internal Core API as specified by the GlobalPlatform TEE specifications. There are several types of user mode TAs, which differ by the way they are stored.

Early TA The so-called early TAs are virtually identical to the REE FS TAs, but instead of being loaded from the Normal World file system, they are linked into a special data section in the TEE core blob. Therefore, they are available even before tee-supplicant and the REE's filesystems have come up. Please find more details in the early TA commit.

Secure Storage TA These are stored in secure storage. The meta data is stored in a database of all installed TAs and the actual binary is stored encrypted and integrity protected as a separate file in the untrusted REE filesystem (flash). Before these TAs can be loaded they have to be installed first, this is something that can be done during initial deployment or at a later stage.

Loading and Preparing for TA Execution

User mode TAs are loaded into final memory in the same way using the user mode ELF loader ldelf. The different TA locations have a common interface towards ldelf which makes the user mode operations identical regardless of how the TA is stored. After ldelf has returned with a TA prepared for execution it still remains in memory to serve the TA if dlopen() and friends are used. ldelf is also used to dump stack trace and detailed memory mappings if a TA is terminated via an abort.

5.1.13 Virtualization

OP-TEE has experimental virtualization support. This is when one OP-TEE instance can run TAs from multiple virtual machines. OP-TEE isolates all VM-related states, so one VM can't affect another in any way. With virtualization support enabled, OP-TEE will rely on a hypervisor, because only the hypervisor knows which VM is calling OP-TEE. Also, naturally the hypervisor should inform OP-TEE about creation and destruction of VMs. Besides, in almost all cases, hypervisor enables two-stage MMU translation, so VMs does not see real physical address of memory, instead they work with intermediate physical addresses (IPAs). On other hand OP-TEE can't translate IPA to PA, so this is a hypervisor's responsibility to do this kind of translation. So, hypervisor should include a component that knows about OP-TEE protocol internals and can do this translation. We call this component "TEE mediator" and right now only XEN hypervisor have OP-TEE mediator.

5.1.14 SPMC

The SPMC is a critical component in the FF-A flow. Some of its major responsibilities are:

- Initialisation and runtime management of the SPs. The SPMC component is responsible for initialisation of the Secure Partitions (loading the image, setting up the stack, heap, ...).
- Routing messages between endpoints. The SPMC is responsible for passing FF-A messages from normal world to SPs and back. It also responsible for passing FF-A messages between SPs.
- Memory management. The SPMC is responsible for the memory management of the SPs. Memory can be shared between SPs and between a SP to the normal world.

Secure Partitions

Secure Partitions (SPs) are the endpoints used in the FF-A protocol. When OP-TEE is used as a SPMC SPs run primarily inside S-EL0. OP-TEE will use

FF-A for it transport layer when the OP-TEE CFG_CORE_FFA=y configuration flag is enabled. The SPMC will expose the OP-TEE core, privileged mode, as an secure endpoint itself. This is used to handle all GlobalPlatform programming mode operations. All GlobalPlatform messages are encapsulated inside FF-A messages. The OP-TEE endpoint will unpack the messages and afterwards handle them as standard OP-TEE calls. This is needed as TF-A (S-EL3) does only allow FF-A messages to be passed to the secure world when the SPMD is enabled. SPs run from the initial boot of the system until power down and don't have any built-in session management compared to GPD TEE TAs. The only means of communicating with the outside world is through messages defined in the FF-A specification. The context of a SP is saved between executions.

Running and exiting SPs

The SPMC resumes/starts the SP by calling the sp_enter(). This will set up the SP context and jump into S-EL0. Whenever the SP performs a system call it will end up in sp_handle_svc(). sp_handle_svc() stores the current context of the SP and makes sure that we don't return to S-EL0 but instead returns to S-EL1 back to sp_enter(). sp_enter() will pass the FF-A registers (x0-x7) to spmc_sp_msg_handler(). This will process the FF-A message.