Providing Root of Trust for ARM TrustZone using On-Chip SRAM

Shijun Zhao
TCA Laboratory, ISCAS
zqyzsj@gmail.com

Qianying Zhang
TCA Laboratory, ISCAS
zhangqy@tca.iscas.ac.cn

Guangyao Hu
Beijing Vion Technology, Inc
guangyaohu@gmail.com

Yu Qin
TCA Laboratory, ISCAS
qin_yu@tca.iscas.ac.cn

Dengguo Feng
TCA Laboratory, ISCAS
fengdengguo@tca.iscas.ac.cn

Abstract

We present the design, implementation and evaluation of the root of trust for the Trusted Execution Environment (TEE) provided by ARM TrustZone based on the on-chip SRAM Physical Unclonable Functions (PUFs). We first implement a building block which provides the foundations for the root of trust: secure key storage and truly random source. The building block doesn't require on or off-chip secure non-volatile memory to store secrets, but provides a high-level security: resistance to physical attackers capable of controlling all external interfaces of the system on chip (SoC). Based on the building block, we build the root of trust consisting of seal/unseal primitives for secure services running in the TEE, and a software-only TPM service running in the TEE which provides rich TPM functionalities for the rich OS running in the normal world of TrustZone. The root of trust resists software attackers capable of compromising the entire rich OS. Besides, both the building block and the root of trust run on the powerful ARM processor. In one word, we leverage the on-chip SRAM, commonly available on mobile devices, to achieve a low-cost, secure, and efficient design of the root of trust.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

TrustZone; Trusted Execution Environment; TPM Service; Root of Trust; on-chip SRAM

1. INTRODUCTION

Mobile devices are offering more and more functionalities, some of which are security-critical, such as e-commerce and banking. Modern mobile OSes are usually equipped with sandbox mechanisms [1, 2, 16] to prevent malicious applications illegally gaining access to sensitive data or compromising other applications, i.e., provides a Trusted Execution Environment (TEE) for mobile applications. However, modern mobile OSes, i.e., the Trusted Computing Base (TCB) that mobile applications rely on, are so complex that it is difficult to ensure the absence of vulnerabilities which hackers can exploit to gain control of OSes and then disable their sandbox mechanisms. Thus, it's far from trivial to provide a TEE for mobile applications.

To address this problem, design of trusted systems providing TEEs for sensitive-critical application code with small TCBs is introduced. Such design can minimize potential security vulnerabilities of TCBs which help attackers to compromise systems. To this end, mainstream CPU designers and manufactures introduce new hardware primitives to their architectures. Intel and AMD propose the late launch technology by extending the x86 instruction set with their respective Trusted eXecution Technology (TXT) [32] and Secure Virtual Machine (SVM) [3] initiatives, which allows a software module running in an environment isolated from the entire OS. Some famous trusted systems have been implemented based on the late launch technology, such as Flicker [46] and TrustVisor [45]. ARM presents TrustZone technology [4], which enables secure services to run in the "secure world" of the processor. Several trusted systems for mobile devices have been implemented leveraging ARM TrustZone technology, such as Nokia's On-board Credentials [20, 34], Sierraware's SierraTEE [57], and TOPPERS Project's SafeG [55].

Actually, a CPU with late launch or TrustZone security extensions only provides an "isolated" execution environment, but not a "trusted" one since it can't attest to the user or an external verifier that the software running inside the environment is untampered and trustworthy. At present, the state-of-the-art for attestation is to compute a signature with an attestation key over the software's measurement, and the software's measurement and the attestation key are securely stored by the root of trust. Thus, the root of trust provides a way to establish trust in the execution environment. So only an isolated execution environment equipped with a root of trust is a real "trusted" execution environment (TEE).

Both Intel and AMD specify the TPM [63] as the root of

trust for late launch. Once the late launch instruction is triggered, the software component that will run in the hardware-protected environment is atomically measured to the TPM. After that, an attestation identity key of the TPM attests the identity of the software component to an external verifier by signing the measurement. However, ARM doesn't specify the root of trust for TrustZone. Current trusted systems and security services [20, 34, 56, 15, 39] based on TrustZone usually assume the availability of a unique device key which is accessible only inside the secure world of TrustZone, and use the device key to serve as the root of trust. Unfortunately, such device keys are not always available on mobile devices. For example, Nuno Santos et al. designed a trusted language runtime [56] which required a device key to serve as the platform identity. They implemented a prototype on Nvidia's Tegra 250 Dev Kit. However, this platform is not equipped with such a device key. So they had to hard-coding a software key in their implementation.

The device key should be stored securely and available after a reboot. After making a survey on popular TrustZone-enabled development platforms equipped with device keys (Xilinx Zynq-7000 All Programmable SoC, Samsang Exynos™ 5 SoC, FreeScale i.MX53, and OMAP™ 3, 4 SoC families), we find that current secure key storage mechanisms for device keys usually rely on the Battery-backed RAM (BBRAM) or eFuse technology. However, the way of using these secure key storage mechanisms to provide a root of trust has the following disadvantages:

1. The BBRAM mechanism requires a battery in order to provide persistent storage across reboot. This approach induces additional cost and requires physical room to add a battery.
2. The eFuse mechanism is inflexible. The eFuse technology is a one-time programmable memory, and once the device key has been designed into the IC it can never be changed again. However, key update is desirable in practice. For example, many mobile systems use regular key updates to prevent side-channel attacks [33, 38], and users can improve their security level by updating their device key to a larger one.
3. These mechanisms only provides secure key storage, and they are not sufficient to serve as a root of trust. Building the root of trust in mobile devices requires a secure random number generator (RNG). For example, RNG is necessary for generating attestation identity keys. Thus, in order to build their roots of trust, mobile devices should be equipped with hardware RNGs, which add product cost. However, to the best of our knowledge, not all devices implement a hardware RNG, such as Zynq-7000 AP SoC.

In this paper, we use a promising technology, SRAM PUFs, to overcome above disadvantages. First, we build a building block in the secure on-chip memory (OCM) which provides the foundations for a root of trust:

- A primary seed extracted by a fuzzy extractor from the start-up value of the on-chip SRAM. The primary seed is the root of the secure storage as we use it to derive the unique device key.
- A truly random seed extracted from the noise contained in the start-up value of the on-chip SRAM. The random seed is used to build a secure RNG for the secure OS.

Besides, the building block also provides secure boot of the secure OS and secure services running inside the secure world of TrustZone. The secure boot process is mandatory for TrustZone as the image of the secure OS and secure services is loaded from non-secure persistent storage such as flash or SD cards, which can be easily tampered by malicious applications in the normal world. Different from current mechanisms providing roots of trust, our approach doesn't require persistent secure key storage and a hardware RNG, but only a few kilos on-chip SRAM, which is available on commodity ARM platforms. Our approach also features flexibility for key updates as it is easy for the fuzzy extractor to replace the primary seed with a new one which can derive a new device key. Besides requiring no secure hardware resources such as BBRAM or eFuse, the building block achieves high security: it resists physical attackers capable of controlling all the interfaces of the SoC in the platform.

Then, we provide the root of trust for secure services and the rich mobile OS by leveraging the device key pair derived from the primary seed:

- We leverage the device key to provide seal and unseal primitives for secure services, which ensure that only the specified secure service and platform can access user data and can also be used to store critical data by secure services. The seal/unseal primitives implicitly attest to the user the state of the platform and the secure service, and can provide secure storage for secure services. Thus, the seal/unseal primitives can be seen as the root of trust for secure services.
- We integrate a software-only TPM service into the secure world of TrustZone, and use the device key as the Endorsement Key (EK) of the TPM service. The TPM service serves as the root of trust for the rich mobile OS running in the normal world.

The root of trust, i.e., the seal and unseal primitives and the TPM service, is implemented in the secure region of the main memory of the device. We don't implement the root of trust in the OCM for the reason that the size of OCM is quite limited. The isolation provided by TrustZone protects the root of trust from software attacks from the mobile OS. For the root of trust, we don't consider physical attacks such as physical tampering the main memory of the device as this kind of attacks falls outside of the protection capabilities of TrustZone.

Finally, we implement above design on real TrustZone hardware, and present a thorough evaluation of our implementation, including the TCB size, a quantitative analysis of the SRAM PUF, randomness tests on the secure RNG, and performance of the root of trust. The results show that the

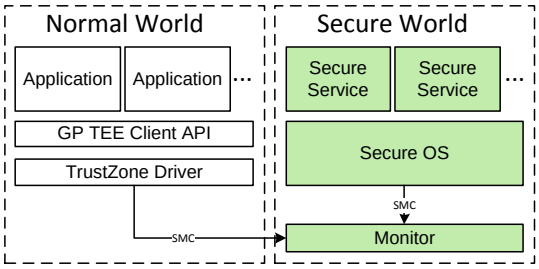

Figure 1: TrustZone Overview

SRAM PUF is secure and unique enough to provide the trust anchor for mobile devices, and can provide foundations for the root of trust by adding only 3.2K lines of code to the TCB. The performance evaluation shows the root of trust is very efficient.

2. BACKGROUND

This section describes ARM TrustZone, the on-chip memory, PUF, fuzzy extractor and truly random number generation (TRNG), which are the key technologies used in our design.

2.1 ARM TrustZone

TrustZone is a set of hardware security extensions to ARM SoC covering the processor, memory, and peripherals [9]. TrustZone Address Space Controllers (TZASC) can partition DRAM into distinct memory regions, and designate a memory region as secure or normal. TrustZone Memory Adapters (TZMA) provide a similar functionality for the OCM. TrustZone-aware DMA controllers prevent a peripheral assigned to the normal world from performing a DMA transfer to or from the secure world memory. TrustZone Protection Controllers (TZPC) can configure peripherals to be secure or normal. These isolation mechanisms partition all of the SoC's hardware resources into two worlds: the secure world and the normal world. The world in which the processor is executing is indicated by an *NS* bit, which is propagated through the system bus. The TrustZone-enabled bus fabric ensures that no secure world resources can be accessed by normal world components.

System designers can leverage TrustZone to run a small secure OS and some secure services in the secure world, and run untrusted software in the normal world. The secure OS manages secure hardware sources, and dispatches the secure services. Usually the secure services are security-sensitive code, and the untrusted software is full blown mobile operating systems such as iOS, Android, and Windows 8. As the processor only runs in one world at a time, to run in the other world requires context switch. This is done via a special instruction called the Secure Monitor Call (*smc*). When the *smc* instruction is triggered, the processor switches into a *monitor* mode which performs the context switch and allows messages exchange between the two worlds. In order to facilitate an application in the normal world to connect to and invoke a secure service in the secure world, the GlobalPlatform consortium develops the TEE client API specification [24], see Figure 1.

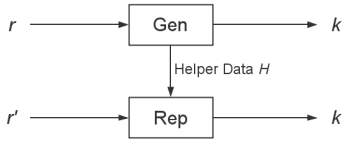

Figure 2: Fuzzy Extractor Overview

2.2 On-Chip Memory

Mobile devices are SoC based. The OCM of an SoC commonly consists of BootROM and SRAM. The BootROM stores the initial boot code when the platform is powered up, which loads a bootloader into the SRAM from external non-volatile memory such as flash or SD cards. In order to establish a chain of trust, the BootROM needs to authenticate the bootloader.

The on-chip SRAM is very fast memory in the SoC and connects the processor via internal connection buses. Since the on-chip SRAM has no address or data lines at device pins, it is secure against physical attackers capable of controlling all the external interfaces of the SoC. In a TrustZone-enabled SoC, the on-chip SRAM can be isolated from the mobile OS, thus it's also capable of resisting software attacks. However, the on-chip SRAM is quite limited, usually only dozens or hundreds of kilobytes. That's why we only implement the building block in the OCM, and put the root of trust in the main memory.

2.3 PUF, Fuzzy Extractor, and TRNG

The concept of PUFs is first introduced by Pappu et al. [50, 51], which describes such hardware components that when evaluated by a stimulus (challenge) they provide a noisy response that depends on manufacturing process variations of the hardware components. Since the introduction, many types of PUFs have been proposed in the literature, e.g. optical PUFs [50], Silicon PUFs [22, 23], Coating PUFs [66], Ring Oscillator PUFs [60], reconfigurable PUFs [36, 19], SRAM PUFs [27, 29], Butterfly PUFs [35], Flip-Flop PUFs [40], Buskeeper PUFs [58], and Flash memory-based PUFs [69].

PUFs should satisfy both high robustness and uniqueness. The high robustness means that when a PUF is evaluated by the same challenge over and over again it should produce responses up to a limited amount of noise. The high uniqueness means that the responses of different PUFs to the same challenge should be independent. These two properties enable each PUF to extract a reliable and unique key by applying a fuzzy extractor introduced by Linnartz et al. [37] (as shielding function) and Dodis et al. [18]. PUFs together with fuzzy extractors present an efficient approach for secure key storage: it directly extracts secure keys from responses of PUFs, eliminating the need for storing keys on secure non-volatile memory. This approach reduces hardware attack surfaces as keys are not present when devices are powered off, and resists clone attacks as PUFs are physically unclonable.

Up to now, many fuzzy extractor implementations have been

proposed [14, 13, 41, 42, 67, 43]. A fuzzy extractor consists of a pair of procedures, generate (**Gen**) and reproduce (**Rep**), see Figure 2. The Gen procedure extracts a key k from the PUF's response r and generates a helper data H which is not sensitive. The Rep procedure reproduces k from a noisy response r' under the help of H. The key k is randomly chosen by the owner or the issuer of the PUF during the Gen procedure, so it is easy to bind a new key k' to the PUF by running Gen again and obtaining a new helper data H'. This feature makes it easy to perform key update mechanisms.

Another application area of PUFs is truly random number generation (TRNG). Take SRAM PUFs for example, part of the SRAM cell bits show noisy behaviour, and the entropy in these noisy bits can be leveraged for random number generation. Several solutions [30, 61, 69, 68] have been proposed in the literature.

Recently, invasive attacks on PUFs and SRAM PUFs in particular have been proposed [28, 49]. However, SRAM PUFs reach at least the same security level as the conventional mechanisms for secure storage, which are inherently susceptible to such invasive attacks as memory contents are retained even when the IC is no longer powered on. Moreover, such invasive attacks on SRAM PUFs require expensive laboratory equipment and the cost is high, so it's uneconomical for attackers to only obtain a device unique secret. Some potential countermeasures [28, 49] against invasive attacks on SRAM PUFs are proposed, and SRAM PUFs are still a promising technology.

3. ADVERSARY MODELS AND DESIGN PROPERTIES

3.1 Adversary Models

We distinguish between an adversary against the building block and an adversary against the root of trust. The former is stronger than the later as it is able to mount some physical attacks.

Adversary Model for the Building Block. We assume a sophisticated adversary with physical access to all external interfaces of the SoC. In particular, the adversary can compromise all software running in the normal world such as applications and the mobile OS, and he can physically attach malicious peripherals such as DDR memory devices and even DMA-capable devices to the SoC as he has access to external interfaces of the SoC. We don't consider side-channel attacks and sophisticated hardware attacks, such as monitoring the high-speed internal buses in SoC using microscopic logic probes and extracting the contents of SRAM at startup by laser stimulation analysis.

Adversary Model for the Root of Trust. The adversary against the root of trust is a strong software attacker. He can compromise the mobile OS and have access to the interfaces of the root of trust, which is provided to the mobile OS through TrustZone mechanisms. However, the adversary cannot launch physical attacks on the root of trust which are outside of the scope of the protection provided by TrustZone technology [8].

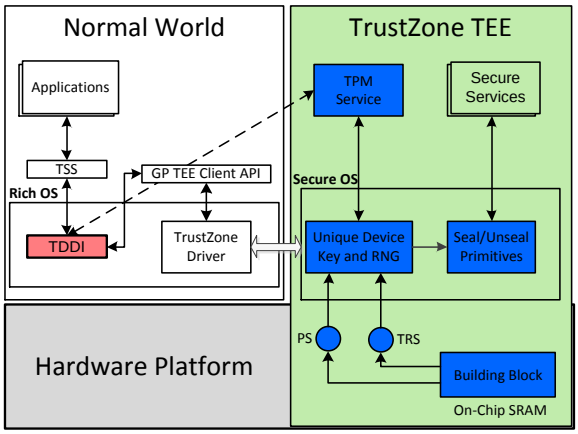

Figure 3: Architecture of our Design

3.2 Design Properties

We describe the desired properties for our design.

1. **Secure.** The root of secure storage for the root of trust, i.e., the primary seed, should resist physical attacks. This guarantees that even if the device is attacked by a physical attacker, we can deploy a new and secure root of trust for the device. The root of trust should be completely isolated from untrusted software such as the mobile OS, and provide trusted computing functionalities for both the mobile OS and secure services.

2. **Efficient.** The root of trust should run on the powerful ARM processor, so as to provide high performance compared to the hardware root of trust, whose computing power might be very limited. Take the TPM 1.2 chip for example, it only operates at 33M Hz, which makes it a bottleneck for many security schemes and renders it impractical for use in situations with demanding performance requirements.

3. **Economical.** No requirements for hardware-based secure storage (e.g., secure non-volatile memory) and the hardware RNG, which decreases manufacturing cost and complexity of mobile devices.

4. **Flexible.** The unique device key should be updated easily even after the device has left the production facility. This property makes it easy to replace the root of trust when the device is corrupted by physical attackers, and to adopt key update mechanisms to prevent side-channel attacks.

4. DESIGN

The goal of this work is to provide the root of trust for TrustZone-enabled platforms in an economical and flexible way, which allows a designer to develop a TEE to provide trusted computing functionalities (for secure services and the mobile OS) with no need for additional security hardware resources, and allows the device owner to re-deploy the root of trust after the device is corrupted by physical

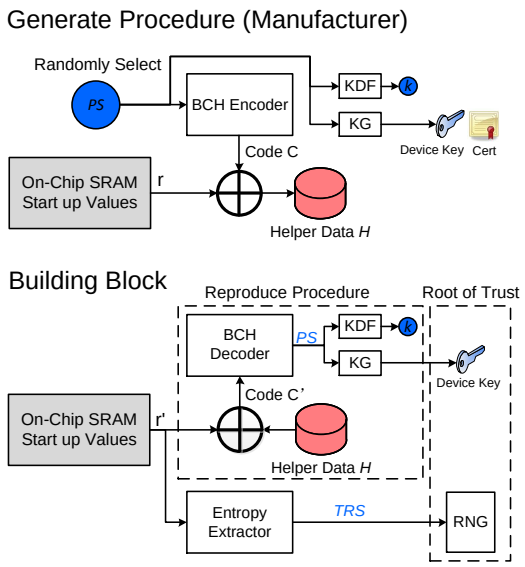

Figure 4: The Generate Procedure and Building Block

attacks. We further seek to establish a chain of trust from the root of trust to the normal world, which enables to boot a clean mobile OS for the normal world. Figure 3 illustrates the detailed architecture of our design, which consists of the following components: the building block, the secure RNG, the seal and unseal primitives for secure services, the TPM service, and a TPM device driver interface (TDDI) providing user-friendly interfaces of the TPM service. The following gives a brief introduction of these components.

The primary goal of the building block is to provide the foundations for building a root of trust: a primary seed (PS) and a truly random seed (TRS) extracted from the start-up values of the on-chip SRAM cells. The building block also provides secure boot for the secure OS and secure services. Secure boot is important for the secure OS and secure services, as their image is stored in external non-volatile memory which is subject to attacks. As the building block resides in the secure on-chip SRAM, it achieves a high security level: resistance against simple physical attacks on the SoC.

Besides securely booting the secure OS, the building block transfers the TRS and the unique device key derived from the PS to the secure OS. Based on the unique device key and the TRS, the secure OS builds the root of trust for both secure services and the rich OS: the seal/unseal primitives providing implicit attestation mechanism and secure storage for secure services in the TEE, and the TPM service which provides rich trusted computing functionalities for the normal world. To facilitate the use of the TPM service, a kernel module called TDDI simulating a hardware TPM driver interface is provided in the normal world.

4.1 Building Block in OCM

The building block consists of the reproduce procedure of a fuzzy extractor and a truly random number (TRN) extractor which extract a primary seed PS and a truly random seed TRS respectively from the on-chip SRAM start-up value. The PS is associated with the device during the generate procedure of the fuzzy extractor. Figure 4 illustrates the generate procedure, which is performed by the manufacturer, and the building block, which consists of the reproduce procedure of the fuzzy extractor and the entropy extractor.

4.1.1 Generate Procedure

This procedure is run by the manufacturer while the device is in the production facility. It takes as input the on-chip SRAM start-up value r (r is a binary string consisting of start-up values of SRAM cells), then performs the following steps:

1. Receive a large value PS which is randomly selected by the manufacturer, then encode PS with the BCH error correction code to obtain a code $C = \mathsf{BCH}_{\mathsf{Enc}}(PS)$.
2. The code C is XOR-ed with r to create the helper data H, which can be stored in insecure non-volatile memory of the device and will be used to reproduce the same primary seed PS during the reproduce procedure.
3. The PS is transferred to a key derivation function (KDF) and a deterministic key generation (KG) algorithm, which will generate a symmetric key k and a unique public/private key pair (pk, sk) respectively. The symmetric key protects the secrecy of the secure OS (including secure services) by encrypting its image, and the encrypted image is stored on the device. The manufacture also issues a certificate $Cert_{pk}$ by signing pk, the standard measurements of the building block and the image of the secure OS. The two measurements will be used to build a chain of trust on this device.
4. Finally, the manufacturer stores the helper data H, the encrypted secure OS, and $Cert_{pk}$ in the non-volatile memory of the device.

In this phase, the device manufacturer implicitly embed the primary seed PS into the device. Here the "implicitly" means that the primary seed is not physically stored on the device, but can be re-generated during runtime. The manufacturer also issues a certificate for the unique device key derived from PS.

4.1.2 Reproduce Procedure of the Building Block

This procedure takes as input the on-chip SRAM start-up value r' (measured and transferred by the BootROM, which is the first code running on the device after powered on). SRAM is a kind of PUF, and its start-up value is noisy because of the manufacturing process variations, so r' is a noisy variant of the initial SRAM start-up value r. Thus the BCH error correction code is used to eliminate the noise. The reproduce procedure first XORs r' with the helper data H to generate a noisy BCH code $C' = r' \oplus H$. Then code C' is transferred to the BCH decoder, which eliminates noise and generates the same PS that the manufacturer selects

during the generate procedure. Finally, the symmetric key k used to decrypt the secure OS and the unique device key are derived from PS.

4.1.3 Entropy Extractor of the Building Block

As not all SRAM start-up bit cells are noisy, we need an entropy extractor to condense the entropy in the noisy SRAM start-up bits. We use the randomness extractor proposed by [17], which stands for the state-of-the-art secure RNG construction, as our entropy extractor to produce a truly random seed TRS full of entropy. The TRS will be given to the root of trust, who will build a RNG by feeding TRS to a cryptographic pseudo-random generator such as AES block cipher in counter mode. Let η be the output length of our entropy extractor, i.e., the length of TRS, h be the min-entropy of SRAM start-up value, and I be an SRAM start-up binary string. According to the construction of the entropy extractor proposed in [17], accumulating n bits entropy to the TRS requires at least $\lceil \eta/h \rceil$ SRAM bits. As the *seed* in our construction also requires η SRAM start-up bits, the length of I is at least $\lceil \eta/h \rceil + \eta$. The pseudocode of our concrete construction is showed in Algorithm 1. To fix the parameter h and thus determine the amount of SRAM bits required by the entropy extractor, we estimate the min-entropy of SRAM start-up value in Section 6.3.

Algorithm 1 Entropy Extractor

`INPUT:` I, η, h
`OUTPUT:` TRS
1: $TRS \leftarrow 1^{\eta}$
2: $seed \leftarrow$ Read first η bits from I
3: **for** $i = 0 \rightarrow \lceil \eta/h \rceil$ **do**
4: $\quad T \leftarrow$ Read next η bits from I
5: $\quad TRS = TRS \cdot seed + T$, where $\cdot$ and $+$ are operations over the field F_{2^η}.
6: **end for**
7: **return** TRS

4.2 Root of Trust in Main Memory

We first show our construction of the secure RNG following the instructions of [17]. Let $||$ denote the concatenation of two stings, $[S]_1^l$ denote the first l bits of S. Our construction leverages AES function in counter mode as shown in Algorithm 2. Based on the unique device key pair (sk, pk) and the secure RNG, we design the seal and unseal primitives and the secure TPM service.

Algorithm 2 Random Number Generator

`INPUT:` TRS
`OUTPUT:` A random number R
1: $X \leftarrow 1^{\eta}$
2: Set state $S \leftarrow TRS$
3: $k = [X \cdot S]_1^{256}$
4: $(k', R) = (\mathsf{AES}_k(0)||\mathsf{AES}_k(1), \mathsf{AES}_k(2))$
5: Set $[S]_1^{256} = k'$
6: **return** R

4.2.1 The Simple Seal/Unseal Primitives

The simple seal and unseal primitives bind secure data with both the platform and the particular secure service through cryptographic encryption and hashing. A user seals his data *data* to some secure service S running on some device D by the following steps. The user first derives a symmetric key from the code identity of S (the cryptographic hash over S's binary): $k_s = \mathsf{KDF}(\mathsf{hash}(S))$, then encrypts *data* with k_s to get $T = \mathsf{Enc}_{k_s}(data)$, and finally encrypts T with the device public key to get an encrypted blob $B = \mathsf{Enc}_{pk}(T)$. Unsealing is the reverse: using D's private key sk to decrypt B to obtain the symmetrically encrypted blob T, deriving the symmetric k_s using the code identity of the secure service, and finally decrypting *data* using the derived symmetric key.

As the user data is first encrypted by the key derived from the secure service, and then is encrypted by the device key, only the device possessing the device key and running the legitimate secure service can obtain the user data. We will show in Section 4.3 that only the platform running the legitimate secure OS can get the device key, so the seal/unseal primitives guarantee that only platform in a secure state can access the user data. In another word, the seal/unseal primitives implicitly attest to users the state of the TEE, i.e., an implicitly attestation mechanism. This mechanism can also be used by secure services to store sensitive data.

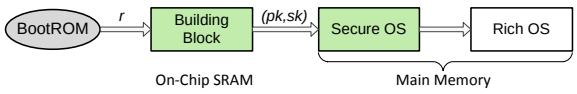

Figure 5: The Chain of Trust

4.2.2 TPM service

The TPM service provides rich trusted computing functionalities for the mobile OS, such as secure storage, measurement, and attestation. These functionalities can be used to bootstrap a trusted mobile OS and further help the OS to extend the chain of trust to applications, i.e., help the normal world run in a trusted state. We will give a detailed description of the chain of trust from the BootROM to the mobile OS in Section 4.3.

The TDDI facilitates the use of the TPM service by simulating a hardware TPM driver interface [64], see Figure 3. It forwards all commands to and receives responses from the TPM service through the GP TEE Client API [24]. TDDI makes the TPM service compatible with hardware TPMs at very low level, so applications previously leveraging hardware TPMs and the TCG Software Stack (TSS) such as Trousers [72] and jTSS [65] can leverage the TPM service without any modification.

4.3 Chain of Trust

Chain of trust is essential for TrustZone, as code running in the secure world is stored in the insecure non-volatile storage of the device, which is vulnerable to attacks from the normal world. Establishing a chain of trust from the BootROM to the mobile OS can protect the secrecy and integrity of the code running inside the secure world, and ensure to boot a trustworthy mobile OS. We now show how to build a chain of trust under our design, see the boot flow under our design in Figure 5.

When the device is powered up, the ARM processor runs in the secure mode and immediately executes the immutable code from the BootROM, which is laid down during chip fabrication and implicitly trusted. The BootROM first verifies the integrity of the building block: this can be done by measuring the image of the building block and using the manufacturer public key to verify whether the measurement is signed by the manufacturer[1]. Then the BootROM reads the start-up value r of the on-chip SRAM, initializes the on-chip SRAM and loads the building block into it. If the integrity verification succeeds, the BootROM transfers r to the building block and runs the building block in the on-chip SRAM, else stops the startup.

The building block reproduces the primary seed PS by feeding the SRAM start-up value r and the helper data H to the reproduce procedure of the fuzzy extractor, and then derive a symmetric key k and the unique device key pair (pk, sk) from PS. The symmetric key k is used to decrypt the image of the secure OS and secure services. Then the building block checks the integrity of the image by verifying whether the measurement of the image is signed by the manufacturer. If the verification succeeds, the building block loads the image to the secure region of the main memory, transfers (pk, sk) to the secure OS, and runs the secure OS, else stops the startup. Before running the secure OS, the building block erases all the information in the on-chip memory, in particular, the SRAM start-up value, the primary seed and the symmetric key.

When the secure OS starts up, it initializes the services contained in the image (including the TPM service). Then it measures the image of the normal OS and extends the measurement to Platform Configuration Registers (PCRs) of the TPM service. Finally, the secure OS runs the normal OS. The normal OS can continually extend the chain of trust with the TPM service.

A Brief Security Analysis. The BootROM ensures the integrity of the building block, the building block ensures both the secrecy and integrity of the secure OS, and the secure OS records the integrity of the normal OS. Thus, a complete chain of trust is established since the power-up of the device. Our design of the TPM service helps the normal world to extend the chain of trust to applications just like using a hardware TPM.

4.4 Key Update

A significant benefit of our design of the root of trust in comparison with existing physical secure key storage mechanisms is that our design enables to deploy flexible key update mechanisms. We propose a key update protocol through which the device owner can change his device key regularly.

Let $x \xleftarrow{\$} S$ denote assigning x a value uniformly chosen from a set S, $\{0,1\}^n$ denote the set of binary strings of length n, η denote the security parameter, D be the device, M be the manufacturer. D has a unique device public/private key pair (pk, sk), and the manufacture has a key pair (pk_M, sk_M) for signing. We denote a signature scheme by a triple $(\mathsf{KG}, \mathsf{SIG}, \mathsf{VER})$ where KG is a key generation algorithm, SIG is a signature algorithm, and VER is a verification algorithm. We use im_B and im_S to denote the image of the building block and secure OS respectively, and use c to denote a mono counter. We define the protocol in Figure 6, the option fields between square brackets are used to resist the downgrading attack which will be described later. Our key update protocol can be summarized in 3 simple steps (in the following description, the secure OS includes secure services running in the secure world):

1. $\mathsf{M} \rightarrow \mathsf{D}$: $nonce \xleftarrow{\$} \{0,1\}^\eta$
2. $\mathsf{D} \rightarrow \mathsf{M}$: $sig = \mathsf{SIG}_{sk}(nonce[,c])[,c]$
3. $\mathsf{M} \rightarrow \mathsf{D}$: M first performs the following steps:
   (a) Verify sig by $\mathsf{VER}_{pk}(sig)$
   (b) Choose $PS' \xleftarrow{\$} \{0,1\}^\eta$
   (c) $H' = r \oplus \mathsf{BCH}_{\mathsf{Enc}}(PS')$ where r is SRAM values collected during the Gen procedure
   (d) $k' = \mathsf{KDF}(PS'[,c])$, $(pk', sk') = \mathsf{KG}(PS'[,c])$
   (e) $Blob' = \mathsf{AES}_{k'}(im_S)$
   (f) $Cert_{pk'} = \mathsf{SIG}_{sk_M}(pk', \mathsf{Hash}(im_B), \mathsf{Hash}(im_S))$
   Finally, M sends $(H', Blob', Cert_{pk'})$ to D.
4. D verifies $Cert_{pk'}$, deletes previous $(H, Blob, Cert_{pk})$, and stores $(H', Blob', Cert_{pk'})$.

Figure 6: Key Update Protocol

1. **Verify the device** (Step 1, 2, and 3.(a)): M sends to D a random challenge $nonce$. D signs $nonce$ with its private device key and sends the signature to M. Then M verifies the signature to check whether D possesses a legitimate device key.
2. **Bind new device key** (Step 3.(b) to 3.(f)): This step actually is a re-run of the generate procedure, during which M implicitly embeds a new primary seed PS' into the device. In this step, M generates a new helper data H', a new encrypted blob of the secure OS using a new symmetric key k' derived from PS', and a certificate binding the new device public key pk' and measurements of the building block and secure OS.
3. **Deploy new device key** (Step 4): D first verifies the certificate $Cert_{pk'}$ using M's public key pk_M, and then stores the triple $(H', Blob', Cert_{pk'})$ in its non-volatile storage. H' will be used to generate the new embedded primary seed PS' during the reproduce procedure, which derives the new device key (pk', sk'). $Blob'$ is the encrypted image of the secure OS using the new symmetric key k'. $Cert_{pk'}$, containing the standard measurement values of the building block and secure OS, will help the device to establish a chain of trust.

When the device reboots after running the key update protocol, the building block will generate PS' under the help of

[1]It's common for BootROM to provide the verification ability in devices supporting secure boot, such as the Zynq-7000 SoC [54] and iOS platforms [6].

the new helper data H'. As PS' is randomly selected by M and independent from the previous device key (pk, sk), the knowledge of the previous key doesn't give the adversary any help in corrupting PS'. So the new device key derived from PS' is secure. Our key update protocol doesn't resist downgrading attacks: an adversary can roll back the device key to an old one by copying previous helper data and encrypted blob of the secure OS back to the device. However, this attack can be easily prevented under the help of a secure mono counter c, see Figure 6. Once running the key update protocol, c is increased by 1, so the building block cannot compute the previous symmetric key k as c has been changed, thus the secure OS cannot be decrypted. Thus, only the new deployed software can boot the system, which prevents downgrading attacks.

4.5 Security Analysis

We now discuss the security of our design, i.e., how the primary seed and the unique device key are protected.

Protection for the primary seed. We now show that the primary seed is secure even under a sophisticated adversary capable of controlling all external interfaces of the SoC. We list all the possible attacks by which the adversary might compromise the primary seed, and argue why these attacks cannot succeed one by one.

1. **Compromise the SRAM start-up value.** The adversary can generate the primary seed itself if it knows the SRAM start-up value of the device. However, the adversary cannot read the start-up value as the value is transferred to the building block by the BootROM through the internal buses in SoC which the adversary cannot monitor.
2. **Software attacks.** The primary seed exists only when the building block is running, and at this time the building block is the only code running on the device. The chain of trust guarantees that only legitimate building block can run in the OCM, so the adversary cannot compromise the primary seed through software attacks.
3. **Attach malicious peripherals.** The OCM is secure storage for SoC as it has no address or data lines at device pins. So malicious peripherals cannot read the contents of the OCM from the pins of the device. Moreover, the OCM is designated as secure, so the TZPC can prevent malicious peripherals from accessing the OCM.
4. **Attach malicious DMA-capable devices.** The last possible attack that the adversary can mount is to attach a malicious DMA-capable device to the SoC to read the primary seed in the OCM. However, as we designate the OCM as secure, the TrustZone-aware DMA Controller can prevent malicious devices from accessing the OCM.

Protection for the device key. The device key is stored in the secure OS which runs in the off-chip main memory such as DRAM. The TrustZone isolation prevents all attacks from the normal world, so adversaries capable of controlling the mobile OS cannot compromise the device key.

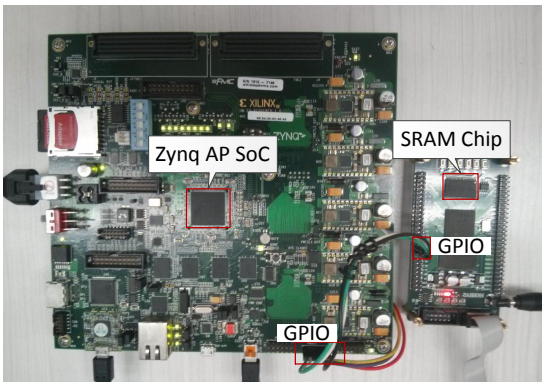

Figure 7: Physical view of our implementation

For the device key, we don't consider physical attacks that fall outside the defense capabilities of TrustZone technology. However, if the device key is compromised by physical attacks, we can deploy a new one by running the key update protocol, which mitigates physical attacks on the device key.

5. IMPLEMENTATION

We now present our implementation on a TrustZone-enabled development board, Zynq-7000 AP SoC Evaluation Kit [71]. This board is equipped with dual ARM Cortex-A9 MPCore, 1GB of DDR3 Memory, and an OCM module consisting of 256 KB of SRAM and 128 KB of ROM (BootROM).

5.1 SRAM PUF

Although Zynq-7000 AP SoC has 256 KB of on-chip SRAM, it is initialized by the BootROM once the board is powered on, preventing us from reading the initial values of the on-chip SRAM. We then use an SRAM chip that is of the type IS61LV6416-10TL [31] to serve as our SRAM PUF. This SRAM chip is equipped in a board [5] whose core is an ALTERA Cyclone II EP2C5T144 chip. Figure 7 shows the Zynq development board, the Cyclone board, and their connection. The SRAM start-up data is transferred to the Zynq development board by an FPGA implementation of Universal Asynchronous Receiver/Transmitter (UART) in Verilog hardware description language. A UART transmitter in the Cyclone board transmits SRAM data via a General Purpose I/O (GPIO) pin, and a UART receiver in the Zynq board receives the SRAM data via a GPIO pin and stores the data to a RAM cache we build in the programmable logic of the Zynq development board. Then CPU can fetch the SRAM data in the RAM cache via the AXI bus.

5.2 Building Block

We implement our building block based on the First Stage BootLoader (FSBL) of Xilinx, which runs immediately after the BootROM. The fuzzy extractor is based on an open source BCH code [48], which can build BCH codes with different parameters. The parameters of BCH codes are $[n, k, d]$, where n is the code size, k is the data size, and $d \geq 2t + 1$ (t is the number of errors that can be corrected). However, the original source code of [48] requires more than 4MB memory, so it cannot directly run in the OCM. We customize a [1020,43,439]-BCH code based on [48], and optimize

the source code to make it require less than 40KB memory. The [1020,43,439]-BCH code can decode a noisy 1020 bits message whose errors are less than $\lfloor 493/2 \rfloor = 219$, and obtain 43 "error-free" bits. As the primary seed is of length 256 bits, we require at least $\lceil 256/43 \rceil * 1020 = 6120$ SRAM cells and need to run the BCH code $\lceil 256/43 \rceil = 6$ times in our building block. For devices whose OCM is quite limited, BCH codes with other parameters can be used. For example, the [511,19,239]-BCH code only requires about 10KB memory after our optimization.

The secure entropy extractor is implemented using a customized Binary finite field library (BFFL) [12] for a finite field of 512 bits. The BFFL consists of only about 300 lines of code, and uses only a small fixed lookup table of 512 bytes. These features make it suitable for our entropy extractor running in the OCM.

We implement KG using the RSAREF library [52] (that we modify to support 2048 bits and whose MD5 hash function is replaced with SHA-2). The generation of the symmetric key for secure boot is similar to the generation of a symmetric primary key in TPM 2.0 [62], so we implement KDF following the _cpri__KDFa() function described in TPM 2.0. We also add RSA verification function [52] and AES decryption function using Byte-Oriented-AES [44] to the building block to support secure boot.

5.3 Root of Trust

In the normal world, we run a Linux OS with kernel version 3.8. In the secure world, we run the Open Virtualization SierraTEE, which provides a basic secure OS running in the secure world of TrustZone and is compliant with the Global Platform's TEE Specifications [25]. The source code of SierraTEE for Xilinx's Zynq-7000 AP SoC now is available from Github [26] under a GNU v2.0 License.

In the secure OS of SierraTEE, we implement the secure RNG described in Algorithm 2 and the simple seal/unseal primitives described in Section 4.2.1. The secure RNG is implemented by the BFFL, and ExpandKey() and Encrypt() from [44]. We locate the secure RNG in a GlobalPlatform TEE Internal API TEE_GenerateRandom() (which is an empty function in the original SierraTEE source code[2]), thus the secure OS and secure services can use our RNG by calling this function. The seal/unseal primitives are implemented by the modified RSAREF library, KDF, SHA256, and the Byte-Oriented-AES.

TPM service is implemented by creating a secure service in the secure world running a software TPM [59] whose Endorsement Key is the unique device key. Figure 8 depicts our implementation. The original software TPM [59] is a daemon application listening on a Unix socket for incoming TPM command requests, and transferring received TPM requests to the main_loop() function who dispatches these requests to corresponding TPM functions. The original software TPM requires TCP/IP software stack, but it's infeasible for the secure world to include such a big software stack. So we move the main() function to the normal world,

[2]It is reasonable that SierraTEE doesn't implement this function: secure RNGs are not commonly available.

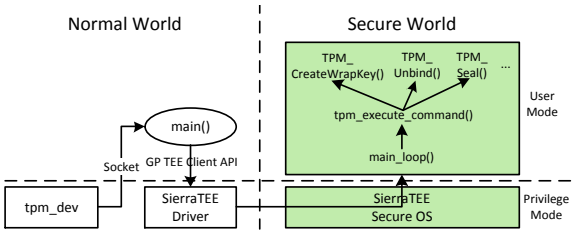

Figure 8: TPM Service Implementation

Table 1: TCB size of our implementation

		Code (LOC)	Code Size
BB	Fuzzy Extractor	0.3K	33.4K
	Entropy Extractor	0.8K	19.7K
	KG and KDF	1.7K	10.7K
	Secure Boot	0.4K	23.8K
	Total	3.2K	63.8K[3]
RoT	Secure RNG	0.8K	20.3K
	Seal/Unseal	1.9K	56.5K
	TPM Service	21.1K	336.5K
	Total	23.8K	413.3K

and leave the main_loop() function as the entry of the TPM service. When the main() function receives a TPM command request, it transfers the request to the main_loop() function though the GP TEE client API. We also port the tpmd_dev Linux kernel module of the software TPM to the normal world, which simulates a hardware TDDI, and connects the TPM daemon through a socket connection. Another technical issue that we meet during implementation is the storage of the persistent data. Storing persistent data of the TPM service happens when a TPM command is successfully processed or the TPM_SaveState() command is called. Unfortunately, the secure world has no persistent secure storage. We solve this issue by encrypting the persistent data using another symmetric key derived from the primary seed and storing the encrypted blob in the normal world.

6. EVALUATION

We present the TCB size of our implementation, and then perform tests on the SRAM PUF and our secure RNG. Finally, we present a performance evaluation of the root of trust.

6.1 TCB Size

We present the number of lines of source code and the binary code size (after compilation) of the building block (BB) and the root of trust (RoT) in Table 1, and all implementations are written in C. As shown in Table 1, the TCB size of the building block and the simple seal/unseal primitives is very small: the total size of the building block is 3.2K and the seal/unseal primitives is 1.9K. The binary code size indicates that only additional 63.8KB ROM and 413.3KB flash is required. One reason for the small TCB size of our implementation is that we leverage some very efficient open source libraries such as BFFL and Byte-Oriented-AES. The TCB size of the TPM service is much bigger as it provides rich trusted computing functionalities.

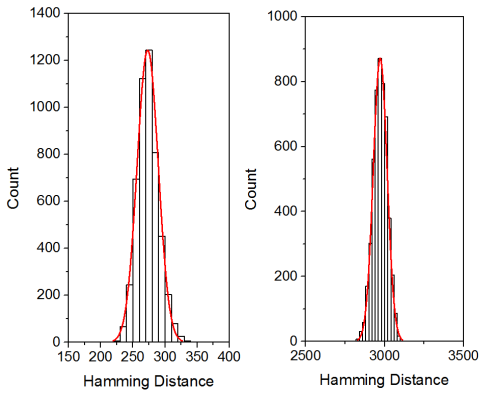

Figure 9: Robustness and Uniqueness Evaluation

6.2 Tests on SRAM PUF

We test the robustness and uniqueness of the SRAM PUF, which are two most important properties of PUFs. For the SRAM PUF, high robustness means that the start-up value from the same address range (the challenge to SRAM PUFs) should not differ significantly between the generate procedure and reproduce procedure. This property guarantees that errors between the generate and reproduce procedure can be corrected by the BCH code, thus the primary seed randomly chosen by the manufacturer during the generate procedure can be reconstructed during the reproduce procedure. The robustness of an SRAM PUF is usually assessed by the Hamming distance between repeated measurements of SRAM cells from the same address range, which is defined by $HD(\bar{x}, \bar{y}) = \sum_{i=1}^{L} x_i \oplus y_i$ where $\bar{x}$ and $\bar{y}$ are two start-up binary strings of the SRAM PUF, L is the length of $\bar{x}$ and $\bar{y}$, and x_i and y_i are the i-th bit of $\bar{x}$ and $\bar{y}$ respectively. We perform 100 measurements on the same address range of 6120 bits in the SRAM chip, compare the 100 measurements to each other, and depict the Hamming Distance histogram of the $100(100-1)/2 = 4950$ comparisons in Figure 9 (left). The analysis shows the average Hamming distance of the SRAM PUF start-up binary strings is 273 (273/6120 = 4.46%), and the maximum Hamming distance is 343 (343/6120 = 5.6%). We now assess the robustness of our fuzzy extractor, i.e., the ability to reconstruct the implicitly embedded primary seed in the reproduce procedure. Under assumption that all SRAM bits are independent, the max bit error probability p is 0.056. Considering experimental results of [27] obtained under the condition that large environment variations are taken into account (and to be conservative), we set $p = 0.15$. Notice that the [1020,43,439]-BCH code can correct up to $t = 219$ errors. Thus the probability that the fuzzy extractor cannot reconstruct the 43 "error-free" bits can be calculated by $P = \sum_{i=219+1}^{1020} \binom{1020}{i} p^i (1-p)^{1020-i} = 1 - \sum_{i=0}^{219} \binom{1020}{i} p^i (1-p)^{1020-i} < 10^{-7}$. As the building block runs the fuzzy extractor 6 times to generate a 256 bits primary seed, the robustness of the building block (i.e., the probability that the building block can reconstruct the primary seed), can be calculated by $P_{BB} = (1-P)^6 > 1-10^{-6}$.

[3]The reason that the total code size is smaller than the sum of above rows is above rows shares some libraries.

The SRAM PUF should achieve high uniqueness, which means that start-up binary strings from different SRAM PUFs should be random and independent from each other. This property guarantees that knowledge of SRAM start-up value of one device doesn't help in the prediction of SRAM start-up value of another device. We use three methods to assess the uniqueness of the SRAM PUF:

1. Hamming distance measure. As we expect that different SRAM PUFs behave independently from each other, the Hamming distance between start-up binary strings from different SRAM PUFs should be close to one half of the length of the start-up binary string.

2. Min-entropy estimation. Min-entropy provides a lower bound for the unpredictability of the SRAM start-up strings. We assume that all bits from the SRAM start-up strings are independent, so each bit can be viewed as an individual binary source. We leverage the method described in NIST 800-90 [11] to assess the min-entropy of a binary bit: $H = -log_2(p_{max})$, where $p_{max} = max\{p_0, p_1\}$ (p_0 and p_1 are probabilities of the binary bit output zero and one respectively). The min-entropy of the SRAM start-up strings is defined by: $H_{total} = \sum_{i=1}^{n} H_i$ where n is the length of the start-up string.

3. CTW compression. Context-Tree Weighting (CTW) [70] is an optimal compression algorithm for stationary sources and is commonly used to estimate entropy.

We implement 100 SRAM PUFs using 100 different address ranges of 6120 bits in the SRAM chip. A histogram of Hamming distances between start-up binary strings of the 100 SRAM PUFs ($100 * 99/2 = 4950$ comparisons) is depicted in Figure 9 (right). Our analysis shows that the Hamming distance distribution closely matches a normal distribution with mean 2972 (which is close to one half of 6120) and a standard deviation of 44 (44/6120=0.7%). The min-entropy and the CTW compression ratio of the 100 SRAM PUFs startup strings are 4835 (4835/6120 = 79%) and 99% respectively. All the results show that our SRAM PUF satisfies high uniqueness.

6.3 NIST Test on the RNG

As we use the noise present in the SRAM start-up value to accumulate entropy for the secure RNG, we first need to estimate the entropy in the start-up value. This time the input of the min-entropy estimation method is not SRAM start-up binary strings from different SRAM PUFs but binary strings from repeated measurements on the same SRAM PUF. We perform 100 measurements on the same address range of 6120 bits, and calculate the min-entropy. The results show that the min-entropy rate of the SRAM chip is about 5.5%. To be conservative, we set the min-entropy rate to be 2%. Notice that our implementation of the entropy extractor extracts a truly random seed of 512 bits, so the required length of the SRAM start-up value is $512/0.02 + 512 = 25.5$K bits.

We then perform the complete set of randomness tests from NIST 800-22 [53] on our implementation of the secure RNG. We use the RNG to generate 128.000.000 bits output, and

Table 2: Performance Evaluation (in ms). Avg. of 100 runs.

Command	Key	Time	Command	Key	Time
Simple Seal	2048	4	Sign	2048	17
Simple Unseal	2048	17		1024	6
TakeOwnership	2048	1056	Seal	2048	4
MakeIdentity	2048	947		1024	4
ActivateIdentity	2048	18	Unseal	2048	18
Quote	2048	17		1024	6
CreateWrapKey	2048	972	Unbind	2048	17
	1024	85		1024	6
LoadKey	2048	18			
	1024	6			

Table 3: TPM chip vs TPM service (in ms, key size: 2048 bits). Avg. of 100 runs.

	LoadKey	Sign	Seal	Unseal	Unbind
TPM 1.2 chip	781	609	63	625	625
TPM Service	18	17	4	18	17

divide it into 125 strings of 1.024.000 bits. The result shows that at least 124 strings (99%) pass all tests.

6.4 Performance of The Root of Trust

We first evaluate the performance delay introduced by the context switch between the secure world and the normal world. This evaluation is done by invoking an empty service running in the secure world. The result shows that the context switch only requires about 2 milliseconds (ms).

We then evaluate the performance of the simple seal/unseal primitives and some TPM commands of the TPM service, and summarize the results in Table 2 (the number in the Key column denotes the key length in bits). We also compare the TPM service with a TPM 1.2 hardware chip produced by National Semiconductor, which is embedded in an IBM ThinkCenter M52 81114 host. The comparison is shown in Table 3. The results of the performance evaluation on the TPM service include the context switch delay. The results indicate that the TPM service is very efficient compared to TPM hardware chips.

7. RELATED WORK

Researchers at Johns Hopkins University Applied Physics Laboratory working with the Trusted Computing Group develop specifications for trusted computing technologies in mobile devices [47]. They define the root of trust and chain of trust as basic requirements for a mobile TPM, which are supported by our design.

ARM defines an architecture [10] for TrustZone-enabled platforms based on GlobalPlatform TEE API standards. The architecture leverages hardware resources such as hardware keys, crypto accelerators, and Secure Element to provide the root of trust. Nokia's On-board Credentials system uses an assumed device key to provide root of trust for the platform. These approaches require special hardware components to provide secure storage and randomness for TEE. However, these hardware components are not always available in devices. Furthermore, the conventional secure storage provided by hardware keys is inflexible when key update is required.

Areno et al. present a methodology that uses PUFs to protect the TEE [7]. Their idea is similar to this work. However, they only concern about the secure-boot process on a TrustZone-enabled platform, and other security functionalities such as attestation and RNGs are not considered. Furthermore, they only discuss how their design can be implemented by FPGA technology and don't give a concrete implementation.

Another related work is TEEM [21], a portable Trusted Computing module that can provide trusted computing functionalities for various computing platforms such as desktop machines and mobile devices. TEEM is designed as a secure TPM service running in the secure world of TrustZone, and a prototype is implemented on a general ARM SoC development board. However, their implementation doesn't isolate TEEM from the Rich OS. Actually, the TEEM runs on an entire Linux OS, which makes the TCB very large. Furthermore, their work ignores the root of trust for TEEM, i.e., they don't consider how to establish a chain of trust for TEEM from powered on.

8. CONCLUSION

In this paper, we presented a research prototype that provides the root of trust for TrustZone-enabled platforms using SRAM PUFs. Our prototype first leveraged the SRAM PUF commonly available on mobile devices to provide foundations for the root of trust (secure storage, randomness, and secure boot) in a very small TCB size of about 3.2K LOC, then built the root of trust running in the secure world of TrustZone, which enabled to establish a complete chain of trust for mobile devices. Another advantage of our prototype is that it enables to deploy key update mechanisms easily. As a result, we demonstrated that the SRAM PUF could provide the root of trust for TrustZone-enabled platforms in a secure, efficient and flexible way.

9. ACKNOWLEDGMENTS

We thank Yevgeniy Dodis, Sylvain Ruhault for their suggestions on building our secure RNG. We especially thank Antonio Bellezza for providing an efficient customized finite field library. We appreciate anonymous TrustED reviewers for their helpful comments.

10. REFERENCES

[1] App Sandbox Design Guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html.

[2] SE for Android. http://selinuxproject.org/page/SEforAndroid.

[3] Advanced Micro Devices. Secure Virtual Machine Architecture Reference Manual. *AMD Publication*, (33047), 2005.

[4] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4), 2004.

[5] Anne's fashion shoes. ALTERA EP2C8F256 Core Board. http://www.aliexpress.com/item/Altera-ep2c8f256-core-board-belt-sdram-sram-fpga-development-board-power-supply-pin/1427214650.html.

[6] Apple. iOS Security. http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf.

[7] M. Areno and J. Plusquellic. Securing trusted execution environments with puf generated secret keys. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1188–1193. IEEE, 2012.

[8] ARM. ARM Security Technology Building a Secure System using TrustZone® Technology.

[9] ARM. Designing with TrustZone® - Hardware Requirements.

[10] ARM. Securing the System with TrustZone® Ready Program. http://www.arm.com/products/security-on-arm/trustzone-ready/index.php.

[11] E. B. Barker and J. M. Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. NIST, 2007.

[12] A. Bellezza. Binary finite field library 0.02. http://www.beautylabs.net/software/finitefields.html.

[13] C. Bösch. Efficient fuzzy extractors for reconfigurable hardware. *Master's Thesis, Ruhr-University Bochum*, 2008.

[14] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls. Efficient helper data key extractor on fpgas. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 181–197. Springer, 2008.

[15] M. Claudio, K. Nikolaos, S. Claudio, K. Kari, and Č. Srdjan. Smartphones as practical and secure location verification tokens for payments. In *NDSS*, 2014.

[16] A. Desnos and P. Lantz. Droidbox: An android application sandbox for dynamic analysis. https://code.google.com/p/droidbox/, 2011.

[17] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs. Security analysis of pseudo-random number generators with input:/dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 647–658. ACM, 2013.

[18] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in cryptology-Eurocrypt 2004*, pages 523–540. Springer, 2004.

[19] I. Eichhorn, P. Koeberl, and V. van der Leest. Logically reconfigurable pufs: Memory-based secure key storage. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 59–64. ACM, 2011.

[20] J.-E. Ekberg, N. Asokan, K. Kostiainen, P. Eronen, A. Rantala, and A. Sharma. Onboard credentials platform design and implementation. *Nokia Research Center Helsinki, Finland*, 2008.

[21] W. Feng, D. Feng, G. Wei, Y. Qin, Q. Zhang, and D. Chang. Teem: A user-oriented trusted mobile device for multi-platform security applications. In *Trust and Trustworthy Computing*, pages 133–141. Springer, 2013.

[22] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Controlled physical random functions. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 149–160. IEEE, 2002.

[23] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[24] Global Platform Device Technology. TEE client API specification version 1.0. http://globalplatform.org, 2010.

[25] GlobalPlatform. GlobalPlatform Device Specifications. http://www.globalplatform.org/specificationsdevice.asp.

[26] J. González. Open Virtualization for Xilinx's ZC-702. https://github.com/javigon/OpenVirtualization.

[27] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls. Fpga intrinsic pufs and their use for ip protection. In *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 63–80. Springer, 2007.

[28] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert. Cloning physically unclonable functions. In *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, pages 1–6. IEEE, 2013.

[29] D. E. Holcomb, W. P. Burleson, and K. Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. *Computers, IEEE Transactions on*, 58(9):1198–1210, 2009.

[30] D. E. Holcomb, W. P. Burleson, and K. Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. *Computers, IEEE Transactions on*, 58(9):1198–1210, 2009.

[31] Integrated Silicon Solution, Inc. IS61LV6416-10TL. http://www.alldatasheet.com/datasheet-pdf/pdf/505020/ISSI/IS61LV6416-10TL.html.

[32] Intel Corporation. LaGrande technology preliminary architecture specification. *Intel Publication*, (D52212), May 2006.

[33] P. C. Kocher. Leak-resistant cryptographic indexed key update, Mar. 25 2003. US Patent 6,539,092.

[34] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 104–115. ACM, 2009.

[35] S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls. The butterfly puf protecting ip on every fpga. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 67–70. IEEE, 2008.

[36] K. Kursawe, A. Sadeghi, D. Schellekens, B. Skoric, and P. Tuyls. Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. In *Hardware-Oriented Security and Trust, 2009. HOST'09. IEEE International Workshop on*, pages 22–29. IEEE, 2009.

[37] J.-P. Linnartz and P. Tuyls. New shielding functions to enhance privacy and prevent misuse of biometric templates. In *Audio-and Video-Based Biometric Person Authentication*, pages 393–402. Springer, 2003.

[38] D. Liu and Q. Dong. Combating side-channel attacks using key management. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[39] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 365–378. ACM, 2012.

[40] R. Maes, P. Tuyls, and I. Verbauwhede. Intrinsic pufs from flip-flops on reconfigurable devices. In *3rd Benelux workshop on information and system security (WISSec 2008)*, volume 17, 2008.

[41] R. Maes, P. Tuyls, and I. Verbauwhede. Low-overhead implementation of a soft decision helper data algorithm for sram pufs. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 332–347. Springer, 2009.

[42] R. Maes, P. Tuyls, and I. Verbauwhede. A soft decision helper data algorithm for sram pufs. In *Information Theory, 2009. ISIT 2009. IEEE International Symposium on*, pages 2101–2105. IEEE, 2009.

[43] R. Maes, A. Van Herrewege, and I. Verbauwhede. Pufky: A fully functional puf-based cryptographic key generator. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 302–319. Springer, 2012.

[44] K. Malbrain. Byte-Oriented-AES. https://code.google.com/p/byte-oriented-aes/.

[45] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[46] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[47] K. N. McGill. Trusted mobile devices: Requirements for a mobile trusted platform module. *JOHNS HOPKINS APL TECHNICAL DIGEST*, 32(2):544, 2013.

[48] R. Morelos-Zaragoza. Encoder/decoder for binary BCH codes in C (Version 3.1). http://www.rajivchakravorty.com/source-code/uncertainty/multimedia-sim/html/bch_8c-source.html.

[49] D. Nedospasov, J.-P. Seifert, C. Helfmeier, and C. Boit. Invasive puf analysis. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 30–38. IEEE, 2013.

[50] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld. Physical one-way functions. *Science*, 297(5589):2026–2030, 2002.

[51] P. S. Ravikanth. *Physical one-way functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[52] RSA Data Security Inc. RSAREF. http://www.homeport.org/~adam/crypto/rsaref.phtml.

[53] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001.

[54] L. Sanders. Secure Boot of Zynq-7000 All Programmable SoC. 2013.

[55] D. Sangorrin, S. Honda, and H. Takada. Dual operating system architecture for real-time embedded systems. In *Proceedings of the 6th International Workshop on Operating Systems*

Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium, pages 6–15, 2010.

[56] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications.

[57] Sierraware. Open Virtualization - ARM TrustZone and ARM Hypervisor Open Source Software. `http://www.sierraware.com`.

[58] P. Simons, E. van der Sluis, and V. van der Leest. Buskeeper pufs, a promising alternative to d flip-flop pufs. In *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*, pages 7–12. IEEE, 2012.

[59] M. Strasser and H. Stamer. A software-based trusted platform module emulator. In *Trusted Computing-Challenges and Applications*, pages 33–47. Springer, 2008.

[60] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[61] G. Taylor and G. Cox. Behind intel's new random-number generator. *IEEE Spectrum*, 2011.

[62] TCG. Trusted Platform Module Library Part 1: Architecture, Family 2.0, Level 00 Revision 01.07, 2014.

[63] Trusted Computing Group. `http://www.trustedcomputinggroup.org`.

[64] Trusted Computing Group. TPM Software Stack (TSS) Specification, Version 1.2. `https://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification`.

[65] TU Graz, IAIK. jTSS–Java TCG Software Stack. `http://trustedjava.sourceforge.net`, 2009.

[66] P. Tuyls, G.-J. Schrijen, B. Škorić, J. Van Geloven, N. Verhaegh, and R. Wolters. Read-proof hardware from protective coatings. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 369–383. Springer, 2006.

[67] V. Van der Leest, B. Preneel, and E. Van der Sluis. Soft decision error correction for compact memory-based pufs using a single enrollment. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 268–282. Springer, 2012.

[68] V. van der Leest, E. van der Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh. Efficient implementation of true random number generator based on sram pufs. In *Cryptography and Security: From Theory to Applications*, pages 300–318. Springer, 2012.

[69] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan. Flash memory for ubiquitous hardware security functions: true random number generation and device fingerprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 33–47. IEEE, 2012.

[70] F. M. Willems, Y. M. Shtarkov, and T. J. Tjalkens. The context-tree weighting method: Basic properties. *Information Theory, IEEE Transactions on*, 41(3):653–664, 1995.

[71] Xilinx. Zynq-7000 All Programmable SoC ZC702 Evaluation Kit. `http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC702-G.htm`.

[72] K. Yoder et al. TrouSerS–Open-source TCG Software Stack. *http://trousers.sourceforge.net*, 2007.