# An Efficient Trustzone-Based In-application Isolation Schema for Mobile Authenticators

Yingjun Zhang[1,2](✉), Yu Qin[1], Dengguo Feng[1], Bo Yang[1], and Weijin Wang[1]

[1] Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing, China
zhangyingjun@tca.iscas.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China

**Abstract.** Mobile devices have been widely used as convenient authenticators for sensitive transactions and user login. It's a challenge to protect authentication secrets and code from malicious mobile operating systems. Although protecting them using hardware privilege isolation like Trustzone and virtualization is a promising countermeasure, existing approaches either have large TCBs with lots of applications and services installed in the privileged software, or provide only coarse-grained isolation unable to prevent intra-domain attacks, or require excessive intervention from the privileged software. We propose a novel mobile authentication schema called TAuth, which creates isolation execution environments in Trustzone normal world, so the system TCB in the secure world remains small and unchanged regardless of the amount of installed authentication applications. The isolation is also fine-grained which only protects the security-sensitive components of an authentication program, thus could defense not only a malicious OS, but also vulnerability threats inside the same program. Designed closely integrated with the intrinsic property of user authentication, TAuth solves two significant technique challenges, including efficient normal world isolation without excessive intervention into the secure world, and securely using of untrusted external functions from inside the isolated environment. Finally, we implement the prototype system on real TrustZone devices. The evaluation shows that TAuth can prevent both in-application attacks like HeartBleed and kernel-level rootkits. It also shows that TAuth achieves much higher system performance than previous Trustzone normal world isolation solutions.

**Keywords:** Mobile authentication · Trustzone · Small TCB
In-application isolation

## 1 Introduction

Mobile devices are increasingly used as authenticators for sensitive transactions and user login. Software authentication tokens free the users from the burdens of carrying multiple hardware tokens at all times. Also, communication ability

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2018
X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 585–605, 2018.
https://doi.org/10.1007/978-3-319-78813-5_30

with on-board peripherals and sensors allows conveniently enrolling new authentication factors, such as geographical locations and fingerprints. Mobile authenticators achieve both flexibility and low cost, hence are commonly seen as an ideal substitution of dedicated hardware tokens.

However, as modern commodity mobile operating systems are increasingly complex with endless kernel vulnerabilities [1], root attackers could easily intercept peripheral channels or compromise the execution of software tokens to steal authentication secrets, like passwords and private keys. Various attacks to mobile authentication applications (MAPs) have been reported [6,29], indicating the serious security challenges.

Researchers have proposed using Trusted Execution Environment (TEE) to protect sensitive applications against OS compromise. Trustzone [7], the most widely used mobile TEE technology, creates two separated execution partitions on ARM devices, the normal world and the secure world. The trusted applications (TAs) in the secure world enjoy hardware-enforced security capabilities against malware in the normal world OS. Trustzone-based authentication solutions have been proposed [17,23,27,37] and are integrated into mainstream authentication specifications like FIDO [8,28].

However, traditional Trustzone solutions face a major challenge, i.e., the security guarantees will be weakened as the attack surface and TCB size will increase along with the number of TAs and system services installed in the secure world. For example, various kernel-level device drivers are integrated into the secure world to support trusted device I/O, which have enormous code size and much higher bug rate than other kernel components. Since the secure world has a higher privilege, a compromised secure world will compromise the whole mobile device. Recent incidents show that exploiting the secure world's vulnerabilities has become a real threat [22,26,30,31,33]. For security concerns, mobile device vendors usually limit Trustzone resources to their own TAs. This makes it hard for third-party service providers to deploy their specific Trustzone-based MAPs, which poses a substantial barrier to their adoptions.

Unlike traditional Trustzone solutions, another kind of virtualization-based privilege isolation method places the TAs and the untrusted OS in the same privilege domain (a guest VM). Thus the system TCB won't increase along with the number of supported TAs. Such shielding systems [12,15,18] have the potential to resolve the defects faced by previous Trustzone-based authentication solutions. However, hardware virtualization is not commonly supported on mobile platforms and the complex commodity hypervisors are already struggling with their own security problems [4,5]. Also, they only provide coarse-grained isolation at an application level, which won't work well under attacks exploiting vulnerabilities inside a victim TA. For example, the Heart Bleed attack, which exploits a memory disclosure vulnerability in OpenSSL, can cause the victim program to leak critical secrets itself, with no need to directly read its memory.

In this paper, we propose a novel Trustzone-based mobile authentication schema, TAuth, which achieves two key advantages compared with previous solutions. First, it creates isolated execution environments in the normal world for the MAPs. Without concrete applications and system services installed in the secure world, the system TCB remains small and unchanged. Second, the

isolation is fine-grained only contains sensitive program components, thus could defense threatens from both the underlying Rich OS[1] and the remaining program components. TAuth aims at addressing the urgent security issues for increasingly popular mobile authentication applications. To achieve these goals, we must solve several challenges.

First, normal world isolation is non-trivial to achieve, given the Rich OS's role in memory management for its applications. Shielding systems leverage hardware MMU virtualization (i.e., the *nested page* mechanism) to achieve exclusive memory access control in their hypervisors. However, the Trustzone normal world, which hosts the Rich OS, has full control over its own resources, including its MMU. Such control would allow the Rich OS to access any normal world memory by manipulating its page tables, including the authentication secrets. Therefore, previous normal world isolation solutions [9,21] require the secure world to intercept frequent page table updates of the Rich OS, which significantly affects the system performance.

Second, in-program partition may not be easy, as commodity software usually has complex semantics, internal interactions, and lots of cross-component function calls. Existing approaches targeting at Pieces of Application Logic (PAL) [19,20] require the PAL being self-contained, thus not supporting calling external functions, which are not suitable for real authentication MAPs. For example, they need to call OS services to communicate with I/O peripherals (storage devices, touch screen, sensors..) to obtain initial authentication secrets. However, under the assumption that the Rich OS and other program components are untrusted, how to guarantee the security of these external calls remains a challenge.

TAuth solves all these challenges, based on the intrinsic property of MAPs' authentication procedure. First, through manual source code analysis and automated taint analysis of several popular MAPs (e.g., Google Authenticator), we found that the critical code which controls the authentication secrets only constitutes a tiny fraction of the whole program, and usually follows fixed patterns. Based on these observations, we propose an efficient isolation mechanism by pre-loading these tiny components into a continuous memory region. When the critical part is running, TAuth applies atomicity protection to it, ensuring it's execution won't be unexpectedly interrupted by the untrusted Rich OS. When it is suspended, TAuth temporarily includes its memory into the secure world by dynamically setting the Trustzone controller, thus ensuring its isolation without frequently intercepting the Rich OS's page table updates. Second, according to execution patterns of the critical authentication code, we divide them into three categories: the storage code, the I/O code, the computation code. Then we design a trusted context switch module in the secure world to ensure the securely calling of necessary external functions from these code. Finally, we apply TAuth to Google Authenticator (GA), tiqr and OpenSSL, and use HeartBleed attack, memory disclosure rootkit to demonstrate its effectiveness and security. In summary, we make the following contributions.

[1] Rich OS represents the commodity operating systems like Linux, Android in Trustzone normal world.

- A novel Trustzone isolation architecture in the normal world, with both enhanced security guarantees and improved efficiency.
- A fine-grained isolation specially designed for mobile authentication applications, which could defense both in-application and OS-level attacks.
- Thorough evaluations on real authentication software and attack samples, which confirm the security and efficiency of TAuth.

## 2 Background

### 2.1 Trustzone

TrustZone is a CPU security extension defined by ARM. It creates two isolated execution domains on ARM platforms: the normal world and the secure world. A new CPU mode called monitor mode is introduced as the only entry point to the secure world. The normal world code needs to call the Secure Monitor Call (*smc*) instruction to enter the secure world. Each world has separated registers and memory and the secure world has a higher privilege with permissions to access all the resources of the normal world, but not vice versa. So it has the potential to control the normal world's behaviors and enjoys the hardware-based protections from attacks that compromise the normal world.

**Memory Isolation.** Trustzone Address Space Controller (TZASC) partitions continuous physical memory regions into secure or non-secure. Note that the protection strategy defined by TZASC is more privileged than that defined by MMU, i.e., the normal world can't access any secure physical memory even if it maps the region accessible in its page tables. This is essential to realize the normal world isolation without intercepting the frequent page table updates.

**I/O Isolation.** TrustZone Aware Interrupt Controller (TZIC) partitions device interrupts into secure or non-secure. By configuring TZIC and some related registers, hardware interrupts can be directly handled in the monitor mode, thus enabling flexible routing of interrupts to either world, which is essential to realize dynamic device I/O isolation. By default, TZIC uses Fast Interrupt (FIQ) as secure interrupt and uses Regular Interrupt (IRQ) as non-secure interrupt.

### 2.2 Mobile Authentication Applications

We explain the aforementioned three types of authentication code using a real-life example, Google Authenticator. The app generates One-Time Password (OTP) tokens using the HMAC-Based (HOTP) and the Time-based (TOTP) OTP generation algorithms. It uses either QR code scanning or manual input to obtain an encoded private key issued by Google and stores it in its database. During the authentication, the key is loaded into memory to calculate a message authentication code (MAC) of a timestamp or a counter to generate OTPs. Then the OTP is displayed to the user to finish the authentication.

As described in this case, the storage code is used to load or store authentication secrets in persistent storage, such as the private key. The I/O code is used

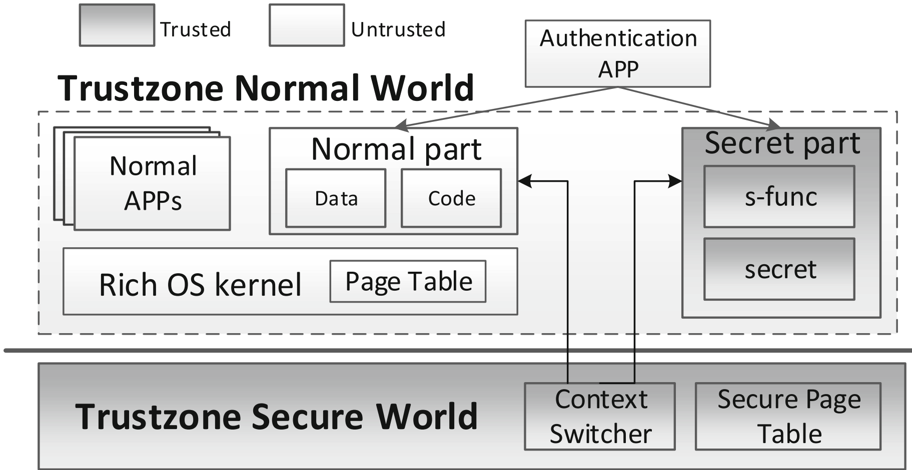


**Fig. 1.** TAuth architecture.

to import raw I/O secrets, such as the QR code and user inputs via keyboard or touch screen. It is also used to display sensitive information to the users. The computation code is used to make computations on the secrets to generate the authentication response, such as the OTP algorithms.

## 3 Threat Model and Security Assumptions

TAuth is designed against both malicious operating systems and in-application vulnerability threatens. TAuth completely removes trust of the Rich OS and assumes it can behave in arbitrarily malicious ways to disclose the authentication secrets, including directly accessing the user-level virtual address space, manipulating the page tables, or launching Iago attacks [11] which cause an application to harm itself by manipulating return values of system calls. It can also hijack or manipulate I/O communications of peripherals. We also assume the adversary can exploit in-application vulnerabilities to launch memory over-read attacks like HeartBleed [2], or control flow hijacking attacks like ROP [32], to disclose the authentication secrets in the address space of the same application. We don't consider complex physical attacks like side-channel attacks, which can't be protected by TrustZone. TAuth doesn't guarantee OS availability. A compromised OS can simply shut down or refuse to schedule apps. However, these disruptive behaviors can be easily detected. We assume TAuth is initialized via trusted booting, so that it can verify its own initial state and bootstrap trustworthy execution. Finally, we assume the protected critical code is trusted and wont deliberately send the secrets out. This is usually true for commodity MAPs like GA as the software itself is designed to keep such secrets.

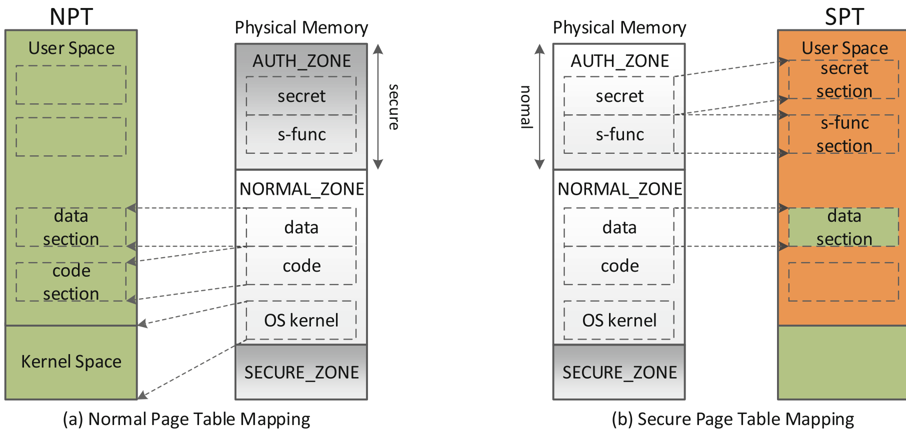


**Fig. 2.** TAuth memory layout.

# 4 System Design

## 4.1 System Overview

Figure 1 shows an overview of the TAuth architecture. In the normal world, the authentication APP is divided into a normal part and several secret parts. Each secret part comprises several authentication secrets and the corresponding critical functions manipulating them. The normal part must call into the secret parts via a trusted context switcher in the secure world. The switcher also allows the secret parts to call necessary external functions. However, TAuth ensures that the secrets can't be accessed by any external entities, including the normal part, other applications and the underlying Rich OS.

## 4.2 Basic Memory Isolation

This section details how TAuth achieves the efficient normal world isolation.

**Physical Memory Layout.** By configuring TZASC, TAuth divides the whole physical memory into three separated zones, i.e., NORMAL_ZONE, AUTH_ZONE, SECURE_ZONE. NORMAL_ZONE represents the normal world physical memory holding the Rich OS, the normal APPs and the normal part of MAPs. AUTH_ZONE is used for the secret parts of MAPs. SECURE_ZONE is used for the core components in the secure world. The security states of NORMAL_ZONE and SECURE_ZONE are always unchanged while AUTH_ZONE will be dynamically configured into either world to achieve the efficient isolation.

**Virtual Memory Layout.** TAuth maintains separated page tables for each MAP. The normal page table (NPT) is used for the normal part and the Rich OS while a secure page table (SPT) is used for every secret part. The overall

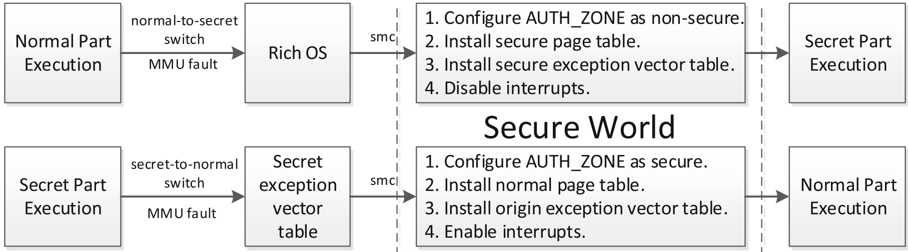


**Fig. 3.** Context switch actions for efficient isolation.

memory hierarchy is shown in Fig. 2. For data mapping, SPT maps all normal data as well as the secrets, since a secret part may also access normal data besides the secrets. All data pages in SPT are set to non-executable so that they cannot be used to inject malicious code. For code mapping, SPT only maps sensitive functions which can access the authentication secrets. These code pages are verified in the setup phase. In NPT, there isn't any valid mapping of the s-funcs and the secrets. Separated page tables allow TAuth to intercept all cross-part control flows, in a way transparent to the MAPs without modifying their source code. Whenever a cross-part code jump happens, an MMU fault occurs and traps the execution into the kernel mode, where an *smc* instruction is invoked to enter the secure world. Then TAuth performs necessary actions for ensuring the isolation, which is shown in Fig. 3.

**Efficient Isolation.** When a normal-to-secret switch happens, AUTH_ZONE is configured as non-secure, so that the secret part will run in the normal world. However, TAuth applies atomicity protection to it, ensuring it won't be interrupted unexpectedly. So other untrusted entities are sure to be suspended during its execution, with no chance to access the secret memory. When a secret-to-normal switch happens, TAuth modifies TZASC to include AUTH_ZONE into secure world, so that the untrusted running entity can't access the secret part, even if it is mapped accessible in NPT. So there is no need to intercept the Rich OS's page table updates into the secure world.

**Atomicity Protection.** In general, the secret part may be unexpectedly suspended in several cases, including hardware interrupts and CPU exceptions. To prevent the secret part from directly switching into the Rich OS, TAuth maintains a secure exception vector table, whose instructions are replaced by *smc*. When a normal-to-secret part-switch happens, TAuth activates the secure vector table to intercept all unexpected events into the secure world. For hardware interrupts, TAuth simply disables unnecessary ones by configuring TZIC, so that the secret part won't be interrupted by them. For CPU exceptions (caused by undefined CPU instructions, MMU faults, etc.), TAuth checks whether it is an MMU fault caused by normal secret-to-normal switch. If it is, TAuth performs a trusted context switch as usual. In other cases, TAuth considers an unexpected fault happens and simply shuts down the secret part, clears the memory contents of AUTH_ZONE.

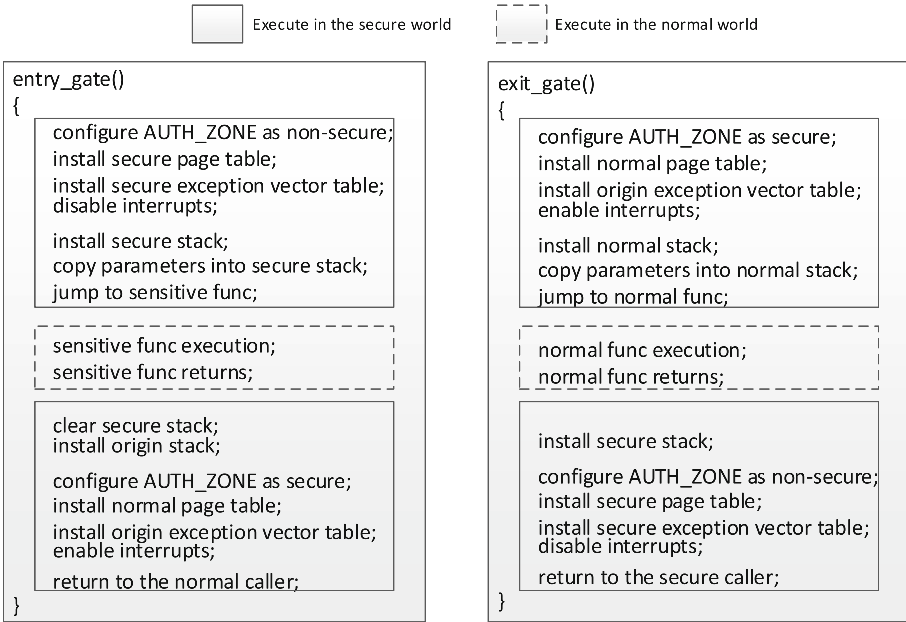


**Fig. 4.** Secure external function call.

**Discussions.** Note that both the secure exception vector table and SPT reside in AUTH_ZONE. So they can't be modified by the Rich OS. They can neither be deactivated during a secret part's execution as untrusted entities are all suspended. Malicious OS may try to access the secrets by mapping it into NPT. However, the dynamic isolation mechanism ensures AUTH_ZONE always resides in the secure world when NPT is activated, thus is always inaccessible to the Rich OS. The Rich OS may also refuse to invoke *smc* to deliver a context switch request to the secure world. This only causes unavailability of the secret part, whereas the secrets still only reside in AUTH_ZONE and won't be leaked.

Our efficient isolation requires a continuous physical memory region reserved as AUTH_ZONE, because TZASC only supports security separation for continuous regions. This will clash with the traditional memory allocation mechanism of commodity operating systems like Linux, i.e., the demand paging mechanism, where physical memory pages are dynamically allocated to the processes in greatest need. A large reserved region will significantly affect the utilization efficiency of system memory resources, because most of the region may not be used immediately and can't be used by other processes either. So our solution is not suitable for large commodity software. Fortunately, TAuth leverages the concept of in-application separation and is specially designed for authentication APPs, whose secret part is usually small, thus won't incur great performance overhead to the overall system.

### 4.3 Securing External Function Call

TAuth divides MAP's program logic into sensitive functions and other code (including application code and OS code). During runtime, functions in the normal part may call sensitive functions, while sensitive functions may also call functions outside of the secret part. As mentioned above, TAuth intercepts every cross-part function call to perform a trusted context switch in the secure world.

Figure 4 shows the whole context switch procedure. When the normal part calls sensitive functions in the secret part, the entry gate code is triggered. TAuth first performs the actions mentioned in Sect. 4.2 to ensure the basic isolation. Then it modifies the statck pointer (the *sp* register) to point to a secure stack residing in AUTH_ZONE, which is used for the execution of the secret part. If the parameter number is larger than four, which is the maximum number of parameters passed via registers, according to AAPCS (Procedure Call Standard for the ARM Architecture), the remaining parameters should be copied to the secure stack. Then the real sensitive function is called. When the sensitive function returns to the caller in the normal part, an MMU fault occurs as the return address of the normal part is inaccessible in SPT, then TAuth takes over control again. It clears the contents of the secure stack, modifies *sp* to point to the origin stack, writes the function's return value in it, and finally returns to the caller.

When a sensitive function is executing, it may call functions outside the secret part, including the ones in the normal part, library calls and system calls of the Rich OS. For function calls of the normal part, which won't access the secrets (otherwise they will be added to the secret part), TAuth performs an exit gate code, which simply reverses the procedure of the entry gate. However, calling library functions or system calls faces more challenges, as they may access the secrets. These untrusted functions usually have complex semantics and implementations. System calls even involve the execution of the Rich OS. So it's hard to guarantee their security. Fortunately, TAuth is specially designed for MAPs who have fixed execution patterns, thus having fixed security requirements. We only provide security guarantees for related function calls.

**Computation Code.** For computation code, library functions for memory operations are needed to perform the authentication algorithms, such as memcpy, strlen. As their implementations are simple and don't rely on the underlying Rich OS, TAuth simply creates a trusted version of these functions and installs them in SPT during a secret part initialization. All these calls will be redirected to the trusted version by TAuth.

**Storage Code.** For storage code, system functions for file I/O (e.g., *read*, *write*) are needed to load or update persistent authentication secrets, such as private keys, passwords in file or database. TAuth provides privacy protections for these secrets. In particular, it ensures they are encrypted using a secure per-device key and their plaintexts only exist in secure memory of AUTH_ZONE. When the secret part needs to store a secret, TAuth checks whether the external call (e.g., a *write* call) belongs to storage code. If it does, TAuth encrypts the secure buffer containing the authentication secret which is specified in the function

parameters, and copies it to the OS's memory page cache, so that external entities can only get the ciphertexts. When loading a secret, the Rich OS first reads the encrypted one into its memory cache, and invokes *smc* to inform TAuth to copy the ciphertext into the secure buffer and decrypt it. Note that a malicious OS may read a wrong secret or directly tamper the secret file, which will cause all authentications unpassed as the secret's integrity has been broken. However, this will be easily detected by the users and won't cause the correct secret being leaked.

**I/O Code.** For I/O code, system functions for device I/O (e.g., *scanf*, *printf*) are needed to import raw secrets (password from keyboard, fingerprints, GPS locations..), or display sensitive information to the users (OTPs, transaction details). Unlike persistent secrets in files, they can only be obtained from I/O devices or displayed to the users in the form of plaintext. As these secrets are transmitted between the MAPs and I/O devices via untrusted device drivers in the Rich OS, TAuth must intercept all I/O flows passing through the data boundary of the Rich OS with the MAPs and I/O devices.

When receiving an external function call for raw data input (e.g., obtain a password from keyboard via *scanf*), ATuth sets the corresponding keyboard interrupt as secure by configuring TZIC. So when a keystroke occurs, the execution of current CPU will trap into the monitor mode in the secure world, which allows TAuth to get the real keystroke before the Rich OS. Then TAuth sends a read instruction to the keyboard to get the key value, stores it in the secure world, writes a dummy value into the data buffer of the Rich OS's keyboard driver, and jumps to the normal interrupt handler of the driver. After the driver obtains all the dummy inputs, it invokes *smc* to inform TAuth to copy the real values into the secure buffer, and configure keyboard interrupt as normal again.

When receiving an external function call for raw data output (e.g., display an OTP), TAuth changes the buffer address in the function parameter to point to a shared buffer with dummy outputs. When the driver is ready for the display, it invokes *smc* to inform TAuth. Then TAuth sends an write instruction to the display device with the real outputs, and resumes the execution of the device driver to finish this function call.

Note that a malicious OS may serve illegally to display wrong outputs, or simply refuse to deliver the correct *smc* instructions. Similar with handling the storage code, this will cause all authentications unpassed as a wrong password or OTP is being used. However, this will be easily detected by the users and won't cause the correct secret being leaked, as Rich OS can only get dummy values.

**Discussions.** Our method allows securely calling external functions while still providing privacy protection to the authentication secrets. Particularly, TAuth creates a trusted data path through the complex device drivers to securely loading or exporting the secrets, with no need to reimplement them in the secure world, hence significantly reduce the system TCB size. Although the untrusted Rich OS may serve illegally to break the secret integrity, or simply launch denial-of-service attacks to block all *smc* instructions, these disruptive behaviors can be easily detected and won't cause any secret leakage.

### 4.4 Lifecycle of a Protected MAP

**Program Launch.** Before deployed into the TAuth system, an MAP must be divided into a normal part and secret parts, in the form of a configuration file, including the secret parts' start virtual addresses, code size, and per-part code hashvalues. The integrity of the file is protected using the device private key. So neither the file nor the sensitive code can be tampered or forged by attackers. The configuration files are loaded and verified in the secure world during system initialization. When an MAP is launched, the Rich OS first loads all secret parts' code into its memory caches, then informs TAuth to check their integrity using the hashvalues. If the check is passed, TAuth moves the code into AUTH_ZONE and installs the corresponding SPT according to the virtual addresses in the configuration file. Note that it also maps several reserved pages in SPT, which will be used as secure heap and stack later. Therefore, all sensitive code can be correctly loaded into AUTH_ZONE via the Rich OS's untrusted file system code and storage device driver, without reimplementing them in secure world.

**Secret Initialization.** During a secret part's initialization, it will allocate a secure buffer from stack or heap for loading every authentication secret. For stack allocation, there need no change as the stack pointer (*sp* register) has pointed to the secure stack. For heap allocation, which needs assistance from the Rich OS via *malloc*, TAuth creates a trusted *secure_malloc* installed in SPT and redirects all *malloc* calls to it to allocate pages from the secure heap. The secrets can be securely loaded via the method described in Sect. 4.3.

**Runtime.** At runtime, code in the normal and the secret part execute concurrently. TAuth ensures that: (1) all authentication secrets and their copies only exist in SPT mappings, (2) they can only be used during secret parts execution. Any attempts to access the secrets memory from the normal part will cause an MMU page fault and will be considered as malicious by TAuth, who takes further measures like shutting down the secret part, or notifying the user. Note that TAuth provides no protections for authentication responses exported from the secret part, whose security relies on the MAP's protocol design, such as using a secure session key shared with the remote authentication server.

**Exit.** When the MAP exits, authentication secrets should also be cleared. If the MAP exits normally, TAuth removes the SPT and releases the secure memory. Even if it exits abnormally or the Rich OS refuses to inform TAuth, the secrets still only exist in AUTH_ZONE and thus won't be leaked.

**Discussions.** Our method relies on the correct partition of the MAP's normal part and secret parts. This assumption is reasonable as mature works exist for automated program partition for privilege separation [10,25,35,36]. MAP providers could leverage these methods to automatically export the configuration file containing a correct and complete closure of all sensitive functions which may access the defined secrets. Moreover, MAPs usually have unified execution patterns and fixed security requirements, making their partition even easier. One of our future work will be integrating automated program partition into TAuth

architecture to generate the configuration file at runtime, thus eliminating the extra partition work for MAP providers.

Note that we can also support partition of dynamic libraries, by modifying Rich OS's loader. Then the virtual addresses in the configuration file will be in-application offsets. We do not assume the loader as trusted. Even if it behaves maliciously by refusing to load sensitive functions or loading them to wrong locations, the secrets will still not be disclosed, as TAuth can reject to load the secrets during the integrity checking phase.

## 5 Security Analysis

TAuth is mainly designed to provide memory, storage, I/O isolation of the authentication secrets. This section discusses several other typical attacks beyond the basic isolation.

**Cloning Attacks.** Cloning attackers aims to impersonate the victims to perform illegal authentications by copying the persistent secrets to their devices. As TAuth encrypts all secrets using a per-device key, they can only be correctly decrypted on the owner's device. As a common solution, most commodity mobile devices equip with a per-device key in hardware secure storage like eFuse, which can only be accessed in the secure world, making cloning attacks hard to success.

**Relay Attacks.** A compromised normal part is an ideal man-in-middle attacker, who monitors and relays the messages between the secret part and the authentication server to perform unexpected authentications. Such attacks could be prevented by requiring an explicit physical user consent (e.g., a user's button press) before any authentication actions. TAuth's I/O isolation mechanism ensures the consent can't be tampered, emulated or masked by the normal world. As the hardware interrupt of the physical consent will be first captured in the secure world.

**Phishing Attacks.** These attacks may display a forged input window to cheat the user to enter his password. Complementary techniques such as a security indicator controlled by the secure world (e.g., an LED light) can be used. Moreover, even if the attacker gets the password, they still can't complete an authentication process as they cannot forge or emulate a physical user consent.

**Rollback Attacks.** The attacker may rollback the MAP software and the corresponding configuration file to an old version, which still has a valid integrity value signed by the device key. Such attacks could be prevented using a secure counter or clock only accessible to the secure world to track the MAP's states. Moreover, even if the rollback of a vulnerable version is success, program bugs are most likely to exist in the normal part, as the secret part often has small code base and simple logic, especially for MAPs. Exploiting these bugs cannot disclose the authentication secrets due to the TAuth isolation.

**Iago Attacks.** Iago attack [11] presents a complete example that the malicious Rich OS can cause a protected application to behave abnormally by manipulating

the return values of mmap system calls, and can further conduct return-oriented programming (ROP) attacks to disclose its secrets. In TAuth, if there is any system call invocation in the secret part, the return values from the exit gate will be checked to avoid malicious ones. The check strategy is shared with existing solutions against these attacks [18].

**ROP Attacks.** ROP attacks tamper the program control flow to cause unusual malicious behaviors without modifying the program code, thus could bypass the code integrity verification. First, there is only very small code base in sensitive functions for an attacker to construct ROP gadgets. Second, as the secure stack used by secret parts is isolated, an adversary has no chance to fake a stack to tamper the control flow. Third, TAuth ensures the secret part can only be called through designated function entries, making gadgets in normal part can only be at the function granularity. Even if the attack succeeds in the normal part, the payload still can't disclose the secrets due to the TAuth isolation.

## 6 Implementation

We develop TAuth prototype system on a Trustzone-enabled development board, Xilinx ZYNQ-7000 AP Soc [34], with a Cortex-A9 dual-core processor, 1 GB external DDR3 RAM and 256 KB on-chip SRAM.

**Normal World.** We run Linux 2.6.38 as the Rich OS in the normal world, with several modifications. (1) We add a kernel parameter *auth_mem* which indicates the memory region used for MAPs, i.e., the AUTH_ZONE. (2) We change the implementation of the *execve* system call to add an process creation routine specially for MAPs, which informs the secure world to perform the MAP program launching mentioned in Sect. 4.4. (3) We change the implementation of the *fork* system call to add an MAP cloning routine, which informs the secure world to copy the SPT and secure memory to an identical clone. (4) We insert some *smc* instructions in the kernel code to perform necessary communications with the secure world, such as the one in page fault handler for context switch, and the one in the universal file system component for secure I/O.

We implement a prototype trusted I/O path using an UART port on Zynq-7000, which can be configured as secure only or shared by both worlds. We connect a PC to the development board via the UART port, whose keyboard and screen are used as the I/O peripherals. We use Tera Term, a PC terminal tool for serial port debugging on PC to transfer the I/O data between them. *Smc* instructions are inserted into the kernel's UART driver code, which inform TAuth to perform secure I/O transactions for MAP secrets.

Our method requires a little modifications to the kernel code, which may not be feasible for closed source systems. However, TAuth is designed as a system-level security solution for device vendors, who usually maintain their own kernel source code. Also, the modifications contain only about 510 LOC to the kernel, which is pretty light-weight, making TAuth practical to be deployed.

**Secure World.** We build TAuth in a bare metal secure world retrenched from an open-source secure kernel, Sierra TEE [13], only reserving its boot code. We modify the boot code to divide the physical memory by configuring TZASC. TZASC in our ZYNQ development board is implemented as a secure control register called TZ_DDR_RAM, which can only be accessed in the secure world at 0xF8000430. The register divides the 1 GB external RAM into 16 regions (so each region has 64 MB RAM), using 16 control bits indicating their security status. TAuth reserves the top 128 MB RAM, the top 64 MB of which is configured as secure for SECURE_ZONE. The other 64 MB is for AUTH_ZONE. Our evaluation result proves that the region is enough for the secret parts of most commodity MAPs.

After system initialization, TAuth boots the normal world's Linux kernel. The kernel first loads the MAP configuration files. Then TAuth verifies their signatures using the device private key and moves them into secure world. During runtime, TAuth will only approve the creation of a valid MAP process whose signature has been verified. As the configuration files define all authorized MAPs, device vendors should sign them in a secure offline environment. Though TAuth fills the gap of security and openness for Trustzone, how to make it commercially available to third-party MAP providers concerns business cooperation, which is out of the scope of this paper.

# 7 Evaluation

## 7.1 MAP Examples

We use three real-world MAPs to perform our security and performance evaluations: *GA*, *tiqr* and *OpenSSL*.

**Google Authenticator.** As mentioned in Sect. 2.2, GA generates One-Time-Passwords (OTPs) for Google users as a second authenticator in addition to their username and password to log into Google services or other sites [14]. It uses a secret key provided by Google (scanned or manually entered) to generate a sha1 HMAC using the key and a timestamp or a counter as the authentication OTP. The secret key is stored in the APP's local database, representing the authentication secret in TAuth, and the computation code includes the OTP generation algorithm.

**Tiqr.** Tiqr is an open-source authentication solution for mobile devices and web applications [3]. It is based on Open Standards from the Open Authentication Initiative (OATH). It performs challenge/response authentication using QR codes. After obtaining the authentication challenge from the QR code, the user needs to enter a pin code to finish the authentication, which represents the secret need to be protected by TAuth.

**OpenSSL.** We also use OpenSSL as a tested MAP for the convenient of security and performance evaluation, by linking its library into a light-weight embedded web server (Nginx) to establish SSL network connection. We use OpenSSL RSA

as the cryptographic scheme. The RSA private key is denoted as BIGNUM data structure, containing the two large prime numbers ($p$ and $q$), and the key's exponent $d$. OpenSSL implements its own heap management function *OPENSSL_malloc*. So all *OPENSSL_malloc* calls for BIGNUM are redirected to *secure_malloc* in TAuth.

### 7.2 Secret Part Size

Since TAuth needs to setup a SPT for each secret part, we calculate how much additional memory is needed for them. First, we use the method introduced in [35] to divide the three MAPs, which combines the use of static taint analysis and dynamic execution track. They have integrated the partition method into their vitalization-based protection architecture and have proved its security. So we believe our partition result is complete and secure, which is proved in our security evaluation. Then we modify the definition of the sensitive functions with different GCC section attribute from *.text*, so that they will be compiled into separated sections. Hence, TAuth could protect them in the page granularity. These MAPs are re-compiled using the arm-linux-gnueabi-gcc cross-compile toolchain to run on our development board.

The memory consumption depends on how many sensitive functions are extracted, and how many secure heap and stack pages are reserved, which is shown in Table 1. The OpenSSL has the biggest secure memory consumption, which is 32 KB (8 pages). The GA and tiqr require less memory as their implementation is simpler than OpenSSL. The consumption is negligible compared with the whole memory, which could hardly affect the system memory utilization efficiency.

### 7.3 System TCB Size

TAuth code in the secure world mainly consists of the boot code, the context switch code, simple low-level device I/O code, and several function emulation (*memcpy*, *malloc*..), without any concrete applications, OS services or complex device drivers. As a result, TAuth only has 2200 lines of code. Moreover, the TCB size doesn't increase along with the number of supported MAPs, which is our greatest advantage compared with other Trustzone-based solutions.

Although several library and OS-feature functions are emulated in the secure world, they will only be called in MAPs after being mapped into their SPTs, which all run in the normal world. These code won't increase the system TCB as they will never be executed in the secure world. TAuch could ensure this by only mapping them as non-executable in the secure world.

### 7.4 Security Evaluation

**Memory Disclosure Rootkit.** We first evaluate to what extent can TAuth achieve the isolation of authentication secrets against disclosure attackers. So we

**Table 1.** Secure memory consumption.

| MAP | Func num | Func page | Sec heap | Sec stack | Total |
|---|---|---|---|---|---|
| OpenSSL | 20 | 5 | 1 | 2 | 8 pages |
| GA | 11 | 2 | 1 | 2 | 5 pages |
| tiqr | 6 | 1 | 1 | 1 | 3 pages |

write a malicious kernel module, which scans the whole normal world memory and tries to find targeted secrets when running these three MAPs. When running in the origin Linux system, there are several secret values found in the program heaps. But when running in TAuth, no secrets could be found. This proves that our program partition is correct, ensuring that no secret operations reside in the normal part and the secrets will only exist in the secure part.

**In-application Vulnerability Exploit.** We use the HeartBleed PoC [2] to launch RSA key disclosure attack targeted on the Nginx server with a vulnerable OpenSSL version 1.0.1f. We get private keys after sending 43 HeartBleed requests when running Nginx in origin Linux. However, when running in TAuth, no fragment of private keys is leaked no matter how many HeartBleed requests are sent. The HeartBleed case proves that TAuch could effectively defense attacks exploiting in-application vulnerabilities.

**I/O Hijacking.** We implement a POC malware acting as a UART logger, who tries to steal the tiqr's pin code entered by the user. It hooks the normal world UART FIQ interrupt handler, and also periodically queries the UART driver buffer to intercept any possible I/O data. When running tiqr in origin Linux, the malware records all the keystrokes. For the TAuth case, the hook code in the FIQ interrupt handler never get executed and only dummy values are obtained from the driver buffer.

### 7.5 Performance Evaluation

**System Overhead.** As TAuth is designed specially for MAPs. We first evaluate whether TAuth has performance effects on other system components. We run LMBench, a series of microbenchmarks for OS services to measure the overall system performance overhead. Table 2 shows the results compared with origin Linux. We also list LMBench results of another similar system from its paper, i.e., a Trustzone normal world isolation solution (SecRet [21]). TAuth produces nearly negligible system performance overhead compared with origin Linux, which proves that the performance effect is localized, only affecting the protected MAPs. By contrast, SecRet incurs much higher overhead, as it needs monitor of global system behaviors, including all page table updates and user-kernel mode switches, which are all omitted by the efficient TAuth isolation.

**World Switch Times.** As the normal part and the secret parts may call each other, we measure the overhead of Trustzone world switches caused by cross-part

**Table 2.** LMBench Results (in microseconds).

| Syscalls | Linux | TAuth | Overhead | SecReT |
|---|---|---|---|---|
| Null | 0.33 | 0.33 | 1x | 3.9259x |
| Read | 0.42 | 0.43 | 1.02x | 3.7273x |
| Write | 0.54 | 0.54 | 1x | 3.7381x |
| Open/close | 6.61 | 6.69 | 1.01x | 1.6264x |
| Fork | 171.25 | 173.12 | 1.01x | 1.1819x |
| Fork+Exec | 194.63 | 201.27 | 1.03x | 1.1791x |

**Table 3.** Trustzone world switch times.

| Nginx/req | | GA/auth | | tiqr/auth | |
|---|---|---|---|---|---|
| N→S | S→N | N→S | S→N | N→S | S→N |
| 64 | 8 | 18 | 21 | 8 | 19 |

function calls. We add a counter in the secure world to record world switch times during a MAP's execution. Table 3 shows the total switch times after the Nginx server processed one request, and GA, tiqr performed one user authentication. We also evaluate one world switch time by invoking an empty service running in the secure world, which is about 2 milliseconds (ms). For the Nginx server, the switch cost is about 144 ms per request, which can be negligible compared with a normal user authentication procedure. The cost for GA and tiqr is less.

**Application Overhead.** We measure the runtime overhead of the three MAPs against running them in origin Linux. We use the standard Apache *ab* benchmark tool to measure the Nginx's overhead. The tool runs on a different client machine connected with the development board over 1 Gbps Ethernet. It sends 5000 requests with 50 concurrent SSL connections, each request asks the server to transfer a 5 KB file. The benchmark result shows that the latency and throughput overhead is 15% and 21%. We also measure the execution time of one user authentication for GA and tiqr, which mainly contain an OTP generation, or a pin code enter. We perform 50 measurements for each case and record the average value. The runtime overhead for GA and tiqr is 10% and 16%.

TAuth introduces a relatively high MAP runtime overhead, which is not less than 10%, mainly due to the extra security operations in the secure world. However, as the whole execution time of one user authentication is usually short, such overhead won't cause obvious degradation for user experience. Moreover, given the high security requirements of MAPs, such performance sacrifice is acceptable. Moreover, such overhead is localized, which won't affect the execution of other system components.

## 8 Related Work and Conclusion

**Trustzone Authentication Solutions.** The OBC system (On-board Credentials) [23] is a TEE-based security architecture for protecting critical user virtual credentials, which allows anyone to design and deploy new credential algorithms and secrets. [27] proposes a location-based second-factor authentication solution for modern smartphones using Trustzone. It is designed for the scenario of point of sale transactions to detect fraudulent transactions. TrustOTP [17] proposes a Trustzone-based secure onetime password solution, which achieves various OTP protections against malicious mobile OS. While these works take advantage of TrustZone, they all deploy the concrete MAPs and necessary OS services, drivers in the secure world, which significantly increase the TCB size.

**Trustzone Normal World Isolation.** Real Trustzone secure world attacks have energized research into moving Trustzone's protection domain to the normal world. TZ-RKP [9] guarantees Rich OS's code integrity relying on a runtime kernel monitor in the secure world. Based on TZ-RKP's kernel protection, SecRet [21] creates an isolated memory region in a normal world process to protect a secure communication key. All these works introduce great performance overhead as they need to intercept frequent global system behaviors, such as page table updates. TrustICE [16] shares a similar isolation method with TAuth while doesn't support securely calling untrusted external functions. Necessary OS services and drivers are still implemented in the secure domain and the TCB size is not effectively reduced. TrustShadow [24] creates zombie processes in the normal world while runs the real code as shadow TAs in the secure world. With only a lightweight runtime module in the secure world kernel, the TCB is effectively reduced but is still threatened by vulnerable shadow TAs.

**Virtualization-Based Shielding Systems.** Overshadow [12], CHAOS [15] and InkTag [18] use a hypervisor to isolate application memory and CPU state from untrusted OS and still support most OS services. However, they all need frequent encryption and hash operations on the application memory. As virtualization is primarily designed to allow multiple OSs to share the same hardware platform at a heavy cost for performance and code size, these solutions are not practical for resource-constrained mobile devices. Also, they only provide coarse-grained isolation at an application level, which won't work well under attacks exploiting in-application vulnerabilities such as HeartBleed.

**Automated Program Partition.** Program partition for privilege separation prevents malicious exploitation of applications that run with maximum privilege. Privtrans requires expert knowledge to specify privileged functions and variables [10]. It annotates the source code and partitions source program into only two components: a privileged one and an unprivileged one. [25] develops an approach for automated partitioning of critical Android applications into client code running in Trustzone normal world and critical TEE commands running in the secure world. SeCage [35] combines static taint and dynamic execution analyses to partition C applications w.r.t. sensitive data, and proposes a

virtualization-based intra-domain isolation architecture integrating their partition method, which is not suitable for mobile devices.

**Conclusion.** We propose a novel Trustzone-based mobile authentication security schema called TAuth, which achieves two key advantages compared with previous solutions, i.e., a normal world isolation with a small and unchanged TCB, and fine-grained in-application isolation which defenses threatens from both the underlying Rich OS and in-application vulnerabilities. Designed specially for MAPs, TAuth solves two significant technique challenges, including efficient isolation without excessive intervention into the secure world, and securely using untrusted external functions. We deploy the prototype system on real Trustzone device, and perform thorough evaluations using real commodity MAPs. The evaluation results confirm the security and efficiency of TAuth.

**Acknowledgements.** Our work was supported in part by grants from the National Natural Science Foundation of China (No. 61602455 and No. 61402455).

## References

1. How to root my android device using vroot. http://www.androidxda.com/download-vroot
2. Poc of private key leakage using heartbleed. https://github.com/einaros/heartbleed-tools
3. Tiqr. http://www.rcdevs.com/downloads/download/1/Utils/rcdevs_libs-1.0.15.tgz
4. Vmware: Vulnerability statistics. http://www.cvedetails.com/vendor/252/Vmware.html
5. Xen: Vulnerability statistics. http://www.cvedetails.com/vendor/6276/XEN.html
6. Dmitrienko, A., Liebchen, C., Rossow, C., Sadeghi, A.-R.: On the (in)security of mobile two-factor authentication. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 365–383. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_24
7. ARM: Building a secure system using TrustZone (2009). http://www.arm.com
8. ARM: Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO) (2015). https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf
9. Azab, A., Ning, P., Shah, J., Chen, Q., Bhutkar, R.: Hypervision across worlds: real-time kernel protection from the arm trustzone secure world. In: Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS 2014) (2014)
10. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: Proceedings of the 13th Conference on USENIX Security Symposium (2004)
11. Checkoway, S., Shacham, H.: Iago attacks: why the system call API is a bad untrusted RPC interface. In: The 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013) (2013)
12. Chen, X., et al.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: The 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008) (2008)

13. Gonzalez, J.: Open Virtulization for Xilinxs ZC-702. https://github.com/javigon/OpenVirtulization
14. Google: Google Authenticator. https://github.com/google/google-authenticator-libpam
15. Chen, H., Zhang, F., Chen, C., Yang, Z., Chen, R., Zang, B., Mao, W.: Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. In: Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, August 2007
16. Sun, H., Sun, K., Wang, Y., Jing, J.: TrustICE: hardware-assisted isolated computing environments on mobile devices. In: International Conference on Dependable Systems and Networks (DSN 2015) (2015)
17. Sun, H., Sun, K., Wang, Y., Jing, J.: TrustOTP: transforming smartphones into secure one-time password tokens. In: Proceedings of the 22th ACM Conference on Computer and Communications Security (CCS 2015) (2015)
18. Hofmann, O., Kim, S., Dunn, A., Lee, M., Witchel, E.: InkTag: secure applications on an untrusted operating system. In: the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013) (2013)
19. Mccune, J.M., Parno, B., Perrig, A., et al.: Flicker: an execution infrastructure for TCB minimization. In: EuroSys (2008)
20. McCune, J.M., Li, Y., Qu, N., et al.: Trustvisor: efficient TCB reduction and attestation. In: Proceedings of the 31st IEEE Symposium on Security and Privacy (2010)
21. Jang, J., et al.: SeCReT: secure channel between rich execution environment and trusted execution environment. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2015) (2015)
22. Keltner, N.: Here be dragons: vulnerabilities in trustzone (2014). https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html
23. Kostiainen, K., Ekberg, J., Asokan, N., Rantala, A.: On-board credentials with open provisioning. In: Proceedings of the International Symposium on Information, Computer, and Communications Security (2009)
24. Guan, L., Liu, P., Xing, X., et al.: TrustShadow: secure execution of unmodified applications with ARM trustZone. In: Proceedings of the 15th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2017) (2017)
25. Rosculete, L., Rosculete, L., Mitra, T., et al.: Automated partitioning of android applications for trusted execution environments. In: Proceedings of the International Conference on Software Engineering (ICSE 2016) (2016)
26. laginimaineb: Bits, please! (2016). https://bits-please.blogspot.com/
27. Marforio, C., et al.: Smartphones as practical and secure location verification tokens for payments. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2014) (2014)
28. Lindemann, R., Hill, D.B., Tiffany, E.: FIDO UAF Protocol Specification v1.0 (2014). https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-protocol-v1.0-ps-20141208.html
29. Roland, M.: Applying recent secure element relay attack scenarios to the real world: Google Wallet Relay Attack (2013)
30. Rosenberg, D.: Reflections on trusting trustzone. In: BlackHat USA (2014)
31. Rosenberg, D.: QSEE trustzone kernel integer over flow vulnerability. In: Black Hat Conference (2014)

32. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007) (2007)
33. Shen, D.: Attacking your trusted core, exploiting trustzone on android. In: BlackHat USA (2015)
34. Xilinx: Zynq-7000 all programmable SOC ZC702 evaluation kit. http://www.xilinx.com/products/boards-and-kits/EK-Z7-ZC702-G.htm
35. Liu, Y., Zhou, T., Chen, K., et al.: Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In: Proceedings of the 22th ACM Conference on Computer and Communications Security (CCS 2015) (2015)
36. Wu, Y., Sun, J., Liu, Y., Dong, J.S.: Automatically partition software into least privilege components using dynamic data dependency analysis. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013) (2013)
37. Zhang, Y., Zhao, S., Qin, Y., et al.: TrustTokenF: a generic security framework for mobile two-factor authentication using trustzone. In: Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2015) (2015)