2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

TrustICE: Hardware-assisted Isolated Computing Environments on Mobile Devices

He Sun*†‡, Kun Sun‡, Yuewu Wang*, Jiwu Jing* and Haining Wang§
*State Key Laboratory of Information Security, Institute of Information Engineering, CAS, Beijing, China
Data Assurance and Communication Security Research Center, CAS, Beijing, China
{sunhe, wangyuewu, jingjiwu}@iie.ac.cn
†University of Chinese Academy of Sciences, Beijing, China
‡Department of Computer Science
College of William and Mary, Williamsburg, VA, USA
ksun@wm.edu
§Department of Electrical and Computer Engineering
University of Delaware, Newark, DE, USA
hnw@udel.edu

***Abstract*—Mobile devices have been widely used to process sensitive data and perform important transactions. It is a challenge to protect secure code from a malicious mobile OS. ARM TrustZone technology can protect secure code in a secure domain from an untrusted normal domain. However, since the attack surface of the secure domain will increase along with the size of secure code, it becomes arduous to negotiate with OEMs to get new secure code installed. We propose a novel TrustZone-based isolation framework named *TrustICE* to create isolated computing environments (ICEs) in the normal domain. TrustICE securely isolates the secure code in an ICE from an untrusted Rich OS in the normal domain. The trusted computing base (TCB) of TrustICE remains small and unchanged regardless of the amount of secure code being protected. Our prototype shows that the switching time between an ICE and the Rich OS is less than 12 ms.**

***Keywords*—Computing Environment; Isolation; TrustZone.**

I. Introduction

As we are rushing toward the post-PC era, the concept of Bring Your Own Device (BYOD) is a growing trend for allowing people to bring personally owned mobile devices (e.g., smart phones and tablets) to workplace and use them to access company's sensitive data. It can reduce an enterprise's running cost and improve its employees' productivity. However, it also brings the risks of potential information leakage that could hold the enterprises back. When both personal and enterprise applications are installed on the same mobile system, adversaries may compromise the mobile OS kernel by exploiting vulnerabilities in personal applications and then steal data from enterprise applications [1], [2], [3].

Recent hardware progresses on ARM processors have enabled mobile devices to protect secure code's execution in an isolated computing environment [4], [5], [6], [7], [8], [9]. ARM introduces a new CPU mode called *Hyp mode*. KVM/ARM [4], [5] utilizes Hyp mode to port Linux KVM on ARM processors and run virtual machines with comparable performance to native execution. One major concern of hypervisor-based solutions on mobile devices is the higher power consumption due to virtualization operations. ARM also introduces TrustZone technology to protect secure code from insecure code by separating them into two isolated execution domains [9].

Current TrustZone-based solutions [10], [9], [11], [12] install a minimal secure OS in the secure domain and run a number of secure applications in the secure OS. It faces two major challenges. First, the attack surface in the secure domain will increase along with the number of secure applications installed in the secure OS. Moreover, the trusted computing base (TCB) of the secure domain will increase along with the number of kernel modules installed in the secure OS in order to provide additional services and support more applications. Since the secure domain has a higher privilege than the normal domain, when the secure domain is compromised, the normal domain will be compromised too. Second, for third-party software developers, it may be an arduous process for negotiating with OEMs and service providers to get their code installed into the secure domain. Though ARM defines an API specification TZAPI [9] for TrustZone, the detailed implementation depends on the software vendors that typically only have interests in providing their own closed source solutions.

In this paper, we propose a novel TrustZone-based isolation framework named *TrustICE* to provide isolated computing environments (ICEs) on mobile devices. The basic idea of TrustICE is to create ICEs in the normal domain rather than in the secure domain. Instead of using a hypervisor, TrustICE relies on TrustZone to ensure that the secure code in ICEs is securely isolated from the untrusted and insecure code including the Rich OS in the normal domain.

TrustICE aims at providing ICEs to protect secure code without enlarging the attack surface and the TCB in the secure domain. By moving the secure code from the secure domain to the normal domain, the attack surface of the secure domain hardly changes, no matter how many pieces of new secure code are installed and executed on the mobile devices. Similarly, the TCB is unchanged and minimized, since it

Dr. Yuewu Wang is the corresponding author.

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.11

only consists of a Boot ROM and a small trusted domain controller (283 lines of code). The trusted domain controller is responsible for ensuring the integrity and authenticity of secure code before being loaded into the memory, enforcing secure isolation of secure code in the normal domain, and achieving a secure switching between ICE and the Rich OS. Therefore, TrustICE provides third-party software vendors and application developers an isolated computing environment for integrating their secure code into the TrustZone's secure domain, without the arduous negotiation with OEMs.

We can protect one ICE from being accessed by the Rich OS and other ICEs by using TrustZone. First, when secure code is running in an ICE, the Rich OS and other ICEs are all suspended and thus cannot access the resource in the active ICE. Second, when one ICE is in the suspend state, since it may contain state information about secure code, we must prevent the running Rich OS and other secure code from reading the memory of the suspended secure code. Instead of using the heavy encryption/decryption mechanisms, we use the hardware-assisted Watermark technique [13] on ARM processors to dynamically protect the memory regions of the suspended secure code.

We implement a TrustICE prototype on Freescale i.MX53 QSB and develop two ICE usage instances to demonstrate the usability of TrustICE. First, we can successfully run a self-contained cryptographic library in one ICE to provide public key operations. Second, we implement a trusted user interface containing a touchscreen driver and a wireless communication driver for users to interact with the ICE.

In summary, we make the following contributions in this paper.

- We design a TrustZone-based isolation framework named *TrustICE* to provide isolated computing environments on mobile devices without using a hypervisor.
- We enhance the system security. TrustICE can reduce the attack surface of the secure domain and minimize the system's TCB by moving secure code from the secure domain to the normal domain. TrustICE's TCB only includes a Boot ROM and a small trusted domain controller, which is protected by TrustZone in the secure domain.
- We can ensure the isolation of secure code in the normal domain. Since all secure code will be executed in the normal domain, we ensure that no matter whether the secure code is running or suspended, the untrusted Rich OS cannot access or manipulate it.
- We implement a TrustICE prototype on Freescale i.MX53 QSB. The Rich OS is a customized Linux 2.6.35 and Android 2.3.4. The experimental results show that our system can switch from the Rich OS to ICE in 10.6 ms, and switch back from ICE to the Rich OS in 0.8 ms.

The remainder of the paper is organized as follows. Section II introduces TrustZone background. Section III describes the threat model and assumptions. We present the TrustICE framework in Section IV. A prototype implementation is detailed in Section V. Section VI presents experimental results. We perform a security analysis in Section VII. The related work is described in Section VIII. Finally, we conclude the paper in Section IX.

II. Background

ARM TrustZone technology is a hardware security extension in ARM processors [9], [14], [15]. Commodity processor chips with TrustZone extension have been introduced by mainstream semiconductor corporations such as Freescale [16], TI [17], and Samsung [18]. Figure 1 shows the TrustZone architecture adopted by most trusted execution environment (TEE) solutions (e.g. MobiCore (Trustonics) [12], SierraTEE [19] and Trusted Logic [20]), which runs untrusted apps on an untrusted Rich OS in the normal domain and protects secure apps on a small customized secure OS in the secure domain. The isolation between two domains is enforced by a secure monitor in the secure domain to ensure CPU state isolation, memory isolation, and I/O device isolation. When the system boots up, a secure boot ensures the integrity and authenticity of the secure OS.

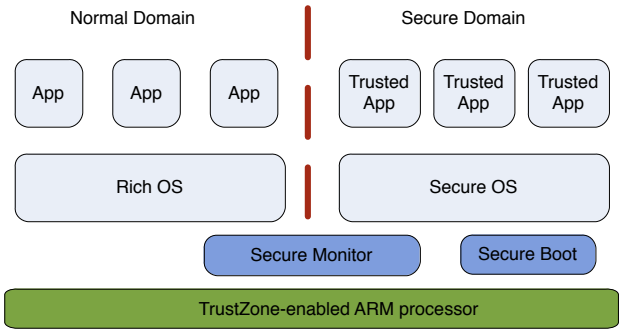

Fig. 1: Traditional TrustZone Architecture

A. CPU State Isolation

TrustZone supports two CPU states, *secure state* and *non-secure state*, for the secure domain and the normal domain, respectively. Two CPU states are separated through a set of banked $CP15$ registers that could be assigned two values. Each state consists of seven CPU modes: *User*, *FIQ*, *IRQ*, *Supervisor*, *Abort*, *Undefined*, and *System*. All the modes, except the User mode, are privileged modes. Mobile applications run in the User mode, and the OS kernel runs in the privileged modes. Secure and non-secure states can be distinguished by setting the NS bit in the Secure Configuration Register (SCR), which can only be modified in the secure state [14]. TrustZone adds a new privileged *Monitor mode* that only runs in the secure state to serve as a gatekeeper managing the switching between the two states. Both states can call a privileged Secure Monitor Call (*SMC*) instruction to enter the Monitor mode and then switch to the other state. Moreover, a hypervisor mode called *HYP mode* has been integrated in ARM Cortex A15 processor family to support virtualization of non-secure operations [4], [5].

B. Memory Isolation

TrustZone provides virtual MMU mechanism to support different virtual memory address spaces in the secure domain and the normal domain. The same virtual memory address in two domains will be mapped to different physical memory addresses. TrustZone allows the secure domain to access the virtual memory address space in the normal domain, but not vice versa. Note that the virtual MMU mechanism can only guarantee the isolation of virtual memory spaces, but not the physical memory spaces. TrustZone includes a TrustZone Address Space Controller (TZASC) to partition DRAM into secure or non-secure memory regions. The normal domain cannot access the physical memory regions assigned to the secure domain. TZASC is accessed through Watermark technique on Freescale's i.MX53 QSB.

i.MX53 QSB provides two Watermark regions for each external DDR memory. One of the Watermark regions is configured to only allow access from the secure domain. The other one is accessible only from the supervisor mode no matter what domain it is in. A Watermark region must be continuous and its size can be configured by setting the Watermark Start ADDR Register and the Watermark End ADDR Register. Moreover, the size of the region cannot exceed 256 MB. With the watermark mechanism, a complete runtime isolation can be implemented between the secure and normal domains.

C. I/O Device Isolation

Hardware Interrupt Isolation. The TrustZone Aware Interrupt Controller (TZIC) is a TrustZone-enabled interrupt controller on i.MX53 QSB, which allows a fine-grained and independent control over each interrupt connected to the controller. There are two types of hardware interrupts: IRQ (interrupt request) and FIQ (fast interrupt request). An interrupt can be set as either secure or non-secure in the TZIC, and a secure interrupt can only be configured by the secure domain while a normal interrupt can be configured by both domains. The secure interrupt can assert FIQ or IRQ while the non-secure interrupt can only assert IRQ. The CPU is responsible for identifying and redirecting the interrupts to the correct domain.

DMA Isolation. Certain I/O devices, such as touchscreen controller and storage controller, can transfer data to and from memory using Direct Memory Access (DMA). A TrustZone-aware DMA controller (DMAC) supports concurrent secure and normal DMA accesses, each with independent interrupt events. The DMAC can prevent a peripheral assigned to the normal domain from performing a DMA transaction on the memory regions of the secure domain.

III. Threat Model and Assumptions

We trust the code in the Boot ROM and the domain controller in the secure domain. An adversary is able to exploit software vulnerabilities to compromise the Rich OS and then launch attacks to compromise the code and data in ICEs. Moreover, we assume the secure code may have unknown bugs, which can be exploited by adversaries to compromise one ICE and then target at compromising other ICEs. Therefore, a malicious Rich OS or compromised ICE may launch the following attacks:

- *Faking an ICE to deceive a user.* When the Rich OS is requested to switch to an ICE, it may fake an ICE and deceive the user to leak sensitive data such as password. A mechanism should be provided to authenticate an ICE.
- *Tampering the static image of secure code.* The Rich OS may tamper with the code image saved on the non-volatile storage. Thus, the code image should be verified before being loaded into memory.
- *Tampering the secure code and data in the memory.* Since TrustICE runs ICEs in the normal domain, the Rich OS has the same level of privilege as the ICEs to access their memory. We must protect the secure code from being accessed by the Rich OS.
- *Preventing the system from switching into the secure domain*. Attackers can launch denial-of-service (DoS) attacks to prevent the system from switching to the secure domain and preparing ICEs.

We assume that the attacker cannot access physical mobile devices or launch local physical attacks, such as removing the Micro SD card.

IV. TrustICE System Design

Figure 2 shows the TrustICE architecture. The Rich OS is still installed and executed in the normal domain. In the secure domain, besides the OEM software, a small trusted domain controller (TDC) is responsible for loading secure code in one ICE, enforcing secure isolation of the secure code from the Rich OS, and achieving a secure switching between an ICE and the Rich OS. Since both the Rich OS and ICEs are running in the normal domain, a secure isolation must be enforced by TDC.

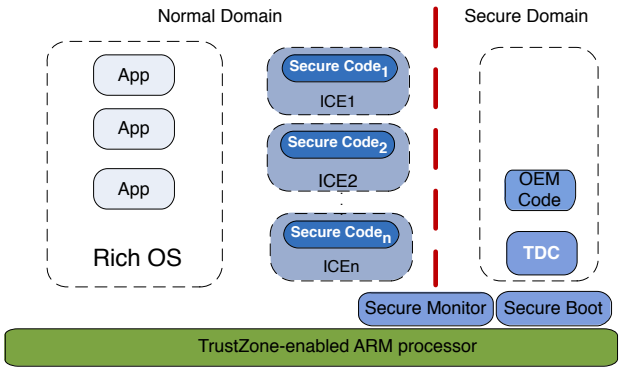

Fig. 2: TrustICE Architecture

Multiple ICEs can be dynamically created in the normal domain to protect the execution of different secure code. We call the code constructing an ICE as *ICE code* or *ICE* in short, and we call the application code to be executed in one ICE as *secure code*. Secure code can vary from simple self-contained cryptographic operations to complex applications that rely on system libraries and device drivers, etc. Accordingly, the ICE code will vary from a simple memory controller to a thin OS.

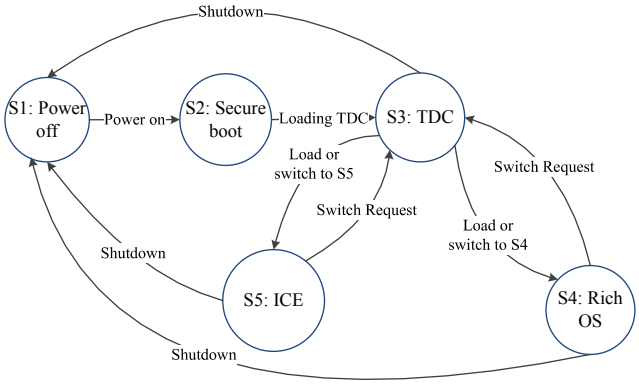

Fig. 3: TrustICE State Machine

A. System State Machine

Figure 3 shows the system state machine of TrustICE. TrustICE has five system states: system power off (S1), secure boot (S2), TDC (S3), the Rich OS (S4), and ICE (S5). At the beginning, system is powered off (S1). When the system is powered on, it first enters the secure domain. Then the system enters into secure boot (S2). A secure bootloader is loaded into the secure domain by the Boot ROM code. The secure bootloader uses TrustZone to set a secure memory region that only allows access from the secure domain and load the TDC image into it. Then, the system enters into TDC (S3), which loads the Rich OS's bootloader to boot up the Rich OS. After the Rich OS boots successfully, the system enters into the Rich OS (S4).

When secure code needs to be executed, it makes an *SMC* system call to switch the system to TDC (S3), which suspends the Rich OS, saves the context information, and disables interrupts and DMA. All these operations can be conducted by configuring processor registers in the secure domain. After verifying the integrity of the secure code and the ICE code, TDC loads them into a secure memory region, which is protected by the Watermark mechanism. Next, TDC switches the system back to the normal domain to execute the secure code in the ICE (S5). When the secure code ends, the system switches back to TDC, which then resumes the Rich OS. As we can see, the entire switching process is controlled by TDC in the secure domain. Moreover, whenever the system is shutdown, it returns to state S1.

B. Secure Isolation

By running TDC in the secure domain, we can utilize TrustZone hardware security support to isolate TDC from the Rich OS and ICEs. On the other hand, since both the Rich OS and ICEs are running in the normal domain, we must ensure a secure isolation that can protect one ICE from being accessed by the Rich OS or other ICEs.

CPU Isolation. When one ICE is running, the Rich OS and other ICEs are all being suspended by TDC. Before the ICE is executed, TDC will save all CPU state information in a secure memory storage. After the ICE accomplishes its mission and switches back to TDC, TDC cleans up the footprint, recovers the CPU state information, and resumes the Rich OS. Therefore, the Rich OS cannot obtain any sensitive CPU states from ICE.

Memory Isolation. When the system exits one ICE, TDC protects both the secure code and the ICE code in a secure memory storage using the memory Watermark mechanism. Therefore, when the Rich OS resumes to run, it cannot access the code and data in the secure memory storage. Since secure code runs in the normal domain, TDC must move the secure code to the normal memory storage before the code can run. When secure code is running, the Rich OS still cannot access its memory since it has been suspended by TDC. We will describe the details of using the memory Watermark mechanism to protect the secure code and the ICE's memory in Section V.

I/O Device Isolation. In the secure domain, we can simply block all external interrupts from arriving at TDC. Thus, we can protect TDC from being interrupted by external interrupts issued by malicious devices. To protect ICE from being intercepted by a malicious external interrupt, TDC disables all the hardware interrupts before going to ICE. However, since one ICE may need some interrupts enabled to interact with I/O devices, we develop a fine-grained interrupt control method by enabling a minimal set of required interrupts and disabling all the other interrupts before switching to ICE.

C. Trusted Path

A trusted path should be guaranteed in both system booting and system switching. A secure boot performs cryptographic checks at each stage of the secure domain's booting process. The trusted code in the Boot ROM first verifies the signature of the secure bootloader image using an RSA public key stored in one electrically programmable Fuse (eFUSE). Since the eFUSE and the RSA code in the Boot ROM cannot be tampered, adversaries cannot manipulate the secure bootloader image without being detected. Thus, a trusted secure bootloader can continue to secure the loading of TDC, which is consequently responsible for ensuring the secure load of the ICEs.

When the system switches from the Rich OS to one ICE, a compromised Rich OS may fake an interface of the ICE to deceive users to leak sensitive information. In this attack, the trusted TDC is totally bypassed. To protect against this attack, we need to make sure that TDC must be involved in the switching process. One potential solution is to present users an exclusive comprehensible signal that can only be controlled by TDC. For instance, an LED light solely controlled by TDC can be used as a trusted display signal to illustrate that one ICE is running in the system.

V. Implementation

We implement a TrustICE prototype using Freescale's i.MX53 quick start board (QSB). i.MX53 QSB comes with an ARM Cortex-A8 1 GHz application processor with 1 GB DDR3 memory and a 4GB Micro SD card. It has a 64 KB Boot ROM, which supports SHA-256 and 2048-bit RSA operations. The touchscreen we use is MCIMX28LCD, a 4.3" 800x480 (WVGA) display with 4-wire resistive touchscreen. The board is equipped with a HUAWEI MC323 CDMA wireless module [21] as the cellular communication component.

A. Memory Map

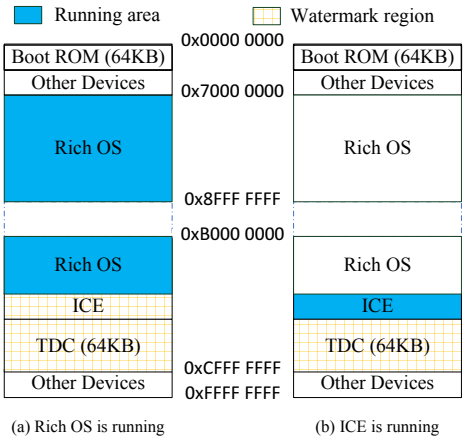

Fig. 4: Memory Map with One ICE

Freescale i.MX53 QSB provides 1GB RAM memory with two external DDR memories: one ranges from `0x70000000` to `0x8FFFFFFF` and the other from `0xB0000000` to `0xCFFFFFFF`. As described in Section II-B, watermark memory is the memory that can only be accessed by the secure domain, so the Rich OS cannot access the watermark memory from the normal domain. The TDC is always protected in the watermark memory. Since the system only defines one watermark memory area on each memory, we design different memory mapping schemes to support different number of ICEs in the system with one watermark.

Figure 4 shows the TrustICE memory map when only one ICE is required in the system. We save the highest 64KB RAM memory from 0xCFFF0000 to 0xCFFFFFFF for TDC. The adjacent memory area is reserved for the ICE. The size of the ICE depends on the complexity of the secure code plus its required system libraries and functions in the ICE, which can be either pre-loaded into memory when the system boots up or dynamically loaded into memory when the secure code needs to run. The remaining RAM memory is allocated to the Rich OS. When the Rich OS is running, TDC extends the watermark region to include the ICE memory, so that the Rich OS cannot access the ICE. When the ICE needs to run in the normal world, TDC dynamically changes the watermark configuration to exclude the ICE memory from the watermark region. Note that the Rich OS has been suspended at this time.

When two ICEs are required in the system, the memory map is depicted in Figure 5. The challenge is to protect three memory areas, namely, the TDC and two ICEs, using only one watermark memory region. In particular, the TDC should be always protected in the watermark memory. We solve this problem by putting the TDC between two ICEs within one continuous memory space and dynamically changing the configurable watermark region. When the Rich OS is running, the TDC and two ICEs are all covered by the watermark. When one ICE is running, the TDC and the other ICE are protected by a new watermark region.

When more than two ICEs are required, we cannot cover all the other ICEs and the TDC while one ICE is running

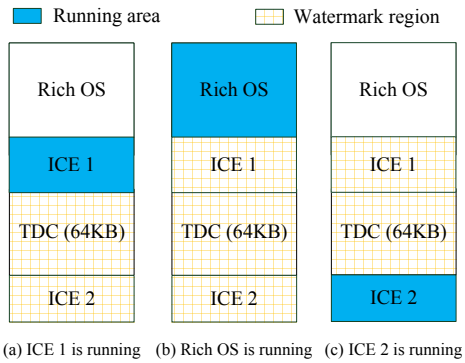

Fig. 5: Memory Map with Two ICEs

using one watermark region, since there is only one watermark region and it must be continuous in the memory. To solve this problem, we designate all ICEs and the TDC in one continuous memory that can be protected by the watermark region, as shown in Figure 6(a); however, whenever an ICE is requested to run secure code, the TDC copies the active ICE to a reserved memory region called *ICE Runtime Environment* to run the secure code, as shown in Figure 6(b). Therefore, even if the Rich OS or one ICE is malicious, the other ICEs are still protected. All the codes of the ICEs are pre-loaded one by one into the adjacent area of the TDC. The ICE Runtime Environment is reserved at the adjacent area of the ICEs, and it is also protected by the watermark when the Rich OS is running. Since we need to copy one ICE to the ICE runtime environment every time when one ICE needs to run, it will increase the switching time between ICEs and the Rich OS. To solve this problem, we would suggest more watermark memory regions should be supported on the ARM platform.

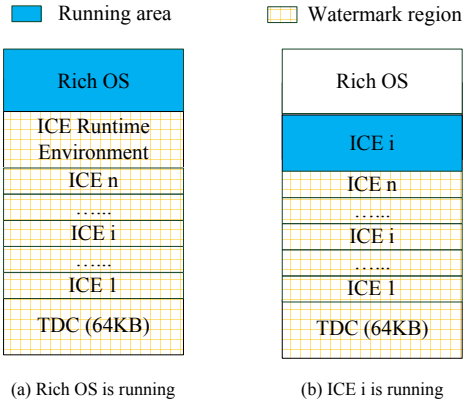

Fig. 6: Memory Map with Multiple ($>$ 2) ICEs

B. Loading TDC

Similar to other embedded systems, i.MX53 QSB starts with the code in the Boot ROM when it is powered on. The code memory address of ROM is between `0x00000000` and `0x0000FFFF`. i.MX53 QSB provides a High Assurance Boot (HAB) to ensure the authenticity and integrity of an image that will be loaded into the processor chips [13], [22]. The Boot

ROM loads the secure bootloader image from the Micro SD card, and then HAB verifies the integrity and authenticity of the secure boot image using SHA-256 and RSA algorithms embedded in the Boot ROM code. The hash of the public key of the root certificate is stored at one electrically programmable Fuse (eFUSE) named SRK_HASH, whose value cannot be changed after eFUSE blowing. The secure bootloader in this work is developed based on Uboot [23]. It is responsible for loading TDC into the watermark region using Uboot $mmc\ read$ command. A certificate signed by root certificate is used as the developing certificate of TDC, and the developing certificate is stored with TDC. The value of SRK_HASH is used to verify TDC's developing certificate. TDC will be loaded only if the verification succeeds.

C. System State Switching

TDC is responsible for enforcing a secure switching from the Rich OS to one ICE and then back to the Rich OS. Figure 7 shows the switching steps when an application needs to execute secure code. When one application is running in the user mode of the normal domain, an SMC instruction must be called to switch from the normal domain to the TDC in the secure domain; however, SMC is a privileged instruction and cannot be called directly by applications. Hence, we add an SMC system call into the Rich OS to allow applications to call the SMC instruction. The system call contains an SMC instruction to forward the request from the application to TDC.

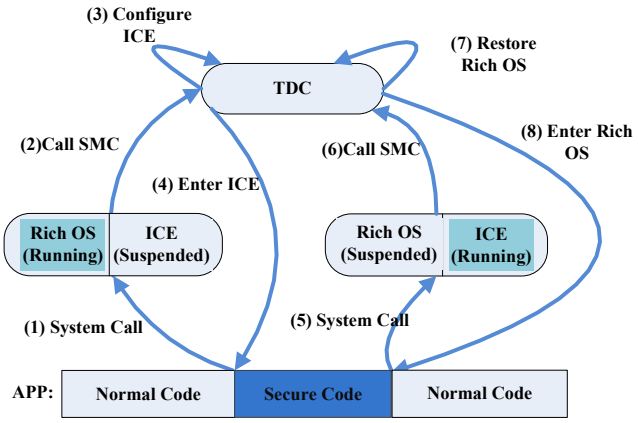

Fig. 7: Switching between Rich OS and ICE

When secure code needs to be executed, the SMC system call is triggered to make the system enter the TDC in the secure domain, and the code indexed by the SMC offset (which is 0x8) in the *exception vector table* of the monitor mode is invoked. The integrity of ICE code is checked by verifying the signature generated by its developing certificate. The signature verification algorithm, RSA in our prototype, is implemented in TDC. The validity of the developing certificate is ensured by SRK_HASH eFUSE. Next, TDC proceeds the switching process through following operations.

1) Backing up the translation tables, the exception vector table, and related registers used by the Rich OS. TDC overwrites the translation table to include the memory map of ICE; the start address of ICE's exception vector table is written into the Non-secure Vector Base Address Register. Then, only the interrupts required by the ICE are enabled while others are masked off in the TZIC.
2) Verifying the signature of the secure code with RSA algorithm. Loading the secure code into ICE and generating the security attestation of ICE execution with TDC's private key.
3) Switching the system back into the normal domain to run the secure code in its ICE.
4) Resuming the Rich OS. At the end of the secure code, the SMC system call will be called again to switch the system into TDC, which cleans up CPU registers and resumes the Rich OS.

With the interrupt control in the TZIC, the execution of ICE will not be intercepted by unnecessary interrupts. The desired interrupts will be handled in the exception vector table of ICE. Since the Rich OS may hijack the SMC system call to deceive the user into a fake ICE to perform sensitive transactions, our prototype uses an LED light that is solely controlled by TDC on the development board to notify the user if one ICE is running or not.

D. ICE Development

Our ICE prototype implements a number of basic library functions for the secure code to call, including a self-contained cryptographic library that supports RSA signature and SHA-1 hash, a user interface using LCD touchscreen, and a network interface based on HUAWEI MC323 CDMA radio module [21].

ICE code is running in the non-secure $Supervisor\ mode$, which allows secure code running in the non-secure $User\ mode$ to call system functions by making system calls. We have defined five system calls numbered from 1 to 5: (1) ICE exit call, (2) RSA signature call, (3) SHA-1 hash call, (4) network interface call, and (5) user interface call. Among the five system calls, only the first ICE exit call is a must for any ICE to return to the Rich OS.

The ICE exit call is implemented to call the SMC instruction to clean up the trace of ICE and resume the Rich OS. The RSA algorithm is ported from an open-source library PolarSSL [24]. In our prototype, SHA-1 algorithm uses the hardware accelerator (SAHARA [13]) on board. The HUAWEI MC323 CDMA radio module includes a complete TCP/IP stack, so using *AT* command, ICE code can easily communicate with a remote server through cellular network. The radio module is an independent device and its logic code is fixed, so it cannot be tampered after being deployed on mobile devices. The user interface call is based on a driver for touchscreen to display pictures in the format of RGB565 on an 800*480-pixel screen. It can render any words and pictures fitting into the screen. When the ICE is running, an interrupt rises as soon as there is a touch on the screen. The touchscreen driver captures the interrupt and reads the X-Y coordinates of the touch point. Figure 8 shows a snapshot of the screen that asks the user to input a passcode before entering an ICE. In our future work, we will support additional system functions in ICE, such as an openGL library and a Wi-Fi driver. Furthermore, more powerful ICEs can be developed by third-party programmers and service providers.

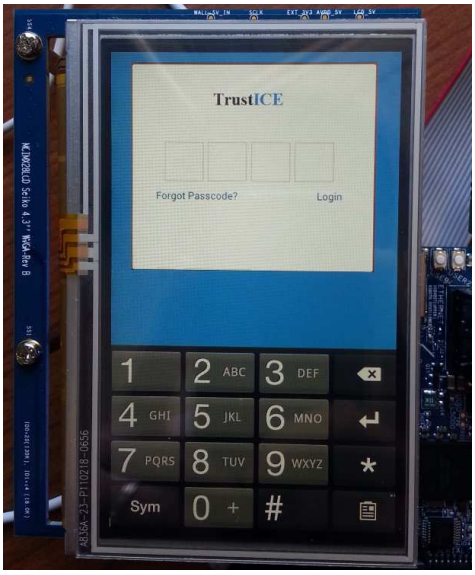

Fig. 8: Snapshot of Passcode Input

The secure code can be a code segment of an application that makes system calls in ICE to request corresponding services. In this case, the beginning and ending of the secure code are both marked by SMC system calls. The system call to mark the beginning of the code is executed in the Rich OS, and the ending system call is the ICE exit call that is handled in ICE. An example of secure code is listed in Appendix A. In the secure code, all privileged functions are implemented in the ICE. In other words, the secure code cannot rely on the untrusted Rich OS.

VI. Performance Evaluation

The implementation of TrustICE should satisfy the following two performance requirements. First, the switching time between the Rich OS and one ICE should be small. Second, the ICE should have small performance impacts on the Rich OS. We conduct experiments to evaluate the performance on two ICEs: the encryption ICE includes a self-contained library for RSA and SHA-1 algorithms and the interface ICE includes a touchscreen interface and a cellular network interface. For the encryption ICE, since no interrupts are needed, all the interrupts are masked. For the interface ICE, only the interrupt of the 4-wire resistive touchscreen is enabled.

The size of the encryption ICE is 46,424 bytes, and the size of the interface ICE is 1,050,892 bytes including the display picture of 1,041,832 bytes. The size of the interface ICE can be further reduced by compressing the image or choosing a low-quality picture. We use the performance monitor in the Cortex-A8 core processor to count the CPU cycles and then convert the cycles to time by multiplying $1\ ns\ /\ cycle$. We repeat the experiments 100 times and take the average value.

A. Switching Time

We measure the switching times from the Rich OS to ICE and from ICE to the Rich OS, respectively. Table I shows the switching time for our two prototype ICEs.

TABLE I: TrustICE Switching Time

Operation	Encryption ICE (us)	Interface ICE (us)
From the Rich OS to ICE	527.77	10611.21
From ICE to the Rich OS	783.47	782.96

The time latency for switching from the Rich OS to ICE includes (1) exiting the Rich OS, (2) loading and verifying the ICE code, (3) loading and verifying the secure code, (4) suspending the Rich OS and configuring the ICE, and (5) entering the ICE. A breakdown of the switching times are shown in Table II. It spends 0.5 ms to switch from the Rich OS to the encryption ICE and 10 ms for switching into the interface ICE. This is because the interface ICE has a larger code base, which makes the TDC spend more time verifying its integrity when loading it into the memory. We use the SHA-1 algorithm to verify the ICE code and the secure code. The switching time will increase along with the code size of the ICE and the secure code. However, since ICE is protected by the watermark when suspended, if we do not worry about ICE being compromised during its runtime, we can skip the stage of verifying the ICE code integrity. If so, the switching time for any ICEs will have a similar value.

TABLE II: Time Breakdown: From the Rich OS to ICE

Operation	Encryption ICE (us)	Interface ICE (us)
exiting the Rich OS	5.84	5.93
verifying secure code	9.76	9.75
verifying the ICE	475.85	10559.37
configuring the ICE	35.05	34.89
entering ICE	1.27	1.27
total	527.77	10611.21

After executing the secure code, the system can switch from the encryption ICE to the Rich OS in 783.47 us and switch from the interface ICE to the Rich OS in 782.96 us. A breakdown of the switching time includes (1) exiting ICE, (2) restoring the Rich OS and DMA, and (3) entering the Rich OS. The breakdown results are listed in Table III. Since we do not need to perform the expensive SHA-1 operations, the switching time is much smaller than switching from the Rich OS to one ICE. Moreover, the switching times are almost the same for different ICEs. Overall, the switching times in both directions are very small and can barely be perceived by a user.

TABLE III: Time Breakdown: From ICE to the Rich OS

Operation	Encryption ICE (us)	Interface ICE (us)
exiting ICE	0.49	0.48
restoring the Rich OS	19.26	19.41
entering the Rich OS	763.72	763.07
total	783.47	782.96

B. Performance Comparison between ICE, TDC and the Rich OS

We compare the performance differences when the system runs in ICE, TDC, and the Rich OS, respectively. We first

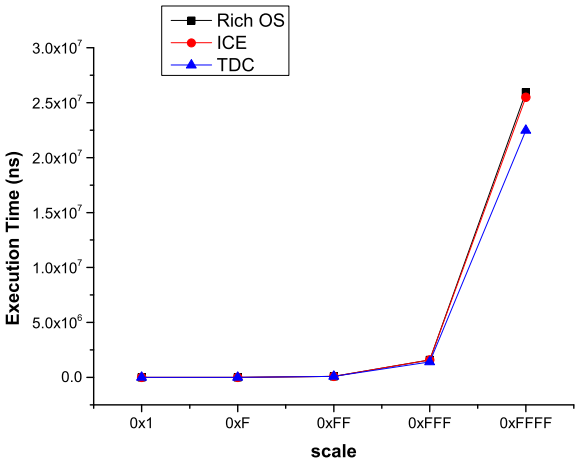

Fig. 9: Absolute time of code execution

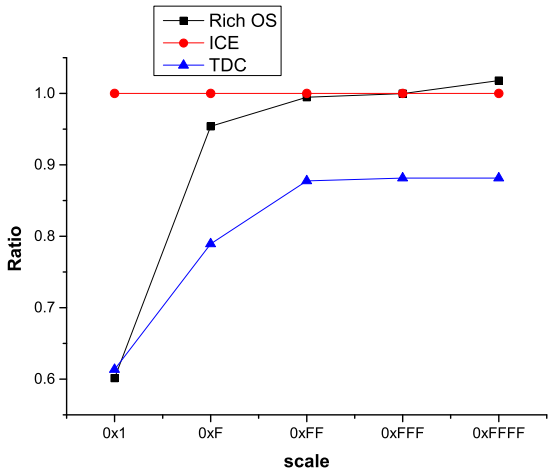

Fig. 10: Relative time of code execution

study the performance in ICE, and then use it as a baseline to compare the other two environments. Then we evaluate the performance of Android in the secure domain and the normal domain, respectively. Also, we compare the power consumption in these three cases.

We run a piece of self-contained code in TDC, ICE, and the Rich OS, respectively, to compare their execution performance. The code is shown in Listing 1. We run the code with five different scales, namely, `0x1`, `0xF`, `0xFF`, `0xFFF`, and `0xFFFF`. In each scale, we run the code in each environment 100 times and show the average results in Figure 9.

Listing 1: The Self-Contained Code

```
unsigned int i,j,scale;
const char* msg="!!!Hello World!!!";
char dest[17];

for(j=0;j<scale);j++){
    for(i=0;i<17;i++)
    dest[i]=msg[i];
    }
```

From the figure, we can see that the code running in the TDC is the fastest. The average execution time ranges from 950 ns to 0.03 s. Hence, we use the time in ICE as the standard time, and generate two ratios by dividing the other two times with the standard time. The comparison result is shown in Figure 10. We can see that the Rich OS is faster than the ICE when the scale is small, and ICE is faster when the scale is larger than $0xFFF$. This is because when the execution time is small, no interrupts in the Rich OS will break the execution of the self-contained code. However, as the scale increases, the execution in the Rich OS will be interrupted more frequently and the execution time thus increases.

Since there are devices that can only be accessed in the secure domain, the Rich OS is designed to run in the secure domain when only the Rich OS is deployed on i.MX53 QSB. In TrustICE, when the Rich OS needs to access those devices that are only open to the secure domain, it issues a request to switch to the secure domain by calling the SMC instruction. Then the secure domain helps access the devices and returns the result back to the Rich OS. To evaluate the impact of moving the Rich OS from the secure domain to the normal domain, we measure the benchmarks of Android in both the normal domain and the secure domain. We install Quadrant [25], a benchmark app for mobile devices, to run on both domains to compare their performance. The app available on Google Play is capable of measuring CPU, memory, I/O, and 3D graphics performance. The overall and categorical Quadrant benchmark scores are shown in Figure 11. The higher the score is, the better the performance is. The results show that ICE has little impact on the Rich OS. The performance of CPU and memory is barely affected. The I/O performance in the secure domain is better than that in the normal domain. This is due to the existence of those I/O devices that are open only in the secure domain. Thus, there is no direct access to those I/O devices in the normal domain while we can directly access all the I/O devices in the secure domain.

We also measure the system power consumption in the three different environments. As the input DC voltage of i.MX53 QSB is approximately fixed at 5 V, we measure the input current to the board and multiply it with the 5 V voltage to derive the power consumption. We record the current data when the current keeps constant. From Table IV, we can see that the system power consumption in three cases are almost the same. Due to the constrained functions and operations in the TDC and the ICE, the power consumptions of these two cases are slightly lower than that in the normal domain.

TABLE IV: Comparison on Power Consumption

System	Power (W)
The Rich OS	2.49
TDC	2.47
ICE	2.47

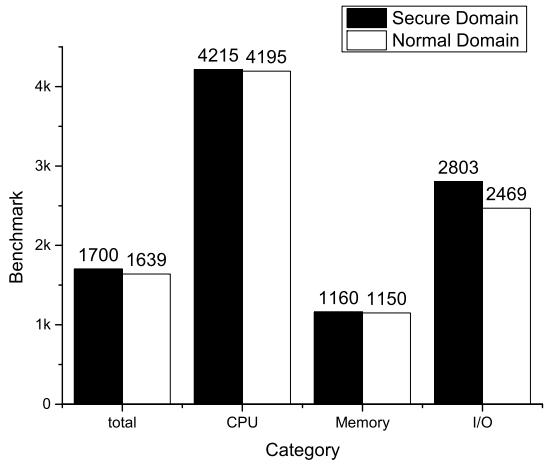

Fig. 11: Android Benchmark

C. Comparison with Other Solutions

Emulated hardware platforms with TrustZone support have been developed [26], [27]; however, those emulation frameworks do not contain many useful security features of TrustZone. Besides, the real platform devices are not very "friendly" [10]. This is because there is little open-source software system for TrustZone, and the security datasheet of hardware is not fully open to the public. This prevents users from utilizing the security features of TrustZone to develop comprehensive TrustZone-based software systems.

TrustZone has been proposed to be used in many ARM architectures [28], [29], [30]; however, there are no detailed performance evaluations available for public access. Some architectures have been proposed in commercial use [12], [20], but they are not open-source. The open-source $SierraTEE$ [19] must be loaded on the emulated Fast Models or on programmable Zynq-7000 AP SOC [31]. Since those experimental results are not collected on real platforms, it is difficult for us to compare our system to their solutions.

D. Performance Comparison between One and Multiple ICEs

We have different memory maps of TrustICE when we load only one ICE or multiple ICEs in the system. Since the memory maps have impacts on setting the watermark regions, we conduct experiments to study and compare their performance. Our experimental results show that the only difference is the switching time when the system switches from the Rich OS to ICE. In one-ICE and two-ICE scenarios, the switching time from the Rich OS to ICE is almost the same, since we only need to reconfigure the watermark registers in TDC. However, in the more-than-two-ICE scenario, the switching time is much larger. In our prototype with 2 encryption ICEs and one interface ICE, it takes 2.85 ms to copy the encryption ICE and 68.44 ms to copy the interface ICE to the runtime environment. This overhead can be reduced if the hardware platforms can provide a flexible watermark solution to protect a large number of memory regions for ICEs.

VII. Security Analysis

As we point out in the threat model, the Rich OS may be compromised by an adversary exploiting known or unknown OS vulnerabilities. Therefore, we must prevent the malicious OS from compromising the code and data in TDC and ICEs.

Data Exfiltration Attack. TDC is the only trusted computing base in our system, which is always protected by the TrustZone hardware security mechanism. We trust the TrustZone hardware design and implementation to protect TDC from being tampered by a malicious Rich OS. TDC guarantees that all ICEs are securely isolated from the Rich OS. When the system switches to ICE, the Rich OS has been suspended and cannot be resumed to run until the system exits ICE. When the Rich OS is running, all ICE's memory regions are protected by the watermark region that can only be configured by TDC. Therefore, even if an adversary can compromise the Rich OS, it still cannot access the code and data in ICEs.

Similarly, TDC guarantees that one ICE cannot access the memory content of another ICE. One ICE may contain malicious code targeting at stealing sensitive information from other ICEs. However, when one ICE is running in the normal domain, it cannot access other ICEs' memory spaces that are protected by the watermark region, which can only be configured by TDC in the secure domain. Due to the limited number and the size of watermark regions on i.MX53 QSB, we cannot prevent an ICE from accessing the resources in the Rich OS. However, it is a minor issue as our goal is to protect the secure code in ICEs.

Secure Boot. TrustICE ensures a trusted path when booting the system and switching between different computing environments. i.MX53 QSB includes a High Assurance Boot (HAB) to ensure the authenticity and integrity of an image that will be loaded on the processor chips [13], [22]. Through validating the signature of image code, HAB can assure that the code is originated from a trusted authority and the code is in its original form. Therefore, if the Rich OS tampers with the images of TDC, ICEs, or the secure code on the non-volatile storage, TrustICE can detect it and refuse to load them into memory.

Spoofing ICE Attack. When the Rich OS is requested to switch to an ICE, it may fake an ICE and deceive a user to input secret credentials such as password in the faked ICE. We use an exclusive hardware signal such as an LED that can only be controlled by TDC in the secure domain to prevent this kind of fake ICE attacks. When the system switches from the Rich OS to one ICE, it is not required to check the integrity of the ICE again. This is because the ICE memory is protected by watermark that can only be controlled by the trusted TDC.

Denial-of-Service Attack. A malicious Rich OS may launch denial-of-service (DoS) attacks to prevent the system from switching into one ICE; however, such attacks can be easily detected by a user. When the ICE cannot be entered, the LED light will not be turned on and the user can notice it immediately. The user then can launch further investigation or reinstall the system.

Side-Channel Attack. The Rich OS can hardly use side-channel to steal sensitive data from ICEs. Though the Rich OS and ICEs all run in the normal domain, when one ICE

is running, the Rich OS is suspended and cannot obtain the system status information. Before the Rich OS resumes execution, TDC has cleaned up the CPU context information. Therefore, it is difficult for the Rich OS to launch side-channel attacks.

Our design can reduce the attack surface in the secure domain and the size of the system's TCB; however, since we run the secure code in the normal domain, our system may not provide equivalent isolation that ARM TrustZone guarantees in the secure domain. ARM TrustZone can ensure isolation on CPU, memory, and I/O devices. We use the hardware-assisted watermark mechanism to successfully ensure the memory isolation between ICEs and the Rich OS. However, for the CPU and I/O devices, we have to rely on the careful design of TDC to clean up the CPU states and control the interrupts for the ICEs. Our design can be integrated with the traditional TrustZone architecture to provide a three-level trusted execution environments, where the secure domain is for the highest-level secure code, the ICE is for the medium-level secure code, and the Rich OS is for the normal code.

VIII. Related Work

A number of research efforts have been attempted to solve the data leaking problem, and they can be classified into two general categories: *access control policy based* solutions (e.g., [32], [33], [34], [35], [36], [37], [38]) and *isolated computing environment based* solutions (e.g., [10], [29], [9], [39], [40], [41], [8]). Current Android relies on Linux discretionary access control (DAC) to achieve application isolation [32]. Furthermore, mandatory access control (MAC) has been integrated into mobile OS kernel to achieve a stronger isolation [33], [34].

An isolated computing environment can be accommodated on mobile devices to protect the execution of secure code. For instance, the SIM card in our cellphones is actually a small computer, with its own memory and even an operating system to protect the credentials stored in the card. Square [7] is another example that processes the credit card sensitive data in an isolated chip that can be connected to the smart phone through the audio interface. ARM processors are extended with a new hardware security support called TrustZone [9] to help construct an ICE using the application processor. TrustZone can isolate a secure OS from a Rich OS into two isolated computing domains. Thus, untrusted applications in a compromised Rich OS cannot access secure applications in the secure OS [10], [9], [11], [12]. Texas Instruments (TI) developed its own TrustZone solution and named it M-Shield [8].

TrustZone has been adopted to secure a number of applications. For instance, a location-based second-factor authentication for mobile payments uses TrustZone to protect its secure enrollment schemes [42]. A TrustZone-based memory acquisition mechanism called TrustDump is capable of reliably obtaining the RAM memory and CPU registers of the mobile OS even if the Rich OS has crashed or has been compromised [43]. Santos et al. [44] took advantage of TrustZone to construct a Trusted Language Runtime (TLR). TLR protects the secure code of .NET mobile applications from the rest of the application, and isolates it from the OS and other apps. It also provides runtime support for the secure code.

Several TrustZone-based systems (e.g., Mobicore/Trustonics [12], Trusted Logic [11], ObCs [29], [40], and KNOX [45]) have been developed to enhance the security of mobile devices. MobiCore [12] is a secure OS for TrustZone enabled ARM controllers including ARM1176 or CortexA8/A9. It provides development tools called Trustlets for third-party application developers. For all existing TrustZone-based solutions, a customized secure OS runs in TrustZone's secure domain to execute secure applications. This type of TrustZone architecture has two major problems. First, the system attack surface increases along with the number of installed trusted applications. Second, it is difficult for third-party developers to get their code into the secure domain. Our solution can mitigate these problems by running the secure code in trusted ICEs in the normal domain.

Virtualization has been adopted to provide isolated virtual machines on mobile devices [46]. Also, there emerges hardware virtualization supports on ARM processors [47], [48], [4]. KVM/ARM [5] utilizes recent ARM hardware virtualization extensions to run virtual machines with comparable performance to native execution. Secure code can be protected in an isolated virtual machine that is protected by a trusted hypervisor. However, due to the size of the Linux kernel, the trusted computing base (TCB) of the hypervisor is still quite large and may contain unknown vulnerabilities that may be exploited to compromise the hypervisor and the virtual machines.

Researchers have investigated on how to use specific hardware supports to create an isolated execution environment from an untrusted operating system on x86 processors (e.g.,Inktag [49], Overshadow [50], TrustVisor [51], SICE [52], and SGX [53]). However, these hardware supports are only available on x86 architecture.

IX. Conclusions

Based on the ARM TrustZone technology, we design a novel TrustICE framework to create isolated computing environments for executing secure code in the normal domain. Contrast to traditional TrustZone solutions, TrustICE has a small TCB that only consists of a Boot ROM and a Trusted Domain Controller (TDC), which are protected by TrustZone. Our design allows application developers to better utilize the TrustZone technology without the arduous negotiation with the OEMs and service providers to get their code into TrustZone's secure domain. Our prototype on Freescale i.MX53 QSB can switch from the Rich OS to ICE in less than 12 ms. In the future, we will study how to develop more powerful system functions or a thin OS in ICE.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments and suggestions. Dr. Kun Sun's work is partially supported by U.S. Army Research Office under Grant W911NF-12-1-0060 and U.S. Office of Naval Research under Grant N00014-15-1-2026 and N00014-15-1-2012. He Sun, Yuewu Wang and Jiwu Jing are supported by National 973 Program of China under award No. 2014CB340603.

REFERENCES

[1] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pp. 95–109.

[2] L. Wu, M. C. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pp. 623–634.

[3] "CVE Details. Google: Android: Security Vulnerabilities," http://cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html, 2013.

[4] L. Rasmusson and D. Corcoran, "Performance overhead of KVM on linux 3.9 on ARM cortex-a15," *SIGBED Review*, vol. 11, no. 2, pp. 32–38, 2014.

[5] C. Dall and J. Nieh, "KVM/ARM: the design and implementation of the linux ARM hypervisor," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pp. 333–348.

[6] M. Reveilhac and M. Pasquet, "Promising secure element alternatives for nfc technology," in *Near Field Communication, 2009. NFC '09. First International Workshop on*, Feb 2009, pp. 75–80.

[7] "Square Security." https://squareup.com/security, accessed in Febuary 2014.

[8] J. Azema and G. Fayad., " M-Shield mobile security: Making wireless secure." *Texas Instruments WhitePaper*, 2008.

[9] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, 2004.

[10] J. Winter, "Experimenting with ARM trustzone - or: How I met friendly piece of trusted hardware," in *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, pp. 1161–1166.

[11] Trusted Logic, "Trusted Foundations by Trusted Logic Mobility," http://www.arm.com/community/partners/display_product/rw/ProductId/5393/.

[12] Giesecke & Devrient, "MobiCore," http://www.gi-de.com/en/trends_and_insights/mobicore/trusted-mobile-services.jsp.

[13] Freescale, "i.MX53 Reference Manual with fusemap addendum," http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX537&fpsp=1&tab=Documentation_Tab.

[14] ARM, "Cortex-A8 Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf.

[15] ARM, "Cortex-A9 Technical Reference Manual," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf.

[16] Freescale, "i.MX53 Processors," http://www.freescale.com/webapp/sps/site/taxonomy.jsp?code=IMX53_FAMILY.

[17] Texas Instruments , "Get Into the Zone: Building Secure Systems with ARM TrustZone Technology," http://www.ti.com/lit/wp/spry228/spry228.pdf.

[18] Samsung Electronics, "Samsung S5PC100, ARM Cortex A8 based Mobile Application Processor," http://www.samsung.com/global/business/semiconductor/file/media/s5pc100_brochure_200902-0.pdf.

[19] Sierraware, "Open Virtualization's SierraVisor and SierraTEE," http://www.openvirtualization.org.

[20] Trusted Logic, "TrustZone Software Porting Kits," http://www.trusted-logic.com/Presentations/Trusted_Logic_TrustZoneSoftwarePortingKits_ccolas_2007Sept13.pdf.

[21] HUAWEI, "HUAWEI MC323 CDMA M2M Module," https://techship.se/products/huawei-mc323/.

[22] Package, Plastic, "i. mx 6solo/6duallite applications processors for industrial products," 2012.

[23] "Das U-Boot," http://www.denx.de/wiki/U-Boot.

[24] Paul Bakker, "PolarSSL," https://polarssl.org/.

[25] Aurora Softworks, "Quadrant," https://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard.

[26] ARM, "Fast Models," http://www.arm.com/products/tools/models/fast-models/index.php.

[27] J. Winter, P. Wiegele, M. Pirker, and R. Tögl, "A flexible software development and emulation framework for ARM trustzone," in *Trusted Systems - Third International Conference, INTRUST 2011, Beijing, China, November 27-29, 2011, Revised Selected Papers*, pp. 1–15.

[28] W. H. W. Hussin, P. Coulton, and R. Edwards, "Mobile ticketing system employing trustzone technology," in *2005 International Conference on Mobile Business (ICMB 2005), 11-13 July 2005, Sydney, Australia*, pp. 651–654.

[29] K. Kostiainen, J. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia, March 10-12, 2009*, pp. 104–115.

[30] M. Pirker and D. Slamanig, "A framework for privacy-preserving mobile payment on security enhanced ARM trustzone platforms," in *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, 2012, pp. 1155–1160.

[31] Zynq-7000, "Zynq-7000 AP SOC," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[32] Android Open Source Project., "Android Security Overview," http://source.android.com/tech/security/.

[33] S. Smalley and R. Craig, "Security enhanced (SE) android: Bringing flexible MAC to android," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*.

[34] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*.

[35] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pp. 328–332.

[36] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*.

[37] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pp. 639–652.

[38] M. Ongtang, K. R. B. Butler, and P. D. McDaniel, "Porscha: policy oriented secure content handling in android," in *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pp. 221–230.

[39] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: What security properties can my mobile platform give me?" in *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*, pp. 159–178.

[40] J. Ekberg, K. Kostiainen, and N. Asokan, "Trusted execution environments on mobile devices," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pp. 1497–1498.

[41] J.-E. Ekberg *et al.*, "Mobile trusted module (mtm)–an introduction," 2007.

[42] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as practical and secure location verification tokens for payments," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*.

[43] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I*, 2014, pp. 202–218.

[44] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM trustzone to build a trusted language runtime for mobile applications," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, 2014, pp. 67–80.

[45] Samsung Electronics, "White Paper: An Overview of Samsung KNOX," http://www.samsung.com/global/business/business-images/resource/white-paper/2013/06/Samsung_KNOX_whitepaper_June-0.pdf.

[46] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pp. 173–187.

[47] P. Varanasi and G. Heiser, "Hardware-supported virtualization on ARM," in *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, p. 11.

[48] ARM, "ARM Virtualization Extensions," http://www.arm.com/products/processors/technologies/virtualization-extensions.php.

[49] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pp. 265–278.

[50] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. S. Dwoskin, and D. R. K. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pp. 2–13.

[51] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pp. 143–158.

[52] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pp. 375–388.

[53] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pp. 267–283.

APPENDIX A
AN EXAMPLE OF SECURE CODE

An example of secure code is shown in Listing 2. Line 7 shows the user-space SMC system call, with the first parameter showing the system call number 366 and the second parameter showing the length of the secure code as $0x18$ bytes. The third parameter indicates which ICE is used. Bit 0 tells TDC to switch the Rich OS to the encryption ICE while bit 1 corresponds to the interface ICE. Then, the secure code between Line 9 and Line 26 is executed in ICE. To call the functions provided by ICE, the secure code just makes the corresponding `SVC` calls and transfers the required parameters. To make the code self-contained, we use inline assembly instead of C functions to make system calls. From line 13 to line 21, the secure code calls the second system call in ICE to generate an RSA signature. The result is stored in `sig[128]`. From line 23 to line 26, the secure code calls the first system call to exit ICE. At last the Rich OS wakes up and the application outputs the string `"app ends"` in the Rich OS.

Listing 2: A typical application containing secure code

```
#include <iostream>
#include <unistd.h>
#include<sys/syscall.h>
using namespace std;
int main() {

    syscall(366, 0x18, 0);
  /*start of secure code*/
    unsigned int length=128;
    unsigned char start[128];
    unsigned char sig[128];

    asm volatile(
    "ldr r0, =0x2\n\t"
    "mov r1, %0\n\t"
    "mov r2, %1\n\t"
    "mov r3, %2\n\t"
    "svc 0x0\n\t"
    :
    :"r"(length),"r"(start),"r"(sig)
    );

    asm volatile(
    "ldr r0, =0x1\n\t
    "svc 0x0\n\t"
    );
 /*end of secure code*/
     cout<<"app ends"<<endl;
    return 0;
}
```