SuiT: Secure User Interface Based on TrustZone

Yang Cai*†‡, Yuewu Wang*†‡, Lingguang Lei*†, Quan Zhou*†, Jun Li§
*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
†Data Assurance and Communication Security Research Center, Chinese Academy of Sciences
‡School of Cyber Security, University of Chinese Academy of Sciences
§Zhongxing Telecommunication Equipment Corporation, China
{caiyang, wangyuewu, leilingguang, zhouquan}@iie.ac.cn, li.jun3@zte.com.cn

***Abstract*—In lots of security-aware scenarios, trusted user interface (TUI) is indispensable. For example, before signing a payment information, user needs to approve the information. Digital right management (DRM) related applications also need TUI supporting. Although current mobile platforms have provided TEE (Trusted Execution Environment) OS to support trusted applications running, introducing additional drivers into TEE OS is not very secure. The additional drivers may increase the code size of TEE OS and expand the attack surface. In this paper, we present a novel secure UI framework called SuiT based on ARM TrustZone hardware security extension. A secure UI driver and a shadow UI driver are implemented in the normal world. In the secure world, only additional switching code is introduced. When an application needs to interact with user in a trustworthy way, the shadow UI driver will take the place of original UI driver to complete the user interaction. During the UI driver switching process, a temporary trusted execution environment for secure UI driver is dynamically built by the switching code in the secure world. The trusted execution environment ensures that the secure UI driver is executed in a secure way and the potential attacks from rich OS can not tamper with the process of user interaction. We also implement a prototype of SuiT based on Android system and Freescale ARM processor with TrustZone extension. Experimental results demonstrate that SuiT can work well with negligible overhead.**

***Index Terms*—TCB(Trusted Computing Base), Trusted User Interface, TrustZone, Shadow Driver**

I. Introduction

User interaction is necessary in many security-aware mobile applications. Unfortunately, user interaction with mobile system is easily tampered by attackers [1]–[3]. In addition, users may leak their private information during insecure interactions. Thus it is necessary to establish a trusted path between users and the application services.

There have been lots of studies on trusted UI [4]–[6]. Most of previous works are based on enhance security of original system architecture. All these works assumed that the operating system is secure. However, the operating system itself is vulnerable because of its large code and high complexity. Once the operating system is compromised, the attacker has full access to the system and the protection provided by the system is gone.

The most immediate solution to this problem is to introduce a secure environment isolated from the operating system. TrustZone technology was proposed by ARM to achieve system-level security solutions. The processor with TrustZone extension isolates a trusted execution environment next to the normal operating system (Rich OS). Due to hardware isolation mechanisms, even malicious code with Rich OS kernel privilege can not disturb the code executed in TEE(Trusted Execution Environment). Rich OS and normal applications are deployed in the normal world, and the trusted applications run in the secure world. The trusted applications often run over a secure OS. Normal applications can access the security functions of the trusted applications in TEE with TEE drivers. Trusted user interfaces provide by secure OS may be called by trusted applications to interact with users in a trustworthy way. VeriUI [7] runs a Linux in TrustZone secure world to provide an attested login for users. Xianyi Zheng et al. [8] proposed a mobile payment framework TrustPAY on TrustZone security enhanced platform. TrustPay implements the Trusted Graphical User (TGUI) interface, which displays the payment interface on the same screen shared with Rich OS.

However, with the TUI(trusted user interface) drivers introduced in the secure world, the amount of code in secure world may increase, this will result in a larger TCB(Trusted Computing Base). The increase in TCB also brings security issues. Software vulnerabilities in the code loaded into the secure world, which is almost unavoidable, could invalidate the protection mechanism for the entire secure world. In fact, there have been many attacks against android TEE [9]–[11].

Our motivation is to build a trusted path between users and the application services with strict control over the size and complexity of TCB. In this paper, we presented a secure UI framework called SuiT to build trusted paths between users and security-aware applications. When the operating system is running, the SuiT code is stored in secure storage and not accessed by the operating system to avoid potential attacks. Whenever SuiT is activated, the operating system will be frozen and a temporary trusted execution environment is dynamically built by the switching code introduced in secure world. Then the SuiT code will be released into the temporary trusted execution environment to support the interaction with user in a secure way.

SuiT has three advantages: First, it do not need to add extra hardware, and reduces costs. Second, we can use the system's best hardware resources to improve efficiency. Finally, unlike previous solutions that run TUI in the secure world, SuiT works in the normal world, which greatly reduces system complexity and attack surface.

We make the following contributions in this paper:

This work was supported by the National Cryptography Development Fundunder Award(No. MMJJ20170215 and No.MMJJ20180222) .

978-1-5386-8088-9/19/$31.00 ©2019 IEEE

1) We design a secure UI framework called SuiT on TrustZone enabled platform and implement a SuiT prototype on Freescale i.MX53 QSB. It provides a trusted path between users and the application servers. Our solution maximizes the use of hardware resources and reduces the TCB.
2) Our secure UI is easy to call for applications. In order to preserve the original Android driver architecture, we deploy a shadow driver in the kernel. It is called as a normal driver for the upper layer and forwards the request to the secure UI.
3) We propose a trusted graphical user interface that displays information and gets user input on the same screen shared with Rich OS. We also implement a crypto module to verify the source of the displayed information and protect user privacy data. Both the secure UI and the crypto module run in an isolated environment that defends against attacks from the Rich OS.

The remainder of the paper is organized as follow. Section II introduces the background knowledge. Section III describes threat model, assumptions and SuiT architecture. The prototype implementation is described in detail in Section IV. Section V discusses the experimental results. We also describe the related work in Section VI. Finally, we conclude the paper in Section VII.

II. Background

A. Android User Interface Architecture

The Android user interface system involves the application framework layer, the runtime library, the hardware abstraction layer (HAL), and the hardware driver layer.

Android applications get display support in the HAL through the services provided by the framework layer. The HAL is at the user space and encapsulates all the operational interfaces to the framebuffer, and only some channels for accessing the hardware are reserved in the kernel driver. When an application needs to display the content to an LCD, it needs to apply for a graphics buffer through the gralloc module in the HAL, then map the graphics buffer to its own address space and write the content. The Android system introduces HAL as an intermediary to operate framebuffer, and the platform-related code can be implemented in the HAL to protect the interests of the mobile phone manufacturers. We know that the Apache License [12] followed by the Android source code does not need to publish the source code when publishing the products.

To make it easy for the applications to call our secure UI, we implement the access interfaces in HAL and kernel space.

B. ARM TrustZone

ARM Trustzone technology [13] is CPU level approach to security. It is supported by a wide range of mobile processors including ARM Cortex-A [14] and Cortex-M23 [15] and Cortex-M33 [16] . TrustZone provide hardware-level isolation between the secure world and the normal world. A Secure Monitor Call (SMC) is used to enter the monitor mode of secure world. The SMC instruction can only be executed in privileged mode of normal world.

TrustZone includes a TrustZone Address Space Controller (TZASC), which can be programmed to configure address space as secure or non-secure. The normal domain can not access a region that is configured as secure. TZASC may be accessed through WaterMark technique on Freescale i.MX53 QSB to protect memory access.

The secure world and the normal world have their own virtual MMUs to manage the mapping of physical addresses. In fact, each world has their own TTBR0, TTBR1 and TTBCR registers. A TrustZone-based processor has three sets of exception vectors: one for the normal world, one for the secure world, and one for the monitor mode. The base address of these three sets of interrupt vector tables can be configured at runtime by the VBAR (Vector Base Address Register) in the CP15 register.

III. System Design

In this section, We first explain the threat model and assumptions, then show the overall architecture of our system.

A. Threat Model And Assumptions

Threat Model: Our work aim to prevent software attacks. The Rich OS including device drivers in the normal world is not trusted. The attacker can execute malicious code in privileged mode to access the system's memory, registers, and peripherals. It is also possible for an adversary to hijack the touch screen interrupts to obtain relevant information. Malicious Rich OS could modify interrupted process state (e.g., the PC register) during exception handling and change the program execution's control flow.

Assumptions: We assume the platform supports TrustZone extension. We assume that the hardware and software resource isolation mechanism provided by TrustZone is trustworthy. We trust the code in the Boot ROM.

Goals: The goal of our work is to provide a secure interaction path between the users and the application services. To this end, we need a secure UI that can defend against software attacks from the Rich OS. Further, we need to provide a trusted execution environment for the secure UI that is isolated from the Rich OS. In order to reduce the size of the TCB, we cannot adopt the solution which deploy a secure UI directly in the secure world, but to build a trusted execution environment in the normal world. In order to protect the security of interactive information at runtime, when the data stream is passed through the Rich OS, we need to protect its confidentiality by an encryption algorithm.

B. System Overview

We design a secure UI framework called SuiT, which uses ARM TrustZone hardware security extension to provide hardware-level isolation. Figure.1 shows the SuiT architecture. The Rich OS is still installed and executed in the normal world. Environment switching code is introduced in the secure world. It is responsible for loading SuiT in the normal world, building a trusted execution environment for the SuiT, and achieving

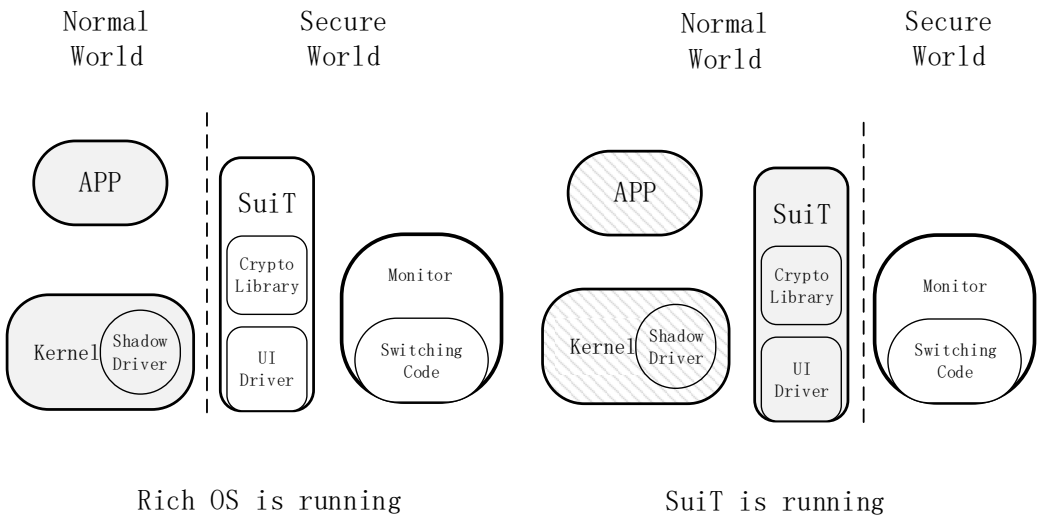

Fig. 1. Architecture

a secure switching between the normal world and the secure world. The SuiT running in the normal world is in the trusted execution environment isolated from the Rich OS.

In the normal world, We deploy a shadow driver as an entry to run SuiT. The shadow driver provides interfaces for the upper layer to access the secure driver. The specific implementation of the driver is in the SuiT code, but for the applications, SuiT is transparent. Applications access the shadow driver through a .so library to complete the secure display call.

We try to reduce the amount of code in the secure world. The monitor code only responsible for some necessary security configuration, and the size of the trusted computing base and the security risks in the security world are reduced.

IV. Implementation

We implement a SuiT prototype using Freescale i.MX53 QSB, which supports ARM TrustZone hardware security extensions. The board is equipped with MC34708 for PMIC. The touchscreen we use is MCIMX28LCD, a 4.3 800x480 (WVGA) display with 4-wire resistive touch screen.

A. Shadow Driver

We deploy the secure UI in a trusted execution environment, which brings two challenges for application call it. First, we need to provide a friendly interface for the application to call the secure UI. Second, the switching between Rich OS and the trusted execution environment needs to execute privileged instruction SMC that cannot be executed by application code.

To address these two challenges, we introduced a shadow driver in Rich OS kernel. The shadow driver defines a virtual device, and provides a way to access secure UI through traditional device files. Applications can operate virtual devices through Java Native Interfaces. A series of `ioctl()` functions are defined in shadow driver to response the UI requests from applications.

These functions are not implemented in the shadow driver. They just forward the requests from applications to the secure UI for execution. In this process, we need to pass the parameters of the function calls and some necessary information to the switching code in the security world. Some simple data can be passed through registers, but if the data is large, such as a picture, then the space of registers is not enough, we need to use a shared memory to pass the data.

We call the APIs provided by the Linux kernel, including `kmalloc()` and `get_free_pages()`, to apply for a contiguous segment of physical memory. The physical address of this memory differs from the virtual address by a fixed offset value. When the system state is in the secure world or in the trusted execution environment, the memory can also be accessed according to the physical address.

The other role of the shadow driver is to call the SMC instruction to switch the system to the secure world. In the secure world, the switching code in monitor mode complete the switching of the execution environment. Since the SMC instruction is a privileged instruction and cannot be directly called by the application in the user mode, we implement the interface of calling the SMC instruction in the shadow driver.

B. Trusted Execution Environment

We build a trusted execution environment for Secure code that coexists with the Rich OS in the normal world.

After the Secure UI code has been safely loaded into memory, it needs to be guaranteed that it will not be accessed by the other code in the normal world. With the WaterMark mechanism of Freescale i.MX53, we can set a contiguous amount of memory to secure memory, which can only be

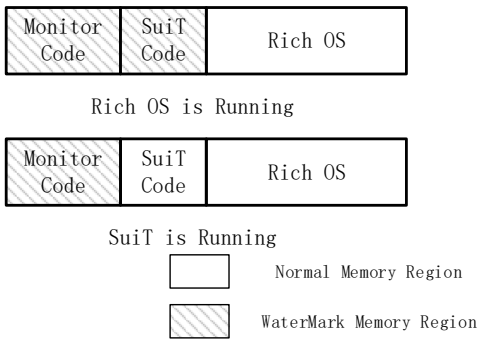

Fig. 2. The Memory Protected By WaterMark

accessed by the secure world. Figure.2 shows the memory area protected by the WaterMark mechanism.

In programming, we set the the WaterMark Start ADDR Register and the WaterMark End ADDR Register to determine the location and size of the WaterMark area. Since the WaterMark region must be contiguous, we load the switching and code into a contiguous piece of physical memory and set this segment of memory to the WaterMark region before the Rich OS boots up. In this way, the Rich OS can not access the code of secure UI at run time. When the secure UI code is running, we set the registers to dynamically adjust the WaterMark region, and the secure UI code will be released from the WaterMark memory area, then secure UI code can run in the normal world. After the secure UI code is running over, before waking up the Rich OS, we adjust the WaterMark configuration again to protect the secure UI in the WaterMark region. It is worth mentioning that the normal world can not modify the WaterMark control register.

In order to protect the executive process from being interrupted by the malicious Rich OS or software, we configure the interrupt vector table of the normal world. This process is performed by the switching code in the secure world. Monitor backs up the exception vector tables and related registers used by Rich OS. The start address of the exception vector table of the trusted execution environment is then written to the non-secure vector base register. Then, in the interrupt controller, only the interrupts needed to the secure code are enabled and mask the other interrupts.

In this way, the secure code executive process will not be interrupted by the Rich OS. By setting the interrupt controller, the execution of the SuiT is not interrupted by unwanted interrupts. The interrupt required by secure UI will be processed in the interrupt vector table of the trusted execution environment. After the execution of the secure UI is completed, the SMC system call is invoked again to return to the monitor mode of secure world. The switching code resumes the rich OS interrupt vector table and related registers and returns the system to Rich OS.

In order to prevent the operating system from disguising the services in the trusted execution environment, we turn on a LED light when entering the trusted execution environment. The LED are configured as a secure device, can not be manipulated by the Rich OS.

C. Secure UI

We implement secure UI in a trusted execution environment. Secure UI resets the image processing unit (IPU) in the secure domain and programs the IPU to display the contents in the secure frame buffer.

The IPU can provide a data channel that connects the memory and the LCD. The key to secure display is to configure the channel of the IPU so that the contents of the secure frame buffer are displayed on the screen. Android system located in the normal world adopts three buffers take turns to provide the display content, which called triple mode. It may be using any of the frame buffer when the system switches to the trusted execution environment to execute the secure code. In order to return to the Rich OS successfully, we must back up the current channel configuration information and buffer content first. Then we disable the IPU, make the required configuration for the IPU, populate the buffer with the content we want to display, and finally enable the IPU to show what we want.

Figure.3 shows the configuration of IPU registers. The configuration of each channel is stored in the corresponding CPMEM. The CPMEM can holds the settings of 80 channels. Each channel's settings are defined by two mega-words. Each mega-word is 160 bits wide. The size and start address of frame buffer are stored in CPMEM. Note that each channel corresponding CPMEM can only store the address of two buffers, but triple mode uses three frame buffer, and the address of the third buffer is stored in mega-word of other free CPMEM. The channel number for storing the third buffer is configured by a group of IDMAC Channel Alternate Address Registers. The information corresponding to channel 23 is stored in IPU_IDMAC_SUB_ADDR_1. This information is a numeric value (69) indicating the channel number in which to store the third buffer. Triple Current Buffer Registers determines which buffer is currently used, and the 14th and 15th bits of Triple Current Buffer Register 1 correspond to 23 channels.

Secure UI code only use one frame buffer. We determine which one to use in the initialization of the IPU and will not change it in the future. When system switch to the trusted execution environment, the Rich OS is suspended, the IPU is still in the pre-switching state, and the LCD stays in the pre-switching screen. We read Alternate Channel Triple Buffer Mode Select 0 Register to determine which buffer is used currently. Then we disable the IPU, fill the base address of the secure frame buffer into the appropriate register, and fill in the contents of the security frame buffer. Next, we check the corresponding configuration information of the IPU, and finally enable the IPU, and the LCD display is completed. This process will not be interrupted by Rich OS in the normal world, the contents of the buffer will not be modified by other malicious programs.

Secure code not only displays images, but also accepts user input. We have implement a secure touch screen driver

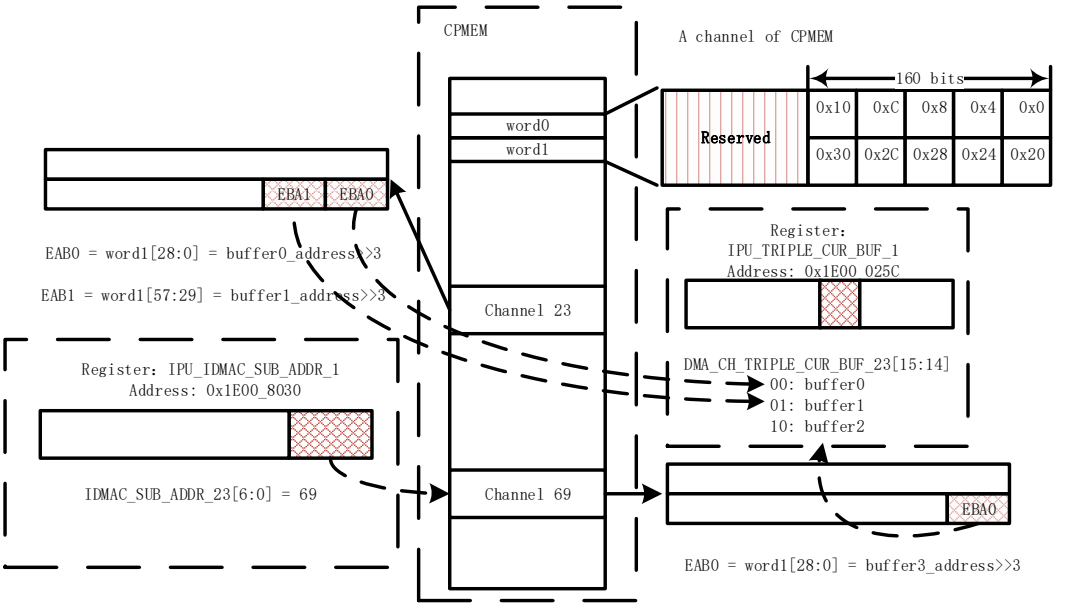

Fig. 3. Partial IPU Register Configuration

to ensure that input will not be hijacked or blocked by the operating system.

Our touch display screen is connectd to MC34708, which is the PMIC designed specifically for Freescale i.MX50 and i.MX53 families. When the finger touches the screen, the two conductive layers contact at the touch point, creating a voltage. The PMIC converts the voltage into digital information and stored in Analog to Digital Converter register. Touch behavior will produce a PMIC interrupt. In the interrupt handler, we call the touch screen driver, and read the analog to digital conversion register to get the touch point position. We can analyze user behavior based on user's touch point location. MC34780 provide I2C interface to access Analog to Digital Converter rigister, Correlation function prototype as shown in Listing 1.

```
1   //reg_addr: address of register
    //val: read the value of register
3   void MC34708_Read(unsigned int reg_addr,
        unsigned int* val)

5   //reg_addr: address of register
    //val: value written to register
7   void MC34708_Write(unsigned int reg_addr,
        unsigned int val)
```

Listing 1. PMIC Function

D. Cryptographic Module

Cryptographic module is necessary for SuiT. First, some information such as a payment messages need to be signed when they are displayed to user. Second, some messages are encrypted when passed to secure UI code to display such as short message verification code. Third, the integrity of secure UI code needs to be verified before loading it into trusted execution environment.

Thus, SuiT includes a cryptographic module in form of function library, which includes SHA1, AES and RSA algorithms. RSA and SHA1 are used to sign and verify the signature. AES is used to encryption and decryption message. The cryptographic module may be loaded together with secure UI code by switching code. Trusted execution environment may be used to assure that the cryptographic module is running in a trustworthy way.

E. Secure Boot

High Assurance Boot (HAB) of i.MX53 QSB is used to ensure a secure booting of the entire system by verifying the integrity of the secure bootloader and SuiT images before loading. Security booting uses a cryptographic algorithm to check the start up of each phase of the security state. The cryptographic algorithm used in this process is based on the public key signature algorithm. Only through the signature verification of each phase, can the code execute. As a result, each module can detect any attempt to maliciously tamper with static code on the Micro SD card before the program is loaded, ensuring a safe start up of the system. The code on the ROM runs first after the system powers on. It loads the secure bootloader from storage to memory. The code then uses the SHA-256 algorithm and the RSA algorithm to verify the authenticity and completeness of the bootloader image file. 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 bootloader in this work is based on Uboot. It is used to load monitor and SuiT image file into the WaterMark area using the Uboot mmc read command. The value of

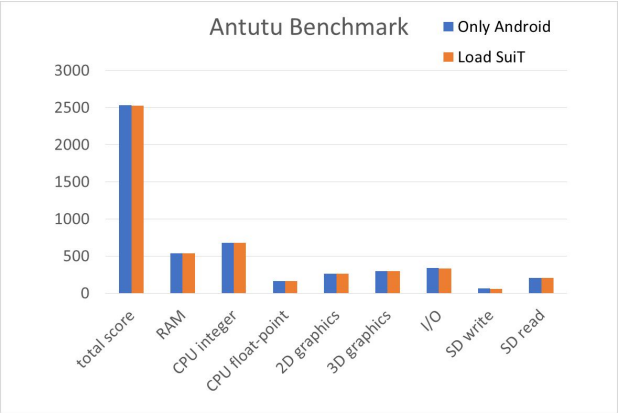

Fig. 4. Performance Evaluated In Antutu

SRK_HASH is used to verify monitors developing certificate. Monitor will be loaded only if the verification succeeds. The remaining stages also use a similar process to verify that a credible chain is formed.

V. Evaluation

In this section we will evaluate our work in three aspects:

i) The impacts of deploying SuiT on Rich OS;
ii) Demonstrate the performance evaluation of the SuiT;
iii) Analyze the security of our system.

A. Impacts on OS Performance

In order to evaluate SuiT's impacts on the Rich OS, we used Antutu to evaluate it, compared with the system on the platforms that do not have SuiT. The experiment result is shown in Figure.4, which represents a score and difference for one aspect of the two systems. The ordinate in the figure represents the score for each item of the system, where the first set of histograms represents the total score and the rest is one aspect of the system. The score of the system equipped with SuiT has a slight decrease. The results showed that SuiT had a very small impact on the system and this result was in line with our expectation. SuiT takes only a small amount of memory , the other resources can be completely used by the Rich OS.

B. SuiT Performance Evaluation

1) Execution Environment Switching Time: The system needs to switch from the Rich OS in the normal world to the secure world, execute the environment switching code, and then switch to the trusted execution environment to execute SuiT. After the end of a call, it also need to switch twice to return to the Rich OS. We measured the time of each switch. The experiment was repeated 30 times and averaged. The results are shown in Table I. In scenarios involving user interaction, such time consumption is negligible.

2) IPU Configuration Time Overhead: In our secure display driver, the configuration of the IPU is the main source of time overhead. The time consumed by this process was a fixed value. We performed a hundred tests and averaged it. The result was 2.2ms.

TABLE I
Switch Time

Switch	Time(μs)
Android to Secure World	0.38
Secure World to Android	0.47
Secure World to SuiT	0.43
SuiT to Secure World	0.45

C. Security Analysis

SuiT is deployed with the Rich OS in the normal world. The monitor running in the secure world builds a trusted execution environment for SuiT. Compared with the solution of deploying a trusted execution environment in the secure world, The framework in this article can integrate with the operating system better, simplify the security module, and reduce the trusted computing base.

Our security architecture can prevent software attacks from Rich OS. With our trusted execution environment, SuiT can completely occupy the resources of the normal world for a certain period of time. When SuiT needs to run in the normal world, we remove it from the WaterMark area. When the SuiT is running, the Rich OS is paused and cannot access the code and data in the trusted execution environment. We rebuild the interrupt vector table of the normal world for trusted execution environment to prevent SuiT's execution from being interrupted by a malicious operating system. Even if the Rich OS is destroyed, the adversary could not access the code and data in the trusted execution environment.

Sensitive information that servers interact with users needs to be passed through the Rich OS and applications in the normal world, and the adversary may destroy the data in the process. SuiT also implemented a crypto module to support the relevant security protocols. In this way, although we cannot guarantee that the data will not be tampered with during transmission, the data that is ultimately presented to the user or server must be correct.

The system's image file is stored in the SD card. Before the system starts up, the adversary may modify the image file. This type of attack can be avoided in this article. High Assurance Boot on the i.mx53 QSB ensures the authenticity and integrity of the image loaded into memory. After SuiT's image file is loaded into memory, access from the normal world is forbidden by the WaterMark mechanism.

Denial of service attacks may also be used by malware, such as hijacking and discarding calls to SuiT. Our system can not defend against denial of service attacks, but it is very noticeable to users.

VI. Related Work

Protect the operating system with TrustZone. TrustZone hardware security isolation can effectively protect the operating system security [17]. TZ-RKP [18] is a novel system that provides real-time protection of the OS kernel using the ARM TrustZone secure world. XU Yan-ling et al. [19] proposes a Trustzone-based secure enhancement framework for embedded system. The framework consists of a multi-policy access control mechanism and a secure reinforcement

method. Sun et al. [20] perform reliable memory dump of the Rich OS in the secure domain.TrustZone can ensure the TrustDumper is securely isolated from the Rich OS, so that a compromised Rich OS cannot compromise the memory acquisition module.

Build trusted execution environment with Trustzone. N Santos et al. [21] implement the Trusted Language Runtime (TLR), a system that protects the confidentiality and integrity of .NET mobile applications from OS security breaches. TLR enables separating an applications security-sensitive logic from the rest of the application and provides runtime support. VeriUI [7] and TrustUI [22] utilizes TrustZone to provide a responsive UI in the secure world. T2Droid [23] does dynamic (runtime) analysis of applications to detect malware on Android-based mobile devices. To prevent malware attacks, it is performed inside the secure world. Virtualization has been adopted to provide isolated virtual machines on mobile devices. Marforio et al. [24] propose a novel location-based second-factor authentication solution for modern smartphones. The solution can be effectively used for the detection of fraudulent transactions.

VII. Conclusions

In this paper, we presented a secure UI framework called SuiT based on the ARM TrustZone technology. SuiT has implemented a self-contained display touch screen driver to provide users with secure input and output. A switch code running in the secure world of TrustZone can freeze the Rich OS and built a dynamic trusted execution environment for secure driver to assure that the interaction between user and mobile system is secure. Different from existing TUI solutions, SuiT runs in the normal world and dose not increase the code size of the secure world. Thus, it does not increase the size of the trusted computing base while enhancing security. Experimental results show that SuiT can work well with negligible overhead.

References

[1] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks," in *Usenix Security Symposium*, 2014.

[2] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," *In Proceedings of 5TH International Symposium on Foundations & Practice of Security (FPS 2012)*, 2012.

[3] H. Shahriar, T. Klintic, and V. Clincy, "Mobile phishing attacks and mitigation techniques," *Journal of Information Security*, vol. 06, no. 3, pp. 206–212, 2015.

[4] Liu, Dongtao, Cuervo, Eduardo, Pistol, Valentin, Scudellari, Ryan, and P. Landon, "Screenpass: secure password entry on touchscreen devices," *Acm Mobisys*, pp. 291–304, 2016.

[5] M. Jakobsson and H. Siadati, "Spoofkiller: You can teach people how to pay, but not how to pay attention," in *The Workshop on Socio-Technical Aspects in Security & Trust*, 2012, pp. 3–10.

[6] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar:piecing together android app guis from memory images," in *ACM Sigsac Conference on Computer and Communications Security*, 2015, pp. 120–132.

[7] D. Liu and L. P. Cox, "Veriui:attested login for mobile devices," in *The Workshop on Mobile Computing Systems & Applications*, 2014, pp. 1–6.

[8] X. Zheng, L. Yang, J. Ma, G. Shi, and D. Meng, "Trustpay: Trusted mobile payment on security enhanced arm trustzone platforms," in *Computers and Communication*, 2016, pp. 456–462.

[9] N. Keltner, "Here be dragons: Vulnerabilities in trustzone," https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html, 2014.

[10] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade attack on trustzone," https://arxiv.org/pdf/1707.05082.

[11] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environment," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[12] "The apache software foundation," https://www.apache.org/licenses/LICENSE-2.0.html.

[13] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[14] ARM, "Processors cortex-a series," https://www.arm.com/products/processors/cortex-a.

[15] "Arm cortex-m23," https://developer.arm.com/products/processors/cortex-m/cortex-m23.

[16] "Arm cortex-m33," https://developer.arm.com/products/processors/cortex-m/cortex-m33.

[17] J. Winter, "Experimenting with arm trustzone – or: How i met friendly piece of trusted hardware," in *IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2012, pp. 1161–1166.

[18] A. M. Azab, J. Shah, J. Shah, J. Ma, J. Ma, J. Ma, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM Sigsac Conference on Computer and Communications Security*, 2014, pp. 90–102.

[19] Y. Xu, W. Pan, and X. Zhang, "Design and implementation of secure embedded systems based on trustzone," in *International Conference on Embedded Software and Systems*, 2008, pp. 136–141.

[20] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, *TrustDump: Reliable Memory Acquisition on Smartphones*. Springer International Publishing, 2014.

[21] N. Santos, H. Raj, S. Saroiu, and A. Wolman, *Using ARM trustzone to build a trusted language runtime for mobile applications*. ACM, 2014.

[22] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C. K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Asia-Pacific Workshop on Systems*, 2014, pp. 1–7.

[23] S. Demesie, G. Q. M. Jr, S. Haridi, and M. Correia, "T2droid: A trustzone-based dynamic analyser for android applications," in *The IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2017.

[24] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. ?apkun, "Smartphones as practical and secure location verification tokens for payments," in *Network and Distributed System Security Symposium*, 2014.