Copyright © 2018 Tech Science Press CMC, vol.55, no.1, pp.17-35, 2018

Securing Display Path for Security-Sensitive Applications on Mobile Devices

Jinhua Cui[1, 2], Yuanyuan Zhang[3], Zhiping Cai[1, *], Anfeng Liu[4] and Yangyang Li[5]

Abstract: While smart devices based on ARM processor bring us a lot of convenience, they also become an attractive target of cyber-attacks. The threat is exaggerated as commodity OSes usually have a large code base and suffer from various software vulnerabilities. Nowadays, adversaries prefer to steal sensitive data by leaking the content of display output by a security-sensitive application. A promising solution is to exploit the hardware visualization extensions provided by modern ARM processors to construct a secure display path between the applications and the display device. In this work, we present a scheme named SecDisplay for trusted display service, it protects sensitive data displayed from being stolen or tampered surreptitiously by a compromised OS. The TCB of SecDisplay mainly consists of a tiny hypervisor and a super light-weight *rendering painter*, and has only ~1400 lines of code. We implemented a prototype of SecDisplay and evaluated its performance overhead. The results show that SecDisplay only incurs an average drop of 3.4%.

Keywords: Mobile device, secure display, virtualization, trusted computing base, display path, trust anchor.

1 Introduction

Smart devices with ARM processors are now widely used in our daily life. For instance, we can use smart phones not only for calling a friend, but also browse websites, take photos and buy products. While smart devices bring us a lot of convenience, they also become an attractive target of cyber-attacks. Thus, the security of smart devices is one of the biggest concerns of users.

The threat to security is exaggerated as commodity operating systems (OSes) support rich functionalities. These Rich OSes usually have a large code base with complicated logic and thus suffer various software vulnerabilities. As an OS has higher privilege level than user-level applications, the security-sensitive data can be readily leaked once the OS is compromised. To protect the sensitive data from being leaked or tampered, various

[1] College of Computer, National University of Defense Technology, Changsha 410073, China.

[2] SMU Labs, Singapore Management University, Singapore 178895, Singapore.

[3] College of Mathematics and Informatics, Fujian Normal University, Fuzhou 350117, China.

[4] School of Information Science and Engineering, Central South University, Changsha 410083, China.

[5] Innovation Center, China Academy of Electronics and Information Technology, Beijing 100041, China.

[*] Corresponding author: Zhiping Cai. Email: zpcai@nudt.edu.cn.

CMC. doi: 10.3970/cmc.2018.055.017 www.techscience.com/cmc

schemes [McCune, Li, Qu et al. (2010); Yu, Gligor and Zhou (2015); Sun, Sun, Wang et al. (2015); Wang, Chen, Wang et al. (2015); Danisevskis, Peter, Nord-holz et al. (2015); Cheng, Ding and Deng (2013); Cheng and Ding (2013); Chen, Garfinkel, Lewis et al. (2008); Cho, Shin, Kwon et al. (2016); Azab, Ning, Shah et al. (2014)] have been proposed during the past years. Among them, the schemes like Fides [Strackx and Piessens (2012)] and Flicker [McCune, Parno, Perrig et al. (2008)] utilize a more privileged kernel to counteract the illegal behaves targeting userland applications, but these mechanisms can unimpededly be disabled by rootkits residing in kernel space at runtime. To this end, schemes based on ARM TrustZone [Sun, Sun, Wang et al. (2015); Alves (2004); Winter (2012); Logic (2012); Azab, Ning, Shah et al. (2014); Tian, Wang, Liu et al. (2017); Guan, Liu, Xing et al. (2017)] and hardware virtualization [McCune, Li, Qu et al. (2010); Yu, Gligor and Zhou (2015); Wang, Chen, Wang et al. (2015); Cheng, Ding and Deng (2013); Cheng and Ding (2013); Cho, Shin, Kwon et al. (2016); Eppler and Wang (2018)] are proposed.

TrustZone is designed as a hardware security extension in ARM processors [ARM (2010)], it has already been adopted by most trusted execution environment (TEE) solutions (e.g. MobiCore (Trustonics) [Logic (2012)], Sierra-TEE [Sierraware (2013)]). TrustZone can build a secure world separated from other software layers including the hypervisor and Rich OS in the normal world, and can configure a secure physical memory space which only can be accessed by the secure world. Therefore, a system rooted on TrustZone surely can provide security guarantees on protecting security-sensitive applications (SecApps). However, the devices providers rarely publish their source code placed in TrustZone, thus make security community difficult to do a good examination. Moreover, the trusted computing base (TCB) of secure world would increment along with the number of kernel modules such as char driver and display driver installed in the OS. A bloated TCB may revoke its reliability in security. Furthermore, for third-party software developers, it may be an arduous procedure for negotiating with OEMs and service providers to place their code into the secure world.

Recent ARM processors like ARMv7-A and ARMv8-A extend their architectures to support virtualization, with which users can efficiently implement a lightweight hypervisor. The immediate benefit of hardware-assisted virtualization is that the hardware resources of a platform can be separated into two isolated domains, and the domain with higher privilege can monitor the activities of the other. Therefore, virtualization has a good availability and becomes a popular choice for a platform to fortify its kernel or application security [Wang, Chen, Wang et al. (2015); Chen, Garfinkel, Lewis et al. (2008); Jiang and Wang (2007); Litty, Lagar Cavilla and Lie (2008); Cho, Shin, Kwon et al. (2016); Azab, Ning, Shah et al. (2014)]. Taking Trusted Display [Yu, Gligor and Zhou (2015)] as an example, it relies on the underlying micro-hypervisor to mediate accesses to sensitive GPU objects by the Rich OS/Apps and emulates these accesses to prevent against arbitrary modifications.

Nowadays, cyber-attacks targeting smart devices start to steal sensitive data by leaking the display content of touchscreen. For instance, screenshot taking attacks [Lin, Li, Zhou et al. (2014)] try to obtain the content of display output stored in the frame buffer, on a purpose to get SecApp’s security-sensitive output. Moreover, the phishing attacks [Chen, Qian and Mao (2014); Bianchi, Corbetta, Invernizzi et al. (2015)] present a dialog on the screen analogous to the user's SecApp to trick the user into leaking security-sensitive information such as login credentials.

This work mitigates such an attack by providing a trust display service. Because of the availability, we exploit the hardware virtualization extensions provided by modern ARM processor to build a trusted world for the service. Specifically, SecDisplay relies on a tiny hypervisor to create a "secure world" separated from the untrusted OS. SecDisplay successfully manage a minimizing TCB by implementing a tiny hypervisor with a super light-weight rendering painter. The rendering painter utilizes the character-image to directly be rendered (~1400 SLoC), avoiding the need of implementing a full-featured char drivers in hypervisor. SecDisplay guarantees that the display content containing the sensitive data is securely protected from being read or modified stealthily by malicious OS.

We implemented a prototype of SecDisplay on Odroid-XU4 QSB equipped with 8 CPU cores, and developed a high-level particular application named SecEditor on Android OS to demonstrate the usability and reliability of SecDisplay.

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

(1) We present a new scheme named SecDisplay to protect the display content from being read or tampered by an untrusted OS running on ARM platform. This scheme exploits ARM Hardware Virtualization extensions to build a "secure world" isolated from the OS kernel, and create a communication channel from SecApp to the display device that is only accessible by the secure world.

(2) We implement a prototype of SecDisplay on Odroid-XU4 QSB with multi-processor architecture. The OS is a customized Linux 3.10.9 and Android 4.4.4. The experimental results show that our system only incurs 3.4% performance overhead.

The remainder of the paper is organized as follows. Section 2 gives a background of ARM Hardware Virtualization, the two stages of memory address translation and the flow of input and output in the context of a display device. Section 3 describes the threat model and assumptions. Section 4 presents the design of our SecDisplay system. We elaborate the prototype implementation in Section 5 and evaluate it in Section 6. The related work is described in Section 7. Finally, we summarize the paper in Section 8.

2 Background

2.1 ARM hardware virtualization overview

Similar to x86 architecture, ARM virtualization extensions enable the efficient implementation of the hypervisor for ARM compliant processors to the latest ARMv7-A and ARMv8-A architectures. Instead of introducing an orthogonal feature to distinguish between the hypervisor and VM operation, ARM extended the existing CPU mode hierarchy, originally just PL0 user mode and PL1 kernel mode, by adding a separate more privileged mode called PL2 (also known as Hyp mode) to run the hypervisor. These PLs have independent memory address spaces and different privileges.

Hyp mode has the responsibility of performing trap-and-emulation operations to support virtualization in the normal world. It holds its own banked registers, as well as additional registers, such as SP, SPSR, and ELR, in which most of critical feature of hardware-assistant CPU virtualization is executed. Using this register set, the hypervisor software running in Hyp mode can configure hardware to trap into Hyp mode on several sensitive instructions and hardware interrupts.

2.2 Stage-2 translation for memory access controlling

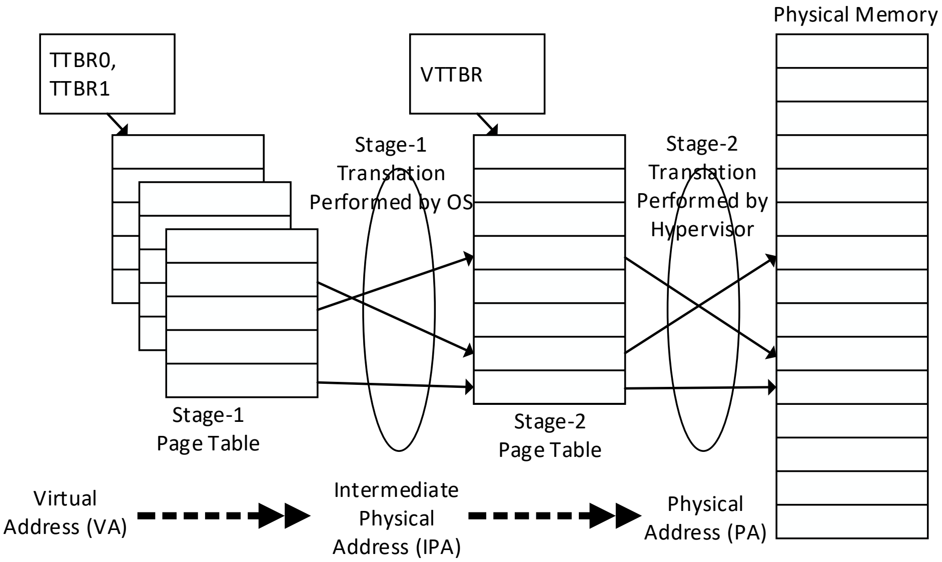

Figure 1: Two stage address translation

In ARM virtualization, ARM provides memory virtualization by adding an extra level translation, Stage-2 translation. With Stage-2 translation enabled, ARM defines three address spaces: Virtual Addresses (VAs), Intermediate Physical Addresses (IPAs), and Physical Addresses (PAs). IPAs are a continuous physical memory space in guest OS's view. Fig. 1 depicts the two-level address translation. VAs in a guest OS are translated to IPAs through the Stage-1 page tables managed by guest OS kernel just like non-virtualized systems. IPAs are further translated to PAs via the Stage-2 page tables maintained by the hypervisor. Each CPU core has two TTBR_0/1 (Translation Table Base Register) and one VTTBR (Virtual Translation Table Base Register), pointing to the Stage-1 and Stage-2 page tables, respectively. While the hypervisor uses a single translation that converts VAs to PAs directly based on another Stage-1 page table for PL2 itself.

The Stage-2 Translation can only be enabled and disabled in Hyp mode, and the hypervisor can flexibly configure which physical memory page needs to be protected through setting the appropriate access permission on the Stage-2 page table entry. Thus, any illegal accesses to protected memory will trigger page faults and be trapped into Hyp mode to handle.

2.3 The input and output for the display devices

Smart devices typically have a touch screen and several functional buttons. The screen is driven by a display controller. It scans an assigned region of memory, interprets the content as a map of color values and feeds them to the screen.

More specifically, when an input event is issued by the human user, the touch screen will actively trigger a hardware interrupt to CPU and delivery the obtained coordinate information to the input buffer mapped into I/O space, then the input driver will further be responsible for parsing and handling the event to response the clicked application. For the output event, in general, the display controller generates a *VSYNC* interrupt on the start of the vertical sync gap to coordinate the system's rendering activities. The display controller driver, which owns the device and receives that interrupt, forwards it to the frame buffer switch which, in turn, passes it onto the active client. After that, the data to be displayed on behalf of the application will be computed and composited by GPU and the Hardware Compositor, respectively. Accordingly, the frame buffers are populated with the blending final pixels, which are transmitted to display device in a DMA channel or other much faster channel by updating the display controller registers.

Moreover, the display controller driver provides an abstraction of the screen. Thereby it partitions the screen into several logical regions, the label region (e.g. cursor, caption or menu) and the client region (e.g. OS window). Using the display controller's support for multiple scan-out regions or overlays, each of the region may be backed by different frame buffers. The driver offers a service to attach arbitrary buffers to the logical screen regions and to retrieve information about the region's geometry and pixel layout.

3 Threat model and assumptions

We require that the smart devices where the SecDisplay is deployed support the Hardware Virtualization Extension, and that their hardware behaves correctly. We trust the code in the Boot ROM where the trust chain is started in the secure world, and the former boot code will always verify the integrity of the latter one. The hypervisor therefore is securely booted and trusted at runtime. An adversary is able to exploit software vulnerabilities to compromise the Rich OS and then obtain the sensitive display content.

We consider that an adversary can leak a SecApp's security-sensitive output through screenshot taking attacks [Lin, Li, Zhou et al. (2014)] whereby the content of display output in the frame buffer is read by a malicious program of a compromised Rich OS during running SecApp. Besides, the adversary can manipulate the display engine's data paths and overlay a new frame buffer over a SecApp's display thereby breaking the integrity of SecApps' display output without touching its contents. In addition, the phishing attacks [Chen, Qian and Mao (2014); Bianchi, Corbetta, Invernizzi et al. (2015)] that present a dialog on the screen analogous to the user's SecApp may trick the user into giving away security-sensitive information such as login credentials. The adversary also could try to eavesdrop on the user input and/or output in the process of transmitting the sensitive information [Xu, Bai and Zhu (2012); Miluzzo, Varshavsky, Balakrishnan et al. (2012)].

In this paper, we assume that the attacker cannot access physical devices or launch local physical attacks, such as removing the MicroSD card. We do not consider side-channels, device peer-to-peer communication and shoulder-surfing attacks [Hoanca and Mock (2005)]. We ignore I/O channel isolation attacks, which have already been addressed in prior study [Zhou, Gligor, Newsome et al. (2012); Jiang and Wang (2007)]. We also omit denial of service (DoS) attacks. For example, an adversary might manipulate the display controller or GPU configurations to disable screen output. However, for a well elaborated SecApp (e.g. SecEditor) it would be difficult to launch a DoS attack that would remain unnoticed by an observant user. Data-only attacks [Hu, Chua, Adrian et al. (2015); Hu, Shinde, Adrian et al. (2016); Davi, Gens, Liebchen et al. (2017)] that modify the data objects are outside the scope of this paper. Other aspects of security requirement [Liu and Li (2018); Tang, Liu, Zhang et al. (2018); Huang, Liu, Zhang et al. (2018); Li, Cai and Xu (2018); Zhang, Cai, Liu et al. (2018); Sun, Cai, Li et al. (2018); Xia, Cai and Xu (2018)] also have no consideration due to weak relevance.

4 System design

Fig. 2 shows the SecDisplay architecture. The tiny hypervisor as the trust display anchor (TDA) is running under the Rich OS, which has a higher privilege than kernel mode and only contain one component: trusted rendering painter. The hardware display device is used to render the data entered by the human user via touchscreen. The owning lowest privileged particular applications are residing on user mode, which are responsible for interacting with the underlying TDA to activate SecDisplay.

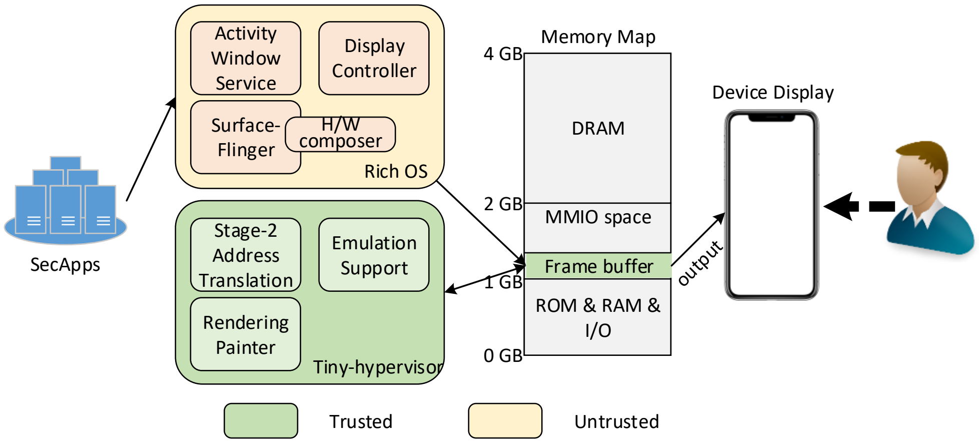

Figure 2: The architecture of SecDisplay. The tiny hypervisor as the TDA to guarantee the security of frame buffer composed by the Rich OS

4.1 Flushing frame buffer

As the repainting operation is performed in the hypervisor space and we do not implement the related display driver in it to reduce the size of TCB, there exists a problem about how to make the data in the new protected frame buffer display on screen quickly. The most straightforward approach is that the trusted rendering painter residing in the hypervisor helps directly repaint the currently used or next frame buffer to be display with the prepared character-image data according to the coordinate stored in the input buffer. Nevertheless, during the SecDisplay working, the input buffer is always locked, which causes the entire touchscreen fail to respond to any request from the Rich OS except for the TDA. Therefore, no any updating operation actively transfers the content of the new frame buffer onto display device. Based on our observation for another Raspberry Pi2 board, the display subsystem is not quite complex than Odroid-XU4, on where the frame buffer update is invoked by display device through forwarding a hardware interrupt to CPU that only periodically fetch the pixel data from frame buffer onto the display device to display. Thus, the former board can save more time-consuming operations and power and display more smoothly but involving in great complexity while the latter one is most likely to occur screen tearing phenomenon but always presenting the new data in frame buffer onto screen. Theoretically, we should implement a complete display driver in the hypervisor space to timely transfer frame data to the display device but this will dramatically increase the size of TCB. Instead of designing a complicated display driver that has the potential to introduce new vulnerabilities, we made several trials below.

Firstly, according to the display controller specification of the particular board, we changed the mappings in SysMMU_DISP1 Page Table that mapped to the physical frame buffer for OS window to the previously allocated new frame buffer address, or created new mappings for SysMMU_DISP1 Page Table, and then observed that the content of the new frame buffer wouldn't be transferred to the display device to display normally after the start and end addresses of display controller registers was exactly configured. While non-cacheable memory attribute for the new frame buffer and the relevant TLB flushing operations have been enforced, the display content repainted by the TDA still did not timely occur on screen until the time updating event arrives.

Secondly, as ARM only supports tracking memory at the 4 KB or even larger granularity, thus, the data structures related to the display controller registers are exactly mapped into the same physical page in current setting, which will involve in extra traps into the hypervisor once happened any access to the arbitrary address within the specific locked page. Moreover, the number of display controller registers is a little more (~18), thus, if we only enable Stage-2 translation lock the physical page that contains the related registers of display controller, and then help emulate these traps according to the syndrome information stored in different syndrome registers (e.g. HSR, HPFAR and HDFAR) and repaint the frame buffer to be displayed in the hypervisor when updating the controller registers by the display driver, the overhead of iterative context switching between SVC and HYP mode will incur much more performance loss.

Therefore, in order to possibly not introduce too much performance overhead or not implement a new display driver in the hypervisor, we make a trade-off between performance and generality for flushing frame data to display device, where we chose to perform the frame buffer updating by invoking the userland android element invalidation interface function from the SecApp. Specifically, we create a dedicated thread to call the invalidation function in polling way, when the TDA is initialized completely, the update status will be set through an installed system call module. At this moment, the TDA does not touch any data in current frame buffer until the touchscreen events are triggered by the human user, where arbitrary access to the protected input buffer from the Rich OS will cause the page fault which is directly delivered into the TDA in HYP mode by hardware. At the same time when the TDA detected the update status, it will immediately repaint the frame buffer allocated for OS window with the character image based on the parsed coordinate in the input buffer.

Because only repainting one frame buffer will occur flickering phenomenon without the particular character-image data in the other two frame buffers, our implementation will firstly repaint the shadowing frame buffer and then copy them to all three frame buffers for OS window when invoked update event in polling thread. While the content in frame buffers may be flushed away due to from the external update events, as the touchscreen is always locked during its working and our SecApp is set to full screen mode, the repainted frame buffers will not be polluted or overwritten.

The frame buffer to be flushed out onto screen that contains the sensitive data can be read or tampered stealthily by the Rich OS, so we exploit Stage-2 translation to constrain such malicious access. Furthermore, considering that GPU also access the frame buffer through SysMMU_DISP1 Page Table. Theoretically, we can make the TDA verify the integrity of the SysMMU_DISP1 Page table to counteract this attack from GPU side. However, since the page table for display device is frequently modified to create the new frame buffer or release the old one, we cannot simply shield it in current setting.

4.2 Quick rendering

Although GPU is quite efficient at accelerating the creation of images in a frame buffer intended for output to a display device, it still needs to occupy a certain number of time slices to render, composite and copy to the frame buffer. To this end, we elaborately devise a series of character images for output and a dedicated soft keyboard for input. Each piece of character image is derived from their complete screenshots through dumping the data of the specific character area from the frame buffer into file. Similar to the hard keyboard on PC or mobile phone, the particular keyboard contains a set of common English characters except the special ones.

With previously prepared input and output resources, it is unnecessary for the TDA to implement a complete functional display driver in the hypervisor space that will dramatically increase the size of TCB. As a result, the trusted rendering painter only need to copy the according character-image data loaded into memory to the current frame buffer when the human user clicks a character on the dedicated soft keyboard. Accordingly, such an operation will enable the sensitive data in frame buffer be quickly rendered onto the display device. Considering that these character images used for output may be tampered by the Rich OS, we put them into the specific memory blocks that are protected against malicious OS access through the Stage-2 memory translation. For the simplicity of implementation, we load these character images with hypervisor image together into the memory region allocated to the hypervisor space during SecDisplay's initializing phase, wherein the integrity is also verified to strictly ensure their security.

4.3 TDA: Trust Display Anchor

Instead of implementing a complicated display driver in the hypervisor, we utilize the TDA as the function module to achieve a series of goals of protecting those elaborated character images, locking the input buffer allocated for touchscreen, handling the page faults and repainting the frame buffer, etc.

4.3.1 Locking the input buffer

The input buffer register as a unit of touch screen controller that is mapped into MMIO space is charge of storing or collecting the coordinate information from touchscreen sensor. When the human user presses down on the touchscreen, the sensor processor unit will forward the computed coordinate into the mapped input buffer which will further be read by OS kernel to handle the touchscreen events. In order to obtain the coordinate prior to the Rich OS, the TDA sets the physical page containing the input buffer to non-accessible through the Stage-2 translation to forbid any malicious access during SecApp's working. The DMA-based attacks can be prevented by verifying the integrity of the SMMU page tables but, in practical, the frequent modifications to those page tables by OS may make it incompetent.

4.3.2 Handling page faults

After finished locking the input buffer, any read or write operations will trigger the page faults that will be forwarded into the handler of offset 0×14 of the exception base address of the hypervisor vector table to further handle. In exception handler, we parse the coordinate information fetched from the input buffer into *x*, *y* values. According to the coordinate we determine the position of the character clicked by the human user on the particular soft keyboard displayed on screen based on the previously computed range of *x* and *y* values, as shown in Tab. 1. Once obtained the knowledge of the position of the clicked character, we will accordingly located the in-memory character image to repaint the frame buffer. After handling the exception, the TDA will perform the ERET instruction to switch back to SVC mode to continue executing the following instructions. With the design of ARM processor pipeline, we have to add the offset of 4 bytes to ELR_hyp register to avoid trapping into the hypervisor again.

Table 1: A small protion of coordinate range based on the elaborated soft keyboard

Characters	Coordinate Range
	Upper-left to Lower-right
0	(1194, 446)~(1282, 488)
1	(24, 446)~(112, 488)
2	(154, 446)~(242, 488)
a	(93, 609)~(185, 655)
b	(724, 685)~(814, 735)
c	(469, 685)~(563, 735)
done	(24, 685)~(182, 735)
delete	(1103, 685)~(1248, 735)

4.3.3 Repainting for frame buffer

In our current design of SecEditor application, a right-size bar located at the top of screen is used to display the sensitive data, accordingly, and the mapped areas in the frame buffer are protected from any malicious access. The trusted rendering painter as a part of TDA is charge of repainting the bar with the clicked character images. Benefiting from our previously elaborated 38 pieces of images, we only directly copy the corresponding image data into the frame buffer instead of spending much more time computing and compositing the final pixels by GPU and hardware compositor, respectively. However, this also introduces the issue of generality that the right-size character images for different screen resolution need to be redesigned. Comparing to the great enhancement to the performance and security, we chose to match the new device through providing a simple versatile tool to produce the matchable images.

5 Implementation

We implement a SecDisplay prototype using Odroid-XU4 quick start board (QSB). Odroid-XU4 is equipped with four big cores (ARM Cortex-A15 up to 2.0 GHz) and four small cores (ARM Cortex-A7 up to 1.4 GHz) with 2 GB LPDDR3 RAM, and supports boot from an eMMC5.0 HS400 Flash Storage or a MicroSD card. The touchscreen we use is Odroid-VU HDMI LCD Display, a 9-inch 1280×800 (WXGA) display with 10-points capacitive touchscreen. Besides, 2×USB 3.0 Host used for faster communicated with the peripherals are also integrated. We run Android 4.4 KitKat with Linux 3.13 on it.

To demonstrate the usability and reliability of our system, moreover, we elaborated a high-level particular application that provides a reliable user input/output interface to ensure the security of the display content. In specific, we will present our implementation as follows.

5.1 Slightly instrumenting to the display driver

To quickly flush the content in the repainted frame buffer to the display device, we slightly modify the source code of the display driver through inserting a stub to check whether the flag of updating the frame buffer is met. When the human user touches the characters on touchscreen via the particular soft keyboard in SecEditor, the page faults will be triggered as the Rich OS tries to read the locked input buffer. In the page fault handler, the TDA will put a global flag into memory to inform the updating thread in SecEditor to invoke the invalidation function of frame buffer. At the same time, another flag for the display driver is also put into memory through the system call interface. Once the instrumented code in the display driver detected the flag set by the updating thread, it will invoke the HVC call instruction to repaint the currently used frame buffer using the prepared character images in the hypervisor. Otherwise, it will branch to the original flow of execution of the display driver.

SedDisplay made a couple of modifications on the OS driver. However, it does not mean SecDisplay is not a practical solution. SecDisplay is compatible with legacy programs, even though a program does not need protection, it can still run on the OS modified by SecDisplay.

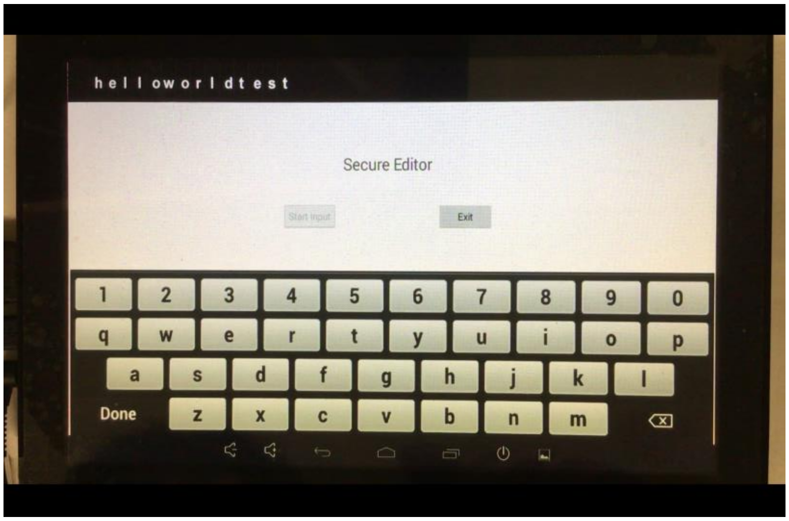

Figure 3: Snapshot of SecEditor based on the SecDisplay

5.2 Repainting based on character image

To quickly display the character clicked by the human user onto screen, we elaborate the common character images instead of producing them via complicated GPU computing and compositing. As is shown in Fig. 3, each character on the top of the picture is composed of a piece of image of suitable size. The 38 pieces of images are derived from their complete screenshots through dumping the data of the specific character area from the frame buffer into file. Thus, when a character needs to be displayed, we only directly copy the corresponding image data into the frame buffer.

Besides, we load these character-image data into the specific memory blocks during initializing the system and verify their integrity to prevent any runtime attacks from maliciously accessing them. Note that when we perform these operations in the TDA, the temporary variables in stack may be asynchronously modified by other CPUs due to running on multi-core platform, thus, the push or pop operation to save or restore the context is most likely to lead to crashing the OS. To this end, we allocate the stack memory for each core and cautiously maintain them.

5.3 Filtering out multi-touch noise

In virtue of different single-touch technology on the display device, when happened to pressing down only one time on the touchscreen, more than one touch events can be tracked down and repeatedly handled in the TDA. This severely constrains the performance of SecDisplay. Based on our prior study to touchscreen driver in Linux kernel, the driver will filter out redundant touch events triggered by the human user through setting a time threshold that is used to drop the events beyond the value.

Similar to the operation performed by the OS kernel, we estimate and adjust the time threshold to filter out the redundant touch events and further avoid the higher overhead from repeatedly repainting the frame buffer for the same touch event. To generalize our design and improve the user experience, only the region belonging to the particular soft keyboard will handle the page faults while the remainder including the gap between the characters on the keyboard and the blank areas on the top of the keyboard will has no any response.

6 Performance evaluation

In this section, we evaluate the proposed system. First, we measured the time required to identify coordinates, repaint shadowing frame buffer, copy frame data to three frame buffers and so on during the system initialization. Secondly, besides these micro-benchmarks, we also conducted the macro-benchmarks. Specifically, we use Android Vellamo (Version 3.2.6) and CF-bench (Version 1.3) benchmarks to evaluate the overall performance impact of the trust display service on the OS/Apps software. Finally, we show the code size of TCB and all major prototype components in SecDisplay. All the experiments were repeated 20 or more times and the average results are reported here.

6.1 Performance on micro-benchmarks

The runtime overhead incurred by SecDisplay includes the hypervisor interceptions or hypervisor calls and the CPU time spent by the hypervisor's execution. To evaluate the former cost, we measure the turnaround time of an empty HVC call which causes the CPU to trap into the HYP mode and return immediately. Our experiments show that the average cost for a round-trip mode switch is around 86 CPU cycles on our board. For the latter cost in the hypervisor, identifying the coordinate clicked by user on touchscreen, locating and repainting the shadow frame buffer and copying the data in shadow buffer into the frame buffers will take a bit of time. The overhead increases around 16 milliseconds on average, as is shown in Tab. 2.

Table 2: The relative time of code execution in SecDisplay, in millisecond

Identifying coordinates	~0.002
Repainting shadow frame buffer	~0.298
Copying to frame buffer	~15.63

6.2 Performance on macro-benchmarks

We use two Android benchmarks Vellamo and CF-bench to evaluate the system-wide performance impact of SecDisplay on the Rich OS/Apps. Vellamo includes two test items: Multicore and Metal while CF-bench actually involves in more, wherein we chose two representative benchmarks: Native and Java. The Multicore on Vellamo extensively measures floating point computing, memory r/w speed, system call operations, Binder IPC and so on, and the Metal mainly aims at CPU performance and networking capabilities. CF-bench is used to measure the performance overhead of DRAM and flash storage in Native and Dalvik environment, respectively. The results are shown in Tab. 3, the higher score means the better performance. The overhead on the overall system performance is quite small.

Table 3: Performance of the trust display service on Vellamo and CF-bench (higher score is better)

	SecDisplay Off	SecDisplay On	Performance loss (%)
Vellamo			
Multicore	963.7	949.1	1.5
Metal	475.5	459.6	3.3
CF-bench			
Native	21802.9	21230.1	2.6
Java	5613.2	5303.6	5.5
Overall	12088.6	11673.7	3.4

As aforementioned explanation, when SecDisplay is enabled, the extra Stage-2 translation will occur on every memory access. Therefore, the time spent on the address translation should be doubled theoretically while the impact is also aggravated accordingly. However, as MMU's TLB caches every mapping that previously has been translated and the data cache and instruction cache also have been enabled for frequent cache hits, the performance loss is still small. In addition, the context switch, TLB invalidation, identifying coordinate and copying the data into the frame buffers also will introduce a certain amount of performance drop, but the measurement results are almost same as off SecDisplay. The reason lay behind that is the two benchmarks never have access to any protected memory (e.g. the input buffer) which will trigger page faults to trap into the hypervisor to handle.

6.3 TCB size

In SecDisplay, the tiny hypervisor is the TCB of the trust display service. To estimate the safety of SecDisplay in terms of TCB size, we counted the number of source lines of our prototype. As is shown in Tab. 4, the SecDisplay hypervisor only consists of <190C SLoC and <1,400 assembly SLoC, therefore, SecDisplay has a smaller TCB than the previous works [McCune, Li, Qu et al. (2010); Yu, Gligor and Zhou (2015); Danisevskis, Peter, Nordholz et al. (2015); Cho, Shin, Kwon et al. (2016); Azab, Ning, Shah et al. (2014)]. Moreover, we also show the statistic of other components, such as the instrumented display driver with 90 SLoC totally, the system call interface comprising of 162 SLoC for interacting with kernel, and the high-level SecApp including java and native code with 223 SLoC.

Table 4: Code size in SecDisplay (in SLoC). The numbers of assembly code lines are in the brackets

Tiny hypervisor	Instrumented drivers	Two syscalls interface	SecEditor
183 (1378)	90	162	223

7 Related work

7.1 Virtualization-based security

The immediate benefit of virtualization is that the hardware resources of a platform can be partitioned two isolated domains, and the domain with higher privilege can monitor the activity of the other. This mechanism has been explored by a diverse of fields including malware analysis [Dinaburg, Royal, Sharif et al. (2008)], kernel rootkits detection and prevention [Li, Wang, Jiang et al. (2010); Riley, Jiang and Xu (2008, 2009)], virtual honeypot [Jiang and Wang (2007)], system security enhancement [Wang, Chen, Wang et al. (2015); Chen, Garfinkel, Lewis et al. (2008); Jiang and Wang (2007); Litty, Lagar-Cavilla and Lie (2008); Cho, Shin, Kwon et al. (2016); Azab, Ning, Shah et al. (2014); Shen, Li, Su et al. (2018)], etc.

In particular, virtualization is a popular choice of platforms to fortify the kernel or applications security. For instance, Patagonix protects the kernel code integrity through virtualization-based code identification [Litty, Lagar-Cavilla and Lie (2008)], while HookSafe further addresses the protection granularity problem through systematic hook redirection [Wang, Jiang, Cui et al. (2009)]. Meanwhile, Overshadow is designed to protect the secrecy of the user data in memory even if the kernel is completely compromised [Chen, Garfinkel, Lewis et al. (2008)].

Trusted Display [Yu, Gligor and Zhou (2015)] and Graphical User Interface [Danisevskis, Peter, Nordholz et al. (2015)] are two approaches based on virtualization to assure the confidentiality and authenticity of content output by SecApp and thus prevent a compromised Rich OS or application from surreptitiously reading or modifying the displayed output. More specifically, the former one mainly provides trusted display on commodity platforms that use modern graphics processing units (GPUs). The latter one provides a trusted and identifiable input and output path between the user and a VM.

Hardware virtualization-based systems are trustable due to their smaller code base and attack surface. However, the bloated code base of modern hypervisors and recent attacks put this assumption into question. This problem is much more serious as a system providing trusted display service needs to support complexity interoperations among several hardware components. Therefore, in this work, we proposed a system with very small code base to provide trusted display.

7.2 Trusted display with other techniques

Several previous approaches provide trusted display services through fortifying the OS kernel, such as Nitpicker [Feske and Helmuth (2005)] and Trusted X [Epstein, McHugh, Pascale et al. (1991)]. Glider [Sani, Zhong and Wallach (2014)] also could be used to provide a trusted display service since it isolates GPU objects in the kernel. However, these approaches extensively modified the OS kernel. Past research efforts to restructure unmodified OSes to support high-assurance security services were failed to meet stringent marketplace requirements on timely availability and maintenance. Moreover, the Rich OS always has a complicated code base and thus potentially contains a number of software vulnerabilities.

Other approaches provide trusted display by exclusively assigning GPU to SecApp.

Recent implementations of trusted path [Zhou, Gligor, Newsome et al. (2012); Zhou, Yu and Gligor (2014)] isolate communication channels from SecApp to GPU hardware. However, once assigned to a SecApp, the GPU cannot be accessed by the Rich OS/Apps until the device is re-assigned to them. Thus, the Rich OS/Apps cannot display their content during SecApp's exclusive use of the trusted display.

Meanwhile, GPU virtualization can provide trusted display services by running SecApps in a privileged domain and the Rich OS/Apps in an unprivileged domain. The privileged domain can emulate the GPU display function in software [Steinberg and Kauer (2010)] for the Rich OS/Apps. However, some GPU functions, such as image-processing emulation, are extremely difficult to implement in software due to their inherent complexity [Tian, Dong and Cowperthwaite (2014)]. As a result, GPU emulation cannot provide all GPU functions to the Rich OS/Apps, and hence this approach is incompatible with commodity software. A mitigation solution named Smowton [Smowton (2009)] paravirtualizes the user-level graphics software stack to provide added GPU functions to the Rich OS/Apps. Unfortunately, this type of approach requires graphics software stack modifications inside the Rich OS/Apps, and hence is incompatible with commodity OS. Basically, these techniques targets for x86 platform while leaving ARM untouched.

Particularly, TrustZone technology on ARM is extensively applied to security enhancement. TZ-RKP [Azab, Ning, Shah et al. (2014)] leverages the Security Extension to protect the kernel running in the normal world. Specifically, it instruments the original kernel to prevent it from executing certain privileged instructions or updating page tables. These operations instead must be handled by the secure world. Certainly, TrustZone technology can also protect the display content for the SecApp which typically runs in the secure world avoiding any interference from the normal world. However, the closed source to TrustZone code makes the research and implementation difficult without any external collaboration.

Some cryptography-based approaches [Yamamoto, Hayasaki and Nishida (2004)] decode concealed display images via optical methods; for instance, by placing a transparency, which serves as the secret key, over concealed images to decode them. These approaches are similar in spirit to the use of one time pads, and hence need physical monitor modification for efficient, frequent re-keying. Other systems [Oikonomakos, Fournier and Moore (2006); Yuan, Li, Wu et al. (2017); Pradeep, Mridula and Mohanan (2017)] add decryption circuitry to displays, and hence also require commodity hardware modification, which fails to satisfy design requirements.

8 Conclusion

In this work, we have proposed a trust display scheme named SecDisplay. It utilizes the hardware virtualization extensions provided by modern ARM processors to protect the display output from being stolen or tampered stealthily by a compromised OS. SecDisplay successfully maintains a minimal TCB while secures the display path between security-sensitive applications and display devices. The design of SecDisplay is fully compatible with commodity hardware, applications, OSes and the display drivers. We implemented a prototype of SecDisplay on Odroid-XU4 QSB, and elaborated a SecApp named SecEditor to demonstrate the usability and reliability of SecDisplay. The performance evaluations conducting on both micro-benchmarks and macro-benchmarks show a negligible overhead.

Acknowledgement: This work was financially supported by the National Natural Science Foundation of China (Grant No. 61379145) and the Joint Funds of CETC (Grant No. 20166141B08020101).

References

Alves, T. (2004): Trustzone: Integrated hardware and software security. *White Paper*.

ARM (2010): *Cortex-a9 technical reference manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388f/DDI0388F_cortex_a9_r2p2_trm.pdf.

Azab, A. M.; Ning, P.; Shah, J.; Chen, Q. (2014): Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 90-102.

Bianchi, A.; Corbetta, J.; Invernizzi, L.; Fratantonio, Y. (2015): What the app is that? Deception and countermeasures in the android user interface. *Security and Privacy*, pp. 931-948.

Chen, Q.; Qian, Z.; Mao, Z. (2014): Peeking into your app without actually seeing it: Ui state inference and novel android attacks. *USENIX Security Symposium*, pp. 1037-1052.

Chen, X.; Garfinkel, T.; Lewis, E. C.; Subrahmanyam, P. (2008): Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 2-13.

Cheng, Y.; Ding, X. (2013): Guardian: Hypervisor as security foothold for personal computers. *International Conference on Trust and Trustworthy Computing*, pp. 19-36.

Cheng, Y.; Ding, X.; Deng, R. (2013): Appshield: Protecting applications against untrusted operating system. *Singapore Management University Technical Report, SMU-SIS-13*, vol. 101.

Cho, Y.; Shin, J. B.; Kwon, D.; Ham, M. (2016): Hardware-assisted on-demand hyper-visor activation for efficient security critical code execution on mobile devices. *USENIX Annual Technical Conference*, pp. 565-578.

Danisevskis, J.; Peter, M.; Nordholz, J.; Petschick, M.; Vetter, J. (2015): Graphical user interface for virtualized mobile handsets. *IEEE S&P MoST*.

Davi, L.; Gens, D.; Liebchen, C.; Ahmad Reza, S. (2017): Pt-rand: Practical mitigation of data-only attacks against page tables. *24th Annual Network and Distributed System Security Symposium*.

Dinaburg, A.; Royal, P.; Sharif, M.; Lee, W. (2008): Ether: Malware analysis via hardware virtualization extensions. *Proceedings of the 15th ACM conference on Computer and Communications Security*, pp. 51-62.

Eppler, J.; Wang, Y. (2018): Towards improving the security of mobile systems using virtualization and isolation. *2018 4th International Conference on Mobile and Secure*

Services, pp. 1-6.

Epstein, J.; McHugh, J.; Pascale, R.; Orman, H. (1991): A prototype b3 trusted *x* window system. *Computer Security Applications Conference*, pp. 44-55.

Feske, N.; Helmuth, C. (2005): A nitpicker's guide to a minimal-complexity secure gui. *Computer Security Applications Conference, 21st Annual*, pp. 85-94.

Guan, L.; Liu, P.; Xing, X.; Ge, X.; Zhang, S. et al. (2017): Trustshadow: Secure execution of unmodified applications with arm trustzone. *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 488-501.

Hoanca, B.; Mock, K. J. (2005): Screen oriented technique for reducing the incidence of shoulder surfing. *Security and Management*, pp. 334-340.

Hu, H.; Chua, Z. L.; Adrian, S.; Saxena, P.; Liang, Z. (2015): Automatic generation of data-oriented exploits. *USENIX Security Symposium*, pp. 177-192.

Hu, H.; Shinde, S.; Adrian, S.; Chua, Z. L. (2016): Data-oriented programming: On the expressiveness of non-control data attacks. *Security and Privacy*, pp. 969-986.

Huang, M.; Liu, Y.; Zhang, N.; Xiong, N.; Liu, A. et al. (2018): A services routing based caching scheme for cloud assisted crns. *IEEE Access*.

Jiang, X.; Wang, X. (2007): "out-of-the-box" monitoring of vm-based high-interaction honeypots. *International Workshop on Recent Advances in Intrusion Detection*, pp. 198-218.

Li, J.; Wang, Z.; Jiang, X.; Grace, M.; Bahram, S. (2010): Defeating return-oriented rootkits with return-less kernels. *Proceedings of the 5th European conference on Computer Systems*, pp. 195-208.

Li, Y.; Cai, Z.; Xu, H. (2018): Llmp: Exploiting lldp for latency measurement in software-defined data center networks. *Journal of Computer Science and Technology*, vol. 33, no. 2, pp. 277-285.

Lin, C. C.; Li, H.; Zhou, X. Y.; Wang, X. (2014): Screenmilker: How to milk your android screen for secrets. *NDSS*.

Litty, L.; Lagar Cavilla, H. A.; Lie, D. (2008): Hypervisor support for identifying covertly executing binaries. *USENIX Security Symposium*, pp. 243-258.

Liu, F.; Li, T. (2018): A clustering-anonymity privacy-preserving method for wearable iot devices. *Security and Communication Networks*.

Logic, T. (2012): *Trusted foundations by trusted logic mobility*. http://www.arm.com/community/partners/display/product/rw/ProductId/5393/.

McCune, J. M.; Li, Y.; Qu, N.; Zhou, Z. (2010): Trustvisor: Efficient tcb reduction and attestation. *2010 IEEE Symposium on Security and Privacy*, pp. 143-158.

McCune, J. M.; Parno, B. J.; Perrig, A.; Reiter, M. K.; Isozaki, H. (2008): Flicker: An execution infrastructure for tcb minimization. *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 315-328.

Miluzzo, E.; Varshavsky, A.; Balakrishnan, S.; Choudhury, R. R. (2012): Tapprints: Your finger taps have fingerprints. *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pp. 323-336.

Oikonomakos, P.; Fournier, J.; Moore, S. (2006): Implementing cryptography on tft technology for secure display applications. *International Conference on Smart Card Research and Advanced Applications*, pp. 32-47.

Pradeep, A.; Mridula, S.; Mohanan, P. (2016): High security identity tags using spiral resonators. *Computers, Materials & Continua*, vol. 52, no. 3, pp. 185-195.

Riley, R.; Jiang, X.; Xu, D. (2008): Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. *International Workshop on Recent Advances in Intrusion Detection*, pp. 1-20.

Riley, R.; Jiang, X.; Xu, D. (2009): Multi-aspect profiling of kernel rootkit behavior. *Proceedings of the 4th ACM European Conference on Computer Systems*, pp. 47-60.

Sani, A. A.; Zhong, L.; Wallach, D. S. (2014): Glider: A gpu library driver for improved system security. *Operating Systems*.

Shen, D.; Li, Z.; Su, X.; Ma, J.; Deng, R. (2018): Tinyvisor: An extensible secure framework on android platforms. *Computers & Security*, vol. 72, pp. 145-162.

Sierraware (2013): *Open virtualization's sierravisor and sierratee*.
http://www.openvirtualization.org.

Smowton, C. (2009): Secure 3D graphics for virtual machines. *Proceedings of the Second European Workshop on System Security*, pp. 36-43.

Steinberg, U.; Kauer, B. (2010): Nova: A microhypervisor-based secure virtualization architecture. *Proceedings of the 5th European conference on Computer systems*, pp. 209-222.

Strackx, R.; Piessens, F. (2012): Fides: Selectively hardening software application components against kernel-level or process-level malware. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 2-13.

Sun, H.; Sun, K.; Wang, Y.; Jing, J.; Wang, H. (2015): Trustice: Hardware-assisted isolated computing environments on mobile devices. *Dependable Systems and Networks*, pp. 367-378.

Sun, W.; Cai, Z.; Li, Y.; Liu, F.; Fang, S. et al. (2018): Security and privacy in the medical internet of things. *Security and Communication Networks*.

Tang, J.; Liu, A.; Zhang, J.; Xiong, N. N.; Zeng, Z. et al. (2018): A trust-based secure routing scheme using the traceback approach for energy-harvesting wireless sensor networks. *Sensors*, vol. 18, no. 3, pp. 751.

Tian, C.; Wang, Y.; Liu, P.; Zhou, Q.; Zhang, C. et al. (2017): Im-visor: A pre-ime guard to prevent ime apps from stealing sensitive keystrokes using trustzone. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 145-156.

Tian, K.; Dong, Y.; Cowperthwaite, D. (2014): A full gpu virtualization solution with mediated pass-through. *USENIX Annual Technical Conference*, pp. 121-132.

Wang, X.; Chen, Y.; Wang, Z.; Qi, Y.; Zhou, Y. (2015): Secpod: A framework for virtualization-based security systems. *USENIX Annual Technical Conference*, pp. 347-360.

Wang, Z.; Jiang, X.; Cui, W.; Ning, P. (2009): Countering kernel rootkits with lightweight hook protection. *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 545-554.

Winter, J. (2012): Experimenting with arm trustzone -- or: How *i* met friendly piece of trusted hardware. *Trust, Security and Privacy in Computing and Communications (Trust-Com)*, pp. 1161-1166.

Xia, J.; Cai, Z.; Xu, M. (2018): An active defense solution for arp spoofing in openflow network. *Chinese Journal of Electronics*.

Xu, Z.; Bai, K.; Zhu, S. (2012): Taplogger: Inferring user inputs on smartphone touch-screens using on-board motion sensors. *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 113-124.

Yamamoto, H.; Hayasaki, Y.; Nishida, N. (2004): Secure information display with limited viewing zone by use of multi-color visual cryptography. *Optics Express*, vol. 12, no. 7, pp. 1258-1270.

Yu, M.; Gligor, V. D.; Zhou, Z. (2015): Trusted display on untrusted commodity platforms. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 989-1003.

Yuan, C.; Li, X.; Wu, Q.; Li, J.; Sun, X. (2017): Fingerprint liveness detection from different fingerprint materials using convolutional neural network and principal component analysis. *Computers, Materials & Continua*, vol. 53, no. 3, pp. 357-371.

Zhang, H.; Cai, Z.; Liu, Q.; Xiao, Q.; Li, Y. et al. (2018): A survey on security-aware measurement in sdn. *Security and Communication Networks*.

Zhou, Z.; Gligor, V. D.; Newsome, J.; McCune, J. M. (2012): Building verifiable trusted path on commodity x86 computers. *Security and Privacy*, pp. 616-630.

Zhou, Z.; Yu, M.; Gligor, V. D. (2014): Dancing with giants: Wimpy kernels for on-demand isolated I/O. *Security and Privacy*, pp. 308-323.