Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 8816023, 13 pages
https://doi.org/10.1155/2020/8816023

Research Article

From Hardware to Operating System: A Static Measurement Method of Android System Based on TrustZone

Xinhong Hei, Wen Gao, Yichuan Wang, Lei Zhu, and Wenjiang Ji

School of Computer Science and Technology, Xi'an University of Technology, Xi'an, China

Correspondence should be addressed to Yichuan Wang; chuan@xaut.edu.cn

Received 21 April 2020; Revised 27 July 2020; Accepted 7 September 2020; Published 21 September 2020

Academic Editor: Qi Jiang

Android system has been one of the main targets of hacker attacks for a long time. At present, it is faced with security risks such as privilege escalation attacks, image tampering, and malicious programs. In view of the above risks, the current detection of the application layer can no longer guarantee the security of the Android system. The security of mobile terminals needs to be fully protected from the bottom to the top, and the consistency test of the hardware system is realized from the hardware layer of the terminal. However, there is not a complete set of security measures to ensure the reliability and integrity of the Android system at present. Therefore, from the perspective of trusted computing, this paper proposes and implements a trusted static measurement method of the Android system based on TrustZone to protect the integrity of the system layer and provide a trusted underlying environment for the detection of the Android application layer. This paper analyzes from two aspects of security and efficiency. The experimental results show that this method can detect the Android system layer privilege escalation attack and discover the rootkit that breaks the integrity of the Android kernel in time during the startup process, and the performance loss of this method is within the acceptable range.

1. Introduction

In recent years, with the rapid development of mobile Internet technology, the number of users using Android mobile devices has increased rapidly. By 2018, the share of the Android system in the global smartphone has reached 85% [1]. According to CVE details [2], in 2017 and 2016, the vulnerability of the Android operating system was 842 and 523, respectively. According to the classification of these vulnerabilities in literature [3], the ratio of kernel vulnerabilities and standard libraries is the largest, accounting for 41% and 32%, respectively. At present, the Android system is mainly faced with cross script attack, privilege promotion attack, malware attack, privacy stealing attack, replay attack, communication attack, NFC attack, denial of service attack, etc. [4–9]. However, for the protection of attacks, most of the current research is in the application layer [10–16], but these solutions cannot fundamentally solve the security problems encountered by the current mobile terminal, and the terminal may still be threatened by malicious attackers and malware, so we should start from the system layer of the mobile intelligent terminal and build a secure and reliable mobile terminal system from bottom to top to ensure the security of intelligent terminal.

At present, there are three main methods for the security research of the Android system layer: SEAndroid, hardware-assisted virtualization technology, and TrustZone technology based on ARM. The introduction of SEAndroid has largely prevented malicious applications from attacking the system, but SEAndroid needs to rely on a trusted kernel and cannot defend against direct attacks from enemies [17]. For the hardware virtualization technology, L4Android [18] adopts the hardware virtualization technology to isolate the Android system on each occasion, but the attack on the system cannot be stopped. The Droid Visor [19] protects the integrity of the static key objects of the kernel and detects the rootkits of processes and modules, but it cannot detect the rootkits that modify the dynamic entropy pool resources. [20] can detect the integrity of the Android system kernel, but it cannot defend against the rights raising attack. For TrustZone technology, Zhang et al. proposed that T-Mac used TrustZone technology to strengthen Mac [21] but did not consider other factors affecting kernel security, such as not measuring the control flow in the kernel. Ahmed proposed a real-time kernel protection mechanism based on the advantage of TrustZone's hardware isolation. Although it has achieved some results against kernel level attacks, it has made significant modifications to the kernel. Ge et al. proposed a core code integrity measurement architecture SPROBES [22] based on the TrustZone architecture. Although it can measure rootkit, the performance loss of single instruction measurement is large. [23, 24] can implement a side-channel cache attack on the Android operating system using TrustZone. Therefore, in order to solve the security problem of the Android system layer, there is an urgent need for a more reliable and secure solution. Because SEAndroid needs to rely on a trusted kernel, hardware virtualization technology is currently considered too expensive and low versatility [25]. Therefore, this paper uses TrustZone technology to study the Android system layer kernel.

From the perspective of trusted computing, this paper proposes and implements a trust static measurement method for the Android system based on TrustZone, which takes bl1.bin image in ARM trusted firmware (ATF) as the trusted root, combines TrustZone technology with the Android system, and measures the kernel modules and executable files in the system startup process statically, and finally, extends the trusted root to the Android system application framework layer that provides a reliable underlying environment for the detection of the Android system application layer. This method can detect the elevated privilege attack of the Android system layer and discover the rootkit that breaks the integrity of the Android kernel in time during the startup process, and the performance loss of this method is within the acceptable range.

To sum up, our main contributions are as follows:

(1) Using the idea of trusted computing, according to the MTM specification, the hardware device is regarded as the source of trust, and the trust chain for Android system startup is designed to solve the problem of trust from the source

(2) A static measurement method for the Android operating system kernel is designed, which transfers the trust of the trusted root to the Android application framework layer through the trust chain

The rest of the paper is arranged as follows. The second section introduces the related knowledge of the technology used in this paper. The third section introduces our overall design. After that, the fourth section introduces the implementation process and gives the evaluation results in the fifth section. Finally, in the sixth section, we summarize this paper and look forward to the future work.

2. Related Work

2.1. Android Trust Chain. Since the establishment of the trusted computing organization (TCG), trusted computing has made rapid development. The establishment and transmission of the trust chain are the basic problems of trusted computing, which involve three points: trust root, trust transmission, and trust measurement. Trust root is the cornerstone of system trust and also the starting point of trust transmission. Trust transmission refers to the function of providing complete trust to the upper layer. The implementation of each layer of the system is based on the trust of the next layer, and the extension of the system's trusted range can be realized through trusted transmission [26, 27]. Trusted measurement refers to the integrity verification of files and their related configuration information to prevent them from being tampered with. The trust chain constructed by these three points gives trust from bottom to top and reduces the trust management of a large-scale system to the root of trust.

2.2. Android Framework Layer. As the middle layer of the application layer and underlying code, the Android framework layer encapsulates standardized modules to provide Java API for the application layer and also includes the JNI method to call underlying library functions to provide some system services; for example, Cameraservice and Mediaplayerservice are closely related to the user's privacy data. In the /system/framework directory of the Android system, there are mainly three types of files: jar package, ODEX file, and boot.art and boot.oat. Jar package provides support for various libraries in the framework layer for some functions of Android; for example, when executing the AM command, the am.jar file will be loaded. From Android version 4.4, Google has migrated the ART virtual machine to Android. After version 5.0, the ART virtual machine completely replaces the original Dalvik virtual machine. To run ART, the boot.art and boot.oat files in the directory are required. When compiling the Android source code, some common classes will be packaged into boot.oat; boot.art contains the pointer to the method code in boot.oat, which is the boot image of the ART virtual machine. The ODEX file in system/framework/oat/arm directory is the result of optimizing some jar packages when compiling source code. For example, services.odex will be loaded when creating system services.

Our goal is to measure the complete Android framework layer, so all files in the /system/framework directory are our goal.

2.3. Selection of Experimental Technology. At present, there are three main methods for the security research of the Android system layer: SEAndroid, virtualization technology, and TrustZone technology based on ARM. Because SEAndroid relies on a trusted kernel and cannot guarantee the security of the underlying system, we will not discuss it in this part. Therefore, we compare virtualization technology with TrustZone technology. The comparison results are shown in Table 1.

2.3.1. Security. All code resources in the trusted execution environment (TEE) are protected, and the management of this code requires certain permissions based on hardware control. The downloading and installation of trusted applications are also based on a certain trust. Particularly for trusted applications developed by third parties, the source of the

Table 1: Contrast result.

	TrustZone	Virtualization technology
Safety	Higher	Lower
SOC implementation	Easily	Difficulty
Ecology	Universal	No standardization
Application scenario	Sensitive applications	Applications that need to improve efficiency

application must be identified and certified before the application is downloaded and installed, so as to reduce malicious software, the attack of Trojan program on a safe operating system.

Compared with the security function of the TEE, virtualization technology allows multiple operating systems to execute on a host processor. Although these operating systems are isolated from each other, they do not make these operating systems have security features. Virtualization does not provide the corresponding interface to deal with security functions, let alone separate security hardware. From the perspective of isolating operating systems from each other to ensure the security of some operating systems, virtualization technology highlights the weakening.

2.3.2. SOC Implementation. For the SOC system, the TEE has the ability to control all hardware peripherals and filter the access to these peripherals under different CPU states, so the system itself needs to be clear about which execution environment is currently accessing which resources. For virtualization technology, the controller is only a software component, which can be directly connected to peripheral devices. The system itself does not perceive virtual machines. Because virtualization is only used to organize software running on the ARM core, it is very difficult to build a complete security system relying on it.

2.3.3. Ecological Creation and Maintenance. At present, the TEE has been deployed in a large number, and the platform it depends on can be completely transparent, and the TEE has the operating system agnostic. No matter what operating system is used by the mobile platform, it will have a set of standard communication interface to ensure that the operating system and the trusted application running in the TEE communicate with each other. On the contrary, virtualization products on mobile platforms do not have a standardized ecosystem to focus on the security needs of the industry. In addition, virtualization will be more intrusive at the following two levels: one is the virtual machine level, and the other is that the controller driver needs to adapt to each new platform monitor version.

2.3.4. Application Scenario. The TEE is generally used to implement sensitive applications, such as DRM, mobile financial payment, and enterprise mobile office. Virtualization technology enables multiple software environments to run on shared physical resources, so its use scenarios are more suitable for those application scenarios that improve efficiency.

In conclusion, TrustZone technology can better achieve the trusted static measurement of the Android system in this experiment.

2.4. TrustZone and OP-TEE. TrustZone is a group of hardware security extensions for ARM. The TrustZone space controller can divide DRAM into different memory areas and specify the memory area as safe or normal. The world executed by the processor is represented by an ns bit, which propagates through the system bus. The trusted bus structure ensures that normal world components cannot access any secure world resources [28]. The Open-source Portable Trusted Execution Environment (OP-TEE) project is implemented by the TEE open source launched by Linaro, which fully complies with the specifications and standards issued by the GP organization for TEE and supports all APIs of document specifications such as TEE client API v1.0 [29].

Therefore, this paper chooses a secure world os(optee_os) in OP-TEE as a trusted execution environment.

2.5. File Encryption Key of OP-TEE. FEK is the file encryption key used by OP-TEE when encrypting data. Each secure file of trusted application has a FEK to encrypt the data of the corresponding file. The generation process is shown in Figure 1.

Secure storage key (SSK): the value of the secure storage key is different in different devices. After the OP-TEE is started, the chip ID and hardware unique key (HUK) will be used to calculate the value through HMAC for use when generating other keys.

Storage Trusted Storage Key (TSK): TSK is the key used to generate file encryption key (FEK). TSK is calculated by HMAC using SSK as the key to the UUID of trusted application. TSK will be used to generate FEK finally.

The generation process of FEK is as follows:

$$\begin{aligned} \text{SSK} &= \text{HMAC}(\text{HUK}, \text{message}), \\ \text{Message} &\coloneqq \text{concatenate}(\text{chip\_id}, \text{string\_for\_ssk\_gen}), \\ \text{TSK} &= \text{HMAC}(\text{SSK}, \text{TA\_UUID}), \\ \text{FEK} &= \text{AES\_CBC}(\text{TSK}, \text{in\_key}), \end{aligned} \tag{1}$$

where in_key is the random number needed to generate FEK.

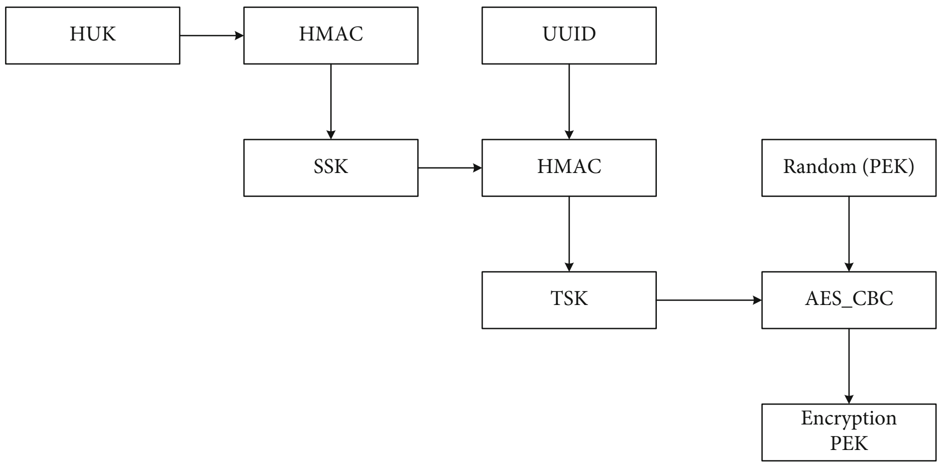


Figure 1: File encryption key (FEK) generation process.

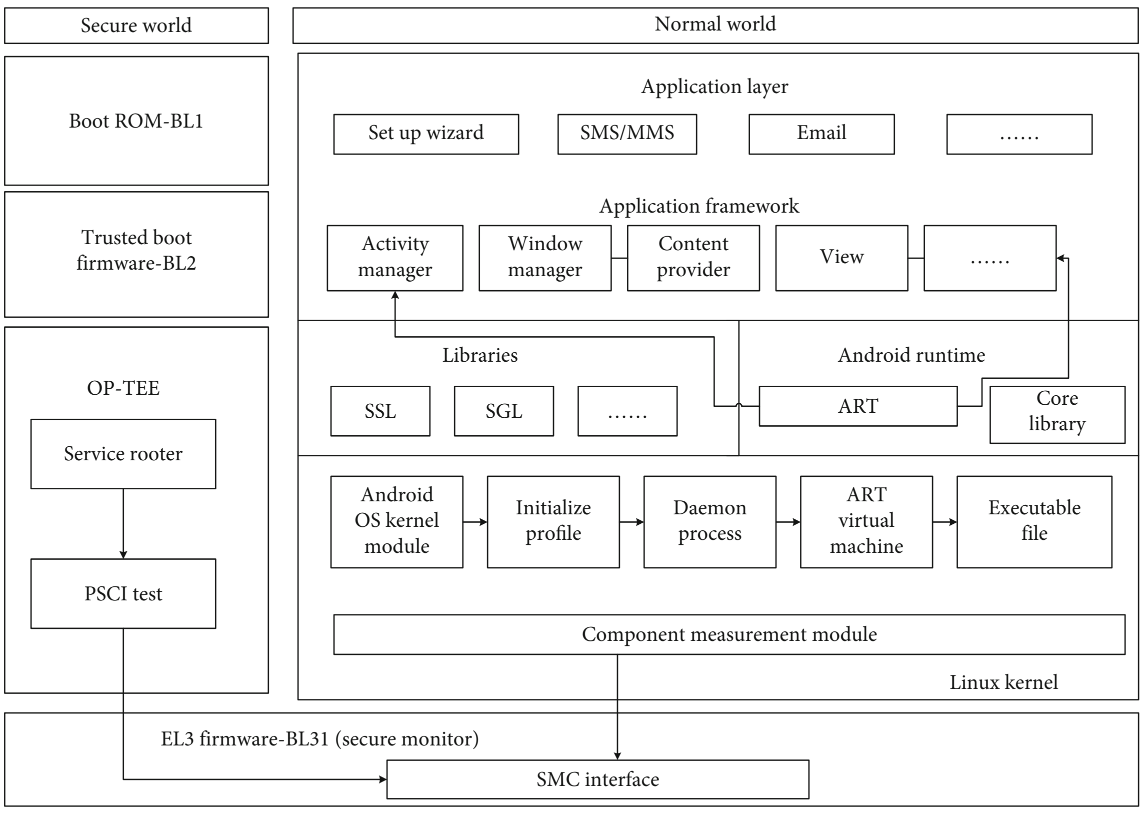


Figure 2: System architecture design.

3. System Design

In this part, we first introduce the design of the system architecture, then describe the trusted verification process and the trusted static measurement method in detail during the startup of the Android system.

3.1. Architecture Overview. According to the MTM standard of a trusted system, to establish the whole system's trust, we need to establish a trusted root first, then form a trusted chain according to the detection, and transfer the trust to each module of the system. In order to achieve the trusted static measurement in the process of Android system startup, we combine ARM trusted firmware with OP-TEE which implements TrustZone technology with the Android system, take bl1.bin image as the trusted root and OP-TEE as the trusted storage root, and add the degree module in the Android system kernel layer to design a trusted static measurement method for the Android system. The overall framework of the system is as shown in Figure 2.

3.2. Trust Delivery Process. According to the architecture chart we designed, the flow of the system integrity verification mechanism we designed is bl1 → bl2 → bl31 → optee_ os → Bootloader → kernel → Android system → APP. The whole process of trusted authentication is shown in Figure 3.

The startup process is divided into trusted execution environment (TEE) side startup and Rich Execution Environment (REE) side startup, which are described in the following two aspects.

3.2.1. TEE Side Start Process. After the system is powered on, it will start to execute the code in chip ROM. Chip ROM will first jump to the bl1.bin image of ATF for execution. After bl1 completes the operation of loading the bl2.bin image into RAM and setting the interrupt vector table, it will perform the signature verification operation on the bl2 image file. During the compilation of ATF, the system will perform the SHA256 calculation on all levels of images in ATF and then sign the generated summary. The private key is the RSA2048 key under the directory file. If the verification is passed, call the EL3, exit function to realize the jump from bl1 to bl2, and enter bl2 to start execution. In bl2, the signature verification module of the image file will be initialized first.

If the signature verification passes, the image file of the bootloader of bl31, OP-TEE, and Android system will be loaded into the memory with corresponding permission. Among them, bl31 is the execution software of EL3, whose function is to call security monitoring mode (SMC) instructions and interrupt processing. After triggering the security monitoring mode call in bl2, bl31 starts to run. bl31 determines whether to load OP-TEE by parsing whether there is an entry function of OP-TEE and verifying the validity of the OP-TEE image signature. If the entry function exists and the image signature verification is passed, OP-TEE will be started. After OP-TEE, the security monitoring mode call will be triggered to reenter bl31 for further execution. bl31 obtains the bootloader image file of the next Android system that needs to be loaded into the Rich Execution Environment (REE) side by querying the link list and verifies the validity of the bootloader file. If the verification is passed, then set the CPU state and running environment when the REE side is running and exit EL3 to enter the bootloader image startup of the Android system. At this time, the trust of the trusted root is transferred from bl1 to the bootloader of the Android system. If any part of the above process fails to be verified, it will directly cause the system to hang up.

3.2.2. REE Side Start Process. When the bootloader starts to start, it enters into the normal world of Android system startup. In the startup of the REE side, as shown in the architecture design in Figure 1, we add a measurement module to the kernel layer of the Android system. In order to formally describe and verify the startup process of the REE side, we refer to the PKI trust model on the basis of reference [30] and first give the following definitions:

Definition 1. Let e^* be the set of all components involved in the safe startup, and m be the OP-TEE,$\forall c_i, c_j \in e^*$, where $i, j \in N$. The following are the propositions:

(1) Integrity measurement capability: $\mathrm{TrustCapa}(c_i, c_j, \mathrm{Integ}) \mid <p_{jm}>$. It indicates that when the constraint condition p_{jm} is satisfied, component c_i believes that c_j has the trusted integrity measurement capability; p_{jm} refers to the trusted measurement capability that component c_j can communicate with OP-TEE

(2) Integrity credibility: $\mathrm{Trusted}(c_i, c_j, \mathrm{Integ})$ indicates that component c_i believes that c_j has a trusted integrity measurement attribute

(3) Integrity measurement: $\mathrm{Meas}(c_i, c_j, \mathrm{Integ}) \mid <\mathrm{RIM}>$ indicates that component c_i measures the integrity value of c_j, which is the same as the reference integrity value (RIM) stored in OP-TEE

Definition 2. Let e^* be the set of all components involved in the safe startup, and m be the OP-TEE, $\forall c_i, c_j, c_k \in e^*$. The following are the propositions:

Rule 1. Integrity measurement capability transfer rule:-
$\mathrm{TrustCapa}(c_i, c_j, \mathrm{Integ}) \mid <p_{jm}> \wedge \mathrm{TrustCapa}(c_j, c_k, \mathrm{Integ}) \mid <p_{km}> \rightarrow \mathrm{TrustCapa}(c_i, c_k, \mathrm{Integ}) \mid <p_{km}>$

Rule 2. Trust delivery rule:-
$\mathrm{TrustCapa}(c_i, c_j, \mathrm{Integ}) \mid <p_{jm}> \wedge \mathrm{Meas}(c_i, c_j, \mathrm{Integ}) \mid <\mathrm{RIM}> \rightarrow \mathrm{Trusted}(c_i, c_k, \mathrm{Integ})$

At the start of trusted start, the external observer c_0 thinks that only bootloader (c_1) in the mobile intelligent terminal is trusted and has integrity measurement capability, so there are initialization conditions as follows:

$$\begin{gathered}\mathrm{Trusted}(c_0, c_1, \mathrm{Integ}),\\ \mathrm{TrustCapa}(c_i, c_j, \mathrm{Integ}) \mid <p_{1m}>.\end{gathered} \tag{2}$$

Android kernel integrity measurement module is responsible for measuring the kernel module loaded in Android intelligent mobile terminal, initialization configuration file, daemons, ART virtual machine initialization process, and all executable files under the framework layer. The measurement process is as follows.

The measurement module measures the Android OS kernel module (c_2) and compares the measurement value with the expected measurement value stored in OP-TEE. If the measurement result is consistent, the next measurement will be continued. At this time,

$$\mathrm{TrustCapa}(c_0, c_1, \mathrm{Integ}) \mid <p_{1m}> \wedge \mathrm{Meas}(c_1, c_2, \mathrm{Integ}) \mid\mid <\mathrm{RIM}> \rightarrow \mathrm{Trusted}(c_0, c_2, \mathrm{Integ})). \tag{3}$$

Since c_2 has started and initialized m, it can be seen from the assumption that

$$\mathrm{Meas}(c_1, c_2, \mathrm{Integ}) \mid <\mathrm{RIMCert}> \rightarrow \mathrm{TrustCapa}(c_1, c_2, \mathrm{Integ}) \mid <p_{2m}>. \tag{4}$$

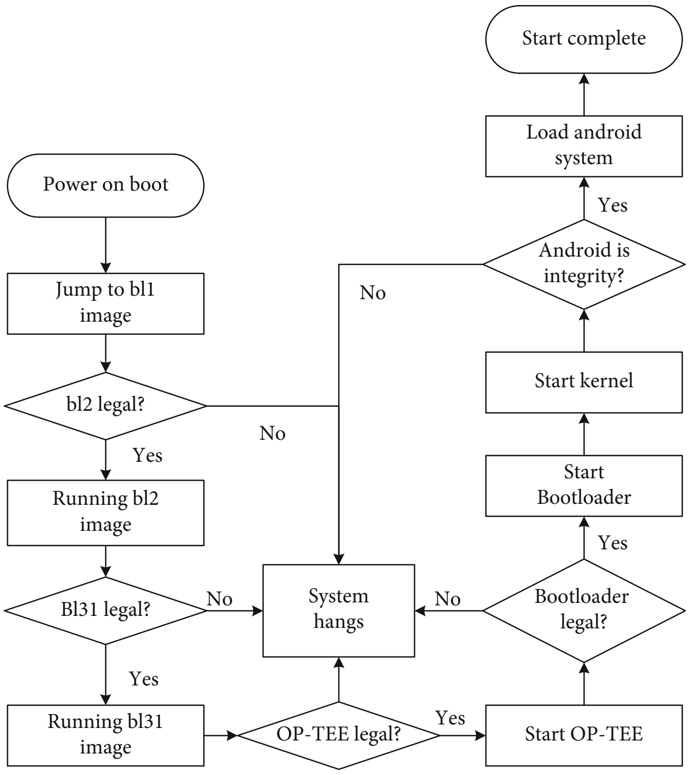


Figure 3: Start process trusted authentication process.

The measurement module measures the Android initialization configuration file (c_3) and compares the measurement value with the expected measurement value stored safely in OP-TEE. If the measurement result is consistent, continue to the next measurement, and the same can be obtained from the above derivation. At this time,

$$\mathrm{Trusted}(c_0, c_3, \mathrm{Integ}), \mathrm{TrustCapa}(c_2, c_3, \mathrm{Integ}) | < p_{3m} > . \tag{5}$$

According to the above method, the daemons (c_4) are measured and the measurement values are verified. If the results are consistent, then

$$\mathrm{Trusted}(c_0, c_4, \mathrm{Integ}), \mathrm{TrustCapa}(c_3, c_4, \mathrm{Integ}) | < p_{4m} > . \tag{6}$$

The measurement module measures the initialization process of the ART virtual machine (c_5) and verifies the measurement value. If the result is consistent, then

$$\mathrm{Trusted}(c_0, c_5, \mathrm{Integ}), \mathrm{TrustCapa}(c_4, c_5, \mathrm{Integ}) | < p_{5m} > . \tag{7}$$

Finally, measure and verify all executable files under the framework layer of the Android system. If the results are consistent, then

$$\mathrm{Trusted}(c_0, c_6, \mathrm{Integ}), \mathrm{TrustCapa}(c_5, c_6, \mathrm{Integ}) | < p_{6m} > . \tag{8}$$

It can be seen from the derivation that in the process of building the trusted start on the REE side, the trust relationship extends from c_1 to the boundary c_6 of the trusted base, indicating that the components on the trust chain in the trusted base are all trusted under the premise that the constraints are met. Therefore, the following conclusions can be drawn:

$$\mathrm{Trusted}(c_0, c_i, \mathrm{Integ}), \forall c_i \in e^* \ 1 \le i \le 6 \ i \in N. \tag{9}$$

By using the initial conditions and the formal deduction of the above formula, it can be seen that the safe startup process on the REE side is safe and reliable, which meets the requirements of integrity and trust verification. At this point, the trusted startup process of the whole system is completed, and the trust is extended from the root of trust for measurement to the framework layer of the Android system.

4. Detailed Description of Scheme

In order to realize the architecture we designed in the previous section, this part describes the process of our specific implementation architecture from the aspects of environment construction, trusted image production, image integrity

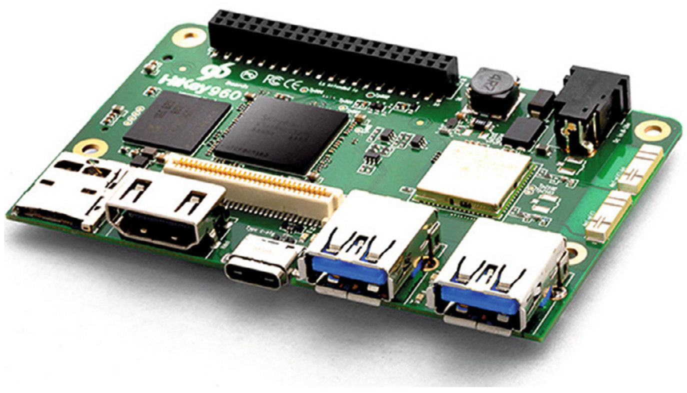


Figure 4: Kirin 960 SOC.

verification, measurement methods, and storage of expected metrics.

In this paper, in terms of the experimental hardware, the Huawei Kirin hikey960 development board based on Kirin 960 SOC shown in Figure 4 is used. The experimental environment is the Ubuntu 14.04 system.

4.1. Environment Building. First, get the latest Android AOSP and OP-TEE code of Google and then carry out MD5 detection and compare it with the official MD5 value to ensure the purity of the code. Then, add the TEE supplicant service in the init.common.rc file of Android source code and add the optee-packages.mk configuration file in the linaro/hikey directory. Add the configuration of OP-TEE in the device-common.mk configuration file and modify conf.mk and platform_config.h files of OP-TEE source code. The purpose is to identify and call the services provided by OP-TEE and provide a trusted environment for the next trusted measurement and safe storage in the Android system startup process. Then, obtain the source code of the underlying firmware ATF officially provided by ARM. The source code of ATF is divided into five parts: bl1, bl2, bl31, bl32, and bl33. bl1, bl2, and bl31 are fixed firmware; bl31 will execute the runtime service init function, which will call the initialization functions registered to all services in EL3. One of them is the TEE service. After the service is initialized, we modify the bl32 init code in bl31 to make the bl32 executed function jump to OP-TEE and start the startup of OP-TEE. After the initialization of OP-TEE, bl31 finds the bootloader of Android that needs to be executed by obtaining the link list of bl2, exits EL3, and enters the bootloader image for execution.

4.2. Production of Trusted Image. According to the Android system startup process framework described in the previous section, ARM trusted firmware, as a newly added stage of the secure startup architecture, not only completes the functions similar to some boot loader functions but also includes the module to verify the image in the next stage and the decryption public key at the time of verification. In order to realize the startup image integrity authentication, we recreate the bl2, bl31, OP-TEE, and bootloader images to be detected and make the trusted startup integrity verification image as shown in Figure 5.

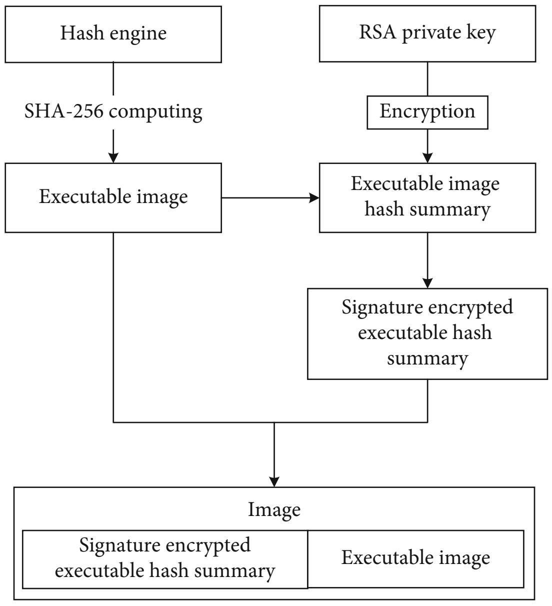

Hash engine
RSA private key
SHA-256 computing
Encryption
Executable image
Executable image hash summary
Signature encrypted executable hash summary
Image
Signature encrypted executable hash summary
Executable image

Figure 5: Production of the integrity verification image.

The steps are as follows: first, prepare the source code transplanted in each stage according to the requirements and compile and generate the executable image file; in the local computer, hash the executable image with the hash engine, which uses the public hash algorithm SHA-256. Get the hash result corresponding to the executable image: the hash summary of the image; then, use the RSA private key provided by the trusted firmware to sign and encrypt the asymmetric algorithm of the hash summary of the image. The encryption algorithm adopts the RSA asymmetric public key encryption algorithm; get the result after signature encryption; finally, the hash summary after signature encryption is relinked with the original executable image to generate the final image file.

4.3. Image Integrity Verification. In order to achieve image integrity verification, it is necessary to verify the source and integrity of the image in the next stage. After power on and startup, the system performs integrity detection to ensure the safe and tamper-free behavior of the image at startup. Figure 6 shows the opverification process in each stage.

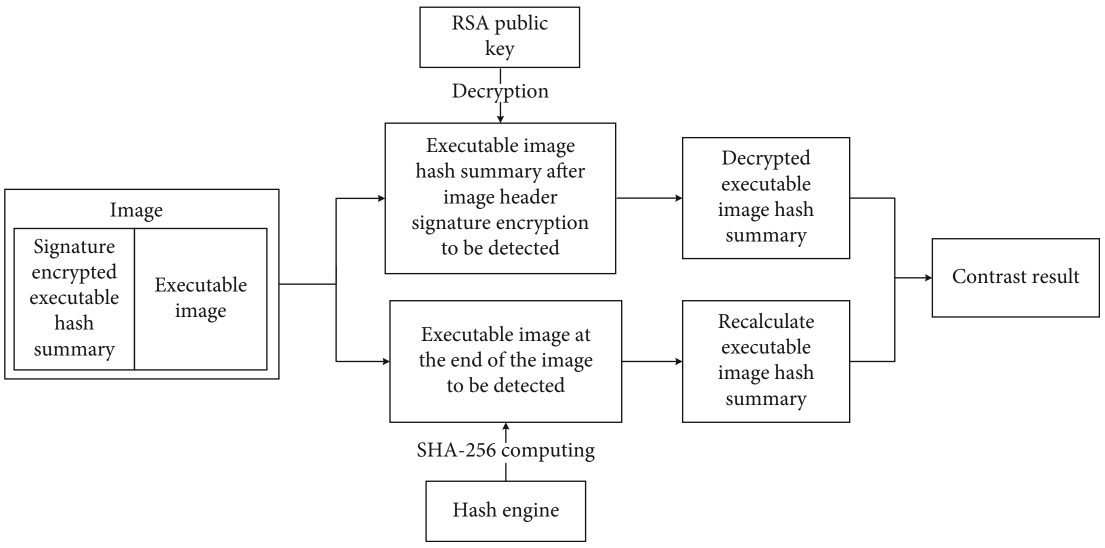

RSA public key
Decryption
Image
Signature encrypted executable hash summary
Executable image
Executable image hash summary after image header signature encryption to be detected
Executable image at the end of the image to be detected
Decrypted executable image hash summary
Recalculate executable image hash summary
Contrast result
SHA-256 computing
Hash engine

FIGURE 6: Image integrity verification diagram.

The verification process is as follows:

(1) First, copy the image of the next stage to the designated memory location according to the requirements of the design startup process

(2) The image is divided into two parts: one is the image head: the encrypted executable image hash summary; the other is the image tail: the executable image

(3) The encrypted executable image hash digest is decrypted according to the public key stored in the executing domain

(4) If it can be decrypted, it means that the image header data source is trusted, and the decrypted result can be obtained: the image hash summary can be executed, if it cannot be decrypted; it means that the image source is illegal and untrusted, and the operation of shutdown can be performed

(5) Then, hash the executable image at the end of the image. The hash algorithm is a public hash algorithm and must be consistent with the algorithm adopted in the local image production to get the recalculated executable image hash summary

(6) Compare the recalculated executable image hash summary with the result of the previous decryption operation

(7) If the two hash values are the same, it means that the image is reliable and complete, and the verification is passed; if the two hash values are different, it means that the image is incomplete, and the shutdown operation is performed

4.4. Implementation of the Measurement Method. We transplant and modify IMA of the Linux kernel to realize kernel measurement during startup. The full name of IMA is integrity measurement architecture; this component uses the hook function provided by LSM to detect files and application codes completely before they are executed or mapped to memory and generates a detection list. By reconstructing IMA code, we use the SHA-1 algorithm to measure kernel module, initialization configuration file, daemons, ART virtual machine initialization process, and executable files under the framework layer; the kernel is configured through making menuconfig; the kernel is recompiled; and the IMA service is started before the mount system partition.

4.5. Storage of Expected Measure List. Use the above measurement method to measure the kernel module, initialization configuration file, daemons, ART virtual machine initialization process, and the executable file of the framework layer of pure Android and generate the measurement list as the expected measurement value. Some expected measurement values generated are shown in Table 2.

Then, we store the generated measure list into the secure file system of the secure operating system as the expected metric list, which is used to start the comparison template for generating the metric list in the future. The security stored procedure steps are as follows:

(1) The REE side initiates the encryption request, and the client CA that executes the TEEC_InitializeContext function initializes the context of the TEE

(2) CA calls the TEEC_OpenSession function opens the session and establishes a connection with the corresponding trusted encryption and decryption program TA in the TEE

(3) CA implements TEEC_RegisterSharedMemory registers a piece of shared memory for communication between CA and TA, which is used to transfer data and commands to the security service in the TEE and receive the results returned by the security

Table 2: Expected measure list.

/system/lib64/libjavacore.so	Sha1:825341bd045d62c15fd7bdc4ec026932ccff4178
/system/lib64/libopenjdk.so	Sha1:fc483a0156f5bafe26bbcc9c90cd38b190516c89
/system/lib64/libvixl-arm.so	Sha1:8e6b911f86c4239a9bbd38df88fc1b91c5387f1d
/system/framework/core-oj.jar	Sha1:811d092eec40e1922af7aaf6189363de0f8a975f
/system/framework/core-libart.jar	Sha1:d2e8e403c1d0ddfecadc2c3ac51197f593e84dc0
/system/framework/okhttp.jar	Sha1:e26a028a129bd9c779667d03e2eeedf9ea6ce6b7

service. If the memory allocation is successful, step (4) is executed; otherwise, step (6)

(4) CA calls TEE_CreatePersistentObject interface, TEE_OpenPersistentObject interface, and TEE_WriteObjectData function, respectively, and writes the data to be transferred into the registered shared memory. After receiving the command, the security service in the TEE first reads the data information in the shared memory, and then OP-TEE sends an RPC request to notify tee_supplicant to complete the operation of the file system on the REE side and stores the security files in the data/TEE directory

(5) Execute the TEEC_ReleaseSharedMemory function to release shared memory

(6) Execute the TEEC_CloseSession function to close the session; the storage result is shown in Figure 7

4.6. Secure Transfer of Measure List. In the nonsecure environment, before the measure list file generated during the startup of the Android system on the REE side is transferred to the TEE security environment for comparison, the measurement data in this stage is also very easy to be intercepted by malicious programs. Therefore, we establish a secure metric list transmission channel between optee_os and Android systems through the TrustZone driver module to ensure that the metric list is transmitted to optee_os security. Figure 8 shows the framework of the security transmission channel of the measure list.

First of all, after generating the measurement list during the startup process of the Android system, REE obtains the key from the security environment and then encrypts the metric list with the aes-256 symmetric encryption algorithm. Then, it calls the CallTrustZone function through the TrustZone driver module to fall into the monitor environment. The monitor switches the execution environment of the system to the secure environment protected by TrustZone, the decryption module is called to decrypt the transmitted ciphertext, and then, the obtained metric list file is compared for the next operation.

4.7. Comparison of Measure List. The comparison phase is divided into two parts: first, decrypt the expected measure list file of the security storage, calling the read interface in TA and calling the syscall_storage_obj_read function to read the data of the security file in the OP-TEE kernel space. The function first obtains the TA session ID, the running context and checks the permissions, and then calls the ree_fs_read function to realize the operation of reading data. The second part is the comparison of measurement list files. SHA-1 operation on the decrypted expected measure list file is performed, and at the same time, SHA-1 operation is also performed on the measure list file decrypted in part 4.6. If the two results are consistent, the result will be returned to the REE side, and the Android system will start normally. If the results are different, a warning will pop up after the Android system starts.

5. Evaluation

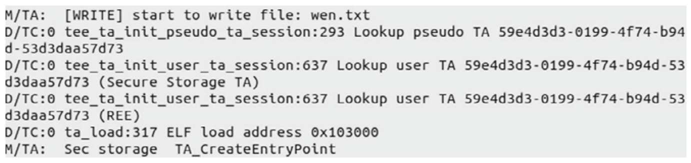
In this part, we discuss the experimental results about the functional effectiveness and performance of our method. All experiments are carried out on the hikey 960 development board.

5.1. Security Assessment. In the five attacks, the first two modified several bytes of the syscall table subroutine to achieve the attack, the third one modifies the system's exception vector table, the fourth one injects malicious code into the trigger mechanism onTouchEvent() function to enhance the permissions of the kernel layer, and the fifth one removes the process from the list of processes in the kernel to hide the process. Reference [20] proposes an android kernel measurement method based on the ARM virtualization extension called DIMDroid. This experiment is compared with the static measurement method in DIMDroid, and the results are shown in Table 3.

From the measurement results, it can be seen that both tampering with kernel static measurement objects such as system call table and interrupt call table and process hiding can be detected. However, DIMDroid measurement cannot detect the privilege attack of the application framework layer and kernel layer.

In the measurement list storage process, we compare our secure storage scheme with the traditional scheme that measure list is stored in ordinary Android files. The results are shown in Table 4.

As the template resource of the Android system startup process measurement list, the expected measure list is the benchmark and basis of the whole comparison process. Because the list of expected measures is stored in a secure isolated area, it can block security threats from nonsecure environments. In addition, in order to prevent other security services in optee_os from obtaining the expected measure list file, the asymmetric encryption algorithm combined with the key stored in the isolated area is used to complete the encryption protection of the expected measure list file.



```
M/TA:  [WRITE] start to write file: wen.txt
D/TC:0 tee_ta_init_pseudo_ta_session:293 Lookup pseudo TA 59e4d3d3-0199-4f74-b94
d-53d3daa57d73
D/TC:0 tee_ta_init_user_ta_session:637 Lookup user TA 59e4d3d3-0199-4f74-b94d-53
d3daa57d73 (Secure Storage TA)
D/TC:0 tee_ta_init_user_ta_session:637 Lookup user TA 59e4d3d3-0199-4f74-b94d-53
d3daa57d73 (REE)
D/TC:0 ta_load:317 ELF load address 0x103000
M/TA:  Sec storage  TA_CreateEntryPoint
```

Figure 7: Storage of expected measure list.

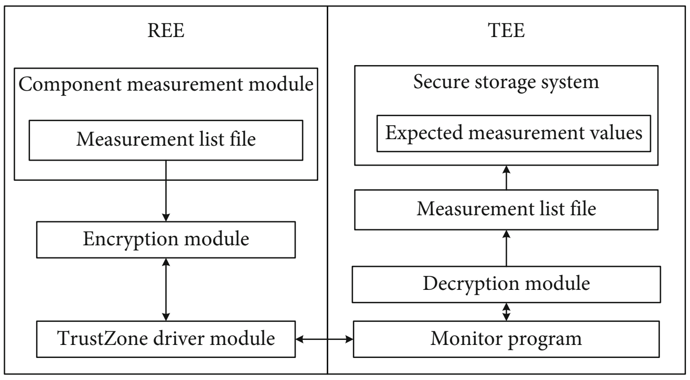


Figure 8: Secure transport framework.

Table 3: Attack experiment measurement results.

Rootkit	Attack function category	Measurement results of this experiment	DIMDroid metric
Rootkit1	Modify some bytes of syscall subroutine	√	√
Rootkit2	Modify some items of syscall	√	√
Rootkit3	Modify SWI software interrupt jump offset	√	√
Rootkit4	Inject malicious code into the onTouchEvent() function and elevate the kernel layer permissions to complete attack	√	×
Rootkit5	Intercept the proc_lookup function to hide the process	√	√

Table 4: Compare results.

Measure list	Our scheme	Traditional scheme
Storage location	Single file	optee_os
Safety	Weak	Strong
Encryption process	Unsafe	Safe

The attack of the Android system starting process metric list transmission process mainly occurs in the stage of transmitting the metric list from the Android environment to the TEE system. The TrustZone driver module will request memory space at the kernel layer and copy the list of metrics generated during startup. Since the list of measurements generated in the whole stage exists in ciphertext, the security of the process is guaranteed.

5.2. Efficiency Evaluation. In the experiment, we need to hash the image file and the file that the Android system needs to be measured. However, which hash algorithm to choose is our first consideration. Therefore, we choose four files with different sizes from the image file that need to hash and the file that the Android system needs to measure and do SHA-256, SHA-1, and MD5 operations on them, respectively. The results are shown in Figure 9.

It can be seen from Figure 8 that with the increase of file size, SHA-256 has the longest calculation time and the largest growth rate for the file, while MD5 has the smallest overall calculation time and the least impact on the calculation rate by the file size. SHA-1 is between the two.

SHA-256, SHA-1, and MD5 are all unidirectional functions, which are almost irreversible. The information that will generate a complete summary is entered. However, it is possible for different information to generate the same summary, which is called a collision. The security of the hash function depends on the ability to resist strong conflict to a great extent. Therefore, to evaluate the security of the hash function, it is necessary to check whether the attacker can find a pair of conflicts under the existing conditions. Table 5 lists the conflict thresholds of three hash functions.

According to Table 5, SHA-256 has the highest security, while MD5 has the worst. Considering the above time and security results, for the hash operation of the image, since the number of images that need to be hash operation is four (bl2.bin, bl31.bin, op-tee os, and uboot..img), the number of images that need to be calculated is small, and we have high

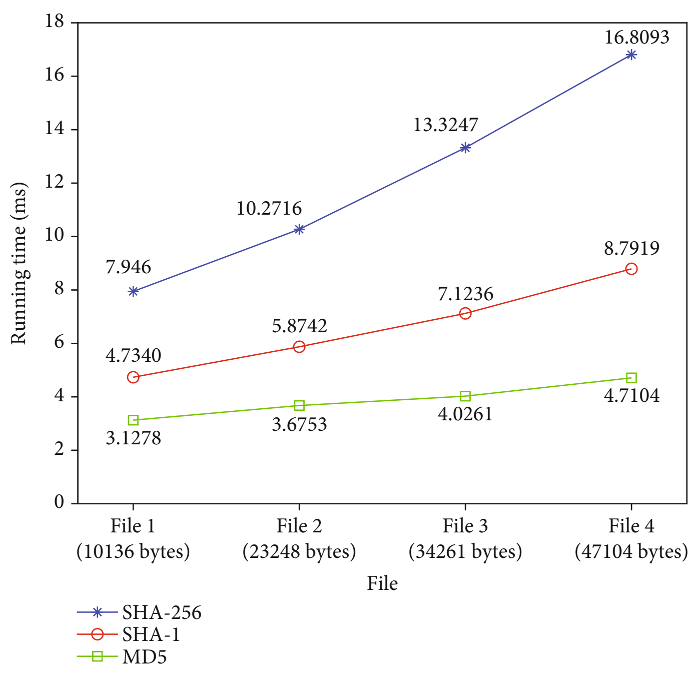


Figure 9: MD5, SHA-1, and SHA-256 encryption time.

Table 5: The collision thresholds for four commonly used hash functions.

Hash function	Function collision threshold
MD5	$2^{64} \approx 1.8 * 10^{19}$
SHA-1	$2^{80} \approx 1.2 * 10^{24}$
SHA-256	$2^{128} \approx 3.4 * 10^{38}$

Table 6: Time required to start components.

Unit (ms)	Bl1, Bl2, Bl31	OP-TEE	Bootloader	Kernel	Android OS
Normal start up	0	0	1226	5331	15052
Trusted startup	1219	2127	1352	5774	16484

security requirements for the image, so we choose the SHA-256 algorithm to generate a summary value for the image. For the measurement of Android system layer files, because the number of files to be measured is hundreds, if SHA-256 is selected, it will cause a lot of performance loss, so we choose SHA-1 operation to measure Android system layer files.

In this paper, for the scheme of trusted measurement of the Android system startup process, its performance impact mainly lies in the signature verification of image, the startup of OP-TEE, the SHA-1 operation on the set file, and the interaction time between the Android system and the OP-TEE. We have done 20 experiments on startup and take the average value of the results, as shown in Table 6.

In 20 experiments, the bootloader, kernel, and Android OS startup time is 10.2%, 8.3%, and 14.8% longer than that of the general Android. Because the startup of the trusted Android involves the time required for the startup of the trusted firmware and OP-TEE, compared with the normal Android startup process, the trusted startup process also increases the additional time overhead for the startup of ATF and OP-TEE. As shown in Figure 10, the starting time range of native Android is 21.1 s-22.8 s, and the starting time range of Android added to this experimental method is 26.3 s-27.6 s. The average starting time of this experiment is 23.4% longer than that of native Android.

In order to judge the impact of the kernel measurement module added on the REE side on the performance of the Android system, this paper uses the AnTuTu benchmark software, which is specialized in scoring Android device phones and tablets. Compared with the performance index of the unused kernel measurement module, the performance index mainly selects several mainstream options at this stage: ram speed, CPU floating-point calculation performance, and CPU integer calculation performance. Use the AnTuTu software test module to measure the kernel 100 times and take the average value. The performance loss ratio is the percentage of the difference between the score of the performance index item measured by the measurement module and the score of the index item measured by the measurement module, as shown in Table 7.

It can be seen from Table 6 that there is a certain performance loss in using the measurement module compared with not using the measurement module, but within the acceptable range, it shows that this method has certain reference significance for ensuring the integrity of the Android kernel.

6. Conclusion

In this paper, we propose and implement a TrustZone-based method to measure the trustworthiness of the Android

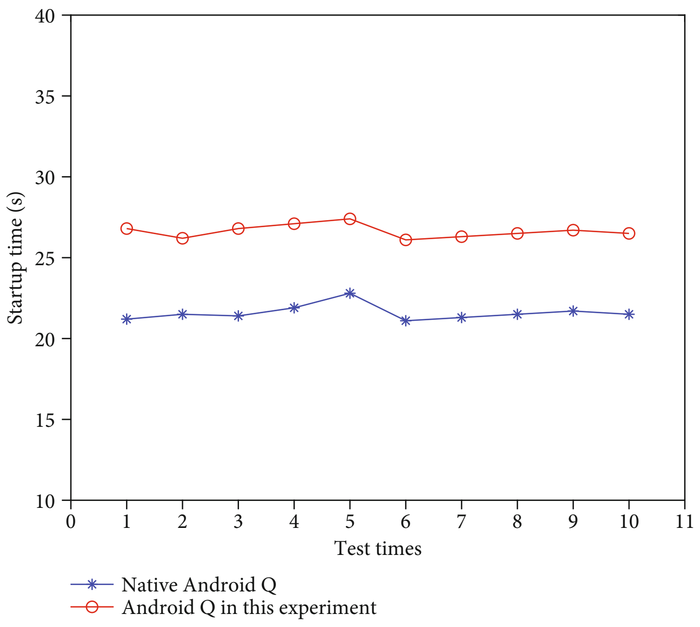


Figure 10: Startup time comparison.

Table 7: Using AnTuTu to test measurement module results.

Test item	Performance loss rate (%)
RAM speed	2.86
CPU floating-point calculation	2.58
CPU integer calculation	6.21

system. We use the bl1 image in ARM trusted firmware (ATF) as the trusted root, combine TrustZone technology with the Android system to measure the kernel modules and executable files in the system startup process, and finally, extend the trusted root to the entire Android platform. The next step is to give different weight values to different files according to the startup relationship, judge the security of the system according to the sum of the weight values, and give a more comprehensive and reasonable measurement verification to the Android system.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research work is supposed by the National Key R&D Program of China (2018YFB1201500), the National Natural Science Foundation of China (61602376, 61773313, 61602374, and 61702411), the National Natural Science Foundation of Shaanxi Province (2017JQ6020, 2016JQ6041), the Key Research and Development Program of Shaanxi Province (2020GY-039, 2017ZDXM-GY-098, and 2019TD-014).

References

[1] Statista, *Global mobile OS market share in sales to end users from 1stquarter 2009 to 2nd quarter 2018[EB/OL]*, 2018, August 2018, https://www.statista.com/statistics/266136/global-market-share-held-bysmartphone-operating-systems/.

[2] MITRE, *Cve details: Android vulneratbilities.[OL]*, 2018, June 2018, https://www.cvedetails.com/product/19997/Google-Android.html.

[3] M. Linaresvasquez, G. Bavota, and C. Escobarvelasquez, “An empiri-cal study on android-related vulnerabilities,” *14th International Conference on Mining Software Repositories*, pp. 2–13, 2017.

[4] Z. Xiaojing, “An autonomous protection algorithm for android malware attacks based on multiple features,” *Proceedings of 2019 International Conference on Information Science,-Medical and Health Informatics(ISMHI 2019).Institute of Management Science and Industrial Engineering*, pp. 573–576, 2019.

[5] G. Ye, Z. Tang, D. Fang et al., “A video-based attack for Android pattern lock,” *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 4, 2018.

[6] M. Youn-A, C. Tae-Mu, and J. M. Kim, “Astudy on Android attack by drive Management,” *Advanced Science Letters*, vol. 23, no. 10, pp. 9926–9929, 2017.

[7] B. Kong, L. Ying, and L.-P. Ma, “PtmxGuard: An Improved Method for Android Kernel to Prevent Privilege Escalation attack,” *ITM Web of Conferences*, vol. 12, p. 05010, 2017.

[8] A. H. N. Woo Hyun, P. A. R. K. Sanghyeon, O. H. Jaewon, and L. I. M. Seung-Ho, “Inishing: a UI phishing attack to exploit the vulnerability of inotify in Android smartphones,” *The*

Institute of Electronics, Information and Communication Engineers, vol. E99, 2016.

[9] J. Gu, C. Li, D. Lei, and Q. Li, "Combination attack of android applications analysis scheme based on privacy leak," in *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, Beijing, China, 2016.

[10] F. M. Faqiry, R. Rahman, and D. S. Tomar, "Scrutinizing permission based attack on android os platform devices," *International Journal*, vol. 8, no. 7, 2017.

[11] W. Bao, W. Yao, M. Zong, and D. Wang, "Cross-site scripting attacks on Android hybrid applications," *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pp. 56–61, 2017.

[12] S. Heuser, M. Negro, P. K. Pendyala, and A.-R. Sadeghi, "Droid auditor: forensic analysis of application-layer privilege escalation attacks on Android," *Proceedings of the 20th International Conference on Financial Cryptography and Data Security*, 2016.

[13] J. Vila and R. J. Rodríguez, "Practical experiences on NFC relay attacks with android," in *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, pp. 87–103, Springer International Publishing, 2015.

[14] M. Kato and S. Matsuura, "A dynamic countermeasure method to android malware by user approval," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pp. 730-731, Kyoto, Japan, July 2013.

[15] S. Y. Shin, Y. W. Kang, and Y. G. Kim, "Android-GAN: Defending against android pattern attacks using multi-modal generative network as anomaly detector. Expert Systems with Applications," *Journal of Engineering*, vol. 141, Article ID 112964, 2020.

[16] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Android-GAN: defending against android pattern attacks using multi-modal generative network as anomaly detector," *Expert Systems with Applications*, vol. 141, p. 112964, 2020.

[17] S. Xinlong, "Mobile device management system based on AOSP and SELinux," in *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, pp. 111–114, Shenzhen, China, June 2017.

[18] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: a generic operating system framework for secure smartphones," *Proceedings of the 1st ACM Workshop on security and Privacy in Smartphones and Mobile Devices*, , pp. 39–50, ACM, New York, 2011.

[19] Y. Yang, Z. J. Qian, and H. Huang, "A lightweight monitor for Android kernel protection," *Computer Engineering*, vol. 40, no. 4, pp. 48–52, 2014.

[20] L. Zicong, X. Kaiyong, G. Song, and X. Jingxu, "Dynamic measurement method of Android kernel based on ARM virtualization extension," *Computer application*, vol. 38, no. 9, pp. 2644–2649, 2018.

[21] D. Zhang, L. Chen, F. Xue, H. Wu, and H. Huang, "T-MAC: protecting mandatory acces control system integrity from malicious execution environment on ARM-based mobile devices," in *International Conference on Information Security*, pp. 348–365, Springer, Cham, 2017.

[22] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: enforcing kernel code integrity on the TrustZone architecture," *Computer Science*, vol. 25, no. 6, pp. 1793–1795, 2014.

[23] B. Lapid and A. Wool, "Cache-attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis," *25th international conference on selected areas*, 2018.

[24] A. M. Azab, K. Swidowski, R. Bhutkar et al., "SKEE: a lightweight secure kernel-level execution environment for ARM," in *Proceedings of Network and Distributed System Security Symposium*, San Diego, CA, USA, 2016.

[25] R. B. Yehuda and N. J. Zaidenberg, "Protection against reverse engineering in ARM," *International Journal of Information Security*, vol. 19, no. 1, 2020.

[26] F. Dengguo, Q. Yu, W. Dan, and C. Xiaobo, "Research on trusted computing technology," *Journal of Computer Research and Development*, vol. 48, no. 8, pp. 1332–1349, 2011.

[27] C. X. Shen, H. G. Zhang, D. G. Feng, Z. F. Cao, and J. W. Huang, "Survey of information security," *Science in China Series F: Information Sciences*, vol. 50, no. 3, pp. 273–298, 2007.

[28] N. Asokan, J. E. Ekberg, K. Kostiainen et al., "Mobile trusted computing," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1189–1206, 2014.

[29] Global Platform Device Technology, *TEE Client API Specification Version 1.0*, 2010.

[30] C. Shuyi, W. Yingyou, and Z. Hong, "Modeling trusted computing," *Wuhan University Journal of Natural Sciences*, vol. 11, no. 6, pp. 1507–1510, 2006.