Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

TC 11 Briefing Papers

TZMon: Improving mobile game security with ARM trustzone

Sanghoon Jeon[a,b], Huy Kang Kim[b,*]

[a] Samsung Electronics, Suwon, Republic of Korea
[b] School of Cybersecurity, Korea University, Seoul, Republic of Korea

ARTICLE INFO

Article history:
Received 15 March 2020
Revised 22 March 2021
Accepted 28 June 2021
Available online 4 July 2021

Keywords:
Mobile Game Security
ARM TrustZone
OPTEE
Application Integrity
Secure Update

ABSTRACT

As the game industry is moving from PC to smartphone platforms, security problems related to mobile games are becoming critical. Considering the characteristics of mobile games such as having short life-cycles and high communication costs, the server/network-side security technologies designed for PC games are not appropriate for mobile games. In this study, we propose *TZMon*, a client-side game protection mechanism based on the ARM TrustZone, which protects the confidentiality and integrity of mobile games. *TZMon* is composed of application integrity protocol, secure update protocol, data hiding protocol, and timer synchronization protocol. To adequately safeguard game codes and data, *TZMon* is designed considering an environment of frequent communications with the game server, a stand-alone operation environment, and an unreliable environment using a rooted OS. Furthermore, flexibility is provided to game application developers who apply security policies by using the Java Native Interface (JNI). In this study, we use Android and the Open Portable Trusted Execution Environment (OPTEE) as the OS platforms for Normal World and Secure World, respectively. After implementing a full-featured prototype of *TZMon*, we apply it to several open-source mobile games. We prove through the experiments that the application of the proposed *TZMon* does not cause any noticeable performance degradation and can detect major cheating techniques of mobile games.

1. Introduction

Mobile games are growing rapidly with the improved performance of smartphones and the expansion of mobile game platforms. They are among the mobile application services used by the most significant number of users AndroidRank (Accessed: March 2021). In the past, mobile games were not attractive objects of attack for hackers because of a lack of accessibility and liquidity. That is, they had been secure by obscurity. However, they are gradually becoming targets of hackers, as many PC games with a large user base can now be enjoyed in the mobile environment, and the accessibility and the liquidity of mobile games are increasing Kim and Kum (2013). Consequently, hacking attacks against mobile platforms are increasing; examples include those for stealing personal information or attacking game applications by obtaining the privileges of smartphones through SMiShing or the distribution of repackaged game applications, and for making monetary profits by exploiting mobile game services SOPHOS (Accessed: March 2021).

* Corresponding author.:
E-mail addresses: sh47.jeon@samsung.com (S. Jeon), cenda@korea.ac.kr (H.K. Kim).
https://doi.org/10.1016/j.cose.2021.102391

To defend against such hacking attacks, PC games are primarily applying server-side or network-side security techniques. Recently, several methods using data mining methods to analyze game play data in game servers Kwon et al. (2017) have been researched to detect game bots Han et al. (2015); Kang et al. (2016, 2013); Lee et al. (2015) and users who commit malicious acts. These detection methods have been frequently applied to PC games because they do not burden the game network and clients. However, they may be insufficient for mobile games because of the overhead Lee et al. (2018a) of the big-data analysis platform and the human resource required for analyzing large volumes of game logs, when the characteristics of mobile games having short life-cycle are considered Martins Bratuskins (Accessed: July 2020). Moreover, network-side security techniques Chen et al. (2009); Hilaire et al. (2010) such as encrypting the gameplay data and the transmission of play data of several features for efficient analysis can cause billing issues because of the additional data transmission in the mobile environment.

The existing client-side security techniques Chen et al. (2018); Fernandes et al. (2015); Rastogi et al. (2016) applied to mobile games include obfuscation for preventing decompile, repackaging detection, and rooting and emulator detection. However, using the obfuscation technique to mobile game applications implemented in Java and other such languages is not as effective as applying it to PC game applications deployed in the C language because these applications are easily decompiled. The repackaging detection method can be easily bypassed because of the lack of root of trust in mobile devices with weak security. Furthermore, rooting and emulator detection is not applicable in practice because it can increase the possibility of user churn by significantly limiting the autonomy of game users.

Our Approach. Approximately 97% Counterpoint (Accessed: March 2021) of Android smart-phones use the v7 Architecture or higher ARM core as Application Processor (AP), which supports the Trusted Execution Environment (TEE) called TrustZone ARM (Accessed: March 2021). The ARM TrustZone technology is widely used to improve mobile device security such as Samsung PAY and Samsung KNOX Samsung (Accessed: March 2021). Thus, in this study, we propose *TZMon*, which is a security mechanism to ensure the integrity and confidentiality of mobile games based on ARM TrustZone. *TZMon* is composed of application integrity protocol, secure update protocol, data hiding protocol, and timer synchronization protocol. The objects of protection are divided into code and data, and techniques to guarantee the integrity and confidentiality of each identified item are presented. In particular, considering the characteristics of mobile games, we studied not only the frequent communications with the game server but also stand-alone operations and operations in an unreliable environment using a rooted OS. Finally, *TZMon* was designed using JNI for ensuring the compatibility of mobile apps that operate in several contexts.

In this study, Android was used as the Normal OS and OPTEE Linaro (Accessed: March 2021) as the Secure OS. Furthermore, we use the HiKey Board embedded with a Kirin 960 chip that supports Octa-Core (4x Cortex-A73, 4x Cortex-A53) as the mobile device environment. A full-feature prototype of *TZMon* was implemented and applied to several open-source mobile games. Besides, the performance overhead of the proposed method was measured, and the detection of cheating techniques was checked.

Contributions. We summarize the contributions of this study as follows:

- *New Mechanism*. To improve the client-side security of mobile games, we proposed *TZMon*, which is a security framework based on the ARM TrustZone. To the best of our knowledge, this work is the first application of the ARM TrustZone to mobile games.
- *Threats and Countermeasures*. We classified the potential security risk of mobile games by the hacker's methods into the categories of memory searching, in-app bypass, application repacking, and abusing. The corresponding countermeasures were then organized into client-side, network-side, and server-side.
- *Implementation and Evaluation*. We implemented a full-feature prototype of *TZMon* and applied it to several open-source mobile games. It was proven through experiments that *TZMon* incurs on average 2.6% overhead to CPU utilization, 6MB overhead to memory utilization, 2.9 fps overhead to frame rate, and increases application loading time by 214 ms, and that it could detect five mobile game cheating techniques.

Organization. The rest of this study is organized as follows. Section II gives the background of the ARM TrustZone and our insights of *TZMon*. Section III presents the detailed design of *TZMon*. The evaluation results of the proposed mechanisms are summarized in Section IV. Section V discusses some limitations of the current model and the possible improvements. Section VI reviews the related work, and finally, Section VII concludes this study.

2. Background

This section is organized as follows. Section II-A briefly describes the ARM TrustZone technology and Section II-B summarizes the mobile game security threat. Section II-C provides the *TZMon* motivations, insights, and technical challenges; and Section II-D discusses the verification scope, and threat models.

2.1. *ARM Trustzone*

The ARM TrustZone is the trusted computing technique ARM Security Technology (2005–2009) applied to ARM processors introduced after the ARM v7 Architecture (e.g., Cortex A8, Cortex A9, and Cortex A15). ARM processors that support the ARM TrustZone operate two separate virtual processors within one physical processor, providing independent software execution environments called Secure World and Normal World. These execution environments include distinct virtual hardware resources, such as memory, cache, bus, register, timer source, and interrupts; and the hardware context is automatically converted when switching from one World to

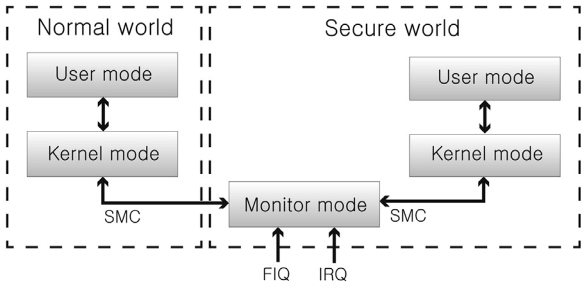


Fig. 1 – World Switching through Secure Monitoring Call.

the other. The ARM processor provides isolated CPU environments, where Secure World can access all system resources, but Normal World can access only resources specifically allocated to it; in particular, Normal World cannot access or change data in memory allocated to Secure World. The TrustZone protection controller divides hardware, including memory and peripherals, to support the asymmetric access for each World. Communication between Normal World and Secure World in user space is achieved through shared memory, which is allocated to the protected memory area and hence protected from other processes.

Fig. 1 shows that the secure monitor enables an ARM Processor to distinguish the worlds, and hence enables context switching. Context switching from Normal World to Secure World is only possible using the secure monitor call (SMC) command. When the SMC command is executed, the ARM CPU runs the secure monitor hardware to implement the context switch. The Monitor Mode operating in Secure World is started when a world switch occurs, and stores the current World hardware contexts, including cache, Translation Lookaside Buffer, and Memory Management Unit; and restores the context in the new World. In the ARM TrustZone, the IRQ is used as an interrupt in Normal World, and the FIQ is used in the Secure World. This ensures real-time OS support (Secure OS) in the Secure World because the FIQ has higher priority than the IRQ.

Fig. 2 shows that the ARM TrustZone requires a separate OS from either of those in Normal World OS, called Secure OS Kwon et al. (2019a). Applications operating in each World are divided into client applications (CA) and trusted applications (TA), where communication between CA and TA is also achieved via shared memory, and the TA can support functions such as secure storage, execution integrity and confidentiality, and end-to-end secure communication Pinto and Santos (2019). In TZIPC (TrustZone Inter-Process Communication), IPC is located in the monitor mode, not within the operating system. In other words, it is in the monitor mode so that the kernel, or other applications, cannot access the IPC code and data. In addition, when performing IPC, it is possible to execute by entering the monitor mode immediately without going through the operating system. As mentioned above, CA and TA can send and receive data through shared memory, and TZIPC improves the reliability of the shared memory by isolating it from other processes. Using the ARM TrustZone to make the shared memory area a completely independent communication channel from other processes, or even operating systems, there is no room for other elements to intervene.

2.2. *Mobile game cheating*

PC Games are vulnerable to several threats Woo and Kim (2012), including Gold Farming, Game Bot, and In-game Hack. Server-side and network-side analyses are the primary countermeasures against these threats. Server-side approaches analyze large game logs using data mining or machine learning techniques to detect these threats; whereas network-side approaches detect abnormalities by analyzing network traffic. Generally, anti-cheating techniques such as code obfuscation and security hardware (e.g., Intel SGX) are used on the client-side. However, in contrast to PC games, mobile games have short lifecycles, making it challenging to implement game log analysis on the server-side; and network-side detection is also challenging due to network charges or network delay issues. Client-side code obfuscation is not as effective for mobile games as it is for PC Games because many mobile game applications are implemented in Java GameACE (Accessed: March 2021).

We review exploits that occur in mobile games, and show how traditional countermeasures may not be appropriate for mobile games as opposed to PC games. Mobile game exploit scenarios use many technologies and tools in several ranges, but can be divided into four categories: memory searching, in-app bypass, application repacking, and abusing, depending on the particular attack.

2.2.1. *Memory searching*

The memory searching technique is the most common mobile game exploit. By using memory searching tool such as GameGuardian GameGuardian (Accessed: March 2021) and GameCIH CheatWare (Accessed: March 2021), even normal users can easily modify the value allocated in the corresponding address. As shown in Fig. 3, the memory searching technique is straightforward for any normal users to access and modify the critical data that can affect the game play such as hit point (HP) value, mana point (MP) value, damage, level, or money. If the attack is not countered, an abusing player can easily win against normal users, destroying the balance of the game, and normal users may lose motivation and leave the game.

Because the memory searching tool can only change data in the CA, it is important to verify the data on the server-side when making changes, particularly when checking billing and ranking-related data. Where the game operates stand-alone or with delayed updates due to network issues, applying the countermeasure on the client-side is essential. In the traditional countermeasure Fernandes et al. (2015), simple protection methods such as memory encryption or checksum are well-known examples.

The use of memory encryption makes it difficult for attackers to find out the critical data by using memory scanning technique. A checksum is useful for detecting any memory manipulation. However, the attackers can easily break the traditional countermeasure if they gain the encryption key from the unprotected storage (i.e., unprotected memory or hard disk). Therefore, the encryption key must be changed frequently for critical data encryption (charging, ranking, etc.) or

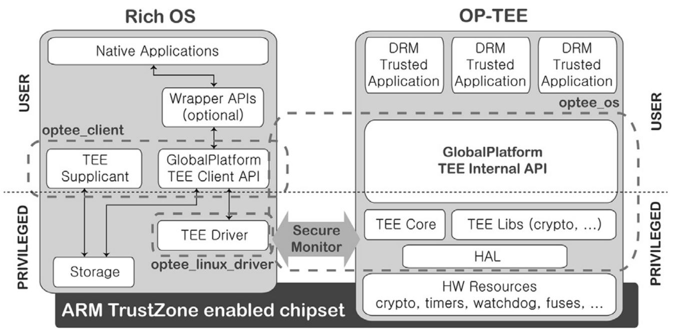
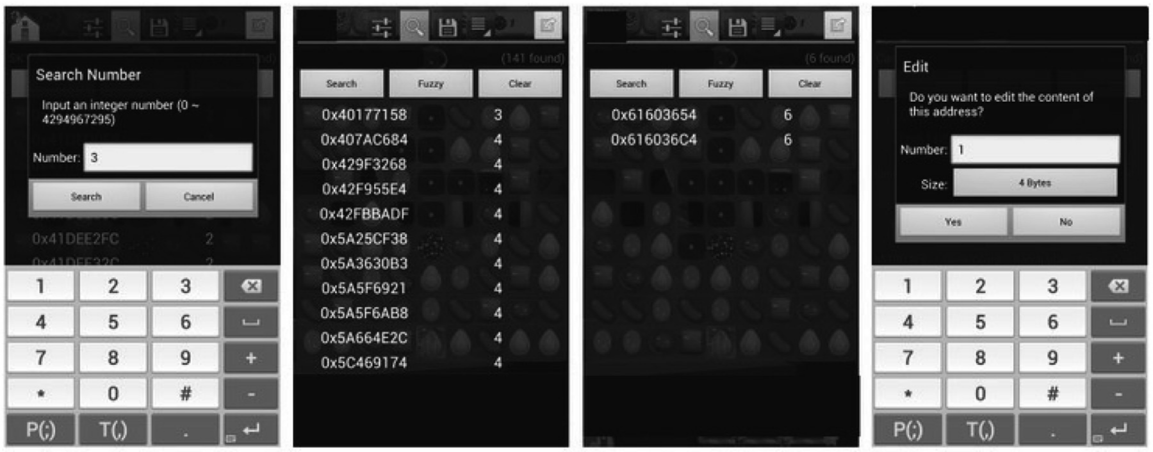


Fig. 2 – Rich and Trusted Execution Environment Architecture.



Search Number
Input an integer number (0 ~ 4294967295)
Number: 3
Search Cancel

(141 found)

```
0x40177158   3
0x407AC684   4
0x429F3268   4
0x42F955E4   4
0x42FBBADF   4
0x5A25CF38   4
0x5A3630B3   4
0x5A5F6921   4
0x5A5F6AB8   4
0x5A664E2C   4
0x5C469174   4
```

(6 found)

```
0x61603654   6
0x616036C4   6
```

Edit
Do you want to edit the content of this address?
Number: 1
Size: 4 Bytes
Yes No

Fig. 3 – Scanning and Modifying using Tool.

stored in another trust-worthy computing environment. One of the critical shortcomings of traditional countermeasures is performance degradation. Frequent updates of encryption keys, code obfuscation, or code verification for every runtime can seriously degrade the overall performance. Resulting is delays and degradation of the user's gaming experience.

2.2.2. *In-App bypass*

This attack changes or reuses transmitted packets or codes to bypass the payment function, allowing items to be used even if payment fails. Fraud detection systems based on log analysis are the most effective defense against this attack, but server-side data analysis is unsuitable for mobile games, as discussed above.

Fig. 4 shows how the LuckyPatcher tool Lucky Patcher (Accessed: March 2021) allows a user to change the menu within the application where the game has an in-app purchase menu, enabling the inclusion of a modulating application. This attack will eventually create a repacking application, so the game application integrity must be verified Chen et al. (2018). Similar to memory searching, these attacks can be defended by enhancing billing information authentication through the server, where server communication is frequent; otherwise, the client must be able to authenticate that a payment has been made, such that this code cannot be bypassed.

2.2.3. *Application repacking*

Application repacking manipulates the game application and redistributes it. An adversary can copy the game source to create a substantially similar game allowing unauthorized use of resources. The attacker usually modulates the reward item table and package name, and eliminates payment codes or advertising; but may manipulate all game details. Even if (for example) LuckyPatcher were not used, reverse engineering tools, e.g., dex2jar dex2jar (Accessed: March 2021) and jadx-gui Skylot (Accessed: March 2021), can analyze the code, modify it to fit the specific purpose, sign it with the keystore, and repack it. Traditional countermeasures include code obfuscation tools, e.g., arXan ARXAN (Accessed: March 2021), ProGuard GuardSquare (Accessed: March 2021), and DexGuard nGenSoft (Accessed: March 2021), which apply algorithms to ensure obfuscation of code layout, data, and control to make source code analysis difficult. Although code obfuscation is not a complete solution to hacking, it remains essential, making it challenging to analyze the source, and hence weakening the hacker's willingness to repack the application Rastogi et al. (2016); Sebastian et al. (2016). Attacks can also be blocked by sending the application signature value to the server at runtime to confirm the application Chen et al. (2015), by restricting access to only applications signed into the Google Play Store. However, checking application signature can be easily disabled by bypassing the relevant code.

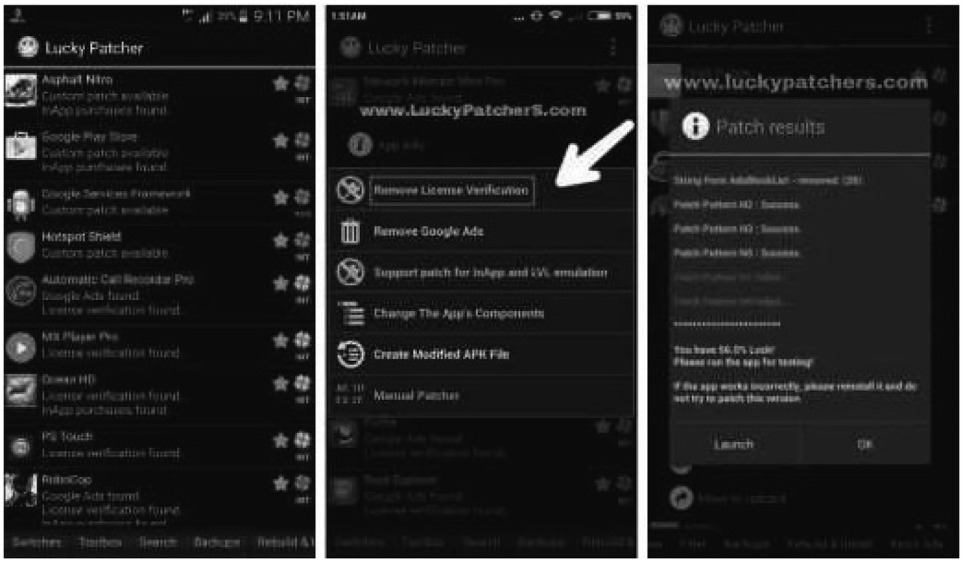


Fig. 4 – LuckyPatcher: Generating the Repacking APK.

Mobile game developers regularly monitor the online repacking community and impose sanctions when repacking applications are identified. However, manual monitoring has limitations due to the wide range of repacking techniques and methods. Application forging can be achieved by downloading a modified APK without rooting, so distribution is speedy and simple. Hence once a repacked application from the hacking community is registered, it can spread to many users in a short time.

2.2.4. *Mobile game abuse*

Abusing is the action of gaining benefit from illegal behavior. In a mobile game, an adversary can gain an unfair advantage by using an emulator, a macro, and speed hacks. Emulators play mobile games using PC programs, such as BlueStack BlueStack Systems (Accessed: March 2021), Nox Player Nox Ltd. (Accessed: March 2021), or GenyMotion GENYMOTION (Accessed: March 2021), which can play many accounts simultaneously and effectively perform infinite autoplay through macros. There is also a higher chance of being exposed to hacking, because PC hacking tools and macro programs are even more widely available than for mobile devices. However, there is some difference of opinion perceived among the game companies whether to view emulators as illegal or as a type of simple gameplay. It has become increasingly possible to allow games in emulators, and Fig. 5 shows that some emulators have been developed and deployed specifically for mobile game applications. This study limits the target to the mobile device, excluding consideration of emulators.

Speed hacking adjusts the device's clock to change the game's running speed. An adversary using speed hacking can finish the game faster, or reduce the game difficulty by slowing it. Speed hacking is based on the concept of delta time, which is used to maintain game speed across devices with varying performances, as shown in List. A.1 in the Appendix A. The algorithm acquires current time to calculate delta time using the OS time function. Therefore, speed hacking manipulates the hardware clock or hooks the time function. Changing the hardware clock is not possible in the mobile environment, because it would cause excessive battery consumption and thermal issues. Therefore, this study only considers hooking into the system time APIs.

Table 1 – List of Speed Hack and Macro Packages.

Package Name	Description
Xmodgames	Automatic and AI-based gameplay
HackerBot	Cheat premium features
Cheat Engine	Customize any of the Games features
GameCIH	Hack and patch Android games
Lucky Patcher	Remove ads, license verification
Creehack	Get unlimited Golds and much more
SB Game Hacker	Increase the points and scores
LeoPlay Card	Get unlimited Golds and much more
Game Killer	Modify whatever values you want in the title
Game Guardian	Speed hacks, searching for encrypted values

The traditional method to protect against such attacks detects and blocks the package installed in the device, as shown in Table 1. Speed hacking can be prevented by periodic time synchronization and deviation checks between the client and the server Christensen et al. (2013). The critical issue is that the code to check time deviation must reside on the server, because, if it resides on the client it can be easily manipulated or bypassed Vidas and Christin (2014).

2.3. TZMon *Design challenges*

Table 2 shows the proposed protection scope to defend against threats that may occur in mobile games, as discussed above. The protection target is data and code, with status separated into run-time and stored-time, and the overall purpose being to ensure confidentiality and integrity.

Fig. 5 – BlueStack: PC Emulator for Android Application.

Table 2 – *TZMon* Protection Scope.

Type	Status	Purpose
Binary Code	Run-time	Confidentiality
Data	Stored-time	Integrity

To design *TZMon*, as a client-side security tool for mobile games, we need to understand the differences between mobile devices and PCs. Mobile devices cannot assume they are continuously connected to the network, and games in particular often operate in standalone or delayed communication modes. To minimize network charges, data other than essential content for application services should be minimized. Moreover, many mobile applications, including mobile games, require fast response times to ensure perceived performance.

Challenges. Therefore, we must solve the following challenges with *TZMon*.

- ***Minimize communication with the game server:*** It is advantageous to check game code and data on the server-side where mobile games frequently communicate with the server. However, server-side checking may cause excessive network charges, and cannot be applied for games that operate stand-alone. Therefore, the *TZMon* protocol must improve mobile game security while minimizing communication with the game server.
- ***Detect disabled security mechanism:*** Mobile devices are relatively insecure compared to PCs, and rooted operating systems can cause and unreliable gaming environment. Thus, client-side only mechanisms without server augmentation can be easily bypassed by attackers. Therefore, the *TZMon* protocol should prevent security mechanisms from being disabled or bypassed, even in unreliable mobile environments.
- ***Minimize source code modification:*** Applying security frameworks can reduce game code maintainability by increasing its complexity, and diverse modification points can increase security problems by increasing the attack surface. Therefore, *TZMon* should minimize source code modification to be efficiently applied to existing mobile game applications.
- ***Lightweight design:*** Security and performance (or usability) are generally trade-offs. Heavy and complex security mechanism can adversely affect mobile game performance, increase network traffic (and subsequent costs) and lead to battery problems in devices. In particular, if the overhead for a sensitive and frequently used algorithm that affects perceived performance is increased, side-effects, such as reduced usability or performance, may overshadow the security improvement. Therefore, *TZMon* should employ a lightweight protocol to minimize perceived performance impacts.

2.4. TZMon *Design considerations*

Unlike other domain apps, mobile games are characterized by frequent updates and many attempts for cheating. Besides, there are many types of research to solve this problem because mobile games have many users, and security is weak due to the characteristics of mobile devices Shabtai et al. (2010). However, the legacy studies are mainly concentrated on the server-side and network-side approaches, and some client-side methods are easily manipulated Kim and Kum (2013). Therefore, it is necessary to design a security framework that reflects the characteristics of mobile games different from other domains and takes into account a mobile game app execution environment.

In the case of a mobile game compared to a PC game or other application, communication is performed using a mobile network (cellular network), and even when the game player moves, the IP address information or connection status may change and play. However, applying safeguards in network-side, in general, is not preferred because it can potentially cause a connection failure. In addition to this, network connection-based safeguards are challenging because mobile game users frequently close the app or switch to another app without explicitly closing the connection or logging out. In other words, the application of network connection-based safeguards will not be effective because the network

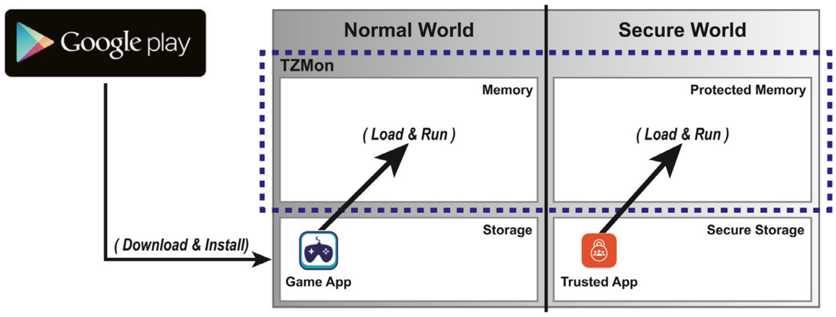


Fig. 6 – *TZMon* Scope and Flow.

connection can suddenly be disconnected due to the user's unexpected behavior.

In the case of mobile games, Some games are provided in a stand-alone type where the game application can be executed without any network connection or a server helps with security. Also, many asynchronous game designs do not continuously communicate with the server until the end phase and deliver only the game results to the server. Therefore, it is not easy to detect only on the server. Besides, the main attack target of hackers is an app installed on the mobile phone where they can economically forge, so it is necessary to block the forgery of the client-side app, which is the most frequent threat.

In the case of the network-side approach, full encryption is performed on the network side. In this case, the network traffic itself may be secured, and the traffic volume itself may be reduced (some sort of compression effect). However, it is not easy to apply encryption/decryption by each session on the server-side with over 100,000 concurrent users per second Lee et al. (2016).

There have been various attempts at the client-side. However, because a mobile device is fully controllable and fully manipulated at the user-side, attempts in the normal world of the client-side have limitations. There is a difference in online games/especially mobile games, with other domains applying TEE. It is common to deploy apps by using obfuscation/anti-debugging tools to the client-side. However, because it is not easy to apply robust existing security methods due to performance/heat/usability issues, a mobile game is a domain that requires the security framework of hardware-assisted methods.

Since mobile games are updated frequently as opposed to other application types that are updated less frequently, it is not adequate to use not TEE but the traditional way for protecting applications and memory. In other words, it is necessary to deploy the updated binary securely by end-to-end, and flexible security policies can be applied every time. Also, due to this frequent update, game users are adapted to the initial delay, so it has a characteristic that it is easier to check security-related in the early stage than other apps.

The goal of *TZMon* is to decrease the overhead of the network & server-side, which is a fine-grain filter by applying a client-side approach, which is a coarse-grain filter. We decided to apply TEE, a hardware-assisted method, considering the characteristics of fully controllable, fully manipulable mobile devices, performance issues, heat issues, and user experience (usability). To the best of our knowledge, this is the first case where the ARM TrustZone has been applied to mobile games. We proved through experiments that the application of the proposed *TZMon* does not cause noticeable performance degradation and can detect major mobile game cheating techniques.

Table 3 – *TZMon* Security Goals related to a Mobile Game .

Security Goals	*TZMon* Protocols
Anti-Piracy	Application Integrity, Secure Update
Anti-Cheating	Data Hiding, Timer Sync.

There are typically two high-level goals related to "game security": (1) anti-piracy and (2) anti-cheating. Thus, there are typically two high-level goals related to "game security": anti-piracy and anti-cheating. Therefore, each protocol proposed in this study is designed to achieve those mobile game security goals, as shown in Table 3. *TZMon* designed the application integrity protocol and the secure update protocol to protect against application repacking attacks for anti-piracy. The data hiding protocol and the timer synchronization protocol protect mobile games against abusive attacks and attempts to cheat, such as Hooking, Speed Hacking, Memory Searching, etc. Details of each protocol are described in Section 3.

2.5. *Scope and threat model*

Scope. We assume the mobile game application is intended to be securely downloaded and installed through the Google Play Store, as shown in Fig. 6. In particular, secure deployment and installation of TA is necessary to get a Root of Trust (RoT). The Trusted Secured Platform (TSP) Trustonic (Accessed: March 2021) and Trusted Application Manager (TAM) intercede (2019) provide 3rd party application owners with a solution to securely install and deploy TA. This paper assumes that the TA is intended to be secured downloaded and installed through previous related works. After installation, *TZMon* targets threats that may occur when loading and executing the normal game application and the TA. The *TZMon* design goal was to provide security mechanisms that consider mobile games characteristics, as distinct from PC games.

Therefore, *TZMon* does not include a protocol using the server, which is an environment similar to that of PC games except for the secure update protocol. Even a secure update protocol assumes that attackers could not exploit the update server. In addition, the application integrity protocol focuses on detecting whether an application is repacked. When it is identified as a repacked application, it aims to report to the server or send a warning message to the client. That is, it is not within the scope of this study to prevent the execution of a repacked application. In general, when a repacked application is detected, the server can take several measures, such as forcibly disconnecting the client application or the server connection, sending a warning message to the client, as well as logging the incident without any warning or action and then suspending the account ID at that specific point in time. Finally, considering the popularity and openness of current offerings, the proposed security mechanism is focused on the Android OS and the OPTEE OS.

Threat Model. We use the same threat model as the ARM TrustZone, i.e., trust only ARM TrustZone applications and related code, including shared memory, protected hardware, and security mechanisms that run in the Secure World. In particular, We do not trust game algorithms, shared libraries, Android platforms, memory, and I/O running in the Normal World. We assume we can detect game algorithms by reverse engineering the game code and data, and we can manipulate the OS and the Shared Library for cheating during gameplay. On the other hand, adversaries assume that any information about code and data operating in the Secure World cannot be acquired. Authentication procedure between game applications and TAs are performed by the ARM TrustZone; thus, this study does not describe these procedures in detail.

However, the security framework proposed here cannot defend against all exploits. Recent studies Zhang et al. (2016) have shown the ARM TrustZone to be somewhat insecure due to side-channel attacks, such as cache side-channel attacks. It was announced in Guilbon (Accessed: March 2021); Shen (2015) that it is possible to manipulate Secure World memory using the SMC return value verification vulnerability. Thus, *TZMon* may not be secure against such attacks due to this fundamental vulnerability in the ARM TrustZone. In addition, as attack methods are advancing, it is possible to attack without changing the application directly through ROP (Return-Oriented Programming) and DOP (Data-Oriented Programming) in a short period of time. Much research such as the shadow stack Burow et al. (2019) and control flow integrity (CFI) Li et al. (2020) is being done to mitigate ROP and DOP attacks, but attack cases Sun et al. (2019) that bypass these defense methods are also being reported simultaneously. Because this study is a method that protects against attacks by application and by data manipulation, *TZMon* cannot defend against attack methods that do not change applications or data such as ROP and DOP.

Authentication Mechanism. TrustZone provides distinct security features by separating resources from Normal World based on hardware access control, but still has several technical problems. Notably, the CA's authentication mechanism is not provided in the TrustZone environment. Therefore, when Normal World is Compromised, TrustZone's TA can be used for malicious purposes because the authentication mechanism can be bypassed through compromised CA, and all subsequent security protocols are disabled.

Several methods Jang and Kang (2019); Jang et al. (2015) have been researched to solve these problems of CA's authentication. The SeCReT, which is composed of SeCReT_T of REE and SeCReT_M of TEE issues a one-time session key for CA and TA, and this session key allows SeCReT to authenticate CA and create a secure channel. That is, SeCReT checks the control-flow integrity of the requested process whenever issuing a one-time session key and can detect malicious CAs through this check. Besides, the session key uses only one-time to protect against a replay attack.

CA's authentication in TrustZone is essential for providing TEE service. Still, since the authentication mechanism and security protocol can be applied independently of each other, this paper does not deal with the authentication mechanism separately. It assumes that *TZMon* uses the existing authentication method. In other words, even if Normal World is compromised, other malicious CAs in the client device cannot use the TEE service. However, even in such a case, exploiting the TEE service by forging or modulating a normal CA itself is possible.

3. Detailed design

Fig. 7 shows the proposed *TZMon* design. The confidentiality and integrity of stored data with the protection scope as in Table 2 are supported through Secure OS storage, i.e., stored code security can be guaranteed through the secure OS's TA control policy. Therefore, detailed design of related content is not described separately; this section focuses on run-time code and data confidentiality and integrity.

Also, it assumes that all the security protocols operate after applying the authentication mechanism, as described in the II-D section. In other words, attacks by other malicious CAs inside the device can be prevented. However, forgery and manipulation attacks against legitimate CAs or attacks that bypass the security protocols may still be possible. In this section, I will explain the security protocol to protect against these attacks.

TZMon supports Application Integrity, Secure Update, Data Hiding, and Timer Synchronization. The *TZMon* library operating in Normal World is formed with a native C library, which makes reverse engineering attack difficult. In the text that follows, 'm' and 'o' following the protocol name means mandatory and optional, respectively. At the end of each protocol section, major considerations in protocol design are addressed as challenging points.

3.1. *Token chaining algorithm*

All security protocols of *TZMon* operate after the authentication mechanism, but attacks that bypass each security protocol can be possible. Therefore, in this paper, to confirm that each protocol is legitimately operated, *TZMon* issue the token, which can be used in the session after each security protocol. Whenever a protocol is repeatedly executed, each token

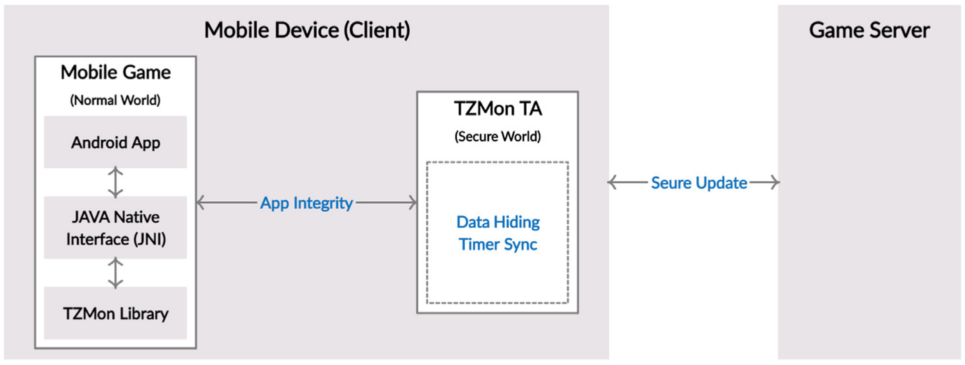


Fig. 7 – Overview of *TZMon*.

is chained to verify that the preceding process has been operated normally.

Theoretically, an attack that bypasses all processes after the authentication mechanism may be possible. Still, in this case, the mobile game cannot be run normally because it cannot access critical data. Besides, due to the chaining token mechanism, an attack that bypasses some protocols is impossible. That is, for the game execution, at least the Data Hiding Protocol must be executed, and the chaining token is validated at every protocol execution; therefore, a protocol bypass attack can be detected. If all tokens, including session key, are sniffed, it may be possible to attack only some protocols. However, since the session key and all tokens are valid only in the session, other attacks except the dynamic attack in runtime are impossible. However, depending on the scenario, there may be a situation in which critical data managed through the Data Hiding Protocol is not requested, so in case the protocol chain is wholly bypassed, Critical data, such as health, position data, of sample users are sent to the server. If there is a difference between the data returned from the client and the data left by sampling from the server by randomly writing logs, it is possible to detect whether the proposed security scheme has been bypassed and update it through subsequent analysis. For example, when calculated on the server, the monster at this point has 10 health, and this user must have 1000 gold.

Fig. 8 shows that the chaining process of tokens used as inputs and outputs when operating each security protocol. That is, as in Algorithm 1, the token generates a new token for each security protocol starting from the session key. The overall process in the figure explains the general procedure. It is not a fixed order except the app integrity protocol, and the applying order of the protocol can be changed according to the mobile game profile.

Algorithm 1 Token Creation and Verification Algorithm.

```
1: procedure TOKEN_CREATE_VERIFY
2:     session_key ← get_session_key()
3:     if is_app_integrity(protocol) then
4:         policy ← choose_policy()
5:         app_sign ← create_sign(policy)
6:         token ← kdf(app_sign, salt)
7:         return token
8:     else
9:         recv_token ← receive_token()
10:        prev_flag, prev_token ← get_prev_info()
11:        if prev_flag == true then
12:            if recv_token == mac(session_key, prev_token) then
13:                token ← kdf(prev_token, salt)
14:            end if
15:        end if
16:    end if
17:    return token
18: end procedure
```

3.2. *Application integrity (m)*

As explained in section II-D, even if the authentication mechanism is applied, attacks that forge or manipulate legitimate CAs can still be possible. Therefore, this section describes the application integrity protocol to prevent such attacks. Application integrity checking is a mechanism to detect mobile game

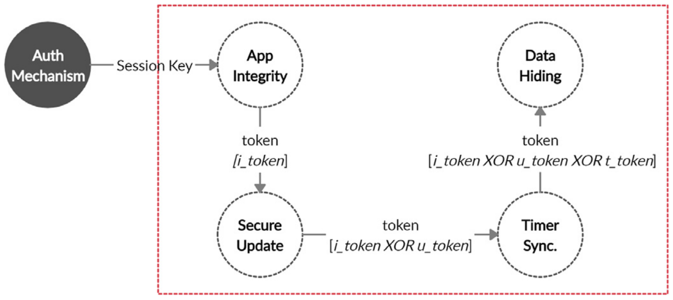


Fig. 8 – Overall Verification Process.

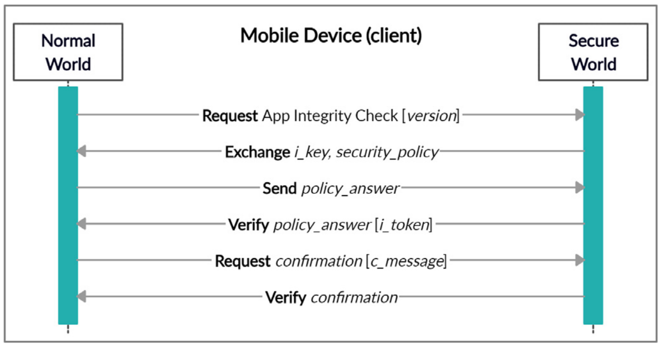

Normal World
Mobile Device (client)
Secure World
Request App Integrity Check [version]
Exchange i_key, security_policy
Send policy_answer
Verify policy_answer [i_token]
Request confirmation [c_message]
Verify confirmation

Fig. 9 – Application Integrity Check.

application forgery or falsification for several purposes or attacks, such as in-app bypass. Previous studies also detected whether the application was repackaged, but the proposed routines can be easily bypassed or require significant communication with server Chen et al. (2018, 2015). Therefore, the *TZMon* application integrity protocol communicates with the Secure OS TA rather than the server, as shown in Fig. 9. The application integrity protocol must be performed immediately after the game application starts.

Step 1. request application integrity **.** The *TZMon* library requests the *TZMon* TA for application integrity verification, and the mobile application version information is transferred to use the application hash corresponding to version. The *ses_token* value in the application integrity protocol is generated, as shown in equation (1), by using the *session_key* made after the authentication mechanism. When the *TZMon* TA receives the request, it generates a *salt* and produces a 256 bit *i_key* using the key derivation function (KDF), as shown in equation (2) and (3), where *m_key* is the pre-installed master key in the Secure OS.

$$ses\_token = session\_key \tag{1}$$

$$i\_pwd = version \oplus m\_key \tag{2}$$

$$i\_key = kdf(i\_pwd, salt) \tag{3}$$

Step 2. exchange i_key and security_policy **.** The *i_key* generated in step 1 is transferred to the *TZMon* library through the shared memory; for security, the key is not copied or stored in non-secure memory. The *i_key* is valid only during the protocol session and is safe from other processes in the shared memory.

To check the randomly selected policy in *TZMon_TA*, the security policy encrypted with i_key is decrypted and parsed (See algorithm 2). The security policy is composed of type, scope, comparison target, *and sync timing information*. The signature type means that the signature of the application is transmitted directly to be checked by the TA, and the comparison type indicates that the result of comparison with the comparison target delivered by the *TZMon_TA* is sent. To increase the randomness of the security policy, the scope value is created using the random function provided by ARM TrustZone in TEE and is transferred to REE. The scope value is created in the

Algorithm 2 The security policy process of Application Integrity Protocol.

```
1:  procedure CHECK_SECURITY_POLICY
2:      $answer \leftarrow null$
3:      $policy \leftarrow$ receive_policy($version$)
4:      $type,\ scope,\ comp\_target,\ sync\_timing \leftarrow$ parse_policy($policy$)
5:      if is_signature($type$) then
6:          $answer \leftarrow$ create_signature($scope$)
7:      else
8:          $signature \leftarrow$ create_signature($scope$)
9:          $answer \leftarrow$ compare($signature$, $comparison\_target$)
10:     end if
11:     return $answer$
12: end procedure
```

form of a mask that targets the entire application area to be applied when creating the app hash later. In addition to this, the comparison target may sometimes include an incorrect signature, so the comparison result does not always return a true return value. A false return value may be the right answer at times. Lastly, sync timing information is used in the timer sync protocol and is also used to mitigate attacks on freely manipulating the normal clock. (See Section 3.5)

Step 3. send policy_answer **.** To prevent a reuse attack, the *TZMon* library uses the application binary and *i_key* to generate a hash value (application hash), which is probabilistic rather than deterministic. According to algorithm 2, an answer for each security policy is generated according to the type and scope and transmit to the *TZMon_TA*. Whenever the Integrity Protocol is executed, a different policy answer is made and sent, so it is possible to block reuse attacks and the hooking attacks that modify/bypass check routines.

$$app\_hash = sha256(app\_binary \oplus scope) \tag{4}$$

$$app\_signature = sha256(app\_hash \oplus i\_key) \tag{5}$$

Step 4. verify application signature **.** The *TZMon_TA* loads the *app_hash* stored in the secure storage using the version information, creates the *app_signature* in the same way as the *TZMon* Library, and compares it with the received *app_signature*. If *verify_value* = 0 it means success, with any other value meaning failure.

$$verify\_value = app\_sign \oplus tz\_app\_sign \tag{6}$$

If the signature is verified, an *i_token* is generated from the *app_signature* value, as in equation (7), and transmitted to the *TZMon* library. Because the *app_signature* is probabilistic, the *salt* can be used as an option. In contrast to *i_key*, *i_token* must be used until the end of the mobile game, i.e., after the protocol session finishes, so that it can be copied from shared memory to normal memory. Although this exposes *i_token*, it is only used to prevent bypass and does not affect security, because it is valid only until the game application session closes.

$$i\_token = kdf(app\_signature, salt) \tag{7}$$

Step 5. request confirmation . The *TZMon* library generates and transfers a confirmation message, as shown below, to inform the *TZMon* TA that the previous step was successful. Because the confirmation message does not need to be secret, it uses predefined result message(s) (*predef_message*) from the *TZMon* library and TA.

$$c\_message = mac(i\_token, predef\_message) \tag{8}$$

Step 6. verity confirmation . The TA generates a confirmation message (*tz_c_message*) in the same way as for step 5 and compares the *tz_c_message* with the received *c_message*. If *c_message* is verified, the *TZMon* TA sets *i_flag*, which is used to check whether the mandatory protocol was executed or not, preventing a protocol circumvention attack.

$$verify\_confirm = c\_message \oplus tz\_c\_message \tag{9}$$

Challenging Point 1: How to mitigate hooking of integrity check function? Hackers mainly use the method of hooking the corresponding routine to manipulate the integrity check. Since the integrity check function returns "true" or "fixed value" (e.g., application signature) in most cases, the routine of the function is modified to return the specified value. Therefore, *TZMon_TA* randomly selects the security policy and send it whenever the mobile game app is loaded. This probabilistic policy prevents hooking attacks that modify the routine to use a fixed value. Besides, the randomness of security policy makes attack levels more complicated.

Challenging Point 2: How to mitigate bypass attack? Hackers can perform an attack bypassing the check routine in addition to the hooking attack above. To prevent such an attack, the token chaining mechanism is applied in this study. The token chaining mechanism traces the protocol execution result in *TZMon_TA*, and the *TZMon_library* chains the request result through the token. Subsequently, when requesting to perform a new protocol, *TZMon_library* sends a chained token to check whether it matches the trace information of *TZMon_TA*.

Challenging Point 3: How to mitigate security policy manipulation? Not only can there be an attack that hooks the check routine, but the hacker can also manipulate the security policy or modify the check routine to leak more security policies. This paper shoes both security policies and the check routines that are created and managed in the TEE, making access more difficult by attackers. In addition to this, when the hacker completely bypasses the security policy and check routine created in TEE, it is possible to detect that the check routine or security policy has been bypassed through the token-chaining mechanism as in the challenging point #2 above.

3.3. *Secure update (o)*

In contrast to conventional secure updates Asokan et al. (2018), securely updating the TA is vital in *TZMon*. Notably, the existing research uses the network of the Normal World, so even if the update through the secure channel, when the Normal World is compromised, end-to-end Security is not guaranteed, so the update itself cannot be secured. Therefore, this paper aims to improve the security of the secure update protocol by using the network of the Secure World (socket API of TEE).

Therefore, to transmit the new TA binary when there is an update from the server, it is encrypted using the key generated in the TA, which must also be forwarded to the mobile application, and decrypted by the TA. We must also only attempt a secure update after an integrity check. The reason the secure update is optional is that it can be applied as optional when the update is not required, depending on the mobile game, rather than for technical reasons. Fig. 10 shows the relevant protocols, and each step is described below.

Step 1. request update . The mobile game application sends an update request message to the *TZMon_TA*, and the request message is XORed using *token*. The *TZMon_TA* verifies by checking the received message and version information against *token* and *i_flag* in secure storage.

$$token = mac(ses\_token, i\_token) \tag{10}$$

$$request\_update = version \oplus token \tag{11}$$

If *i_flag* is true, step 1 returns *u_compare*; whereas if *i_flag* is false, it returns false without *u_compare*, because the integrity application mechanism has been bypassed.

$$u\_compare = request\_update \oplus version \oplus token \tag{12}$$

$$(i\_flag == true)\ ?\ u\_compare : false \tag{13}$$

Step 2. send update info . *TZMon* TA creates a network channel with the game server, creates *u_key*, as below, concatenates the *update_info* with version information, and transfers it to the game server. Besides, as mentioned in the challenging point 2 above, information for randomly performing the server-side approach is transmitted from the server. The message entropy delivered from the client to the server is kept similar, and the device was not able to check what information the server uses to verify. Every communication channel with the game server assumes secure protocol such as transport layer security (TLS). If a secure protocol is not available, handshake and message encryption processes should be added between server and CAs. To confirm that the update server is a legitimate update request, TEE on the client device and the update server pre-install the same mac_key. The MAC value is

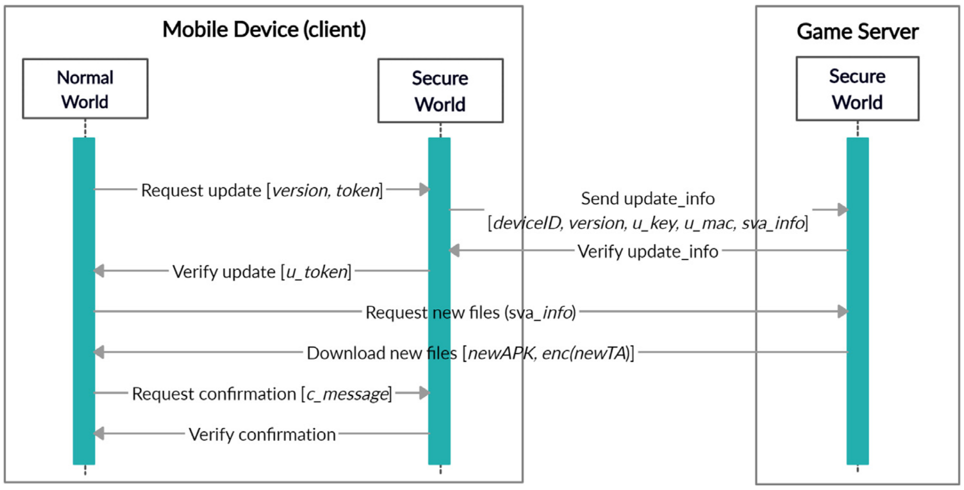

Mobile Device (client)
Normal World
Secure World
Game Server
Secure World
Request update [version, token]
Send update_info [deviceID, version, u_key, u_mac, sva_info]
Verify update_info
Verify update [u_token]
Request new files (sva_info)
Download new files [newAPK, enc(newTA)]
Request confirmation [c_message]
Verify confirmation

Fig. 10 – Secure Update.

generated using this mac_key and transmitted to the update server.

$$u\_pwd = token \tag{14}$$

$$u\_key = kdf(u\_pwd, salt) \tag{15}$$

$$update\_info = deviceID \parallel u\_key \parallel version \parallel [sva\_info] \tag{16}$$

$$u\_mac = mac(mac\_key, update\_info) \tag{17}$$

Step 3. verify update info . First, the update server finds the pre-installed mac_key corresponding to the device ID and verify the received MAC value. Then, the game server confirms the received version information and determines whether an update is required. If required, the new TA binary is encrypted using the received *u_key* and *u_token*, and prepared for forwarding to the mobile application with the APK file. The encrypted TA binary is not delivered directly to *TZMon_TA* but rather to the mobile application, because if the network session between the server and the *TZMon_TA* remains open for a long time, the Android kernel watchdog can force-kill the *TZMon_TA*. Therefore, we maintain server communication with the mobile application rather than the *TZMon_TA* for (potentially) time-consuming operations such as file transfer.

$$u\_mac' = mac(mac\_key, update\_info) \tag{18}$$

$$(u\_mac == u\_mac')?true : false \tag{19}$$

Step 4. verify update . If all preceding steps are executed successfully, the *TZMon_TA* generates *u_token* using *u_key* as the pre-token, which is transferred to the *TZMon* library through shared memory, and it can be copied to normal memory in a manner similar to the integrity check protocol.

$$u\_pretoken = u\_key \tag{20}$$

$$u\_token = kdf(u\_pretoken, salt) \tag{21}$$

Step 5. request new files . The *TZMon* library determines if *u_token* requires an update, and if so requests an update APK and encrypted TA from the game server. As in step 2 above, the device delivers sva_info according to the server-side approach, and the server randomly determines whether cheating is performed using the information. The server-side approach is not the scope of this study, so the detailed server-side approach and random selection policy are not described.

$$update = (u\_token == 0)\ ?\ false : true \tag{22}$$

Step 6. download new files . The game server checks the request for update from the mobile game application and sends the APK and encrypted TA to the *TZMon* library. The encrypted key for TA is *u_key*, which the *TZMon* library does not know, so it is impossible to forge or reverse engineer a new TA key.

$$iv = msb16(u\_token) \tag{23}$$

$$encTA = encrypt(u\_key, iv, newTA) \tag{24}$$

$$mac = hmac(u\_key, newTA) \tag{25}$$

$$new\_files = newAPK \parallel encTA \parallel mac \tag{26}$$

Step 7. request confirmation . The *TZMon* library forwards a confirmation message (*c_message*), the MAC value generated using *u_token* as the key and encrypted TA, to the *TZMon_TA* to indicate that the preceding steps were successful.

$$mac = mac(u\_token, encTA) \tag{27}$$

$$c\_message = encTA \parallel mac \tag{28}$$

***Step 8. verify confirmation* .** The *TZMon_TA* verifies *c_message*, and if no problem is detected, decrypts the encrypted TA using *u_key* and *u_token*. Subsequently, the TA sets *u_flag* to confirm the update completed normally.

$$iv = msb16(u\_token) \tag{29}$$

$$newTA = decrypt(u\_key, iv, encTA) \tag{30}$$

Challenging Point 1: How to create end-to-end secure network channel? Since the secure update is performed through network communication between the client and the server, it is crucial not only for the security of the network channel but also for the end-to-end security. It is challenging to detect manipulated clients even if the secure network protocol such as TLS is used when forging traffic in the communication port of the client end-device. Therefore, in *TZMon*, a secure end-to-end network channel was created by allowing *TZMon_TA* to communicate with the server using the socket API (TEE). Also, since a communication overhead may occur when using a socket API, it is designed that only the essential step is in charge of the socket API, and then performs communication in the normal world (REE).

Challenging Point 2: How to utilize light-weight server-side approach? In general, the updating of mobile game applications is frequently executed, unlike other domain applications. Therefore, when the client communicates with the server during updating, a server-side approach can be used that can supplement the limitations of the client-side approach while minimizing the burden on the server. To this end, when requesting from *TZMon_TA* and *TZMon_Library* to the server, the server-side approach is performed by passing information that conforms to the policy set by the server, such as trace information about behavior. After that, it is possible to check the cheating according to the result of the corresponding operation. However, to reduce the burden on the server, rather than to perform all devices and requests, it is necessary to select the device and update requests randomly considering the computational resources of the server. (Although the *TZMon* research scope does not include the server-side approach, we explicitly write the corresponding step in the protocol.)

3.4. *Data hiding (m)*

To prevent memory manipulation without synchronizing with the server, run-time data must be encrypted to make memory searching difficult. However, if the encryption scheme is exposed through reverse engineering, memory searching difficulty can be reduced. Therefore, even if the data masking scheme is exposed, the *TZMon_TA* randomly generates the crypto-key to ensure difficult memory searching.

There are two modes in the data hiding protocol. In the tee_mode, critical data is stored, and all functions that require changing data, such as modification or deletion, are operated in the TEE, too. In the ree_mode, critical data is stored in the TEE, but all data processing routines are conducted in the REE. The ree_mode makes the memory searching attack difficult by XORing the one-time key generated from TEE when data is calculated in REE, as shown List. A.2 in Appendix B.

The reason for managing separately in two modes is that the data hiding protocol can be selectively applied according to a mobile game profile because of performance overhead. In other words, in the case of the tee_mode, even though performance overhead, the security is much improved. Therefore, we recommend to apply it to a mobile game where there are few performance issues, and security is essential. The ree_mode has some limitations in the runtime memory searching attack, an adversary reverse engineering attack, or delta modifying attack, but it has lower performance overhead compared with the tee_mode. Therefore, it is necessary to select REE mode together with TEE mode to be applied in consideration of the balance between performance and security according to the profile of a mobile game. Therefore, the ree_mode is recommended to be applied only to a performance-critical mobile game. Fig. 11 shows the relevant protocols, and each step is described below. Unlike the ree_mode, after the initial request, the tee_mode sends the command, including operation type and existing data, to verify whether the critical data is forged or manipulated in the Secure World. Therefore, the following steps are explained based on tee_mode.

***Step 1. request initial hiding data* .** The *TZMon_TA* generates a suitable encryption key to hide data in the mobile game application, such as level, cash, life-span, etc. The data is XORed with previously generated *token* values *i_token* and *u_token*, to determine whether the previous mechanisms have been bypassed thereby avoiding exposure of data values. After that, its requests to the TEE by XORing the token to the name of critical data that needs to be hidden.

$$token = token \oplus previous\_token \tag{31}$$

$$init\_msg = data\_name \oplus token \tag{32}$$

***Step 2. verify initial hiding data* .** The *TZMon_TA* randomly generates *h_key* after checking several flags and validating the received message token, and then passes *h_key* and data to the *TZMon* library. The mobile application can then encrypt using *h_key* when referencing data, as shown List. A.2 in Appendix B. Encrypting data will increase memory searching difficulty, and the key can be changed irregularly or every time the application starts, ensuring robustness against reverse engineering attack. The key, *h_key*, is valid during the protocol session and should be managed only in shared memory.

$$h\_pwd = token \tag{33}$$

$$h\_key = kdf(h\_pwd, salt) \tag{34}$$

$$init\_ret\_val = (data \parallel h\_key) \oplus token \tag{35}$$

***Step 3. request hiding data* .** Requests for various operations on hiding data after the initial command must send operation_type, data_name, and data. When the TEE receives the request, it checks that the data and data_name are not forged or manipulated through h_key and responds with the data.

$$msg = (op\_type \parallel data \parallel data\_name) \oplus h\_key \tag{36}$$

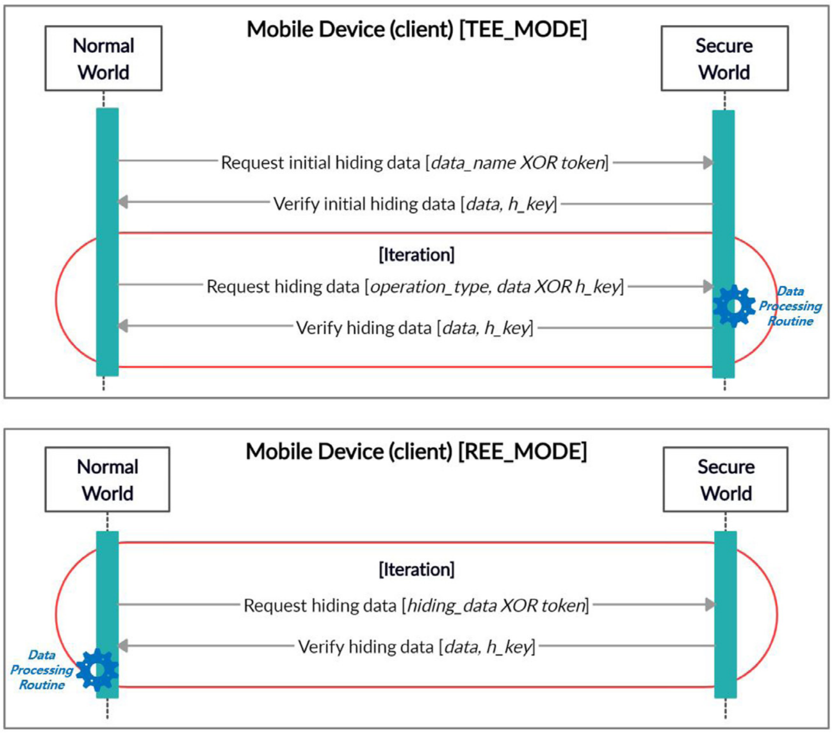

Mobile Device (client) [TEE_MODE]
Normal World
Secure World
Request initial hiding data [data_name XOR token]
Verify initial hiding data [data, h_key]
[Iteration]
Request hiding data [operation_type, data XOR h_key]
Verify hiding data [data, h_key]
Data Processing Routine
Mobile Device (client) [REE_MODE]
Normal World
Secure World
[Iteration]
Request hiding data [hiding_data XOR token]
Verify hiding data [data, h_key]
Data Processing Routine

Fig. 11 – Data Hiding (tee_mode & ree_mode).

Step 4. verify hiding data **.** After verifying the msg received in Step 3, it responds with the XOR operation using old_h_key on the data, as shown in equation (34). As described in the above algorithm 3, new_h_key is created with h_key as a password. After that, it concatenates with hiding data and XORs with old_h_key.

$$h\_pwd = h\_key \tag{37}$$

$$new\_h\_key = kdf(h\_pwd, salt) \tag{38}$$

$$ret\_value = (data \parallel new\_h\_key) \oplus h\_key \tag{39}$$

In contrast to previous protocols, the procedure to generate *h_flag* and *h_token* is omitted because hidden data is in secure storage. Thus, circumvention of this mechanism is not possible. Data is frequently accessed during the game, so a light-weight protocol is essential to minimize performance impacts.

Challenging Point 1: How to hide encryption scheme for critical data? A common method to defend against memory search attacks is to apply an encryption scheme to critical data. However, this method has a problem that the hacker can break the scheme through reverse engineering of the game app. Therefore, hiding data is protected by safely managing the key used in the scheme or changing the key from time to time. (See algorithm 3).

Challenging Point 2: How to utilize behavior data? The game application generates behavior data essentially. Therefore, behavior data is designated as the hiding data, and when executing the game application must execute *TZMon_TA* at least once to prevent attacks that bypass all schemes. The token chaining mechanism plays a role in detecting bypass protocols.

Challenging Point 3: How to make light-weight protocol? Even if the overhead of Data Hiding processing itself is not significant, if the data is frequently used when running the game application, it dramatically affects the performance of the game. Therefore, Data Hiding Protocol should be kept lightweight than other protocols. Therefore, game application can select TEE Mode and REE Mode according to the critical level and apply selectively to essential data rather than to all data. Critical data, such as hit point (HP) value, mana point (MP) value, damage, or money, should especially have a data hiding protocol applied. In addition, when even non-critical data can

Algorithm 3 The hiding_key update algorithm of Data Hiding Protocol.

```
1: procedure HIDING_KEY_UPDATE
2:     if is_init() then
3:         $token \leftarrow$ receive_token()
4:         $h\_pwd \leftarrow$ is_verify($token$)
5:     else
6:         $h\_pwd \leftarrow$ get_old_h_key()
7:     end if
8:     $salt \leftarrow$ random()
9:     $h\_key \leftarrow$ kdf($h\_pwd$, $salt$)
10:    return $h\_key$
11: end procedure
```

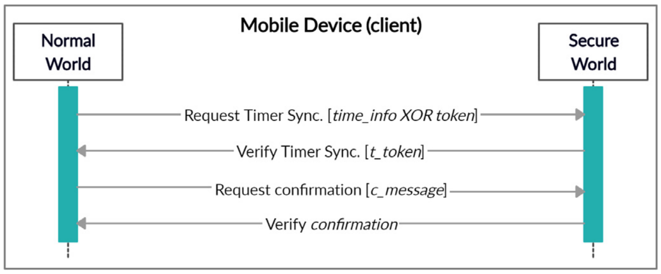

Mobile Device (client)
Normal World
Secure World
Request Timer Sync. [time_info XOR token]
Verify Timer Sync. [t_token]
Request confirmation [c_message]
Verify confirmation

Fig. 12 – Timer Sync.

indirectly affect critical data, the data hiding protocol could also be applied.

3.5. *Timer sync. (o)*

Although retrieving the package name will prevent speed hacking, a more efficient method is to synchronize timer information with a trusted target. Conventional PC games can easily detect speed hacking by timer sync with the server Christensen et al. (2013), but timer sync with the server is inappropriate for mobile games, as discussed Section II-B. Therefore, the proposed *TZMon* system performs timer sync in the *TZMon_TA* using the secure ARM TrustZone clock to eliminate communication with the server. We also use an irregular sync interval to ensure an unpredictable routine run-time. Fig. 12 shows the detailed protocols, and each step is described below.

Step 1. request timer sync . The *TZMon* library sends timer information from the Android OS to the *TZMon_TA* to prevent speed hacking. It generates *token* in the same way as for the token procedure, XORs it with the normal timer, and transfers *t_message* to the *TZMon_TA*.

$$token = token \oplus previous\_token \tag{40}$$

$$t\_message = normal\_time \oplus token \tag{41}$$

Step 2. verify timer sync . *TZMon_TA* first checks the security policy that was created in the application integrity protocol and checks whether the requested timer sync protocol was operated in a normal execution cycle. In addition to this, if the timer sync protocol is requested, according to the access to time-sensitive position data, it will be set to operate in advanced mode. In this case, *TZMon* additionally checks the delta value of the position data. Afterward, the *TZMon_TA* uses the stored flags and token information to generate the *token* in the same way as the *TZMon* library and obtains *normal_time*, XOR *t_message* with *token*. After verifying normal timer information, it validates the information by comparing with TZ timer information from the secure ARM TrustZone clock and checking agreement within a *gap_threshold*, set considering processing overhead. If operating in advanced mode, an algorithm that calculates the ratio (ratio) of the delta value and the secure clock value for the position data managed by the data hiding protocol and compares it with the amount of *ratio_threshold* can be added after the above process to increase the security level. (See algorithm 4)

$$t\_gap = secure\_time - normal\_time \tag{42}$$

$$t\_result = t\_gap < gap\_threshold\ ?\ true : false \tag{43}$$

The *TZMon_TA* then generates *t_token* as follows and transfers it to the *TZMon* library.

$$t\_pwd = token \tag{44}$$

$$t\_token = kdf(t\_pwd, salt) \tag{45}$$

Step 3. request confirmation . The *TZMon* library transfers a confirmation message to the *TZMon_TA* indicating that steps 1 and 2 were successful. Because the confirmation message does not need to be secret, it uses predefined result message(s) (*predef_message*) from the *TZMon* library and TA.

$$c\_message = mac(t\_token, predef\_message) \tag{46}$$

Step 4. verify confirmation . The *TZMon_TA* generates a confirmation message (*tz_c_message*) in the same way as the other protocol and verifies it by comparing with the received *c_message*. If verification is successful, it sets *t_flag*.

$$verify\_confirm = c\_message \oplus tz\_c\_message \tag{47}$$

Challenging Point 1: How to verify by using only reliable data? When the hacker attacks a speed hack, it uses a method of arbitrarily adjusting the speed by hooking the time-related system API. In this case, it is possible to detect speed hacking by comparing the time delta of the secure clock and the normal clock. The comparison with the normal and secure clock is not a perfect mitigation method because the attacker also can forgery the time delta of the normal clock, although the attack complexity increases. Therefore, this paper adds the algorithm that detects speed hack using delta information and secure clock of position (coordinates) information that is traced in the secure world through the hiding data protocol. (See algorithm 4).

Challenging Point 2: How to determine the timing of verifying the clock? A hacker can manipulate the normal clock's delta value, and bypass the delta value when checking it with a specified cycle. Manipulated delta values can be changed to a normal value before the check routine. To prevent this bypass, we added the execution cycle (it changes every time

Table 4 – Mobile Game Applications.

Type	App Name	Line of Code	Protection Type	Protection Data	Access Count
Racing	dodge & chase	27.8K	coordinate,	7EA, (192B)	15(6.2)
Shooting	space shooter	27.7K	health,	6EA, (204B)	14(7.5)
Defense	archer defence	28.1K	level	11EA, (312B)	23(10.8)
Puzzle	blockinger	30.9K		4EA (116B),	12(5.1)

Algorithm 4 The Verify Algorithm of Timer Sync Protocol.

```
1: procedure VERIFY_TIMER_SYNC
2:     t_result ← true
3:     ree_delta ← receive_ree_time_info()
4:     tee_delta ← get_secure_clock()
5:     t_gap ← ree_delta - tee_delta
6:     if check_for_timing_clock() || t_gap ≥ gap_threshold then
7:         t_result ← false
8:     end if
9:     if is_advance_mode(mode) then
10:        pos_delta ← get_position_data()
11:        t_ratio ← pos_delta / tee_delta
12:        if t_ratio ≥ ratio_threshold then
13:            t_result ← false
14:        end if
15:    end if
16:    return t_result
17: end procedure
```

whenever an application is launched) of the timer sync protocol into the security policy, created in the application integrity protocol. That is, the timer sync protocol is performed based on the execution cycle in the security policy. We also added an execution cycle that runs every time the time-sensitive position data managed by the TEE was accessed to prevent an attack that disables the timer sync protocol by acquiring the execution cycle during application execution. In this case, *TZMon* checks the delta value of the normal clock and the secure clock, and the delta value of the position data simultaneously.

4. Implementation and evaluation

In this section, we describe the implementation of a full-featured prototype of *TZMon* and evaluate its performance overhead, effectiveness in cheat detection, and source code modification considering the *TZMon* design challenges discussed Section II-C. Besides, we design a series of experiments to evaluate the effectiveness of TZMon by investigating the following research questions:

RQ1: Can we minimize overhead when the *TZMon* apply to the mobile games? To check the overhead of *TZMon*, we experiment with measuring CPU usage and memory usage in terms of the resource overhead of the mobile device, and measured frame per second and app loading time in the operation overhead aspect of the mobile game. **(Section 4.2)**

RQ2: Is the *TZMon* safe from known mobile game hacking? To check the mitigation effect from *TZMon*'s hacking risk, we identified the known mobile game hacking techniques described in the Table 5 to confirm whether *TZMon* can detect or protect. **(Section 4.3)**

RQ3: Is it easy to apply *TZMon* to mobile games? To check the easy of applying *TZMon*, the amount of source code modification was measured. Additionally, the principles for using *TZMon* were summarized according to the characteristics of each mobile game. **(Section 4.4,** Section 4.5)

4.1. *Implementation and experimental setup*

Mobile game applications are open-source code written in Java with modified applications using Android Studio (v3.4.2). The *TZMon* library and TA are written in C/C++ using opteeSDK (manifest_hikey960_v3.4.0). Android Open Source Project (android_9.0.0) is used as the Normal OS and OPTEE (hikey960_v2.5) as the Secure OS. Furthermore, we use the Hikey960 board supporting Octa-Core (4x Cortex-A73, 4x Cortex-A53) and ARM TrustZone v8 as the mobile device environment.

For evaluation, we selected several representative games for each type, which are classified as racing (dodge&chase Dodge-And-Chase (Accessed: March 2021)), defense (archerdefence Archer Defence (Accessed: March 2021)), shooting (spaceshooter Space Shooter (Accessed: March 2021)), and the puzzle (blockinger Blockinger (Accessed: March 2021), 2048 2048 (Accessed: March 2021)) in the top 30 open-source mobile games. There are two main reasons for this selection. First, we can evaluate several performance overhead according to the characteristics of game types. Second, many different types of mobile games cheats are readily available for our evaluation.

The Table 4 below describes mobile game applications with *TZMon* information used in the experiment. The game app for the test applies all of the *TZMon* protocols, and protection types are composed of coordinate, life, and level (grade) data for all four games. In the protection data field, the number in front of parentheses means the amount of data, and the number in brackets indicates the size of data. Lastly, the number in front of parentheses in the access count field means the maximum access count of protection data for 5 seconds, and the number in brackets indicates the average access count during the experiment time.

We are happy to share all source code of *TZMon* at GitHub TZMon (Accessed: March 2021). Mobile game applications include modifying source code to apply *TZMon*. *TZMon*'s implementation has 9.2K lines of code.

4.2. *Performance overhead*

We evaluate the performance overhead incurred by *TZMon* by measuring CPU utilization, memory utilization, frame rate,

Table 5 – Cheat types and scenarios in the mobile game.

No	Cheat Type	Cheat scenario	Detected&Protected
1	App repacking	Modify mobile apps	App integrity
2	In-App Bypass	Modify mobile apps	App integrity
3	Disable security policy	Secure update without previous protocols	Secure update
4	Secure update of malware	Secure update attempt without TA	Secure update
5	Abusing (Hooking)	Attached process & Memory searching	Partial-detected
6	Abusing (speed hacking)	Customize system APIs	Timer sync.

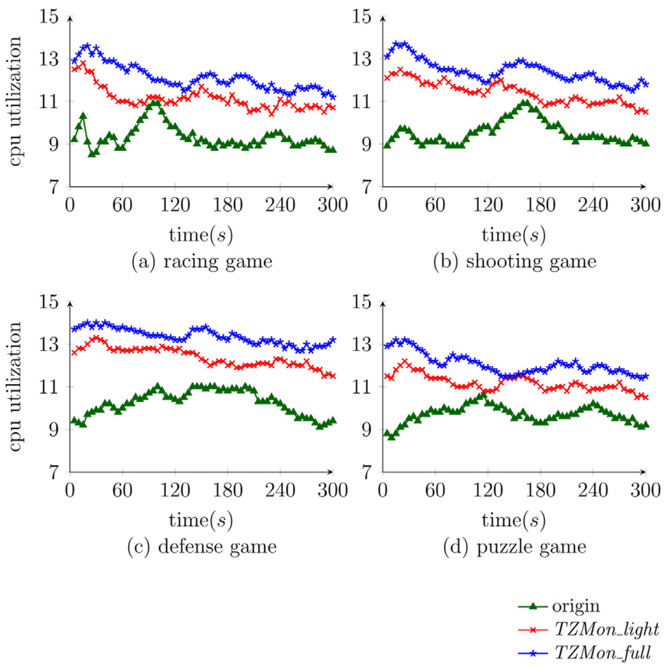


Fig. 13 – CPU utilization; green (origin), red (with *TZMon_light*), blue (*TZMon_full*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

and application loading time in three experimental settings: playing a mobile game 1) without *TZMon*, 2) with *TZMon_light* (including mandatory protocols and ree_mode data hiding), and 3) with *TZMon_full* (including all protocols and tee_mode data hiding). We measure the performance overhead 10 times every 5s at the client-side and report each result during the game play, each game running for 5 minutes. The count of critical data applying the data hiding protocol is three in all types of mobile games. However, since the usage of critical data is not taken into account, performance overhead may differ depending on the mobile game.

In this study, HiroMacro Heo et al. (2015), an auto touch macro program, was used to reproduce the execution scenario. The game was run and measured several times under the same conditions for each game. That is, the scenarios executed while playing the game for five minutes were applied equally to the origin, *TZMon_light*, and *TZMon_full*. It was possible to experiment with the same conditions, except for the application of *TZMon*, by using an auto touch macro program. For reference, we made an execution scenario macro based on the loading time of *TZMon_full*, which has the longest loading time, because a difference may occur in the execution scenario due to the delayed loading time.

CPU overhead . Fig. 13 shows the average CPU utilization (percent) by each game process when running with and without *TZMon*. In the worst case, when using *TZMon_full*, we observe that CPU utilization overhead ranges from 2.8% to 4.2% with an average value of 3.3%. On the other hand, using *TZMon_light*, we observe an average overhead of 1.9%.

In particular, the section (e.g., racing: 240s, shooting: 120s, defense: 120s, puzzle: 150s) in which the gap is small between the CPU overheads of *TZMon_full* and *TZMon_light* means the part which is less or no access to data managed by *TZMon_full* or *TZMon_light*. The section (e.g., racing: 100s, shooting: 170s, puzzle: 110s) in which the gap is small between the CPU overheads of *TZMon* and the original app means the part which is less or no access to data managed by the data protection protocol of *TZMon*.

In the case of Defense Game, the increase rate of CPU utilization is higher than other types of games, because the use of data used for data hiding and the operation frequency is higher than those of different kinds of games. The puzzle game has a low CPU overhead relatively compared with other game applications because of less access and count of critical data. In other words, the overhead of the game can vary depending on the number and frequency of critical data, so

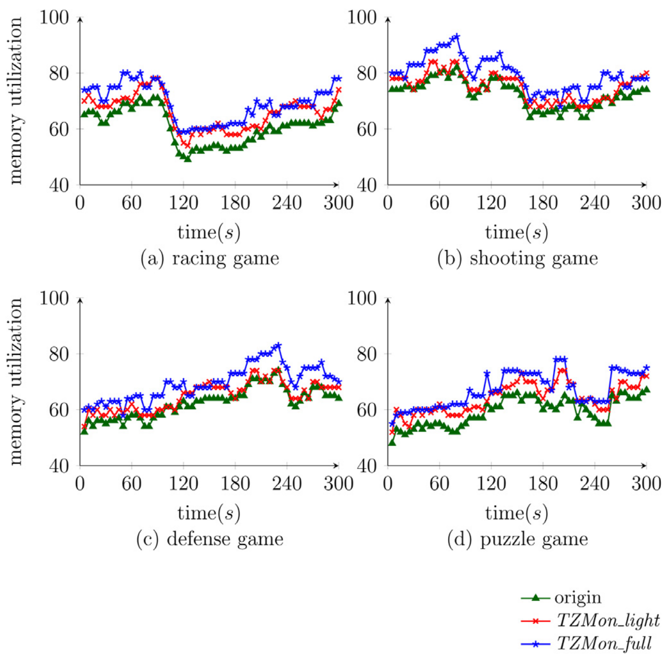


Fig. 14 – Memory utilization; green (origin), red (with *TZMon_light*), blue (*TZMon_full*). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

it must be selectively applied according to the impact risk of critical data.

In all four game types, CPU overhead is high at the beginning and end of the game. The high overhead at the beginning of the game execution is because protocols such as the integrity check and the secure update are executed, and the high cost at the end of the game execution is because of access for saving changes to protected data increases. As such, we can confirm that the CPU overhead usually occurs at the beginning and end of a game, so it does not affect delay or cost during the game run that has a significant impact on usability.

Memory overhead **.** Fig. 14 shows the average memory utilization (MB) for each game process when running with ad without *TZMon*. Using *TZMon_light* and *TZMon_full*, we obtain an average overhead of 4MB and 7MB, respectively. In contrast to CPU utilization, memory utilization does not increase significantly in the initial phase of the game. The memory overhead is uniformly increased because *TZMon*'s shared memory uses fixed-size memory.

The shooting game and defense game has a low memory overhead of *TZMon_light* and origin, and a high memory overhead of *TZMon_full*. This is because the number of protected data managed by *TZMon_light* is not large, and the number of protected data managed by *TZMon_full* is large. On the contrary, the racing game and puzzle game do not have a significant memory overhead of *TZMon_full* and *TZMon_light* because the number of protected data managed by *TZMon_full* is relatively small. Comprehensively with the results of the previous CPU overhead, the defense game has a relatively high CPU overhead, and the puzzle game has a low CPU overhead. This means that the memory and CPU overhead depends on the data amount.

Frame rate **.** In mobile games, frames rendered per second, i.e., the frame rate, is a measure of game performance experienced by the user. These frames are rendered as fast as CPU cycles are available for execution. Thus, the frame rate is one of the critical measures for evaluating the effect of *TZMon*. In the case of FPS, the worst-case performance is more critical than the average index of performance, so the experimental result of FPS used the worst-case performance. Fig. 15-(a) shows that frame rate overhead ranges from 2.7 fps to 3.1 fps with an average value of 2.9 fps. The frame rate of the defense game with *TZMon_full* is 31 fps, which is the worst performance. However, mobile game users cannot perceive the frame rate's degradation if the rate is over 30 fps; thus, the frame rate overhead of *TZMon* is negligible.

As mentioned earlier, it is essential to manage the worst-case performance so as not to fall below the threshold (recommend 30) because FPS is a significant indicator of usability. According to the experiment result, the FPS of the game app with *TZMon* does not go below the threshold.

Application loading time **.** We assume that all protocols except the data hiding protocol operate at the initial function of the mobile game. Due to the characteristics of the game application, execution of operation such as version check and update check is typical, so the initial loading overhead of the game is not sensitive to usability. Fig. 15-(b) shows that the average loading time ranges from 182 ms to 245 ms with an average value of 214 ms.

Although the above result can be considered to be relatively high overhead than the original loading time, the increase in loading time due to the application of *TZMon* is less than 1 second, which is fully acceptable. However, depending on the game, the acceptable loading time may be different. If the ap-

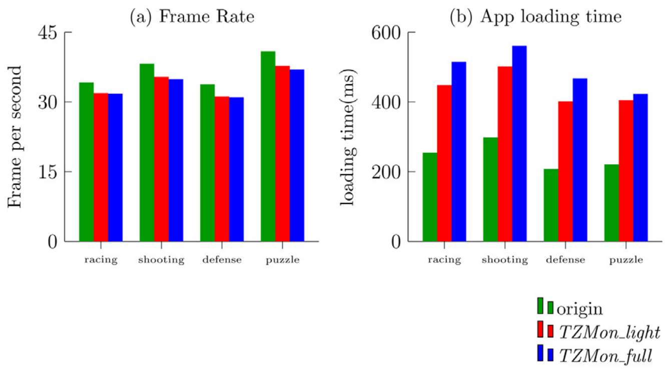


Fig. 15 – Frame Rate and Application loading time.

plication loading time exceeds the adequate time due to the applying of *TZMon*, the loading time can be managed by adjusting when other protocols (e.g., sync timer protocol) except for the integrity check protocol are applied.

4.3. *Effectiveness of the cheat detection and risk analysis*

This section presents an assessment of the effectiveness of *TZMon* in detecting the cheats in mobile games. Table 5 lists six types of cheating used in our evaluation. Application repacking and in-app bypass did not use a cheating tool, but after adding hacked code in the mobile application, we checked whether those protocols could detect an anomaly. *TZMon* detected secure update attempts without the previous protocols such as an application integrity checking. Another type of attack is a scenario in which a malicious application attempts to update by directly accessing the update server. In this experiment, we check if the server detects anomaly by verifying the MAC value when making and sending fake versions, u_key, device ID, and MAC to the update server after implementing malware CA.

Abusing such as hooking is possible in various attack scenarios. Among the possible attack scenarios, the malicious application is hooking the mobile game and attach it to the mobile game process. In case the debugging option is activated, the attache prevention feature is disabled to attach to the process of the mobile game in malware. But even if mobile game is hooked by malware, *TZMon* can detect cheating of protected data. In this experiment, the attache prevention feature is disabled to attach to the process of the mobile game in malware, and the debugging option is activated to facilitate the process attach. In addition, speed hacking verification implemented the cheating scenario by manipulating system time functions to change the delta time.

Risk Analysis. The goal and intention of *TZMon* are to protect cost-effective data, address, and functions, so it is not necessary to protect all things. A forgery of unmonitored data may occur, but usually, these data do not significantly affect the game. Besides, depending on the game genre, since unmonitored data is initialized when the stage changes, it is possible to significantly increase the security of the game application by ensuring integrity for sensitive data that must be maintained at all times. Therefore, *TZMon* was not intended to prevent all types of detoured access (hooking).

In addition to the verification of the cheating scenario for the mobile game applications, we derived additional risk analysis for *TZMon* and countermeasures. The experiment for the above cheating scenario is to check the detection ability of *TZMon* according to the type of the known cheating scenarios. On the other hand, Risk Analysis and Mitigation of Table 6 aim to identify security threats that take into account the characteristics of *TZMon* and to determine how much these threats can be defended.

The biggest threat to the mobile game app with *TZMon* is when a hacker attacks a game app that includes *TZMon_lib*. To disable the integrity check routine first, a hacker can either bypass the routine or hook the function by returning the same result value as the integrity check result. *TZMon* can detect the bypass attack using the token chaining mechanism and randomly use the server-side approach of the server to prevent attacks that bypass the entire protocol in specific scenarios. Besides, to detect hooking attacks, *TZMon_TA* applied a different security policy each time to detect an attack that returned the same value, thereby increasing the complexity of the exploit.

In the case of the Secure Update Protocol, there may be a threat to install malicious game binary that hackers want. However, these attacks use the Socket APIs that communicate directly with the server through the TA, so it is impossible to access from servers other than the authenticated game server. Also, the threat that transmits fake data to the server can be defended because of sending a socket from TEE.

The threat to the timer is performed through hooking of system API, and Timer Sync. Protocol detects through double checks. First, *TZMon* checks by comparing the time delta of the secure clock with the time delta of the normal clock. Also, to prevent the manipulation of the normal clock delta value, *TZMon* detects by calculating the ratio of the delta value of the position data managed by TEE to the secure clock. This protocol can be said to be a significant scenario using TEE in consideration of the characteristics of the mobile game app, and it can be used for hooking detection in similar apps.

Result . *TZMon* detected an integrity check anomaly and a secure update anomaly without an integrity check. In particular, in case of the secure update protocol, when a direct update

Table 6 – Risk Analysis and Mitigation.

Risk	Description	Mitigation
App Integrity	Hooking integrity check routine	Dynamic Security Policy
Secure Update	Replace to malicious game binary	Socket API
Data Hiding	Modify data value	Use secure data on TEE
Sync. Timer	Hooking system APIs	Use secure data and clock on TEE
Bypass protocol	Bypass partial OR all protocols	Token Chaining Algorithm
		Data Hiding (for critical data)
		AID server_side approach
Manipulate network end-point	Send fake data to game server	Socket API

Table 7 – Source code modification.

Game App	LoC	Mobile game source	*TZMon* library	*TZMon* TA
Racing	27.8K	+0.39K (*TZMon* I/F)	+3.2K (JNI)	+5.6K (TA Binary)
Shooting	27.7K	+0.39K (*TZMon* I/F)	+3.2K (JNI)	+5.6K (TA Binary)
Defense	28.1K	+0.42K (*TZMon* I/F)	+3.2K (JNI)	+5.6K (TA Binary)
Puzzle	30.9K	+0.38K (*TZMon* I/F)	+3.2K (JNI)	+5.6K (TA Binary)

request is made from malware to the update server without TA, it confirmed that secure update is not normally executed by detecting anomaly through MAC authentication.

In the abusing attack scenario, after attaching a process using various attack methods, a specific value of the memory is eventually read/write, and a memory search attack is attempted in this case. As a result of the attack, there was no protocol for detecting the hooking process, so it could not be detected, and because the attack scenario did not use the TEE Service, the prevention mechanism through authentication or *TZMon* did not work. However, when the memory search for critical data is performed with encryption value, it is difficult to predict and change the value in the attack scenario accurately. Besides, as the key value used in the calculation was renew for each session, the complexity of the attack was increased.

Finally, in case of speed hacking, when the system time API is changed, *TZMon* could detect the difference between secure time and manipulated time. However, when the system time is changed within the threshold, *TZMon* could not detect speed hacking. Therefore, threshold should be adjusted according to the mobile game.

4.4. *Source code modification*

We measure the amount of source code modification to present portability and ease of use for *TZMon*. Table 7 shows the added or modified lines of code (LOC) to apply *TZMon*. The Table 7 is the value measured without the data hiding protocol. In the case of hiding data, it is meaningless to calculate the quantified value because it varies depending on the type and number of applied data, and the characteristics of the mobile game. However, even if the data hiding protocol is applied to multiple data, code modification is not complex as it can be applied by simple source code modification through the API function proposed by *tzmon*.

Table 7 shows that the difference in *TZMon* Modification according to the size of the Game App is not vast. Source code modification of the mobile game is about 0.4K LOC, which is very simple with *TZMon* library API calls. The modification for *TZMon* can be divided into the code block to add *TZMon_library* and the code block to apply and check the protocol of *TZMon* (See listing A.3). The modification of the Data Hiding Protocol depends on the protection data. On the other hand, since *TZMon_library* uses the same API, it can be easily applied regardless of the genre or size of the game app. Table 7 shows that the same *TZMon_TA* operating in the TEE can be applied regardless of the mobile app as like *TZMon_library*. Therefore, *TZMon* library (3.2K LOC) does not modify game source code, adding the JNI library to the mobile game application. A black-box reuse of source code published in GitHub is possible. Given such limited source code modification, mobile game applications can apply *TZMon* easily.

In summary, source code modification can vary according to the type and number of critical data to apply the data hiding protocol, but this can be applied with low complexity by using *TZMon* API. Also, the rest of the protocols that can be applied independently to the mobile game was proven that *TZMon* could be used with little effort, as shown in Table 7.

4.5. *Guidelines for applying* TZMon

Mobile games can be divided into offline single-player games and online multiplayer games. A stand-alone type game means an offline single-player game that is a game application only executed without any network connection or the help of a game server. According to the game profile, online multiplayer games can be classified into delayed-critical network-based games and graphical performance-oriented games.

To minimize possible performance degradations, we suggest selectively applying *TZMon* depending on mobile game profiles as shown in Table 8. This study evaluated mobile

Table 8 – Guidelines for applying *TZMon*.

Profile Type	Integrity	Update	Hiding	Timer
Stand-alone	Init	△	tee_mode	gameplay
Delay-critical	Init	Init	ree_mode	X
Graphic-performance	Init	Init	ree_mode	Init

games, which are classified as racing, defense, shooting, and puzzle games. Considering performance, however, it is practical to apply *TZMon* depending on mobile game profiles as discussed below.

Stand-alone game. Stand-alone mobile games do not require high performance and are vulnerable to manipulation of data. Given the stand-alone game profile, we suggest executing application integrity protocol at the initial loading time. The tee_mode of data hiding protocol and timer sync protocol are recommended to run during gameplay. Especially, critical data such as hit point (HP) value, mana point (MP) value, damage, or money should apply a data hiding protocol. Finally, *TZMon* runs a secure update protocol by user selection to minimize network overhead.

Delay-critical network-based game. In delay-critical network-based games such as first-person shooter (FPS) games, it is recommended that the ree_mode of data hiding protocol be applied selectively because packet delays affect win or loss of games. Instead, we use the secure network with a TEE socket API for end-to-end security. Delay-critical network-based games communicate with the server frequently. Given this feature, the timer sync protocol is optional because it can be executed on the server. Finally, *TZMon* use application integrity, and secure update protocol at the initial loading time that do not affect network delay.

Graphic-performance-oriented game. In graphic-performance-oriented games, *TZMon* runs the application integrity and the secure update protocol at the initial time as for other game profiles. Graphic-performance-oriented games are required to have high-performance. When the data hiding and timer sync protocols are applied to the data requiring a real-time update, graphic performance degradation occurs. Thus, it is recommended that timer sync protocol is applied once an initial time and the ree_mode of data hiding protocol be applied lightweight during gameplay. Moreover, the encryption key update interval is also a configurable value to optimize performance.

4.6. *Experiment summary*

Overall, we evaluate that *TZMon* incurs on average 3.3% overhead to CPU utilization, 7MB overhead to memory utilization, 2.9 fps overhead to frame rate, and increases application loading time by 214 ms. Given such a small set of overhead value, *TZMon* is a performance-efficient solution for mobile games. Moreover, *TZMon* detected all cheating types effectively, discussed in Section II-B, in the mobile game. Despite the effectiveness of *TZMon* in cheating detection, the amount of game source modification is only 0.4K LOC. This study proved through experiments that *TZMon* could enhance mobile game security without large-scale data analysis on the server-side or the network-side, and without performance degradation.

5. Discussion and limitations

Although *TZMon* can defend against many threats that can occur with mobile games, there are still avenues for future improvement. This section presents the limitations and their workarounds for the current *TZMon* and discusses enhancements for future work.

Secure network transmission. The first potential threat is an insecure network transmission between game clients and game servers. If mobile devices are compromised easily, attackers will succeed in applying network exploits even though the devices are using a secure protocol such as TLS. On the other hand, a secure socket API (working in the TEE) support end-to-end security because all procedures of the network are run under the Secure OS.

To provide secure network transmission between game clients and game servers, this study ported the current secure network library to the Secure OS. However, the secure update protocol uses the secure network only when making an encryption key because porting the secure network library is a non-compliant with the TEE specifications Global Platform (Accessed: March 2021). According to the TEE specifications, a secure socket API (working in the TEE) of OPTEE will be provided. If we use the official socket API, then we expect that several threats residing in the network-side can be resolved (although not yet resolved in this study).

Preventing TZMon bypass. The second potential threat is an attacker disabling either all or specific *TZMon* protocols. Because the *TZMon* library is running on the Normal OS, attackers can avoid security protocols. To detect a disabled *TZMon*, we use the result chaining mechanism in this study, which is to prevent protocol skip. In other words, when every protocol is starting, *TZMon* always checks the result flag of previous protocols. Moreover, if a powerful attacker can bypass entire protocols of *TZMon*, we can use minimum server-side approach without gaming performance degradation, as described below. *TZMon* writes the encrypted log file (especially network API or function call records as a piece of evidence available for hooking or tamper detection) to local storage and sends the log file when doing so does not interfere with gameplay, such as when ending or updating the game. If an abnormal user is found, the server can detect that based on this audit log.

Cheating detection. Cheating tools including game bots represent the third potential threat. As discussed in section II-B, various cheating tools are available that can be exploited to mobile games. As the prevention method using the blacklist method has a lot of methods for bypass, it is necessary to research the prevention based on cheating behavior. Of course, the server-side approach is the most effective, but to reduce the server-side burden and more effective prevention, research for basic filtering methods such as static rule-based methods is essential. Especially, game bots represent another potential threat. Game bots using macros are also increasing for mobile games. Recent PC games use big-data or AI analysis on the server-side for game bot detection, but server-side analysis is not sufficient for mobile games. Server-side burden should be reduced by applying detection techniques on the client-side van Kesteren et al. (2009), Gianvecchio et al. (2009). Aside from auto-play features officially provided by the game application, all user inputs to the mobile game must be made through the touchpad or keypad. Therefore, game bots can be detected on the client-side with high accuracy by checking whether game inputs are entered periodically using the Arm TrustZone trusted user interface (TUI) Li et al. (2014a).

Extensibility beyond the ARM TrustZone. The last potential threat is the inability to apply *TZMon*. The ARM TrustZone is a dominant security framework, supported by more than 97% Android smartphones. However, *TZMon* should also apply to security framework other than the ARM TrustZone to ensure scalability. For example, an Android mobile device may use Secure Element Gemalto (Accessed: March 2021) rather than the ARM TrustZone; and iPhones use the Secure Enclave TheiPhoneWiki (Accessed: March 2021), which is similar to the ARM TrustZone, but not identical. Therefore, future *TZMon* implementations should extend to other security frameworks.

6. Related work

We review related research regarding trusted services based on the ARM TrustZone in mobile devices for several domains, and game cheating countermeasures for server, network, and client-side.

6.1. *ARM Trustzone based trusted services*

The ARM TrustZone is a widely used security framework for mobile devices and has been widely studied. TrustZone-based trusted services are divided into secure storage, authentication, TUI, and OS enhancements.

Secure storage. An essential class of trusted service is to provide secure storage to safely store files in the mobile device even if the OS was compromised. *TZMon* supports secure storage for stored game data confidentiality and integrity. Secure storage is the most important function to securely store file elements, and many studies have considered it. For example, González and Bonnet (2014) and Gonzalez and Bonnet (2014) only operated virtual file systems in the secure OS to reduce file management overhead, and encrypted file elements were stored in the Normal World. DroidVault Li et al. (2014b) proposed an isolated data protection manager for Android that operated in a trusted domain for secure file management. Critical files were encrypted and signed by DroidVault before being stored on the untrusted Android OS, ensuring the files could be securely managed even if the Android OS was compromised. A similar approach has been proposed Hein et al. (2015) with a more scalable secure storage solution and without requiring the Android OS.

These studies aimed to provide data confidentiality, hence they provided secure file management and all operations were in the trusted environment. In contrast, the proposed *TZMon* system secure storage provides only data integrity and confidentiality to minimize overhead during runtime. Thus, *TZMon* can distinguish between run-time and stored data, and hence can minimize data overheads while ensuring confidentiality and integrity using secure storage as required.

Authentication. *TZMon* uses the Secure World crypto function to protect runtime data. Most crypto functions as trusted services focus on authentication. For example, the Android key-store Google (Accessed: March 2021) keys that can be used as authentication factors (pattern, fingerprint, password, etc.) are generated in a trusted container (keystore) and controlled in the key master abstraction layer module. TrustOTP Sun et al. (2015) generates a one-time password based on time and a counter using the trusted crypto hardware intellectual property (IP) managed by Secure World. TEEBAG Balisane and Martin (2016) proposed a device-based authentication scheme Rijswijk-Deij and Poll (2013), implementing two-factor authentication using the ARM TrustZone and using information the user knows and has.

Previous studies used the Secure World crypto function primarily for authentication. However, *TZMon* used the crypto function to guarantee runtime data integrity and confidentiality, rather than authentication.

Trusted user interface (TUI). The trusted service also provides a TUI to ensure a secure input path. The TUI is one of the most challenging domains and is lightly studied compared to other services. For example, Li et al. (2014a) proposed a user interface secure from DDoS for fake display and input peripherals. The FIDO Coombs (2015) white paper proposes that the TUI should be applied to biometric authentication user interfaces. Trustonic Coombs (2015), a commercial secure OS, proposed a TUI using SAMSUNG KNOX. This study also described a user interface using the TUI to detect game bots in the client-side as future work.

Operating system enhancement. Some trusted services provide users with trusted functions by extending OS features. For example, ASSURED Asokan et al. (2018) proposed a protocol to update a firmware or service application in IoT devices if no security issues were found by remote attestation of the network domain that the server and peripheral devices belonged to. In contrast, the proposed *TZMon* system update approach minimizes server communication, once the installed application is verified through the proposed protocol.

A framework has been proposed Brasser et al. (2016) that can securely use remote machine resources to solve this problem for devices with restricted space. TrustDump Sun et al. (2014) implemented a memory monitoring module in Secure World to allow safe memory dump even if the OS crashed or was compromised. MIPE Chang et al. (2017) and kernel protection Azab et al. (2014) can also verify kernel integrity for Normal World. Reference Williams (2015) proposed a model for detecting rootkits, and Jang et al. (2018) proposed the Private Zone model to simplify using the ARM TrustZone to address environmental constraints where the ARM TrustZone could be used. A recent system Kwon et al. (2019b) enabled secure monitoring with low overhead for several purposes using the ARM 64-bit architecture. However, this study focused on utilizing the ARM TrustZone, rather than expanding OS functionality.

TrustShadow Guan et al. (2017) proposed a security platform that can be applied without modification of security-critical applications using ARM TrustZone. A novel method of TrustShadow helped increase the security level by applying it to various applications, including IoT devices. Because this study presented an application-level security solution regarding mobile game characteristics, *TZMon* can coexist with TrustShadow. That is, unlike TrustShadow, that is a security platform for general applications, *TZMon* proposed security solutions specialized for cheating scenarios, which mainly occur in mobile games, so it is still a useful method to help improve the security level of mobile games.

6.2. Anti-Cheating in mobile game

Mobile game anti-cheating studies can be categorized as server, network, and client-side approaches. Although users and developers are having difficulties due to mobile game security issues Kim and Kum (2013), few studies have considered these issues. This section reviews the literature, expanding to include online games to check anti-cheating techniques available for mobile games.

Server-side approach. Server-side anti-cheating techniques primarily focus on detecting gold farming or game bots. When a user plays a game, play logs are transmitted to the server and stored, and abnormal behavior can be identified by analyzing the log. Server-side detection techniques can be applied as often and whenever desired, with minimal exposure of the detection technique to adversaries. Therefore, if the game has a long lifecycle, there is enough log data, and real-time detection is not required, the server-side approach is the most effective method.

Previous studies Han et al. (2015); Kang et al. (2016, 2013); Lee et al. (2015) have also considered game bot detection using data mining or machine learning approaches applied to log data, such as user movement, chatting, transaction, battle/hunting, login, and action sequences. Reference Kwon et al. (2013) detected an entire network structure of gold farmers and gold farmer characters by tracing the trading network formed by the gold farmers to minimize collateral damage. Reference Chun et al. (2018); Lee et al. (2018b) analyzed game black markets using large scale gold farming group (GFG) analysis. Other approaches J. Woo, A. R. Kang, and H. K. Kim (2013); Ki et al. (2014); Woo et al. (2012) distinguished game bots from normal users by analyzing malicious behavior diffusion in the game.

Network-side approach. Network-side anomaly detection captures network traffic between client and server, and analyzes relevant traffic information, including traffic extension, network response, and traffic interval times. Reference Chen et al. (2009) analyzed traffic regularity and explosiveness on the network-side, extracted them as a feature, and classified game bots in Massively Multiplayer Online Role-Playing Game (MMORPG) using a statistical approach. Time intervals and packet sizes were extracted by Hilaire et al. (2010), and the decision tree algorithm was applied to detect game bots in MMORPG.

However, server and network-side anti-cheating methods are based on data-driven analyses using big-data. Although data-driven analysis may be suitable for some long-lifecycle games, it is unsuitable for most mobile games.

Client-side approach. A client-side anti-cheating study in Hong et al. (2006) extracted window event sequences, including client-side keyboard and mouse input sequences, and detected game bots using the decision tree algorithm. It was confirmed in Han et al. (2013) that game bots were detected with over 90% accuracy using assembly code generation rules. In Bauman and Lin (2016) a security framework to enhance PC game security using Intel SGX was also proposed. However, these studies were specific for PC games, and these approaches would be challenging to apply to mobile applications due to detection overhead.

Mobile game client-side research has been relatively active compared to research with PC games. For example, Google authenticates users through the license validation library (LVL) Google (Accessed: March 2021) to prevent illegal replication of applications. However, LVL can be circumvented simply by manipulating the executable code that identifies the message received from the market application. Obfuscation solutions, such as ProGuard GuardSquare (Accessed: March 2021), DexGuard nGenSoft (Accessed: March 2021), and ArXan ARXAN (Accessed: March 2021), prevent internal logic vulnerabilities from being exposed, but cannot wholly avoid analysis code through reverse engineering because they cannot change control flows. To prevent application repackaging, Chen et al. (2018) proposed self-defending code techniques to generate keys using the application's checksum and detect repackaged applications by encrypting part of the application with that key. AppInk Zhou et al. (2013) and DroidMOSS Zhou et al. (2012) proposed detection of repackaging using watermarking code in the original application and the application's instruction sequence, respectively. However, these methods require additional network traffic with the server, and detection code may be easily disabled or circumvented. In contrast, the proposed *TZMon* mechanism executes modules in Secure World, making it difficult to disable or circumvent the detection logic.

Summary. Overall, traditional countermeasures have shortcomings such as the potential to bypass countermeasures (client-side), delays in packet transmission (network-side), and gaming performance degradation (server-side). Moreover, packet delays in the FPS genre game are critical because it affects win or loss of games, and verifying every dataset that comes from the client on the server-side incurs significant gaming performance degradation, which can cause users to leave the game. In this study, we proposed *TZMon* based on the ARM TrustZone to enhance mobile game security and minimize performance overhead.

7. Conclusion

We have presented *TZMon*, a security mechanism based on the ARM TrustZone, to improve the security level of mobile games. Many related works have introduced an application that utilizes ARM TrustZone. Still, in the case of mobile games, there have been no cases where ARM TrustZone has been applied to mobile games because of a lack of security design considering the characteristics of mobile games and concerns about performance reduction.

To the best of our knowledge, this is the first case where the ARM TrustZone has been applied to mobile games. The ARM TrustZone is used to protect the integrity and confidentiality of game code and data. The proposed *TZMon* is performed in the Secure World, which makes an attacker unable to bypass security protocols. For evaluation, we implemented full-featured proposed protocols. We evaluated *TZMon* with open-source mobile games of several types. Table 5 shows that *TZMon* can detect several cheating techniques for mobile game applications, which are discussed in Section II-B, with negligible overhead.

We are happy to share the source code of *TZMon* on GitHub. Our implemented *TZMon* can be utilized to easily integrate other mobile games for game industry practice.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A

A1. *Delta time*

Delta time is used to maintain game speed across devices with varying performances, as shown in List. A.1. Delta time is usually used for updating variables based on the elapsed time since the game last updated, which will vary depending on the speed of the computer, and how much work needs to be done in the game at any given time.

A2. *Sample code for data hiding*

The mobile application can encrypt using key when referencing data, as shown List. A.2. Encrypting data will increase memory searching difficulty, and the key can be changed irregularly or every time the application starts, ensuring robustness against a reverse engineering attack.

A3. *Example source code of applying* TZMon

The modification for *TZMon* can be divided into the code block to add *TZMon_library* and the code block to apply and check the protocol of *TZMon*.

```
int  lastTime, curTime;

while  {
     curTime  =  time();
     deltaTime  =  curTime  -  lastTime;
     lastTime  =  curTime;

     character.xPos  +=  speed  *  deltaTime;
}
```

List. A1 – Sample code for Delta Time.

```
//  Normal  Version
class  Character  {
     int   HP;

     setHP( int   HP) {
          this.HP  =  HP;
     }

     int   getHP() {
          return   this.HP;
     }
}

//  Encrypt   Version
class  Character  {
     int   HP;
     //  created   by  TA
     int   h_key  =  0x10101010;

     setHP(int  HP)  {
          this.HP  =  HP  XOR  this.h_key;
     }

     int  getHP()  {
          return  this.HP  XOR  this.h_key;
     }
}
```

List. A2 – Sample code for Data Hiding.

```

public static final boolean tzmonUse = true;
static {
    System.loadLibrary("tzMonJNI");
}

public native boolean tzmonInitKeyNFlag();
public native boolean tzmonCheckAppHash();
public native boolean tzmonSecureUpdate();
public native boolean tzmonAbusingDetection();
public native boolean tzmonSyncTimer();

public native boolean tzmonHidingSetup();
public native int tzmonGetHKey(String data);

public static void alertDialog(final Activity a,
                               String message)
{
    AlertDialog.Builder alert = new AlertDialog.Builder(a);
    alert.setCancelable(false);
    alert.setPositiveButton("Exit!",
                    new DialogInterface.OnClickListener()
    {
        @Override
        public void onClick(DialogInterface dialog,int which)
        {
            dialog.dismiss();
            a.moveTaskToBack(true);
            a.finish();
            android.os.Process.killProcess(
                                android.os.Process.myPid());
        }
    });
    alert.setMessage(message);
    alert.create().show();
}
```

List. A3 – Example code for applying *TZMon*.

CRediT authorship contribution statement

Sanghoon Jeon: Conceptualization, Methodology, Software, Writing - original draft. **Huy Kang Kim:** Writing - review & editing, Supervision.

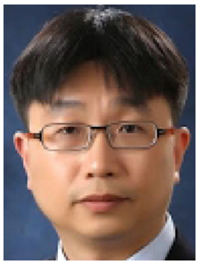
REFERENCES

2048, Accessed: March 2021. GitHub.
AndroidRank, Accessed: March 2021. Android application ranking: All applications.
Archer Defence, Accessed: March 2021. GitHub.
ARM, Accessed: March 2021. Introducing ARM TrustZone.
ARM Security Technology. In: Technical Report. Building a Secure System using TrustZone Technology. ARM; 2009.
ARXAN, Accessed: March 2021. arXan.
Asokan N, Nyman T, Rattanavipanon N, Sadeghi A-R, Tsudik G. Assured: architecture for secure software update of realistic embedded devices. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 2018;37(11):2290–300.
Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G, Ma J, Shen W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2014. p. 90–102.
Balisane RA, Martin A. Trusted execution environment-based authentication gauge (teebag). In: Proceedings of the 2016 New Security Paradigms Workshop. ACM; 2016. p. 61–7.
Bauman E, Lin Z. A case for protecting computer games with sgx. In: Proceedings of the 1st Workshop on System Software for Trusted Execution. ACM; 2016. p. 4.
Blockinger, Accessed: March 2021. GitHub.
BlueStack Systems, Accessed: March 2021. BlueStacks.
Brasser F, Kim D, Liebchen C, Ganapathy V, Iftode L, Sadeghi A-R. Regulating arm trustzone devices in restricted spaces. In: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services. ACM; 2016. p. 413–25.
Burow N, Zhang X, Payer M. Sok: Shining light on shadow stacks. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE; 2019. p. 985–99.
Chang R, Jiang L, Chen W, Xiang Y, Cheng Y, Alelaiwi A. Mipe: a practical memory integrity protection method in a trusted execution environment. Cluster Comput 2017;20(2):1075–87.
CheatWare, Accessed: March 2021. GameCIH.
Chen K, Zhang Y, Liu P. Leveraging information asymmetry to transform android apps into self-defending code against repackaging attacks. IEEE Trans. Mob. Comput. 2018;17(8):1879–93.
Chen KT, Jiang JW, Huang P, Chu HH, Lei CL, Chen WC. Identifying mmorpg bots: a traffic analysis approach. EURASIP J Adv Signal Process 2009;2009:3.

Chen Q, Wang J, Wang Y. An online approach for detecting repackaged android applications based on multi-user collaboration. In: 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom). IEEE; 2015. p. 312–15.
Christensen J, Cusick M, Villanes A, Veryovka O, Watson B, Rappa M. In: Technical Report. Win, lose or cheat: The analytics of player behaviors in online games. North Carolina State University. Dept. of Computer Science; 2013.
Chun S, Choi D, Han J, Kim HK, Kwon T. Unveiling a socio-economic system in a virtual world: a case study of an mmorpg. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web; 2018. p. 1929–38.
Coombs R. Securing the future of authentication with arm trustzone-based trusted execution environment and fast identity online (fido). ARM White paper 2015.
Counterpoint, Accessed: March 2021. Global Smartphone SoC Market Crossed US$8 Billion in Q3 2017, A Third Quarter Record.
dex2jar, Accessed: March 2021. dex2jar.
Dodge-And-Chase, Accessed: March 2021. GitHub.
Fernandes E, Aluri A, Crowell A, Prakash A. Decomposable trust for android applications. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE; 2015. p. 343–54.
GameACE, Accessed: March 2021. What Is The Best Language For Game Development?
GameGuardian, Accessed: March 2021. GameGuardian.
Gemalto, Accessed: March 2021. embedded Secure Element.
GENYMOTION, Accessed: March 2021. GenyMotion.
Gianvecchio S, Wu Z, Xie M, Wang H. Battle of botcraft: fighting bots in online games with human observational proofs. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. p. 256–68.
Global Platform, Accessed: March 2021. TEE Specification.
Gonzalez J, Bonnet P. Tee-based trusted storage. Itus Skriftserie 2014.
González J, Bonnet P. Versatile endpoint storage security with trusted integrity modules. Itus Skriftserie 2014.
Google, Accessed: March 2021. Android Key Store.
Google, Accessed: March 2021. App Licensing: License Validation Library.
Guan L, Liu P, Xing X, Ge X, Zhang S, Yu M, Jaeger T. Trustshadow: Secure execution of unmodified applications with arm trustzone. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services; 2017. p. 488–501.
GuardSquare, Accessed: March 2021. ProGuard.
Guilbon, J., Accessed: March 2021. Attacking the ARM's TrustZone.
Han ML, Kang B-T, Kim HK. A game bot detection using the assembly code generation rules of the x86 processor. Communications of the Korean Institute of Information Scientists and Engineers 2013;31(7):41–6.
Han ML, Park JK, Kim HK. Online game bot detection in fps game. In: Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems-Volume 2. Springer; 2015. p. 479–91.
Hein D, Winter J, Fitzek A. Secure block device–secure, flexible, and efficient data storage for arm trustzone systems, 1. IEEE; 2015. p. 222–9.
Heo GI, Heo CI, Kim HK. A study on mobile game security threats by analyzing malicious behavior of auto program of clash of clans. Journal of the Korea Institute of Information Security & Cryptology 2015;25(6):1361–76.
Hilaire S, Kim HC, Kim CK. How to deal with bot scum in mmorpgs?. In: 2010 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2010). IEEE; 2010. p. 1–6.
Hong S, Kim H, Kim J. Identification of auto programs by using decision tree learning for mmorpg.. Journal of Korea Multimedia Society 2006;9(7):927–37.
intercede. In: Technical Report. Trusted application management as a service. intercede; 2019.
J. WooA. R. Kang, and H. K. Kim. The contagion of malicious behaviors in online games. ACM SIGCOMM Computer Communication Review 2013;43(4):543–4.
Jang J, Choi C, Lee J, Kwak N, Lee S, Choi Y, Kang BB. Privatezone: providing a private execution environment using arm trustzone. IEEE Trans Dependable Secure Comput 2018;15(5):797–810.
Jang J, Kang BB. Securing a communication channel for the trusted execution environment. Computers & Security 2019;83:79–92.
Jang JS, Kong S, Kim M, Kim D, Kang BB. Secret: Secure channel between rich execution environment and trusted execution environment.. In: NDSS; 2015. p. 1–15.
Kang AR, Jeong SH, Mohaisen A, Kim HK. Multimodal game bot detection using user behavioral characteristics. Springerplus 2016;5(1):523.
Kang AR, Woo J, Park J, Kim HK. Online game bot detection based on party-play log analysis. Computers and Mathematics with Applications 2013;65(9):1384–95.
van Kesteren M, Langevoort J, Grootjen F. A step in the right direction: Botdetection in mmorpgs using movement analysis. In: Proc. of the 21st Belgian-Dutch Conference on Artificial Intelligence (BNAIC 2009); 2009. p. 129–36.
Ki Y, Woo J, Kim HK. Identifying spreaders of malicious behaviors in online games. In: Proceedings of the 23rd International Conference on World Wide Web. ACM; 2014. p. 315–16.
Kim HK, Kum YJ. Mobile game security issue in android. Journal of The Korea Institute of Information Security and Cryptology 2013;23(2):35–42.
Kwon D, Seo J, Cho Y, Lee B, Paek Y. Pros: light-weight privatized secure oses in arm trustzone. IEEE Trans. Mob. Comput. 2019.
Kwon D, Yi H, Cho Y, Paek Y. Safe and efficient implementation of a security system on arm using intra-level privilege separation. ACM Transaction on Privacy and Security (TOPS) 2019;22(2) 10:1–10:30.
Kwon H, Mohaisen A, Woo J, Kim Y, Lee E, Kim HK. Crime scene reconstruction: online gold farming network analysis. IEEE Trans. Inf. Forensics Secur. 2017;12(3):544–56.
Kwon H, Woo K, Kim H-c, Kim C-k, Kim HK. Surgical strike: A novel approach to minimize collateral damage to game bot detection. In: Proceedings of annual workshop on network and systems support for games. IEEE Press; 2013. p. 1–2.
Lee E, Kim B, Kang S, Kang B, Jang Y, Kim HK. Profit optimizing churn prediction for long-term loyal customer in online games. IEEE Transactions on Games 2018.
Lee E, Woo J, Kim H, Kim HK. No silk road for online gamers!: Using social network analysis to unveil black markets in online games. In: Proceedings of the 2018 World Wide Web Conference on World Wide Web; 2018. p. 1825–34.
Lee E, Woo J, Kim H, Mohaisen A, Kim HK. You are a game bot!: Uncovering game bots in mmorpgs via self-similarity in the wild. In: NDSS; 2016. p. 1–15.
Lee J, Lim J, Cho W, Kim HK. In-game action sequence analysis for game bot detection on the big data analysis platform. In: Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems-Volume 2. Springer; 2015. p. 403–14.

Li W, Ma M, Han J, Xia Y, Zang B, Chu C-K, Li T. Building trusted path on untrusted device drivers for mobile devices. In: Proceedings of 5th Asia-Pacific Workshop on Systems. ACM; 2014. p. 8.
Li X, Hu H, Bai G, Jia Y, Liang Z, Saxena P. Droidvault: A trusted data vault for android devices. In: 2014 19th International Conference on Engineering of Complex Computer Systems. IEEE; 2014. p. 29–38.
Li Y, Wang M, Zhang C, Chen X, Yang S, Liu Y. Finding cracks in shields: On the security of control flow integrity mechanisms. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security; 2020. p. 1821–35.
Linaro, Accessed: March 2021. OPTEE.
Lucky Patcher, Accessed: March 2021. Lucky Patcher.
Martins Bratuskins, Accessed: July 2020. Mobile game lifespan is shortening, but there's a way to extend it.
nGenSoft, Accessed: March 2021. DexGuard.
Nox Ltd., Accessed: March 2021. NoX Player.
Pinto S, Santos N. Demystifying arm trustzone: a comprehensive survey. ACM Computing Surveys (CSUR) 2019;51(6):130.
Rastogi S, Bhushan K, Gupta B. Android applications repackaging detection techniques for smartphone devices. Procedia Comput Sci 2016;78:26–32.
Rijswijk-Deij Rv, Poll E. Using trusted execution environments in two-factor authentication: comparing approaches. Open Identity Summit 2013.
Samsung, Accessed: March 2021. Device-side Security: Samsung Pay, TrustZone, and the TEE.
Sebastian SA, Malgaonkar S, Shah P, Kapoor M, Parekhji T. A study & review on code obfuscation. In: 2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave). IEEE; 2016. p. 1–6.
Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C. Google android: a comprehensive security assessment. IEEE Security & Privacy 2010;8(2):35–44.
Shen D. Exploiting trustzone on android. Black Hat USA 2015.
Skylot, Accessed: March 2021. jadx.
SOPHOS, Accessed: March 2021. When Malware Goes Mobile.
Space Shooter, Accessed: March 2021. GitHub.
Sun B, Liu J, Xu C. How to survive the hardware assisted control flow integrity enforcement. Blackhat Asia 2019 2019.
Sun H, Sun K, Wang Y, Jing J. Trustotp: Transforming smartphones into secure one-time password tokens. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM; 2015. p. 976–88.
Sun H, Sun K, Wang Y, Jing J, Jajodia S. Trustdump: Reliable memory acquisition on smartphones. In: European Symposium on Research in Computer Security. Springer; 2014. p. 202–18.
TheiPhoneWiki, Accessed: March 2021. Secure Enclave.
Trustonic, Accessed: March 2021. Trustonic Secured Platform.
TZMon, Accessed: March 2021. GitHub.
Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. ACM; 2014. p. 447–58.
Williams J. Inspecting data from the safety of your trusted execution environment. BlackHat USA 2015.
Woo J, Kang AR, Kim HK. Modeling of bot usage diffusion across social networks in mmorpgs. In: Proceedings of the Workshop at SIGGRAPH Asia. ACM; 2012. p. 13–18.
Woo J, Kim HK. Survey and research direction on online game security. In: Proceedings of the Workshop at SIGGRAPH Asia. ACM; 2012. p. 19–25.
Zhang N, Sun K, Shands D, Lou W, Hou YT. Truspy: cache side-channel information leakage from the secure world on arm devices.. IACR Cryptology ePrint Archive 2016;2016:980.
Zhou W, Zhang X, Jiang X. Appink: watermarking android apps for repackaging deterrence. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM; 2013. p. 1–12.
Zhou W, Zhou Y, Jiang X, Ning P. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on Data and Application Security and Privacy. ACM; 2012. p. 317–26.

Sanghoon Jeon received his MS degree in Digital Media Communication Engineering from Sungkyunkwan University in 2011. He received a BS degree in Computer Science from Korea University in 2006. Since March 2019, he has studied for a doctor's degree in the School of Cybersecurity, Korea University. He is a security engineer at Samsung Advanced Institute of Technology (SAIT) of Samsung Electronics (2006 ~ present). His research interests include malware analysis, embedded system security, and data driven vulnerability detection.



Huy Kang Kim received his PhD in industrial and systems engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2009. He received an MS degree in industrial engineering from KAIST in 2000. He received a BS degree in industrial management from KAIST in 1998. He founded A3 Security Consulting, the first information security consulting company in South Korea in 1999. Also, he was a member and the last leader of KUS (KAIST UNIX Society), the legendary hacking group in South Korea. Currently he is a professor in School of Cybersecurity, Korea University. His recent research is focused on solving many security problems in online games based on the user behavior analysis. Before joining Korea University, he was a technical director (TD) and a head of information security department of NCSOFT (2004 ~ 2010), one of the most famous MMORPG companies in the world.