AppIS: Protect Android Apps Against Runtime Repackaging Attacks

Lina Song†, Zhanyong Tang†*, Zhen Li†, Xiaoqing Gong†, Xiaojiang Chen†, Dingyi Fang†, Zheng Wang‡
†School of Information Science and Technology, Northwest University, Xi'an, China
‡School of Computing and Communications, Lancaster University, UK.

***Abstract*—Apps repackaged through reverse engineering pose a significant security threat to the Android smart phone ecosystem. Previous solutions have mostly focused on the detection and identification of repackaged apps. Nevertheless, current app anti-repackaging services can only protect applications at a coarse level and have significant performance overhead. These approaches can neither meet the performance requirements of Android nor achieve fine-grained protection against cumulative attack[1] at the same time. Specifically, these solutions rely on a fix-structure detecting engine and then will execute the same path at different times, which lead to the whole protection performs poorly when faced with dynamic cumulative attack, which is typical in real-world attack.**

This paper introduces the AppIS, a reinforced anti-repackaging immune system, that is robust to app-repackaging attack scenarios. Unlike past work, which mostly focuses on simple protection only from just one respect, our design exploits an interlocking guarding net with time diversity for the tamper-proofing of Android applications. The intuition underlying our design is that a dynamic and static combining method can provide a multi-level protection for the codes, core algorithm and sensitive data. We analyze and classify the existing threats on Android platform and furthermore abstract then model the repackaging attack scenarios. We then adapt a random controller used by the dispatcher to randomly construct guarding net with different structure every time. We have built a prototype of our design using Java Native Interface cross-layer calling mechanism for performance requirement. Results from a deployment of AppIS on three kinds of popular apps demonstrate that the new design can prevent our apps from cumulative attack without extra performance cost.

I. Introduction

How many times have you found that your mobile phones are installed useless bundled software by default but you only download a legitimate one? Unfortunately, that legitimate one was repackaged. Protecting apps from repackaging like this occasion is also a major problem in the official marketplaces (e.g., Google Play and Apple Store) as well as third-party alternative Android marketplaces, where repackaged apps are hosted. The industry is seeking a solution to this problem and recently trys to adopt an online security service for app reinforcement. For example, Bangcle, a new mobile app security service provider, is able to provide the app developers with security assessment, application reinforcement, channel monitoring and other services. Current app anti-repackaging services, however, can only provide a coarse-grained checksum protection but will greatly reduce the performance because of the insertion of a lot of other code, which is still inadequate in resisting cumulative attack. In addition, there are many other potential threats in the process of app uploading. For example, some attackers may use cumulative attacks to get the execution control flow and then get the key code or algorithm by constantly debugging trace. At the same time, developers also hope that their apps are strong enough to deal with the cumulative attack in the absence of any performance cost. Fine-grained anti-repackaging protection with high performance is also indispensable for some application marketplaces like official Android markets and other third-party alternative marketplaces.

App anti-repackaging protection has recently received much attention. There are two main research directions in this field, i.e., detection and recognition of infected apps, as well as reinforcement before being released to markets. Some researchers have implemented these methods. However, while the detection and recognition of repackaged apps is a belated protection which seriously damages the interests and enthusiasm of the majority of app developers, the current app reinforcement can only address the symptoms temporarily just one time, but cannot cope with sophisticated cumulative attack. Indeed, these attacks are widely existed in real life.

This paper introduces AppIS, a scalable and effective countermeasure for anti-repackaging, which is an app reinforcing framework like biological immune system of humans. In line with common practice in application protection, AppIS employs added security units as guards with interlocking relationship between each other. To better cope with cumulative attacks, both static and dynamic mechanism should be invoked here. The challenge however is how to design a guard net with time diversity, which means constructing different net structures at different times without extra performance overhead.

Unlike past approaches, which mostly focuses on considering the complexity of protecting scheme to protect apk file while ignoring the high performance overhead brought to Android, AppIS exploits an interlocking dynamic-dispatching guard net of time diversity, whose net structure is produced from static defense net, invoking more efficient C/C++ codes in native layer via Java Native Interface(JNI), to achieve multi-level protection and high performance of dual purpose for the tamper-proofing of apps. Specifically, our system deploys a

*Corresponding author. Email address: zytang@nwu.edu.cn

[1]In this paper, we define the attack process that requires the attacker to repeatedly initiate attack for accumulating attack lessons as the *cumulative attack*.

dispatcher to use a random controller to randomly construct guard net with different structures. That is to say, constructing the different execution path. By doing this, it can make the protected app execute in different path at different times. There is no doubt that AppIS will cost an attacker more time and energy to complete a successful attack, because he can not accumulate the attacking knowledge under the protection of dynamic guard net. Hence, AppIS can greatly increase the intense of protection in resisting cumulative attack.

To illustrate the creation and spread of infected apps caused by repackaging attack, Fig. 1 shows an actual scenario of infected apps' creation and diffusion. In general, developers create apps and submit them to official markets without solid protection (①). Anyone can gain access to these apps, including adversaries (②). Past experience shows, most android apps themselves do not have enough protection so that they are vulnerable to attack by an attacker with reverse engineering experience. In Section II, we will show the real attack scenarios through several mainstream applications (e.g. Temple Run, a hot game). An attacker releases these infected apps to the third party application market(③). In real life, due to the improper security regulation of app market, it is easy for users to be ignorant of downloading the repackaged version of the legal app(④). What is worse, in order to spread these infected apps and lure users to download them, adversary also trends to process them by using sociology and psychology. So developers need to strengthen their own applications (⑤), which can protect not only the interests of their own, but also the interests of users.

So how can we implement the anti-repackaging immune system to resist cumulative attack with high performance to prevent the generation and dissemination of malicious apps? To do so, we need to adopt a dynamic and static combining method as well as Java Native Interface (JNI) cross-layer calling mechanism, which can provide apps with multi-level protection against cumulative attacks without extra performance cost. AppIS eliminates the high probability of infection by raising the immunity of the app itself. An intuitive way to carry out this straightforward idea is to add some secure units, Guards we called. These guards are pre-deployed into the codes and will be sequentially executed to detect and make timely response to the modification of app when running. In this case, any infecting behaviors, including modification and injection to app code or data, could be detected and handled by the app itself. However, this intuitive approach has two limitations: Firstly, all the guards are related to the execution path and will be invoked when running the app. Thus, the more guards, the higher the security, but also the lower operating efficiency, which is a serious conflict especially on the performance-first platform like Android. Secondly, the layout of guards in app is fixed and static so adversary could find and disable this protection by Cumulative Attack [1] in an accepted time. To address above limitations on scalability, efficiency and robustness, AppIS allows more guards without increased influence on performances by using the JNI cross-layer calling mechanism. Additionally, the built-in random dispatcher of

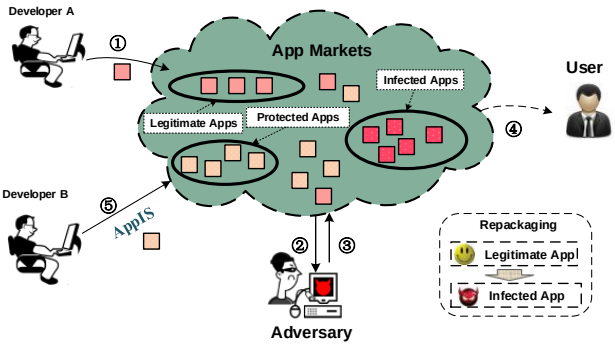


Fig. 1: The Scenario of infected apps creation and spread: Adversary downloads the hot applications from markets and creates infected versions after repackaging, then spreads their APKs to various alternative markets and lures the users to download and run them.

AppIS allows to randomly construct the layout of embedded guards for each app, which can enforce flexible security to arbitrary apps by raising the difficulty of repackaging.

In summary, this paper makes the following contributions:

- To leverage the exploitations of current vulnerable app ecosphere on Android, we devised two realistic attacks on real apps, which can bypass the coarse protection and repackage the popular apps.
- We designed AppIS as a practical solution that would protect the app without making any changes to the underlying Android architecture, making AppIS easy to adopt.
- The results of the quantitative evaluation show that AppIS is a strong enough protection scheme. We measured its security on app and conducted thoroughly evaluations on app storage cost, launch delays, memory use, etc., which shows that AppIS, compared with other approaches, is more effective and has lower memory and runtime overhead.

II. Background And Related Work

In this section, we present related work in the following category:

Detection of repackaged app on Android platform: To isolate the user from repackaged apps, research recently mainly focused on the detection and identification of repackaged apps. Yajin Zhou [2] found most of malware coming from repackaging attack. DNA Droid, a detection framework developed by J Crussell, can find the repackaged version by comparing the code similarity of two apps. [3] Zhou Wu [4] also put forward a repackaging detection mechanism, unlike the former, it was mainly based on checking the coupling relationship between modules instead of app comparison. Admittedly, repackaging detection and identification, can block the spread of repackaged version, and alleviate the trend of app tampering. Moreover, many researches were dedicated to detecting malicious code [5]–[10] in app market because one of the main purposes of repackaging is to generate malware.

Software tamper proofing: To prevent attackers from illegally tampering with apps and abusing the key information, research in this field often follows the detecting-responding process.Typically, static and dynamic tamper proofing are two important techniques. In static protection, confusion is often used to increase the reverse difficulty,which can safeguard key information in program to some extent, such as control flow obfuscation [11], lexical confusion [12], etc. While in dynamic field, Integrity verification is often used. And checksum, Hash value [13]–[15], signature, etc. are indicators of integrity verification. Besides, a software tamper-proofing model based on the idea of guarding net, first proposed by Hoi [16], is also a critical dynamic tamper proofing technique. In this model, the guarding net is formed by embedding multiple security units as guards into the program to achieve self-protection. Different guards are designed to complete different tasks, for example, some guards are responsible for detection, while others are used for response. Unlike other schemes, these guards can protect each other, so there is no isolated guard..

App protection on mobile terminal: In official Android SDK, Google provides developers with ProGuard for preliminary code confusion, increasing the complexity of code logic. Additionally, Android system can verify the identity of app developer with signature verification. Signature is useful for malware or attack detection. [17]–[19].These two protections, however, can only provide a primary protection and it's easy for attackers to bypass. Some online reinforcement services emerge as the times require, such as DexGuard abroad, Bangcle at home. They can provide security services including shell protection, encryption, dynamic loading and anti-debugging technology, greatly increasing the protection intensity. But their shortcomings can not be underestimated. First, although these protection have joined Integrity Verification (or IV, for short) module, these check points are still lack of self-protection and can not hide them very well, which cannot defense cumulative attack as well as effectively prevent the verification module from bypassing or destruction. Secondly, this add-on manner brings tremendous performance overhead to the protected app, including startup time consumption, runtime performance cost and space overhead, etc.What's more, because of the online manner, developers need to upload the original app, which may cause security risk about developers' copyright and privacy when uploading.

III. Attack Model

Repackaging an application refers to that attacker modifies then redistributes the legitimate app. At present, repackaging attack can be classified into three categories according to different attack purposes as follows:

Malware: In this attack case, attackers usually embed malicious code into the legitimate apps for malicious purpose, and then these normal apps become a Trojan or virus host.

Here we analyze a malicious repackaged app bundled with the Trojan DroidKungFu[2] to understand the process of repackaging, as shown in Fig.2(a). Attackers usually download the legitimate APK, then decompile it and analyze the *smali* code to find an appropriate place to inject DroidKungFu payload. Then modify the corresponding configuration file On the other hand, the bulk of malicious payload will be placed in the *smali* or *libs* directory. Eventually, attackers repackage the modified app and sign it again for redistribution.

[2]DroidKungFu is one of the most widely spread, the most influential Trojan on Android platform.

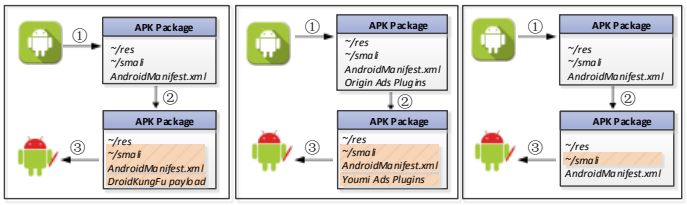


Fig. 2: Three types of repackaging attack according to different attack purposes.

Fake Advertisement: In this scenario, the reasons why attackers repackage apps are as follows.

1) While playing hot games, they cannot tolerate the annoying in-app ads so they remove them.
2) Malicious authors repackage apps to replace original ads or embed new ads to "steal" or re-route ad revenues driven by interests.

Fig.2(b) shows the injection of Youmi ads into a normal app without any ads before. The attack process is similar to the one of malware, but to display fake ads on the interface the former usually needs to modify the apk layout file while the latter needs to hide their behavior as well as avoiding being perceived, so they cannot change the host interface.

In-App Purchase Crack: For "free" use of privilege service of apps instead of paying for use, attackers either crack these apps through repackaging and bypass the authentication module, or modify the pay logic to be "free", as shown in Fig.2(c).

IV. AppIS Overview

AppIS is a reinforced anti-repackaging system that can provide a multi-level protection for Android applications, much safer than the traditional mechanism.

A. Key Ideas

Following a common practice in software protection, AppIS leverages the traditional guard net to provide multi-level safeguard for apps. AppIS's infrastructure also includes a static net consisting of several guards (or safety units with defending functions) to furnish basic security services. Further, AppIS adds a time-diversity module as a value-added function to cope with cumulative attack.

To reinforce an app, at a high level AppIS goes through the following steps:

- AppIS analyzes an app to find out all the functions to be protected, and then sets up some security guards to perform different tasks.

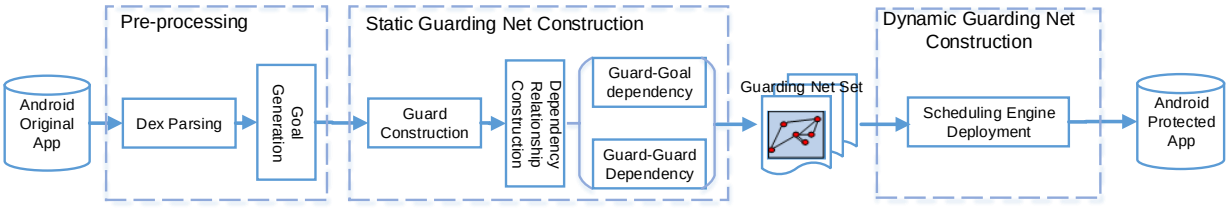


Fig. 3: The overall architecture of AppIS, which is comprised of three main stages, including *pre-processing*, *static guarding net construction* and *dynamic guarding net construction*: The basic idea of AppIS is to embed dynamically-scheduling guarding net into original app.

- AppIS constructs dependency relationship among these guards to form a static guarding net.
- AppIS constructs different executing paths at different times through a scheduing dispatcher. This can form a static and dynamic combining guarding net to achieve multi-level protection.

B. Overall Architecture of AppIS

The overall architecture of AppIS is shown in Fig.3, in which you can see there are mainly three stages.

In the *app pre-processing* stage, we parse the *DEX* or *APK* for app to be protected for generating its corresponding funtion call graph, then choose some key API as protection target, named Goal. In the *static guarding net construction* stage, firstly, we construct some guards according to the guard template, then build the dependency relationship to form a guarding net, which includes the Guard-Goal dependency and Guard-Guard dependency. We choose different guards to build diversified dependency relationship for security, then generate a collection of guarding net. In the *dynamic guarding net construction* stage, we introduce a scheduling engine to dynamically select different guarding nets. In ordinary protecting schemes, there can be a large number of vulnerable points for cumulative attack. This type of attack is pretty common in real life. In next section, we describe AppIS's dynamic guarding net to deal with serious cumulative attack. In the end, a protected app with dynamic guarding net is generated.

V. Constructing Static Guarding Net on Android

The static guarding net was originally used in software protection in PC platform, which is designed as a sophisticated structure in the form of a network of interlocking security unit, not a single security module, to work together for preventing malicious attacks. In this section, we focus on how to transfer this kind of protection mechanism to Android platform, that is to say, how to construct a static guarding net on Android platform.

Static guarding net made up of many safety units can be used for software protection, like in PC platform. Such an approach however is just adopted in the mature platform, which has not been implemented on the new Android facility yet. We plan to program several safety units, or guards we called, to do certain security tasks. Meanwhile, not single unit but a network of these guards is used in our guarding framework in order to reinforce the protection of each other

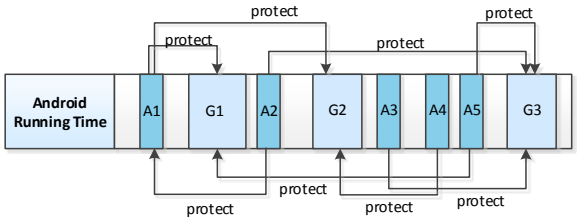


Fig. 4: Memory layout of the guarded program: G denotes the target API to be protected, and A denotes corresponding guard.

by creating mutual-protection. In order to make this idea works well on Android, we need to construct the guarding net step by step.

To visually understand this idea, let us again consider the simple possible guarding scenario shown in Fig.4. Fig.4 shows the memory image of the protected app, in which there are three security-sensitive regions as attack goals, *G1*, *G2* and *G3*. And they are protected by our preset guards *A1*,...,*A5* in an interlocking manner. For example, the vulnerable crack point *G1*, is guarded by *A1*, which is also protected by another guard, *A2*. Once AppIS finds that the *G1* in the program is being cracked, *A1, A2* and a sequence of guards will be invoked as a chain reaction, which greatly increases the difficulty of attack.

In this simple scheme, guard can prevent an attacker from attacking the object directly. However, due to the relative isolation of the guard, the lack of security of the guard itself, for the attacker, you can locate and attack guard to make it fail, and thus affect the protection of the goals. Therefore, how to increase the protection of the guard to improve the safety of its own is a challenge need to solve for improving the reliability of the program.

A. Guards construction

In AppIS, we design a interlocking network of safety units(or guards) embedded within an app. Each guard is deployed to safeguard certain goals, commonly the key code, resources or data. So the first nut for us to crack is how to construct the guards well. Given the fact that most code in Android application is mainly programmed with Java, so the Java code segment is the most important protection target.

Intuitively, Android program is mainly developed in Java code, so we can use Java to achieve guards in the source code,

for example, Fig.5 is a template of the guard, it verifies the integrity of the goals to provide protection. It will dynamically obtain the checksum of the goals and compare it with the stored value in advance. If the two are different, it means the current program is suffering from a heavy repackaging attack. And the guards will respond to this, for instance, terminate the program directly.

```
    Guard:
1   calcHahs = guard.getHashofGoal(goalId);
2   preHashValue = guard.getPreHashValue(goalId);
3   if(calcHash!=preHashValue)
4     System.exit(1);
```

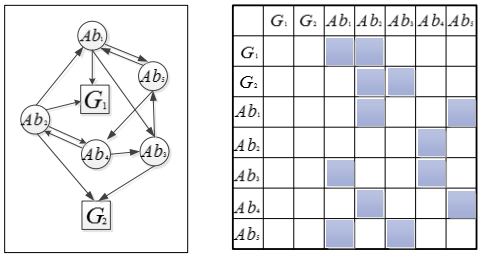
Fig. 5: Template of guard.

The security of guard itself, however, should be also taken into consideration. Hence, the guard should monitor each other and protect each other. We have designed several guards with the same structure, which are composed of the triggering module in Java layer, the main function module in native layer and the key information module. Taking into account that the Java layer is easy to hack, the implementation of main function module of guards is written in C/C++ language, and besides the trigger module will then call the *JNI* interface, thus this cross-layer calling mechanism can enhance the strength of guards against attack. Consequently, we designed the following three guards.

- ***J_Guard***: *J_Guard* is a special code snippet mainly used to protect user defined Java code. Its triggering module is implemented in Java. The *J_Guard*s will be embedded into a Java function to bind the Java function for the purpose of protection, and these guards will be executed when the Java function is running, then the main function module written in C/C++ will be triggered. The main function module checks the integrity of Java function through hash calculation. In addition, the key information module is used to record the key information related to the guards, and it will be initialized when constructing the guarding net.
- ***N_Guard***: *N_Guard* is used to protect other safety guards. It mainly aims at the protection of native function. Its triggering module written in Java will trigger the main function module in native layer. Different from *J_Guard*, the *N_Guard* is designed for native function protection.
- ***D_Guard***: *D_Guard* is a safety unit which is primarily responsible for the reinforcement of the *DEX* executable file on Android. Its original design basis is the fact that all the trigger modules in guards are stored in *DEX* file, so the safeguard of the *DEX* file is the safeguard of the trigger module in guards.

B. Guarding Net Construction

In practice, it is infeasible to deploy a series of single guards to provide unilateral reinforcement protection for Android apps. AppIS aims to achieve multidimensional protection with a interlocking guarding net. In other words, once an app is tampered with, even if the guard is cracked, other guards can still be triggered to take a sequence of protective actions. In this way, all the guards form a complete multi-level protecting barrier.



(a) Dependency Graph (b) Dependency Table

Fig. 6: (a)(b) shows the dependency relationship between Goals and Guards

To visually understand this idea, let us again consider the dependency graph between guards in Fig.6. Fig.6(a) shows a dependency graph between guards while Fig.6(b) shows a dependency table, which explains the main idea in a totally different form. As you can see, *G1* and *G2* are protected by other guards, $Ab_1, \ldots, Ab_5$. This intricate mesh structure is sufficient to deal with ordinary attacks, because even the guard like Ab_1 is attacked, its adjacent guards will still work and notice this attack, and then turn to protect the Ab_1.

Now we formally construct a static interlocking guarding net, where all of the guards are organized into a mutual protection mode to strengthen the reinforcement. Next we will introduce the corresponding guarding net construction algorithm. Then we start by explaining its potential safety weakness. After that, we will present AppIS's further optimization against this weakness.

C. Guarding Net Construction Algorithm

When constructing a relationship between guards in a guarding net, a series of constraints must be met, as follows:

1) Each Goal is protected by at least one guard J_Guard;
2) Each guard is protected by at least two guards (one J_Guard and one N_Guard);
3) When the program is executed, the guard that performs the protection function must be able to execute in a timely manner near the execution time of the protected node.

For the first condition, the Goal in guarding net is generally a custom Java function in Android application, so the Goal should be associated with at least one guard(J_Guard) which can protect Java function.

For the second condition, through the use of JNI programming, the code portion of each guard comprises a trigger module in Java layer and the main functional modules in Native layer, so in order to prevent guard code from tampering, each Goal requires at least two guards for protection, one is J_Guard, which is responsible for protecting the guard's trigger

module, the other is N_Guard, responsible for protecting the main function module of guards.

For the third condition, each Goal or Guard needs to consider the response time of the protection function when choosing its own guard for protection. The so-called response timeliness means that when the protected node is tampered with, the corresponding guard can monitor and respond to the attack in a timely manner. Therefore, you should select the guard near the execution of the protected node as the guard.

Through the detailed analysis of the above constraints, the main algorithm of the guarding net construction is as shown in Algorithm 1, 2. As you can see, Algorithm 1 illustrates the association algorithm between protection target G and J_Guard, while Algorithm 2 shows the association algorithm between these guards to form a complicated guarding net. In Algorithm 1, there are two scenarios. In Scenario 1, when the number of Goals is less than the number of J_Guard, first get the calling chain with length L of the Goal, then in the calling chain, randomly select a function other than the Goal as a binding function, and finally generate the corresponding *gL* table entry. Scenario 2 is similar to Scenario 1, but the difference lies in that if there are duplicate entries in the binding function in each entry, the binding function will be re-selected for replacement. In Algorithm 2, choose two guard in S, one is Node, the other is Node1, firstly get the calling chain callChain with a length of 2L + 1 centered on Node, then get the calling chain callChain1 with a length of 2L + 1 centered on Node1. Finally, if callChain has an intersection with callChain1, the node1 is set to a one-way protection association of the Node, and the ID of the Node1 is assigned to the Node's oid.

Algorithm 1 Association algorithm between protection target G and guard J_Guard

Input:
$FILE$: Java function call graph description file of the protected application
$gArray[]$: A user-defined array of Goals
N: Number of user defined J_Guard
L: Initial length of the user-defined call chain
Output:
gL: J_Guard information list,a vector {Goal ID, Java layer function ID bound to the trigger module}

$M = len(gArray[])$;
if $M <= N$ **then**
  for $(i = 0; i < N; i++)$ **do**
    $chainArray[L] =$
    $getCallChain(FILE, gArray[i\%M], L)$
    $bindMethodID =$
      $selectBindMethodByRandom(chainArray[],$
        $random(L))$;
    $gL[i] = \{gArray[i\%M], bindMethodID\}$
    while $(i >= M)\&\&(gL[x].[0] = gist[i][0])\&\&(gL[x].[1] = gL[i].[1]))$ **do**
      $gL[i].[1] =$
        $selectBindMethodByRandom(chainArray[],$
          $random(L))$;
    end while
  end for
end if
if $M > N$ **then**
  for $(j = 0; j < M; j++)$ **do**
    $chainArray[L] = getCallChain(FILE, gArray[j], L)$
    $bindMethodID =$
      $selectcBindMethodByRandom(chainArray[],$
        $random(L))$
    while $(j >= N)\&\&(isRepeat(gL[j\%N], bindMethodID))$ **do**
      $bindMethodID =$
        $selectBindMethodByRandom(chainArray[],$
          $random(L))$
    end while
    $gL[j\%N] = add(gL[j\%N].[1], bindMethodID)$
  end for
end if

Algorithm 2 Association algorithm between guards

Input:
S: Guards collection
$FILE$: Java function call graph description file of the protected app
L: Initial length of calling chain
Output:
$RList$: Protection association list between guards, a vector {node ID, Guard ID for protecting Goal, Guard ID for protecting other guards }, denoted as {id, iid, oid}*/

while $Node.oid < 1$ **do**
  $callChain = getCallChain(FILE, Node, L)$
  $callChain1 = getCallChain(FILE, Node1, L)$
  if $isIntersect(callChain, callChain1)$ **then**
    $Node.oid = Node1.id$
  end if
end while

D. Safety Weakness: Not Against Cumulative Attack

A static guarding net can only coarsely protect app from repackaging, and there are still some security risks. The most prominent safety weakness is that if an attacker attack a specific app continuously, even if other guards can make some timely response, an attacker is still able to accumulate a certain degree of attack knowledge in each attack process. In the end, he will use the accumulated knowledge of the attack to successfully crack your application. The process is called cumulative attack.

Cumulative attack, refers to the attacker through repeated attacks to obtain the accumulation of attack information. The process of attack on an app is essentially a process of accumulation and extraction of information by one or more reverse attacks. For those apps with complex logic, it is difficult for an attacker to get enough knowledge through just one-time execution of the procedure. Usually, they need to attack repeatedly till they get enough knowledge to crack the app. It is only when the accumulated knowledge exceeds a certain threshold that they have completed a successful attack.

VI. Dynamic Guarding Net: An approach against cumulative attack

To defend against cumulative attack, we further explore the mesh structure of our AppIS, by attaching a randomized scheduling engine. Therefore, every time the attacker runs the protected app, the scheduler will invoke different guards to construct a totally different guarding net with different path. Each time the application is executed, the structure of the guarding net changes, so we call this kind of guarding net with time diversity.

Time diversity was firstly presented by Christian Collberg, and he believed that the time or space diversity is the necessary condition to ensure software suffer lasting protection [1], [20]. The so-called time diversity refers that the program after

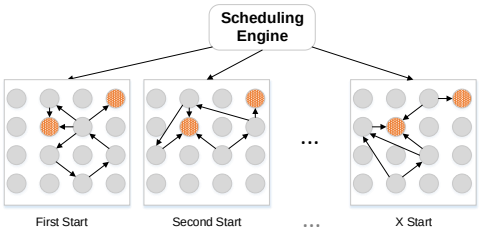


Fig. 7: Dynamic guard network construction via scheduling engine: Each time the app runs, the scheduling engine randomly selects a guarding net structure that is completely different from the last boot from the guarding net set generated from the *static guarding net construction* phase.

protection will execute different paths in different time [21].

Therefore, we can combine the conservation ideas of time diversity to effectively combat the cumulative attack. Time diversity in AppIS is reflected in that when the protected program is executed at different times, the guarding net structure is entirely different in the actual execution. When the application detects an attack withdrawal, the guarding net structure will change after app restarting, so that the previous attack knowledge is no longer valid, so the attacker can not accumulate enough attack knowledge to crack the application. Due to the dynamic scheduling of the time diversity module, the scheduling engine will select a completely different guarding net structure every time the program is executed, so no matter how many attacks the attacker had launched before, the previous attack information can no longer be used for a new round of attacks. Therefore they cannot accumulate either attack information or attack progress and the attack failed.

A dynamic guarding net is shown in Fig.7. The scheduling engine is responsible for randomly selecting one structure from the guarding net collection. Therefore, every time you run the app, there will be a totally different mesh structure constructed in the protected app, that is to say, the guard net is changing dynamically, so called dynamic guarding net.

VII. A Future Optimization Based on Guarding Net

In order to further enhance the protection strength, we design a series of security modules integrated in a daemon process in our AppIS, mainly three. The following is a detailed description of these three modules.

- **Dynamic encryption and decryption module**: In order to prevent the attacker through the static analysis to obtain a collection of guarding net, you need to set up dynamic encryption and decryption module. The guarding net is usually stored in ciphertext in the program, and will be performed a temporary decryption when needing, finally the net will be encrypted for storage again after protection.
- **Environment detection module**: Normally users run an app on a real Android machine directly, only some of the attackers will run it in the simulator for debugging. And what is more, attributes of real machine and the simulator is different in some ways. Based on this cognition, we design an approach to identify the simulator by checking the properties of the running environment.
- **Anti-debugging module**: Static analysis and dynamic debugging are two important means of reverse engineering. So in order to protect the main process of AppIS from attackers debugging analysis, we add an anti-debugging module to perform anti-debugging protection for the scheduling engine. In this paper, this module is mainly implemented based on the Ptrace mechanism. Ptrace is a system call provided by Linux system to make it possible that a parent process can access and manipulate a child process. And the Android kernel is based on the Linux kernel, so it inherits the Ptrace mechanism. Ptrace mechanism only allows just one process to debug another process. At present a lot of debugging tools adopt the ptrace principle to carry on the debugging process. So we can let our daemon process and main process add Ptrace to each other so that other debugger cannot attached to the main process.

VIII. Evaluation

In this section, we will focus on the relevant settings for evaluating the security and performance of AppIS to validate its design and implementation. Firstly, we will give the experiment setting and then we will use several case studies to evaluate the effectiveness of the methodology. Finally, we will demonstrate AppIS's good compatibility by deploying the system in different Android platforms.

A. Experimental environment

Our testbed is a personal computer with windows7 system, equipped with different versions of Android prototype system. To evaluate our design and implementation, we downloaded several non-protected apps covering almost every category from F-Droid, an open-source application repository. Meanwhile, we also developed our own app for test. The app collection is as shown in TABLE I.

TABLE I: Information about App Collection Set.

Apk Name	Size	Function Description	Sample Source
CoolReader.apk	1525k	A e-book reader app	Third-party
CrackMe.apk	44k	A login authentication cracking app	Third-party
NISLContact.apk	1311k	A communication tool for lab staff	Independent writing
2048.apk	852k	A puzzle game	F-Droid
EPMobile.apk	2142k	A set of tools for electrophysiologists and health care works	F-Droid
ABCore.apk	1042k	A wrapper for Bitcoin Core for Android	F-Droid
APhotoManager.apk	1131k	Search photos in local media store for viewing and maintenance	F-Droid
aMetro.apk	2554k	aMetro shows the maps of transit systems all over the world-subways, metro, buses, trains, and other.	F-Droid

B. Case Study: CoolReader.apk

CoolReader is a e-book reader from third-party market. In this section, we will show the details about how AppIS protects CoolReader.

In *app pre-processing* stage, we use Androguard to get the Function Call Graph *gexf* for CoolReader.apk. The apk has a total of 204 APIs. Then a python script is exploited to obtain the Key API as Goal ($indegree + outdegree > 5$). Fig.8(a) shows the parsed gexf using Gephi, with a total of 30 key

APIs, colored red. Fig.8(b) is a detail view of Fig.8(a) with some tags.

After the Goals are determined, it needs to get the calling chain in which each Goal is, (for example, let Goal as the center, have the maximum step size keep 5), then randomly select a node on this calling chain to insert a protecting Guard.

Why choose to insert guards in the calling chain of the goals?

The reason is that the guard in the calling chain can ensure that the corresponding guards will be invoked to prevent some guards can not be implemented at the appropriate time. For example, B is often executed and A is not executed. As a result, A-guard is often executed with B, which is obviously not reasonable in considering the reliability and efficiency of protection. In addition, random insertion can improve the whole security.

After that, check whether the two call chains are crossed, if they are, establish a mutual protection association between the two guards.

And finally we call those guards who have not been guarded by other guards isolated guards, and then continue to increase the step size of the calling chain as well as continue the process above until there is no isolated guard left.

Finally, we embed this guarding net into the original app in the form of shared library. The final effect after protection is shown in Fig.8(c), where the green node and the orange node represent 30 goals, and the yellow node and the orange node represent 27 guards.

C. Measurement Index

In order to make a fair evaluation of the effect of the system, we choose the following indicators as evaluation criteria.

- **Validity**:Validity refers to the effectiveness of the protection effect, which is usually reflected in the degree of safety of the protected application. As we all know, an application's security level is reflected in its ability to withstand malicious attacks. So we will quantify the risk of the attack to measure its effectiveness.
- **Performance overhead**: The performance overhead here refers to the extra cost of performance, which includes two aspects: the space cost and the time cost. For Android app, the space overhead here is the size of app itself(or the size of *APK*), while the time overhead is mirrored in the app start time delay compared to the one before protection.
- **Correctness**: Correctness means that the protected app can be run correctly. Due to the serious fragmentation in Android system, the correctness requires that the protected app can be normally executed and compatible with all the system version.

D. Validity

In order to objectively evaluate the effectiveness of AppIS, we use Microsoft's risk assessment model, named DREAD to quantify the risk of attack.

TABLE II: DREAD rating for threats in AppIS.

Threat	D	R	E	A	D	Total	Rating
Attacker obtains static guarding net collection by reversing engineering.	3	1	1	3	1	9	Medium
Attacker launches cumulative attack to AppIS.	2	1	1	2	1	7	Low

DREAD is an acronym for Damage Potential, Reproducibility, Exploitability, Affected users and Discoverability these five words.

In AppIS, we consider the following two threats:

- Attacker obtains static guarding net collection by reversing engineering.
- Attacker launches cumulative attack to AppIS.

TABLE II shows the DREAD rating for both threats. For every threat, we use the DREAD model to calculate risk from D(Damage Potential), R(Reproducibility), E(Exploitability), A(Affected users) and D(Discoverability) these five aspects. In DREAD model, every index's values can be rated from 1-3. So the result can fall in the range of 5-15. Just as TABLE II shows, the overall ratings of the first threat is 9, which can be treated as Medium risk; while the second threat can be treated as Low risk.

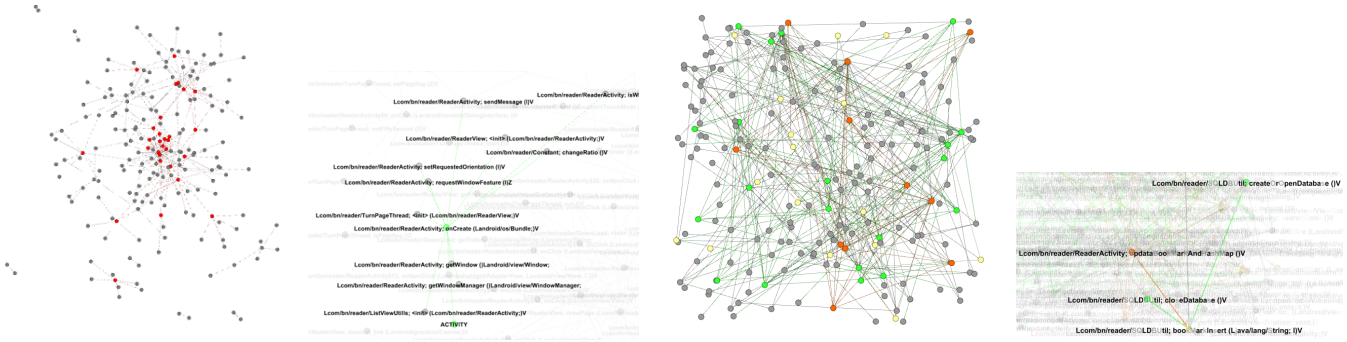
As can be seen from the table above, AppIS is strong enough against potential threats, which proves its validity.

Case Study: Security Evaluation of CoolReader after AppIS protection

The following Fig.9 shows the entire protecting flow of CoolReader under AppIS protection. In the event of an attack, the guards pre-deployed in libnativeSec.so will begin to monitor the key API in real time, such as the *onPrepareDialog()* function, to calculate the integrity of its hash value, and when the hash value is not matched, a warning is issued and drop out.

Fig.8(c) is the guarding map after AppIS protection of Fig.8(a), (b). Zoom in on its part, the yellow guard *Lcom/bn/reader/SQLDBUtil;bookMarkInsert(Ljava/lang/String;I)V* is responsible for protecting the two green goals, while the yellow node itself is protected by orange nodes. This cascading interlocking protective mesh can effectively prevent repackaging and tampering attack.

AppIS's own security issues: As the core algorithm of AppIS is implemented with C++ in native layer, so the attacker is difficult to perform static analysis. Even if the attacker tried to directly remove the .so to bypass AppIS, that is also not feasible, because of its internal implementation function *antiDeleteSo()*. Additionally, it has *deleteCallPointinJava ()* to delete the native call point in Java layer. For the defense of dynamic debugging, as mentioned in Section VII, AppIS uses the ***ptrace*** mechanism to prevent an attacker from using IDA Pro for dynamic debugging, which is the most commonly used reverse tool for an attacker.



(a) FCG of original CoolReader.apk (b) FCG with key API tag. (c) FCG of AppIS protected CoolReader.apk (d) Partial magnification of (c)

Fig. 8: Funtion Call Graph Comparison before and after AppIS protection observed by Gephi: (a)The red node indicates the key api to be protected.(c)The green node and the orange node represent 30 goals,and the yellow node and the orange node represent 27 guards.

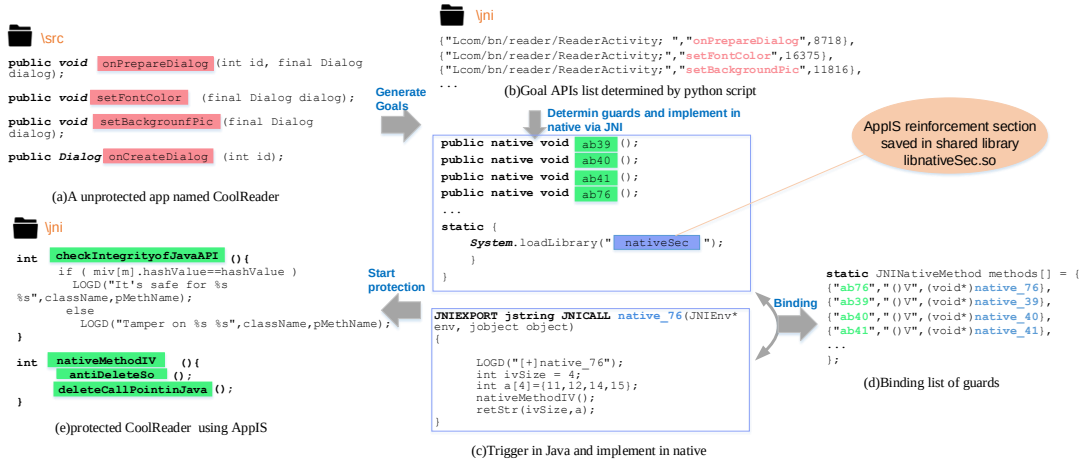


Fig. 9: The details about AppIS protecting process: The red box indicates the key API to be protected, the green box represents the safety protection section of AppIS, and the blue box denote the shared *.so* library where the core algorithm of AppIS will eventually be packaged.

E. Performance and Correctness

In this section, we place emphasis on performance evaluation and correctness evaluation. For the performance evaluation, we use two evaluation criteria, including the size and startup time of the protected application. For the correctness evaluation, we conduct a compatibility test.

To show the experiment results, we have chosen two popular reinforcement suppliers to reinforce the experimental object as reference samples.

At the beginning of the experiment, we separately measured the size and startup time of each application in the collection of applications. Later, respectively, these applications will be input to AppIS, Bangcle and 360 to protect. After protection, we again counted the respective sizes and startup times of these applications. At the same time, we calculated the result of $(protectedappsize)/(originalappsize)$ and the percentage of growth rate in the start time for each application. Fig.10 shows the comparison result in space overhead after protection. The data in this figure reveals one key point about AppIS, that it, compared to the Bangcle and 360 protection, the extra space costs brought by AppIS can be very smaller, which indicated that AppIS can minimize the negative impact of protection on the original application.

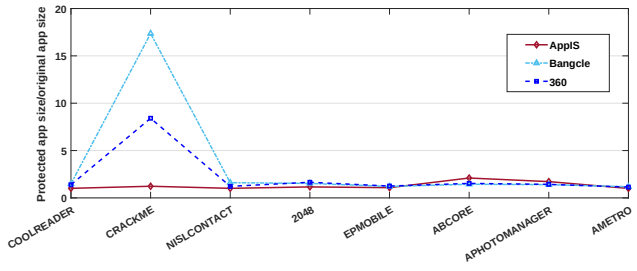


Fig. 10: Space overhead comparison on file size (KB) of AppIS, Bangcle and 360.

Fig.11 shows the startup time difference of the original app, the app protected by AppIS, Bangcle, 360 to check the time overhead. In this paper, for each sample app, we test its startup time for 100 times and then calculate the average value. In

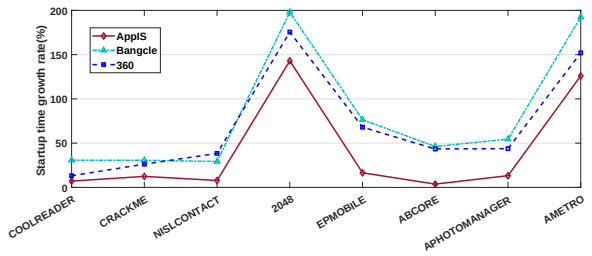


Fig. 11: Time overhead comparison on startup time (ms) of AppIS, Bangcle and 360.

general, our AppIS brings less time overhead than other two protection schemes according to Fig.11.

Compatibility: due to the various Android system version, influence caused by the differences between the versions on the normal execution of the application determines the feasibility of the protection scheme. Therefore, in order to verify the feasibility of the AppIS protection scheme, we have chosen four mainstream Android system version, that is Android4.0, 4.4, 5.0, 6.0, for testing the compatibility of the apps. And the experimental results shows that our system can be compatible with most versions.

IX. Conclusion

With the rapid growth of Android apps, security issues have become increasingly prominent, in order to improve the Android apps own security, app anti-tamper protection has become an urgent issue. This paper based on the Android platform, puts forward a high-performance and scalable anti-repackaging protection, named AppIS. It can protect app from two aspects of static and dynamic, and effectively enhance the app's own security. In addition, the validity and feasibility of the method are verified by the combination of theory and experiment.

In this paper, the implemented AppIS protection scheme can greatly improve the Android app security and the strength against repackaging and cumulative attack, however, there are still some deficiencies need to improve.

The first one reflects in the evaluation of the protection scheme effectiveness. This paper makes a quantitative evaluation about the risk value of the same threat before and after protection, and measures the effectiveness from the point of view of the risk rating. However, the risk assessment of DREAD risk assessment model has some subjective factors, and it is easy to cause errors. Therefore, we should carry out sufficient attack experiments to further evaluate the effectiveness.

The second deficiency lies in the security of the time diversity module. This paper uses a dynamic guarding net with time diversity to enhance the dynamic protective effect. However, during the implementation of the system prototype, the concrete realization of the time diversity module is completed by the guarding net collection and daemon process. Therefore, how to effectively protect the guarding net collection and daemon process from being debugged and cracked by attackers is a concern of us. Meanwhile, because this problem is not the main problem need to be solved in this paper, this paper only puts forward three basic strengthening measures, and the more advanced solutions still need further improvement.

References

[1] C. Collberg, "The case for dynamic digital asset protection techniques," 2011.

[2] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy*, 2012, pp. 95–109.

[3] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symposium on Research in Computer Security*, 2012, pp. 37–54.

[4] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *ACM Conference on Data and Application Security and Privacy*, 2013, pp. 185–196.

[5] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *ACM Symposium on Applied Computing*, 2013, pp. 1808–1815.

[6] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon:evaluating android anti-malware against transformation attacks," in *ACM Sigsac Symposium on Information, Computer and Communications Security*, 2013, pp. 329–334.

[7] M. Grace, S. Zou, S. Zou, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *International Conference on Mobile Systems, Applications, and Services*, 2012, pp. 281–294.

[8] K. O. Elish, D. Yao, B. G. Ryder, and X. Jiang, "A static assurance analysis of android applications," 2013.

[9] A. Aldini, F. Martinelli, A. Saracino, and D. Sgandurra, "Detection of repackaged mobile applications through a collaborative approach," *Concurrency & Computation Practice & Experience*, vol. 27, no. 11, pp. 2818–2838, 2015.

[10] M. Zhang and H. Yin, "Semantics-aware android malware classification," in *Android Application Security*. Springer, 2016, pp. 19–43.

[11] Y. Peng, J. Liang, and Q. Li, "A control flow obfuscation method for android applications," in *Cloud Computing and Intelligence Systems (CCIS), 2016 4th International Conference on*. IEEE, 2016, pp. 94–98.

[12] A. Chawdhary, R. Singh, and A. King, "Partial evaluation of string obfuscations for java malware detection," *Formal Aspects of Computing*, vol. 29, no. 1, pp. 33–55, 2016.

[13] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, "Oblivious hashing: A stealthy software integrity verification primitive," in *Information Hiding, International Workshop, Ih 2002, Noordwijkerhout, the Netherlands, October 7-9, 2002, Revised Papers*, 2002, pp. 400–414.

[14] H. Y. Chen, T. W. Hou, and C. L. Lin, *Tamper-proofing basis path by using oblivious hashing on Java*. ACM, 2007.

[15] M. Jacob, M. H. Jakubowski, and R. Venkatesan, "Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings," in *The Workshop on Multimedia & Security*, 2007, pp. 129–140.

[16] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, 2001, pp. 160–175.

[17] R. H. Niazi, J. A. Shamsi, T. Waseem, and M. M. Khan, "Signature-based detection of privilege-escalation attacks on android," in *Information Assurance and Cyber Security*, 2016, pp. 44–49.

[18] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor, "Androsimilar: Robust signature for detecting variants of android malware," *Journal of Information Security & Applications*, vol. 22, no. 11, pp. 66–80, 2015.

[19] Z. Wang and F. Wu, "Android malware analytic method based on improved multi-level signature matching," in *International Conference on Information Science and Technology*, 2015, pp. 93–98.

[20] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F.-Y. Wang, "Toward digital asset protection," *IEEE Intelligent Systems*, vol. 26, no. 6, pp. 8–13, 2011.

[21] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *IEEE Symposium on Security & Privacy*, 2007, pp. 231–245.