Prison Break of Android Reflection Restriction and Defense

Zhen Ling , Ruizhao Liu , Yue Zhang[yz], Kang Jia , Bryan Pearson[z], Xinwen Fu[yx], Luo Junzhou
School of Computer Science and Engineering, Southeast University
Email: {zhenling, ruizhaoliu, kangjia, jluo}@seu.edu.cn
[z]Department of Computer Science, Jinan University
Email: zyueinfosec@gmail.com
[x]Department of Computer Science, University of Central Florida, USA
Email: bpearson@knights.ucf.edu
[y]Department of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA
Email: xinwen_fu@uml.edu

***Abstract*—Java reflection technique is pervasively used in the Android system. To reduce the risk of reflection abuse, Android restricts the use of reflection at the Android Runtime (ART) to hide potentially dangerous methods/fields. We perform the first comprehensive study of the reflection restrictions and have discovered three novel approaches to bypass the reflection restrictions. Novel reflection-based attacks are also presented, including the password stealing attack. To mitigate the threats, we analyze these restriction bypassing approaches and find three techniques crucial to these approaches, i.e., double reflection, memory manipulation, and inline hook. We propose a defense mechanism that consists of classloader double checker, ART variable protector, and ART method protector, to prohibit the reflection restriction bypassing. Finally, we design and implement**

Novel attacks. Powered by these restriction bypassing approaches, various reflection-based new attacks are presented to demonstrate the potential threats posed on the Android user privacy and security, including the password stealing attack and Bluetooth bond removal attack.

New defense. To fight against the threats caused by the reflection bypassing approaches, we propose a novel defense mechanism. We first dissect the five Android restriction bypassing approaches, including our three novel approaches and two existing ad-hoc approaches [26], and summarize the crucial techniques used in these approaches, i.e., double reflection, variable manipulation, and inline hook. Then we design and implement the defense mechanism to respectively circumvent the restriction bypassing via the three techniques.

Extensive Experiments. Expensive experiments are performed to evaluate the feasibility of our reflection restriction bypassing approaches and the effectiveness of our defense enabled Android systems. We build an automatic static analysis framework to analyze 100,000 apps and discover 5,531 apps using reflection. The apps are dynamically executed in our defense enabled Android system. The defense takes effect and is robust.

The rest of this paper is organized as follows. We introduce the reflection and existing Android restrictions on reflection in Section II. Then we present the three approaches to bypass the restrictions and various reflection-based attacks in Section III. In Section IV, we elaborate on the design and implementation of our defense against the reflection powered attacks. Extensive experiments are performed to evaluate the feasibility of the attacks and effectiveness of the defense mechanism in Section V. We review related work in Section VI. Finally, this paper is summarized in Section VII.

II. Background

In this section, we briefly introduce reflection in Android and the reflection restriction mechanisms.

A. Reflection in Android

Android apps are written in Java, an object-oriented programming language that supports the reflection mechanism. The reflection mechanism enables developers to call a method or inspect a field of a given class at runtime. Listing 1 provides an example, where `targetMethod()` is the method an app wants to invoke via reflection, and the class that the target method belongs to is `targetClass`. In the first line, the app first obtains the instance of an object representing `targetMethod()`. In the third line, the app uses the instance to invoke the target method. Unlike a standard method/field access, reflection does not need to know the method/field at compile time. For example, "targetMethod" is a string variable that can be replaced by another string, e.g., "anotherMethod", from user inputs or a remote server. If this is the case, `anotherMethod()` will be invoked at runtime. One use case of reflection is testing compatibility of Android devices [10], [18]. When Android updates its SDK, developers may use reflection to test if a method is available on a specific device at runtime before the app runs the method.

Although developers with reflection have the flexibility of accessing hidden methods/fields that are reserved by the Android system, the use of Android reflection opens an attacking surface against Android ecosystem. For example, Kywe *et al.* [17] report that reflection can be abused to bypass the Android permission model and collect sensitive information such as device ID, telephone service status, and SIM card status.

```
1 Method mtargetMethod =
2         Class.class.getDeclaredMethod("targetMethod"
      ); // obtain the instance of targetMethod
3 mtargetMethod.invoke(); // run targetMethod()
```

Listing 1. Code fragment of reflection

B. Restrictions on Reflection

The Android Security Team is aware of the potential security risks brought by the reflection. Therefore, starting from Android 9, restriction mechanisms against reflection techniques have been introduced. With such restrictions in position, Android limits the use of a specific set of hidden methods/fields [4]. Whenever an app attempts to access to a restricted method/field via reflection, Android throws an exception to crash the app. Specifically, Android adopts four restriction mechanisms.

Whitelist, graylist, and blacklist. Android groups all hidden methods/fields into three categories, called the whitelist, the graylist, and the blacklist. (i) Whitelist. The methods/fields in the whitelist are considered non-risk and free to access via reflection. (ii) Graylist. Methods/fields in the graylist can be used via reflection depending on the current Android system version. Specifically, each method/field in the graylist is assigned a maximum accessible API level number [3], which functions as an identifier for the programming framework API offered by the corresponding version of the Android system (e.g., the API level number of Android 10 is 29). An app may use a graylisted method/field via reflection if and only if the target API level of the app is not higher than the maximum accessible API level of the hidden method/field. Accordingly, the set of methods/fields that **can** be accessed in the graylist is referred to as the lightgreylist, while the remaining methods/fields that **cannot** be accessed are in the darkgreylist. (iii) Blacklist. Methods/fields that are in neither the whitelist nor the graylist are in the blacklist and prohibited to be accessed via reflection. *Based on this information, methods/fields in the darkgreylist and blacklist cannot be accessed via reflection.* In this paper, we refer to the methods/fields in the darkgreylist and blacklist as the **protected methods/fields**.

Trusted caller. Android allows a trusted caller to access the protected methods/fields. If a class of a caller is loaded via `BootStrapClassLoader`, the caller is referred to as a trusted caller. Consequently, Android determines whether the app can access the protected methods/fields via reflection by checking whether the classloader of the caller is `BootStrapClassLoader` or not.

Enforcement policy. Android specifies a set of principles, termed enforcement policy, including `kNoChecks`,

kJustWarn, kDarkGreyAndBlackList, and kBlacklistOnly, so as to allow an app to invoke some protected methods/fields via reflection under these four conditions. The enforcement policy is set during the initialization of the Android Runtime (ART) [2]. For example, the enforcement policy is configured to kNoChecks by Android when an app is in the debug mode. Then the app can access the protected methods/fields without any ue2ekations. In addition, the enforcement policy can also be set to kJustWarn, where Android system prompts warnings whenever an app accesses the protected methods/fields via reflection. When the enforcement policy is set to kDarkGreyAndBlackList, the system prohibits all protected methods/fields. When the enforcement policy is set to kBlacklistOnly, the system only prohibits the methods/fields in the blacklist.

Exemption list. Android also specifies an exemption list mechanism for pre-installed apps (e.g., Settings) to access to protected methods/fields. Only a pre-installed app can invoke a method named setHiddenApiExemptions(.) to add a protected method/field into the exemption list so as to access to the method/field via reflection. However, since the setHiddenApiExemptions(.) is not public to third party apps, they do not have the privileges to call setHiddenApiExemptions(.) and cannot access a protected method/field.

C. Android Sandbox

Android implements Linux-based resource isolation [16], where each app is assigned a unique user ID (UID) in order to set up a kernel-level sandbox. In the sandbox, the kernel prevents an app with a specific UID from interacting with another app with different UID. Meanwhile, since the sandbox is implemented in the kernel, the enforcement also applies to the native code (i.e., the underlying C/C++ code) of Android, which the Android framework is based on. Therefore, the mechanism cannot be bypassed through native techniques such as the Java Native Interface [20]. However, the resources owned by one app (e.g., the memory, OS libraries, OS framework, and Android Runtime) run within the sandbox and Android does not restrict how an app may access these resources. Therefore, an app may freely access these resources.

III. Reflection Restriction Bypassing Approach

In this section, we first present reflection restriction bypassing approaches one by one and then introduce the potential attacks caused by the reflection.

A. Approach Intuition

As discussed in Section II-B, Android specifies regulations to restrict the abuse of reflection. Particularly, whenever an app attempts to access a hidden method/field via the reflection (called the ”target method/field”), ART will determine whether the attempt can succeed by checking if one of several conditions is fulfilled. Particularly, Figure 1 illustrates the workflow of the checking process:

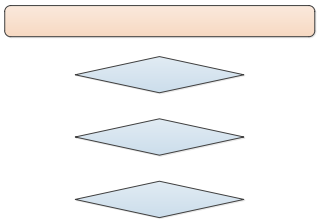

Fig. 1. Workflow of the reflection checking process

1) Android checks if the target method/field is in the whitelist. If yes, Android permits the access.
2) Android checks whether the attempt is initiated by a trusted caller that is loaded via BootStrapClassLoader. If yes, Android permits the access.
3) Android checks if the enforcement policy (e.g., kNoChecks) is configured to allow the target method-/field via reflection. If yes, Android permits the access.
4) Android checks if the target method/field is in the exemption list. If yes, Android permits the access.
5) Android checks if the target method/field is in the lightgraylist as discussed in Section II-B. If yes, Android permits the access.

It can be observed the reflection attempt is granted when any one of the five conditions is satisfied. If none of these requirements is fulfilled, the app crashes and throws exceptions such as “NoSuchFieldException” or “NoSuchMethodException”. Although the checking process is conducted by the ART, the library implementation of ART (i.e., libart.so) is loaded in the app-level memory space of an app. This can be manipulated by the app itself due to the sandbox mechanism as discussed in II-C. Based on this intuition, we craft novel approaches to bypass the reflection restrictions, and each approach works against one specific condition.

B. Methodology

We present our new approaches for bypassing the reflection restrictions, including a whitelist bypassing approach, an exemption list checking bypassing approach, and a graylist checking bypassing approach. All the approaches discussed in this section have been tested on a Google Pixel 2 running the latest Android 10 operating system.

Whitelist checking bypassing approach: Whitelist checking bypassing approach works against the whitelist checking condition. With this approach, an app may trick Android to determine the protected method/field is in the whitelist, which will grant access to the method/field. Specifically, when an app attempts to access a hidden method/field via reflection, Android enters the checking process and first uses GetDexFlags(.) to determine whether the method/field being accessed is in the whitelist or not. GetDexFlags(.) accepts the name of the method/field being accessed as the in-

put, and will return an integer named kMaxTargetR[1], if the method/field is in the whitelist. Therefore, the app can modify the return value of GetDexFlags(.) to kMaxTargetR so as to obtain the access. To this end, we employ an inline hook tool named SandHook [13] to hook GetDexFlags(.) and modify the return value of GetDexFlags(.). SandHook allows a running app to record, alter, and replay the inputs and the return value of a target method by modifying the code fragment in memory of the app. Moreover, SandHook can be integrated into an app without requiring root permission to be granted. Once GetDexFlags(.) is hooked via SandHook, we force the method to always return kMaxTargetR so as to access a protected method/field.

Exemption list checking bypassing approach: Exemption list checking bypassing approach works against the exemption list condition. Since ART is loaded in the app-level memory space of an app, the app may manipulate its own memory space that stores the exemption list so as to intentionally add a protected method/field into its exemption list. To this end, the app first (i) obtains the JNI (Java Native Interface) environment pointer (i.e., JNIEnv*) via the Java Native Interface. The JNI environment pointer is implicitly passed as an argument from the Java layer to the native layer, allowing the app to use the JNI environment pointer within a JNI native method. (ii) The JNI environment pointer can be employed to locate the memory address of the ART object. Then we analyze the souce code of the ART to derive an offset to further locate the address of a vector named hidden_api_exemptions that is a class member variable of the ART. (iii) hidden_api_exemptions maintains the methods/fields that have been added into the exemption list. Each method/field is represented using its unique signature. For example, the signature of method "android.widget.CompoundButton.getButtonDrawable()" is "Landroid/widget/CompoundButton;.getButtonDrawable". The app then appends the signature of the method/field of interest into the vector. In such a way, the method/field is added to the exemption list.

The exemption list checking process uses the longest prefix match algorithm to determine if the signature of the method/field being accessed matches the string in an entry of the vector of the exemption list. Since all the signatures start with the letter "L", we can simply append a new entry that only includes a letter "L" into the vector. Consequently, all protected methods/fields are considered in the exemption list and can be accessed without restrictions.

Graylist checking bypassing approach: Graylist checking bypassing approach works against the graylist condition. An app may use graylist bypassing approach to make Android determine the protected method/field being accessed is reachable under the graylist policy. Specifically, the graylist checking process is conducted in a method named GetMemberActionImpl(.) and the method returns an integer kAllow [2] if the method/field being accessed is reachable under the graylist policy. Similar to the whitelist checking bypassing approach, a malicious app may employ SandHook to hook GetMemberActionImpl(.) and modify its return value. Consequently, the caller can access a protected method/field of interest via reflection.

C. Other Reflection Restriction Bypassing Approaches

It is worth noting that there are other approaches for bypassing Android's reflection restrictions. These ad-hoc approaches [26] work against either the trusted caller condition or the enforcement policy and are summarized as follows.

Enforcement policy checking bypassing approach. An approach in [26] works against the enforcement policy. In the approach, an app manipulates the variable of the ART in the memory (i.e., a variable named hidden_api_policy_) so as to configure the enforcement policy to NoChecks.

Trusted caller checking bypassing approach. In [26], the authors also demonstrate that an app can use a technique named double reflection to disguise a trusted caller. In this paper, we refer to reflection recursively used twice as double reflection. Specifically, the app first derives an instance of a crucial reflection method (i.e., getDeclaredMethod() or getDeclaredField()) that is in the whitelist via the first reflection to "disguise" a trusted caller, since the classloader of the derived reflection method is BootStrapClassLoader. Then the app uses the derived reflection method to obtain an instance of the target protected method/field via the second reflection. In this way, a protected method/field can be invoked so as to bypass the checking process.

Although these approaches demonstrate additional techniques that work against the restriction on reflection, they do not propose any systemic insights from a scientific prospective. In our work, we analyze the principles of the restrictions and the root cause of these reflection restriction bypassing approaches. Moreover, we demonstrate three novel approaches against the restrictions on reflection. This provides a more comprehensive look at the issue.

D. Reflection-based Attacks

The Reflection restriction approaches demonstrated in this paper can be utilized as building blocks to design various reflection-based attacks, violating user privacy and impairing the Android ecosystem. Particularly, we have validated four attacks on a Google Pixel 2 running the latest Android 10, as shown in Table I. Among these attacks, permission-free device fingerprinting attack and breaching user privacy attack were introduced in [17]. Android disabled these attacks via restrictions on reflection. However, such defenses can be thwarted by using our approaches. We also identified password/keystroke stealing attack and bluetooth bond removal attack. These attacks have never been discussed in the previous works.

Permission-Free Device Fingerprinting Attack. A malicious app may use our attacks to collect sensitive information and deploy the device fingerprinting attack, which

[1]The value of kMaxTargetR is 6.

[2]The value of kAllow is 0.

TABLE I
ATTACKS USING REFLECTION. "—" MEANS THE AN ATTACKER DOES NOT NEED OUR APPROACHES. "✓" MEANS "YES"; "✗" MEANS "NO".

Attacks	Related Method or Field	Disabled	Enabled by Our Approaches
Permission-Free Device Fingerprinting	getDeviceId(.) getRingerModeInternal(.) getInputDevice(.)	✗	—
Breaching User Privacy	getMode(.)	✓	✓
Password/Keystroke Stealing	FLAG_NOT_TOUCHABLE	✗	—
Bluetooth Bond Removal	removeBond(.)	✓	✓

uniquely identifies a specific Android device. Some of this information is protected by Android, such as the device ID and SIM card state; therefore, an app may need specific permissions to access the information. However, using reflection restriction bypassing, the information can be accessed without requiring permissions to be granted as reported in [17]. We have validated the approach, and find methods such as `getRingerModeInternal()`, `getInputDevice()`, and `getDeviceId()` can be abused to collect sensitive information and fingerprint a device using reflection.

Breaching User Privacy. Some APIs can be abused to breach user privacy and as a result, Android blacklists these APIs to prevent them from being invoked via reflection. However, a malicious app may use the approaches introduced in our paper to bypass the restrictions [17] and violate user privacy. For example, the method `getMode(.)` of `IAudioService` can be abused to monitor the phone status, indicating whether there is an incoming call/VOIP call.

Password/Keystroke Stealing Attack. An attacker may abuse the approaches to steal user password and keystrokes. In this attack, the malicious app may use a customized touchable toast so as to receive user inputs without requiring permissions. A toast is a special type of view for user feedback in Android and does not require any permissions when apps use it. On the other hand, other overlay based attacks may require multiple permissions as discussed in [12]. After a toast is created by an app, it stays on the foreground for a few seconds (i.e., 2.5 s or 3.5 s) before Android destroys it. Toast can be customized to appear as arbitrary content such as a keyboard-like view. This can threaten user privacy, since such a keyboard-like view may spoof a user to believe it is a genuine keyboard, and a user may type sensitive information such as credentials into it. Therefore, Android attempts to restrict a toast from receiving user input. However, an app can disable the flag "FLAG_NOT_TOUCHABLE" of a toast via reflection, so as to make the toast receive user input.

Bluetooth Bond Removal Attack. With our approaches enabled, a malicious app may also deploy the Bluetooth bond removal attack via reflection, where the malicious app intentionally removes all Bluetooth bonds existing on the smartphone so as to derive denial of service attack or even downgrade the encrypted Bluetooth link into plaintext. In Bluetooth, a bond is the key used for encrypting the communication. A smartphone may pair with a peer device through the pairing process and generate a bond for securing communication. However, the malicious app can call `removeBond()` via reflection to delete an existing bond on a smartphone. Without a bond, the further communication between the smartphone and a previously bonded device may be subject to attacks such as eavesdropping or spoofing attacks as discussed in [28].

TABLE II
REFLECTION RESTRICTION BYPASSING APPROACHES AND USED TECHNIQUE

Reflection restriction bypassing approaches	Technique
Whitelist checking bypassing	Inline hook
Enforcement policy checking bypassing	Variable manipulation
Trusted caller checking bypassing	Double reflection
Exemption list checking bypassing	Variable manipulation
Graylist checking bypass	Inline hook

IV. COUNTERMEASURE

In this section, we first introduce the basic idea of our countermeasure against reflection-based attacks, including *Classloader Double Checker*, *ART Variable Protector*, and *ART Method Protector*. Then we present the challenges and solutions of our defense methods. Finally, we illustrate the implementation of the methods in detail.

A. Overview

Our defense works against the reflection restriction bypassing approaches so as to defend against various reflection-based attacks. It can be observed from Section III that all the reflection restriction bypassing approaches adopt one of three types of techniques including double reflection, memory manipulation, and inline hook. Specifically, Table II summarizes the approaches and the corresponding techniques. Therefore, if we can design solutions against these techniques, the reflection-based attacks can be stopped. By applying our defense, the app will terminate if it detects an attempt at reflection restriction bypass.

Our defense mechanism consists of three main components, including *Classloader Double Checker*, *ART Variable Protector*, and *ART Method Protector*, corresponding to the three techniques used by the reflection restriction bypassing approaches, respectively. We now explain the principles of each component: (i) *Classloader Double Checker* defends against the double reflection. The Trusted caller checking bypassing approach is made possible as Android only checks the classloader of the class that invokes a hidden method/field via reflection, while Android fails to construct a chain of callers and trace back to the class that initiates the reflection process. *Classloader Double Checker* can identify such an initiator so as to defend against the approach. (ii) *ART Variable Protector* works against the variable manipulation. Particularly, at runtime, it identifies the relevant variables in memory the attacker intends to modify, and configures the variables as non-writable before the attacker has a chance to access them. (iii) Recall that inline hook requires modification of methods in memory. Therefore, similar to the approach adopted by *ART Variable Protector*, *ART Method Protector* can locate the methods to be protected in memory and configure them as non-writable.

B. Challenges and Solutions

The implementation of our defense should be made within the Android System to defend against the reflection-based

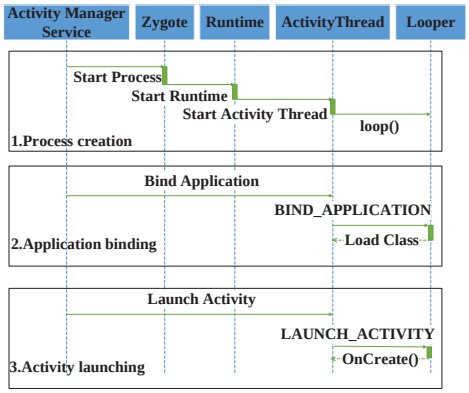

Fig. 2. Workflow of the app startup

attacks. However, prior to our implementation, there are a few technical challenges to be solved.

First, for *ART Method Protector* and *ART Variable Protector*, we should select an appropriate time to load our defense; however, identifying such a time point can be tricky. This is because these defenses involve the modifications of ART memory and the corresponding modifications should occur after the ART initialization. Otherwise, the modifications made by the defense will result in the failure of the ART initialization. At the same time, these modifications should be made before a malicious app can obtain the access of these ART memories so as to avoid the malicious ART memory manipulation. To address this issue, we go through the source code of Android Open Source Project referring the launch process of an Android app, so as to find an appropriate position where our defense can be located, enabling the defense to be loaded at the right time. Figure 2 shows three phases an app goes through when launching, including process creation, application binding, and activity launching. (i) In the process creation phase, AMS (Activity Manager Service), a system service handling the life-cycles of all activities, creates a process via Zygote, which is a special process responsible for creating processes for apps, for the app to be launched. During the process creation, `libart.so`, the implementation of ART, is loaded in the memory of the app and initialized by the Android kernel. Then an `ActivityThread` object is built and a new process is created at this point. `ActivityThread` initiates a message loop by invoking the method `Looper.loop()` to handle possible messages continuously via a message queue. (ii) In the application binding phase, AMS notifies the `ActivityThread` to send a `BIND_APPLICATION` message to associate the app with the created process and loads all classes of the app to be launched into memory. (iii) In the activity launching phase, AMS notifies the `ActivityThread` to send a `LAUNCH_Activity` message to the process attached to the app to start the corresponding activity by calling the `onCreate()` method, so that a user can interact with the app. Therefore, our defense should be positioned in the process creation, particularly, at the time the process enters the loop method. We implement the *ART Method Protector* and *ART Variable Protector* at `Looper.loop()`, right after the process enters the loop.

Second, the Android System may run various apps, and therefore, when the defense is in position, it should not impair the running of normal apps. For example, *ART Variable Protector* configures variables in a data segment of the ART memory such as `hidden_api_policy_` to non-writable. However, when *ART Variable Protector* performs such an operation, due to the memory management policy enforced by Android, *ART Variable Protector* can only manipulate the memory at the granularity of pages. Therefore, it has to configure the entire memory page where the variable locates so as to achieve the goal. Observably, this may cause issues since the pages may include variables used for other purposes, and setting the entire page as non-writable may unintentionally crash the app. In order to minimize the impact, we ensure that only the variable to be protected is contained in an entire page and if the page is set to non-writable, other variables in the memory of the ART are not impacted. Moreover, for ART methods to be protected, *ART Method Protector* configures the page containing a code segment of methods of interest as non-writable directly without extra efforts to be taken. This is because normal running process of an app should not involve the modification of any methods.

C. Implementation

Classloader Double Checker: To implement the *Classloader Double Checker*, we force the current checking process to go further so as to discover the genuine initiator of reflection. Fortunately, we find Android uses a method named `VisitFrame()` to perform the checking process over a call stack. When a method on the top of the call stack attempts to access a hidden method/field, the original checking process is to use `VisitFrame()` to check whether the classloader of the class of the method is `BootStrapClassLoader`. However, the double reflection technique used in the trusted

TABLE III
VALUE AND MEANING OF VM_FLAGS

name	value	meaning
VM_READ	0x00000001	readable
VM_WRITE	0x00000002	writable
VM_EXEC	0x00000004	Executable
VM_SHARED	0x00000008	Shareable
VM_MAYREAD	0x00000010	VM_READ can be set
VM_MAYWRITE	0x00000020	VM_WRITE can be set
VM_MAYEXEC	0x00000040	VM_EXEC can be set
VM_MAYSHARE	0x00000080	VM_SHARED can be set

found that we only need to protect two variables; therefore, the overhead is only 16KB. Next, *ART Variable Protector* locates the address of the variables to be protected through the JNI environment pointer and then computes the starting address of the page that contains the variable to be protected. We can compute the starting address of the page by

$$A_s = A_v - (A_v\%4096), \quad (1)$$

where A_v is the address of the variable and A_s is the starting address of the page. Then, *ART Variable Protector* configures the page containing the variable as non-writable using the starting address of the page A_s. This can be achieved through the modification of a specific struct. In Linux based OSes such as Android, each memory page maps to a `vm_area_struct`, which contains a member variable named `vm_flags`. The value of `vm_flags` ultimately determines whether the referred page is readable, writable, executable, and shareable as shown in Table III. Once the bits (i.e., both `VM_WRITE` and `VM_MAYWRITE`) are set to 0, the memory page is configured to non-writable, and a malicious app cannot reset the configuration via the system call `mprotect(.)`. As a result, a malicious app cannot manipulate the memory page during the lifecycle of the app. The `VM_MAYWRITE` bit is used to determine whether the `VM_WRITE` bit of the memory page is able to be set to non-writable or not. However, the Android system does not provide a system call to modify the `VM_MAYWRITE` bit of a page. To this end, we first revise the Android kernel source code to customize a system call that can set the properties of target page(s) to non-writable and ensure that the properties cannot be reset again by inputting two parameters of the starting address of the page and the number of pages and setting both the `VM_WRITE` and `VM_MAYWRITE` of the page(s) to 0. Then we revise the source code of the Android framework (i.e., the method `Looper.loop()`) to call the customized system call so as to set the target page(s) to non-writable. Consequently, the variable manipulation based attacks including enforcement policy checking bypassing and exemption list checking bypassing based attacks can no longer work.

ART Method Protector. This component addresses the abuse of inline hook. We observe that to achieve inline hook based attacks, the malicious app must intentionally manipulate the return values of specific methods such as `GetDexFlags(.)` or `GetMemberActionImpl(.)`. These methods, however, are all included in the `libart.so`, a core library used by ART. Therefore, to achieve inline hook, SandHook must manipulate the code segments in this library at runtime which contains the binary code of the methods. As a solution against inline hook, we can locate the code segments of `libart.so` in the memory and configure the corresponding memory as non-writable. Particularly, a configuration file named "proc/self/maps" stores the start and end addresses of the app's code segments. Moreover, these code segments are classified into several segments of memory space. Each segment has the executable property and the VM_WRITE bit of the page is set to 0. However, the VM_MAYWRITE bit is set to 1 by default. Therefore, a malicious app may use the system call `mprotect(.)` to set VM_WRITE to 1 and reset the memory as writable. To address this, we revise the source code of the method `Looper.loop()` to read the file "proc/self/maps" and search the content containing "libart.so" with an executable and a non-writable property so as to locate the code segments before the app initiates. Since a code segment may cross multiple pages in memory, we must set all these pages as non-writable. Fortunately, using the "proc/self/maps" configuration file, we can compute the number of pages in the code segment and derive the starting address of each page using the starting and end address of the code segment. As a result, we can use the method introduced in implementing *ART Variable Protector* to set all pages containing the code segment as non-writable.

V. EXPERIMENTAL EVALUATION

In this section, we first present experiment setup, and then evaluate the presented approaches and countermeasures.

A. Experiment Setup

To measure the generality of our approaches against different mobile devices, we use 6 Android mobile devices from different manufacturers. We implement our defense on a Google Pixel 2 Android device running Android 9.0 based on the Android Open Source Project (AOSP) [14]. We conduct a large-scale static analysis over 100,000 apps from the AndroZoo dataset [1] to demonstrate the current status and the potential risks of our approaches. We create five virtual machines on a Linux machine running Ubuntu 16.04 equipped by Intel Core Xeon (2.10 GHz) CPUs and 128 GB RAM. Each of the virtual machines runs Ubuntu 16.04 and is equipped by 16 GB RAM. Four of the virtual machines are used to download the apps and the last one is used to perform the static app analysis. We perform a large-scale dynamic analysis over 5,531 apps using reflection on the smartphone with our defense in order to measure the performance of the patched system. We also use a Google Pixel 2 with an unpatched Android system to run the apps as comparison.

B. Experimental Results

Feasibility of the bypassing approaches. As listed in Table IV, we deploy different approaches against multiple mobile devices in order to confirm the feasibility of the reflection restriction bypassing approaches. We only assessed our approaches on Android 9 and Android 10, since Android 8 and earlier do not provide reflection restrictions. Whitelist

TABLE IV
APPROACHES WORK AGAINST DIFFERENT SMARTPHONES. FOR CONCISENESS, "WCB" IS REFERRED TO AS WHITELIST CHECKING BYPASSING; "GCB" IS REFERRED TO AS GRAYLIST CHECKING BYPASSING; "EXLP" IS REFERRED TO AS EXEMPTION LIST CHECKING BYPASSING.

Brand	Version	Bypassing approaches		
		WCB	GCB	EXLB
Google Pixel 2	9.0	✗	✓	✓
Google Pixel 2	10.0	✓	✓	✓
HUAWEI	10.0	✓	✓	✓
OnePlus	10.0	✗	✗	✓
Vivo	10.0	✗	✗	✓
XiaoMi	10.0	✗	✗	✓

checking bypassing does not work on Android 9, since the method `GetDexFlags(.)` used for the approach is not included in the `libart.so` of Android 9. Inline hook based approaches including whitelist checking bypassing and graylist checking bypassing fail in Vivo, XiaoMi and OnePlus, since the SandHook [13] no longer works on these customized systems. However, exemption list checking bypassing works against all these devices without any changes.

Apps using reflection. We build an automatic analysis framework based on soot [23] to analyze extensive apps downloaded from the AndroZoo and determine whether the apps use reflection or not. Our observation is that an app uses reflection to bypass protected methods/fields (i.e., the methods/fields in the blacklist or graylist) and must initiate the methods such as `getDeclaredMethod()` and `getDeclaredField()` so as to invoke reflection. For example, an app may use `getDeclaredMethod()` to invoke a hidden method and use `getDeclaredField()` to inspect a hidden field. However, our detection does not check the use of methods such as `getMethod()`, which are used to call or insect public methods/field of a class. These methods are out of our focus, since the restrictions on reflection do not work against the public methods/field. Particularly, we find that $5,531$ out of $100,000$ apps adopt reflection. Further, among these apps, all of the $5,531$ apps use `getDeclaredField()` and $3,485$ apps use `getDeclaredMethod()`. Our experiment indicates that the use of reflection is prevalent among apps.

Sources of apps using reflection. Table V illustrates the relationship between apps using reflection and the markets where these apps come from. Since the AndroZoo provides interfaces to allow us to obtain the market information of an app, we query the market information so as to shed light on the relationship between the apps using reflection and their source. It can be observed that more than 82.8% of these apps come from Google Play and only 13 of them are come from 1mobile Market. Based on these results, Google Play may be more susceptible to reflection restriction bypass.

Trend of apps using reflection. We then analyze the relationship between the numbers and the released time of the reflection-based apps. Similarly, we use the interfaces provided by AndroZoo to query and obtain such information. We exclude a few apps with an invalid released date (e.g., an app with a released time 1980). Figure 3 shows the result. It can be observed that from the year before 2013 to the year 2014, the number of apps using reflection were increasing. However, the numbers of apps using reflection decreased after 2015. This may be because people or app markets started to notice the risks brought by reflection. For example, more reflection detection techniques were introduced in the year 2015 and 2016 as reviewed in Section VI. The number of apps using reflection decreased further in 2019 and 2020. This is reasonable since Android's reflection restrictions were introduced in 2018. These restrictions indicate that Android discourages modern apps from using reflection due to potential security issues.

Effectiveness of our defense. To evaluate the effectiveness of our defense, we launched an app with our approaches introduced in Section III and confirmed that these approaches failed under the patched Android system. When the app attempts to run the approaches, the patched OS forces the app to crash so as to defend against possible reflection-based attacks. As a comparison, we run the same app on the smartphone without our defense and found that the five approaches work with no change. Our experiments indicate that our defense can prevent reflection attacks effectively.

Robustness of our defense. We use the collected apps using reflection to assess the robustness of our defense. We install the $5,531$ collected apps using reflection one by one into a smartphone with our defense enabled system and another one with an original system to conduct a comparative experiment. Monkey [5] is used to simulate the clicks of users on the screen. The simulation using monkey lasts one minute to ensure that the reflection of the apps can be triggered. Due to the backward compatibility issues, 414 apps cannot be installed in our system as these apps are developed for running on system that is earlier than Android 9.0. 4,124 apps can successfully run on both two systems as the apps do not access the protected methods/fields and our defense does not affect them. 678 apps are crashed on both systems. In addition, 162 apps are only crashed on the unpatched system and 153 apps are only crashed on the patched system, since the Monkey cannot cover all of the app UI operations on both patched and unpatched systems. We analyze the system logs of the 993 crashed apps and find that these apps do not access protected methods/fields via reflection. The main reasons that they are crashed includes bugs and backward compatibility issues that are unrelated with our defense.

Performance of our defense. We also evaluate the performance of our defense by measuring the delay of our defense. To this end, we set up a smartphone with our defense and one without it, and run 2400 apps including 1200 apps using reflection and 1200 apps without reflection collected from the previous experiment on both smartphones. As discussed in Section IV-B, our defense is initialized during the launch time of an app. We therefore measure the launch time of each app. Specifically, we record and calculate the time required to execute from `Looper.loop()` to `OnResume()` when the app has been completely launched. We use monkey [5]

TABLE V
NUMBER OF THE APPS USING REFLECTION ACROSS DIFFERENT MARKETS

Market	Number of apps	Ratio
Google Play	4579	82.8%
Appchina	322	5.8%
AnZhi Market	473	8.5%
PlayDrone	807	14.7%
1Mobile Market	13	0.2%
VirusShare	65	1%
Mi Store	51	0.9%
F-Droid	15	0.3%

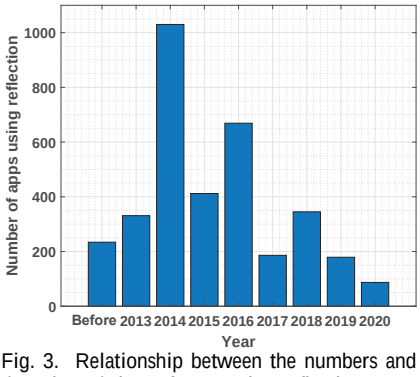

Fig. 3. Relationship between the numbers and the released time of apps using reflection

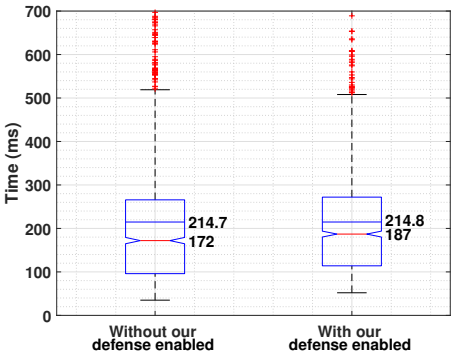

Fig. 4. Performance of our defense enabled system

to automatically start the app. Figure 4 demonstrates the comparison of the launch time of the apps with and without our defense enabled system. It can be observed that the median of the launch time on the smartphone with our defense is 187 ms, while that on the smartphone without our defense is 172 ms. The average of the launch time on the two smartphones are 214.8 ms and 214.7 ms, respectively. Such a delay is negligible for the typical use of apps on mobile.

VI. RELATED WORK

To the best of our knowledge, there is no systematic study of Android reflection restriction bypassing and defense. We now briefly review related work on the use of Android reflection.

REFERENCES

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 468–471, 2016.

[2] Android Open Source Project. Android runtime (art) and dalvik. https://source.android.com/devices/tech/dalvik. Accessed: 2020-08-16.

[3] Android Open Source Project. Api level. https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels. Accessed: 2020-08-16.

[4] Android Open Source Project. Restrictions on non-sdk interfaces. https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces. Accessed: 2020-08-16.

[5] Android Open Source Project. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey. Accessed: 2020-08-16.

[6] Avast Threat Intelligence Team. Mobile spyware uses sandbox to avoid antivirus detections. https://blog.avast.com/mobile-spyware-uses-sandbox-to-avoid-antivirus-detections. Accessed: 2020-08-16.

[7] S. Badhani and S. K. Muttoo. Evading android anti-malware by hiding malicious application inside images. *International Journal of System Assurance Engineering and Management*, 9(2):482–493, 2018.

[8] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, and M. d'Amorim. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[9] M. Collin, W. Robertson, and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[10] S. Conder and L. Darcey. Learn Java for Android Development: Reflection Basics. https://code.tutsplus.com/tutorials/learn-java-for-android-development-reflection-basics--mobile-3203, 2018.

[11] T. Emre, D. Kurt, and A. Güleç. Android obad. *Technical Analysis Paper*, 2013.

[12] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 1041–1057, 2017.

[13] ganyao114. Sandhook. https://github.com/ganyao114/SandHook. Accessed: 2020-08-16.

[14] Google. Android open source project. https://source.android.com/. Accessed: 2020-08-16.

[15] H. Hao, V. Singh, and W. Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[16] T. Kim and N. Zeldovich. Making linux protection mechanisms egalitarian with userfs. In *Proceedings of the USENIX Security Symposium (Security)*, 2010.

[17] S. M. Kywe, Y. Li, K. Petal, and M. Grace. Attacking android smartphone systems without permissions. In *Proceedings of the 14th IEEE Annual Conference on Privacy, Security and Trust (PST)*, 2016.

[18] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[19] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. Reflection-aware static analysis of android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[20] S. Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[21] A. Mohannad, Q. Yan, and H. Bagheri. Dina: Detecting hidden android inter-app communication in dynamic loaded code. *IEEE Transactions on Information Forensics and Security (TIFS)*, 15, 2020.

[22] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[23] Soot. Soot - a framework for analyzing and transforming java and android applications. https://soot-oss.github.io/soot/. Accessed: 2020-08-16.

[24] R. Vaibhav, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[25] C. Valerio and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*, 2015.

[26] weishu. Freereflection. https://github.com/tiann/FreeReflection. Accessed: 2020-08-16.

[27] Y. Xue, G. Meng, and Y. Liu. Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security (TIFS)*, 12(7), 2017.

[28] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In *Proceedings of the 29th USENIX Security Symposium (Security)*, pages 37–54, Aug. 2020.