ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime

Valerio Costamagna
Università degli studi di Torino
valerio.costamagna@di.unito.it

Cong Zheng
Palo Alto Networks
Georgia Institute of Technology
cozheng@paloaltonetworks.com

Abstract

Various static and dynamic analysis techniques are developed to detect and analyze Android malware. Some advanced Android malware can use Java reflection and JNI mechanisms to conceal their malicious behaviors for static analysis. Furthermore, for dynamic analysis, emulator detection and integrity self-checking are used by Android malware to bypass all recent Android sandboxes. In this paper, we propose ARTDroid, a framework for hooking virtual-methods calls supporting the latest Android runtime (ART). A virtual-method is called by the ART runtime using a dispatch table (vtable). ARTDroid can tamper the vtable without any modifications to both Android framework and app's code. The ARTDroid hooking framework can be used to build an efficient sandbox on real devices and monitor sensitive methods called in both Java reflection and JNI ways.

1 Introduction

The analysis of Android apps becomes more and more difficult currently. Both benign and malicious developers use various protection techniques, such as Java reflection, dynamic code loading and code obfuscation [RCJ13], to prevent their apps from reverse-engineering. Java reflection and dynamic code loading techniques can dynamically launch specific behaviors, which can be only monitored in dynamic analysis environment instead of static analysis. Besides, in obfuscated apps, static analysis can only check the API-level behaviors of apps rather than the fine-grained behaviors, such as the URL in network connections and the phone number of sending SMS behavior.

Without above limitations of static analysis, the dynamic analysis approach is usually used for deeply analyzing apps [SFE+13]. Currently, dynamic analysis uses hooking techniques for monitoring behaviors of apps. The hooking techniques can be divided into two main types: 1) hooking Android framework APIs by modifying Android system [ZAG+15][EGH+14], and 2) hooking APIs used in the app process by static instrumentation [BGH+13][DC13]. Both of them have limitations to analyze trick samples. The first hooking technique has a drawback that it cannot be used on other vendors' devices except for Google Nexus devices or emulators. This gives a chance for malicious apps, which uses the device fingerprint and anti-emulator technique to evade the dynamic detection. Even though the second hooking technique does not have this drawback, but it becomes useless when apps apply anti-repacking techniques. Apps can check the integrity of themselves in runtime and enter the frozen mode to prevent dynamic analysis if the integrity is broken. Moreover, if malicious apps implement malicious behaviors in native codes, the second hooking technique still cannot detect them.

To build a better hooking framework for dynamic analysis, we design **ARTDroid**, an framework for hooking virtual-method calls under the latest Android runtime (ART). The idea of hooking on ART is tampering the virtual method table (vtable) for detouring virtual-methods calls. The vtable is to support the dynamic-dispatch mechanism. And, dynamic dispatch, i.e., the runtime selection of a target procedure given a method reference and the receiver type, is a

In: D. Aspinall, L. Cavallaro, M. N. Seghir, M. Volkamer (eds.): Proceedings of the Workshop on Innovations in Mobile Privacy and Security IMPS at ESSoS'16, London, UK, 06-April-2016, published at http://ceur-ws.org

central feature of object-oriented languages to provide polymorphism. Since almost all Android sensitive APIs are virtual methods, we can collect the apps behavior by using ARTDroid to hook Android APIs methods.

To summarize, this paper makes the following contributions.

- We propose ARTDroid, a framework for hooking virtual-method calls without any modifications to both the Android system and the app's code.
- We discuss how ARTDroid is made fully compatible with any real devices running the ART runtime with root privilege.
- We demostrate that the hooking technique used by ARTDroid allows to intercept virtual-methods called in both Java reflection and JNI ways.
- We discuss applications of ARTDroid on malware analysis and policy enforcement in Android apps.
- We released ARTDroid as an open-source project [1].

The rest of this paper is organized as follows. Section 2 introduces the background about Android and the new Android runtime ART. The ARTDrod framework is introduced in Sec. 3 and its implementation is discussed in Sec. 4. Performance evaluation is presented in Sec. 5, and discussion and applications are in Sec. 6. Section 7 discuss some related works, and we conclude this paper in Sec. 8.

2 Background

Android apps are usually written in Java and compiled to Dalvik bytecode (DEX). To develop an Android app, developers typically use a set of tools via Android Software Development Kit (SDK). With Android's Native Development Kit (NDK), developers can write native code and embed them into apps. The common way of invoking native code on Android is through Java Native Interface (JNI).

2.1 ART Runtime

ART, silently introduced in October 2013 at the Android KitKat release, applies Ahead-of-Time (AoT) compilation to convert Dalvik bytecode to native code.

At the installation time, ART compiles apps using the on-device **dex2oat** tool to keep the compatibility. The `dex2oat` is used to compile Dalvik bytecode and produce an *OAT* file, which replaces Dalvik's *odex* file. Even Android framework JARs are compiled by the `dex2oat` tool to the `boot.oat` file. To allow preloading of Java classes used in runtime, an image file called `boot.art` is created by dex2oat. The image file contains pre-initialized classes and objects from the Android framework JARs. Through linking to this image, OAT files can call methods in Android framework or access pre-initialized objects. We are going to briefly analyze the ART internals, using as codebase the Android version 6.0.1_r10.

```
1 // C++ mirror of java.lang.Class
2 class MANAGED Class FINAL : public Object {
3
4  [...]
5  HeapReference<IfTable> iftable_;
6  HeapReference<String> name_;
7  HeapReference<Class> super_class_;
8  HeapReference<PointerArray> vtable_;
9  uint32_t access_flags_;
10 uint64_t direct_methods_;
11 uint64_t virtual_methods_;
12 uint32_t num_virtual_methods_;
13 [...]
14 }
```

Figure 1: ART Class type

The ART runtime uses specific C++ classes to mirror Java classes and methods. Java classes are internally mirrored by using **Class**[2]. In Figure 1, virtual-methods defined in *Class* are stored in an array of `ArtMethod*` elements, called `virtual_methods_` (line 11). The `vtable_` field (line 8) is the **virtual method table**. During the linking, the vtable from the superclass is copied, and the virtual methods from that class either override or are appended inside it. Basically, the vtable is an array of `ArtMethod*` type. Direct methods are stored in the `direct_methods_` array (line 10) and the `iftable_` array (line 5) contains pointers to the interface methods. We leave interface-methods hooking for future work. The Figure 2 shows the definition of **ArtMethod** class[3]. The main functionality of `ArtMethod` class is to represent a Java method.

```
1 class ArtMethod FINAL {
2  [...]
3  GcRoot<mirror::Class> declaring_class_;
4  uint32_t access_flags_;
5  uint32_t method_index_;
6  [...]
7  struct PACKED(4) PtrSizedFields {
8      void* entry_point_from_interpreter_;
9      void* entry_point_from_jni_;
10     void* entry_point_from_quick_compiled_code_;
11   } ptr_sized_fields_;
12 }
```

Figure 2: ART ArtMethod type

By definition, a method is declared within a class, pointed by the `declaring_class_` field (line 3). The method's index value is stored in the `method_index_`

[1] `https://vaioco.github.io`

[2] art/runtime/mirror/class.h

[3] art/runtime/art_method.h

field (line 5). This value is the method's index in the concrete method dispatch table stored within method's class. The `access_flags_` field (line 4) stores the method's modifiers (i.e., public, private, static, protected, etc...) and the `PtrSizedFields` struct, (line 7), contains pointers to the ArtMethod's entry points. Pointers stored within this struct are assigned by the ART compiler driver at the compilation time.

2.2 Virtual-methods Invocation in ART

In this paragraph we describe how ART runtime invokes virtual-methods by choosing the virtual-method `android.telephony.TelephonyManager`'s `getDeviceId` as an example. Figure 3 shows that the getDeviceId method is invoked on TelephonyManager object's class (line 4). Figure 4 shows dumped compiled codes for arm architecture by *oatdump* tool.

```
1 package org.sid.example;
2 public class MyClass {
3  public String callGetDeviceId(TelephonyManager tm){

4   String imei = tm.getDeviceId();
5   return imei;
6  }
7 }
```

Figure 3: Call to method getDeviceId

```
1 CODE: (code_offset=0x002d93b5 size_offset=0x002d93b0
       size=60)...
2       0x002d93b4: f5bd5c00 subs    r12, sp, #8192
3       0x002d93b8: f8dcc000 ldr.w   r12, [r12, #0]
4       suspend point dex PC: 0x0000
5       GC map objects:  v1 ([sp + #36]), v2 (r6)
6       0x002d93bc: e92d40e0 push    {r5, r6, r7, lr}
7       0x002d93c0: b084     sub     sp, sp, #16
8       0x002d93c2: 1c07     mov     r7, r0
9       0x002d93c4: 9000     str     r0, [sp, #0]
10      0x002d93c6: 9109     str     r1, [sp, #36]
11      0x002d93c8: 1c16     mov     r6, r2
12      0x002d93ca: 1c31     mov     r1, r6
13      0x002d93cc: 6808     ldr     r0, [r1, #0]
14      suspend point dex PC: 0x0000
15      GC map objects:  v1 ([sp + #36]), v2 (r6)
16      0x002d93ce: f8d00234 ldr.w   r0, [r0, #564]
17      0x002d93d2: f8d0e02c ldr.w   lr, [r0, #44]
18      0x002d93d6: 47f0     blx     lr
```

Figure 4: Compiled native code of callGetDeviceId

Before discussions on native code, in Fig. 4, we briefly introduce the devirtualization. To speedup runtime execution, during the on-device compilation time, virtual-methods calls are devirtualized. Devirtualization process uses method's index to point to the relative element inside the vtable within `receiver` instance's class. As result, compiled code contains static memory offset used to get the called ArtMethod's memory reference.

Now, we discuss the native code generated for the method *callGetDeviceId*. The line 4 in Figure 3 is compiled in lines 11-18 in Figure 4. The TelephonyManager instance (an Object[4] type) is stored in the register *r2*. Then, the instance's class is retrived from address in *r2* and stored in the register *r0* (line 13). The method getDeviceId (an ArtMethod type) is directly retrived (line 16) from memory using a static offset from address stored in *r0*. Finally, the getMethodId's entrypoint is called using the ARM branch instruction *blx* (line 18). The entrypoint's address is also retrived by using a static memory offset from the ArtMethod reference (line 17).

In Java, it is allowed to invoke a method dynamically specified using `Java Reflection`. Reflection calls managed by ART runtime use the function `InvokeMethod`[5]. This function calls `FindVirtualMethodForVirtualOrInterface` which returns a pointer to the searched method by looking in the `vtable_` array of receiver's class.

A Java method can also be invoked by native code using the `Call<type>Method` family functions, exposed by JNI. For instance, function `CallObjectMethod(JNIEnv* env, jobject obj, jmethodID mid, ...)` [6] is used to call a virtual-method which returns an Object type. When a Java method is invoked from native code using a function from `Call<type>Method` family, the ART runtime will go through the `vtable_` array to find a matched method matching.

There are two different ways to get a Java virtual-method's reference. One is through the reflection APIs exposed by `java.lang.Class`. For instance, the method `getMethod` returns a reference which represents the public method with a matched method signature. All java.lang.Class' methods, which permits to get a virtual-method reference, can use the `virtual_methods_` array to lookup the requested method. The other way is offered by the JNI function `FindMethodID`. It searches for a method matching both the requested name and signature by looking in the `virtual_methods_` array within the class reference passed as argument.

3 Framework Design

The goal of ARTDroid is to avoid both app's and Android system code modifications. So, the design of ARTDroid is oriented towards directly modify the app's virtual-memory tampering with ART internals representation of Java classes and methods. ARTDroid consists of two components. The first component is the core engine written in C and the other one is the Java side that is a bridge for calling from

[4] art/runtime/mirror/object.h
[5] art/runtime/reflection.cc
[6] art/runtime/jni_internal.cc

user-defined Java code to ARTDroid's core. The core engine aims to: find target methods' reference in virtual memory, load user-supplied DEX files, hijack the vtable and set native hooks. Moreover, it registers the native methods callable from the Java side. ARTDroid is configured by reading a user-supplied JSON formatted configuration file containing the target methods list.

Suppose that you want to intercept calls to a virtual-method. You have to define your own Java method and override the target method by using ARTDroid API. All calls to the target method will be intercepted and then go to your Java method (we call it `patch code`). ARTDroid further supports loading *patch code* from DEX file. This allows the patch code to be written in Java and thus simplifies interacting with the target app and the Android framework (Context, etc).

ARTDroid is based on library injection and uses Android Dynamic Binary Instrumentation toolkit[ADB] released by Samsung. The ABDI tool is used by ARTDroid to insert trace points dynamically into the process address space.

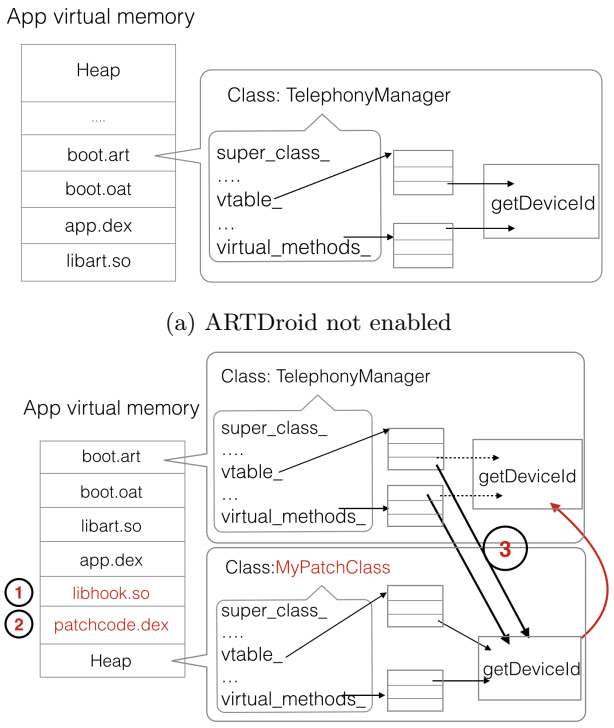
ARTDroid requires the root privilege in order to inject the hooking library in the app's virtual memory, and the hooking library can be injected either in a running app or in the Zygote[LLW+14] master process.

Now, we explain the framework design in figures. Figure 5a shows the app's memory layout without ARTDroid. The class *TelephonyManager* is loaded within the boot image (boot.art). This Class contains both the *vtable_* and *virtual_methods_* arrays where the pointer to method getDeviceId is stored. Instead, Figure 5b represent the app's memory layout while ARTDroid hooking library is enabled. First, the hooking library is loaded inside the app's virtual memory (step 1), and then ARTDroid loads the user-defined patch code by DexClassLoader's methods (step 2). After this, ARTDroid uses its internal functions to retrive target methods reference. So, it can hook these methods by both `vtable_` and `virtual_methods_` hijacking (step 3).

As discussed in 2.2, the `vtable_` array is used by the ART runtime to invoke a virtual-method. Instead, the `virtual_methods_` array is accessed to return a virtual-methods reference from memory. ARTDroid exploits these mechanisms to hooking virtual-methods by both `vtable_` and `virtual_methods_` hijacking means.

4 Implementation

To get the target method's reference, ARTDroid uses the JNI function `FindMethodID`.



(a) ARTDroid not enabled

(b) ARTDroid enabled

Figure 5: App virtual memory layout

ARTDroid overwrites the target method's entry within both the `vtable_` and `virtual_methods_` array by writing the address of the method's patch code. The original method's reference is not modified by ARTDroid and its address is stored inside the ARTDroid's internal data structures. This address will be used to call the original method implementation.

When ARTDroid hooks a target method, all calls to that method will be intercepted and they will go to the patch code. Then, the patch code receives the *this* object and the target method's arguments as its parameters. To call the original implementation of target method, ARTDroid exports the function `callOriginalMethod` to the Java patch code. Internally, ARTDroid's core engine calls the original method implementation using the JNI `CallNonVirtual<type>Method` family of routines. These functions can invoke a Java instance method (non-static) on a Java object, according to specified class and methodID. The original method implementation is invoked by ARTDroid using its address internally stored before the hooking phase. To guarantee a reliable hooking, ARTDroid uses ADBI features to hook the functions of `CallNonVirtual<type>Method` family. By doing this, all calls to these functions are checked by ARTDroid to block calls to an hooked virtual-method only if these calls do not come from ARTDroid's core engine.

5 Evaluation

5.1 Performance Test

To measure the effectiveness of virtual-methods hooking, we firstly need a test set of sensitive methods. SuSi[RAB14] provides sensitive methods in Android 4.2. To verify how many of these methods are declared as virtual, we firstly test them in Android emulator in version 4.2. We use Java reflection to call these methods at runtime. The result of our experiment shows that a remarkable number of virtual-methods could be used to threaten user privacy. The following list describes our experiment results:

- 65.1% of these methods are declared as virtual
- 6.6% are non-virtual
- 28.3% methods not found

Unfortunately, the only methods list available from SuSi is from Android version 4.2. To overcome this limitation, we analyze the sensitive methods list offered by PScout[AZHL12]. The methods of PScout are available from version 2.2 to version 5.1.1. Our another test is on Android 5.1.1 codebase and it is carried on a Nexus 6 running Android 5.1.1. After analyzing them, we know that only 1.0% of methods are non-virtual.

- 59.2% of these methods are declared as virtual
- 1.0% are non-virtual
- 39.8% methods not found

However, some methods cannot be found via Java reflection because corresponding classes or methods are not visible to normal apps. They belong to the Android system apps. So, we can conclude that most of sensitive methods are virtual from our test results. ARTDroid can cover all sensitive methods except 1.0% methods on Android 5.1.1.

The overhead introduced by ARTDroid depends much on the behavior of the patch code. To measure the overhead, we developed a test app, which repeatedly calls sensitive methods or APIs. In particular, this applciation attempts to perform the following operations by calling Android APIs (both via Java reflection and JNI) : initiate several network connections, access sensitive files on the SD card (such as the user's photos), send text message to premium numbers, access the user's contact list and retrive the device's IMEI. We used the profiling facilities offered by Android calling the *android.os.Debug*'s *startMethodTracing*/*stopMethodTracing*. Then, the produced traces can be analyzed using either *traceview* or *dmtracedump*. To measure the effective overhead due to ARTDroid, we call the methods using both Java reflection and JNI in addition to the normal invocation. We ran the test 10,000 times for each method, once with ARTdroid disabled and then with ARTDroid enabled mode. The average running time for each call to an hooked method is showed in the following Table 1.

The most of overhead in ARTDroid is caused by the JNI call, which is internally used to invoke the original method implementation. We registered a worst case overhead of 25% for each hooked method. Therefore, the total overhead of a call to an hooked method is around 0.25 seconds. This overhead could be decreased by adding an internal cache to store methods' reference called by ARTDroid, instead of using JNI function *FindMethodID* at each call. We leave these improvements as future work.

Table 1: Performances

ARTDroid enabled?	Invoke type		
	Normal	Reflection	JNI
Yes	1.12 s	1.39 s	1.19 s
No	0.88 s	1.14 s	0.94 s
overhead	0.24 s	0.25 s	0.25 s

5.2 Case Study

Now, we show a case study by hooking *TelephonyManager*'s *getDeviceId* in ARTDroid.

```
1 {"config": {
2     "debug": 1,
3     "dex": [{"path": "/data/local/tmp/dex/target.dex
          "}]
4     "hooks": [
5     {
6         "class-name": "android/telephony/
              TelephonyManager",
7         "method-name": "getDeviceId",
8         "method-sig": "()Ljava/lang/String;",
9         "hook-cls-name": "org/sid/example/HookCls"
10    }]
11 }}
```

Figure 6: ARTDroid configuration file

Figure 6 shows the configuration file which contains the definition of methods to hook. This file is used to define the information requested by ARTDroid, which are: method's name and signature and the class' name where the patch code is defined in. The patch code called instead of method *getDeviceId* is showed inFigure 7.

```
1 public String getDeviceId() {
2     String key = "android/telephony/
          TelephonyManagergetDeviceId()Ljava/lang/
          String;";
3     Object[] args = {};
4     return (String) callOriginalMethod(key, this,
          args) + " IMPS2016!!";
5 }
```

Figure 7: Patch code for method getDeviceId

To restore the original call-flow, ARTDroid exposes to Java patch-code the native function `callOriginalMethod`. This function receivers as first argument the string key to identify the target method in the dictionary of hooked methods, internally managed by ARTDroid. Second argument represents the *this* object and the last argument is the array of method's arguments. All future calls to method getDeviceId will be redirected to the patch code, independently if these calls are made using Java reflection mechanisms or JNI.

6 Discussion

We note that the main goal of our work is to propose a novel technique to hook Java virtual-methods, our approach can be used to enforce fine-grained user-defined security policies either on real-world devices or emulators as well. Previous research has shown that even benign apps often contain vulnerable components that expose the users to a variety of threaths: common examples are component hijacking vulnerabilities[LLW+12], permission leaking [GZWJ12],[Jia13] and remote code execution vulnerabilities[PFB+14].

Suppose the target app is implementing the following features:

1. dynamic code loading
2. code obfuscation (Java reflection, code encryption, etc...)
3. integrity checks (i.e, due to copyright issue)
4. invoke of security-sensitive Java methods via JNI
5. detection/evasion of emulated environments (i.e, due to copyright issue)

An approach based only on static analysis cannot properly extract security relevant information due to the use of 1, 2 and 4. Moreover, all existing approaches based on bytecode rewriting techniques cannot analyze that app mainly for the use of integrity checks. Note that since the use of 5, in contrast to ARTDroid, all the existing approaches based on emulated environments can not properly analyze the behavior of that app. Instead, ARTDroid is still able to analyze that app. Obviously, ARTDroid has its limitations and corner cases. The main limitations is due to the running of the hooking library inside the same process space of the target app. In a scenario where an attacker want to bypass our approach, it can directly invoke a syscall through inline assembly code to gets sensitive results bypassing ARTDroid. We note that the direct system call is not a common technique used by current daily Android malware. Nevertheless, we envise that ARTDroid can be used in conjunction with existing works like [TKFC15],[ZAG+15], [XSA12] to provide an additional layer of analysis.

Even though Java direct methods are almost not used for both malicious and security-sensitive behaviors, our future work will support both interface-methods and direct-methods hooking. A possible solution is that we can statically instrument the `dex2oat` and replace the system original one once we get root privilege. The instrumented `dex2oat` can intercept all interface-methods and direct-methods.

Since ARTDroid hooking library can be injected directly either in Zygote or when the target app is going to be spawned. Even if the app under testing can tamper with the `vtable_` , it can not get the original method's address. In fact, after ARTDroid is enabled, the original method is no more pointed by both the `vtable_` and `virtual_methods_` arrays.

In section 5, we have presented an evaluation about the effectiveness of virtual-methods hooking in the Android system by analyzing results obtained from both SuSi[SuS] and PScout[AZHL12] projects. Research results indicate that there is a considerable percentage of sensitive methods which are virtual. Since, ARTDroid can hook virtual-methods and tamper with their arguments, it could be used to define security policies to verifiy apps' behaviors at runtime. For instance, ARTDroid can be used to automatically identify apps which are sending SMS to premium numbers.

Since the main downside of dynamic analysis techniques is the code-coverage issue, we envise that ARTDroid can be integrated with automatic exploration system like Smartdroid[ZZD+12], proposed by Cong et al.

In the following, we show some applications of ARTDroid:

- Collect apps behavior at runtime. Analysis of Android API function calls permits the extraction of information about the behavior of apps.
- Verify security policies at runtime. When users install an app, they can enforce some policies in ARTDroid, so that the new app's sensitive behaviors, such as sending SMS, can be restricted by ARTDroid.

- Android malware analysis. Some trick malware use a lot of dynamic analysis evading techniques. But in ARTDroid enforced sandbox, our hooking technique cannot be bypassed by current evading techniques. Also, we can easily build our ARTDroid sandbox either on Android emulator or on real devices.

7 Related Work

Several approaches have been proposed to provide methods hooking on Android. A family of approaches is based on bytecode rewriting technique. The app can be instrumented offline by modifying the app bytecode. AppGuard[BGH+13] proposed by Baches et ak, uses this approach to automatically repackage target apps to attach user-level sandboxing and policy enforcement code. Zhou et al. proposed AppCage[ZPW+15], a system to confine the runtime behavior of the thid-party Android apps. Davis et al. proposed Retroskeleton[DC13], an Android app rewriting framework for customizing apps, which is based on their previous work, I-ARM-Droid[DSKC12].

While these approaches are valuable and each of them has its own advantages as well as disadvantages, they have different significant down sides. This approach is not feasible against apps that verify their integrity at runtime. This kind of defense (anti-tampering) is also used in benign apps as well. To be able to replace API-level calls with a secure wrapper, bytecode rewriters need to identify desidered API call-site within the target app. As mentioned in [HSD13],[ZAG+15], apps that use either Java reflection or dynamically code loading can bypass the app rewriting technique. Moreover, apps which are using JNI to call Java methods can bypass this techniques as well.

A different approach to implement methods tracing can be achieved by using a custom Android system or by using an emulated environment (e.g., a modified QEMU emulator). Enck et al. proposed TaintDroid[EGH+14], an Android modified system to detect privacy leak. StayDynA[ZAG+15] a system for analyzing security of dynamic code loading in Android, uses a custom system image which can be used only on Nexus like devices. Tam et al. presented CopperDroid[TKFC15], a framework built on top of QEMU to automatically perform dynamic analysis of Android malware. These families of approaches, which are based on emulators, can be bypassed by emulation detection techniques [PVA+14] [VC14]. A comparison on Android sanbox has been published by Neuren et al. in [NVdVL+14].

Mulliner et al. proposed PatchDroid[MORK13], a system to distribute and apply third-party securities patches for Android. This system uses the DDI[DDI] framework. DDI allows to replace arbitrary methods in Dalvik code with native function call using JNI. In [MRK14], Mulliner et. al. shown an automated attack against in-app billing using the DDI capabilities to control the in-app billing purchase flow. Note that the methods used to achieve in-app billing are defined as virtual.

Frida[Fri], a dynamic code instrumentation toolkit, Xposed framework [Xpo] and Cydia substrate for Android [Cyd] share similarity with the DDI intrumentation approach. These projects were created for device modding and, in contrast with DDI, require replacing of system components suck as zygote. Currently, Xposed compatibility with ART runtime is actually in beta stage[7] and the framework installation condition is to flash the device by a custom recovery image. While these approaches are very suitable under the Dalvik VM, they are totally limited for using under the ART runtime. In fact, both DDI, Frida and Cydia substrate are not able to work under the ART runtime.

Aurasium [XSA12] builds a reference monitor into application binaries. The Dalvik code is not patched, but new classes and native code are added to ensure that the instrumentation code is run first. Clearly, such approaches are not effective if the code is obfuscated and protected against static analysis and disassembly. Also note that the package signature of the instrumented applications are broken when they are patched statically. In comparison, our approach does not need to repack the app, our modifications are in-memory only and thus we do not break code signing.

Recent works proposed novel approaches that aim to sandbox unmodified apps in non-rooted devices running stock Android. Boxify[BBH+15] presented an approach that aims to sandbox apps by means of syscall interposition (using the ptrace mechanism) and it works by loading and executing the code of the original app within the context of another, monitoring, app. A similar work, [BFKV15] uses the same approach to sandbox arbitrary Android apps. The approach presented in both of these recent works, represent one of the most promising and interesing future work direction.

8 Conclusion

In this paper, we present ARTDroid, a framework for hooking virtual-methods under ART runtime. ARTDroid supports the virtual-method hooking without any modifications to both Android system and app's code. ARTDroid allows to analyze apps even if they employ anti-tampering techniques or they use ei-

[7] http://forum.xda-developers.com/showthread.php?t=3034811

ther Java reflection or JNI to invoke virtual-methods. Moreover, ARTDroid can be used on any real devices with ART runtime once getting the root privilege. The applications of ARTDroid include dynamic analysis of Android malware on real devices or security policies enforcement.

References

[ADB] Android dynamic binary instrumentation. `https://github.com/Samsung/ADBI`. Accessed: 2016-02-27.

[AZHL12] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[BBH+15] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 691–706, 2015.

[BFKV15] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38. ACM, 2015.

[BGH+13] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 543–548. Springer, 2013.

[Cyd] Cydia substrate for android. `http://www.cydiasubstrate.com`. Accessed: 2016-02-27.

[DC13] Benjamin Davis and Hao Chen. Retroskeleton: retrofitting android apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 181–192. ACM, 2013.

[DDI] Dalvik dynamic instrumentation framework. `https://github.com/crmulliner/ddi`. Accessed: 2016-02-27.

[DSKC12] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-armdroid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012, 2012.

[EGH+14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[Fri] Frida.re. `https://frida.re`. Accessed: 2016-02-27.

[GZWJ12] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[HSD13] Hao Hao, Vicky Singh, and Wenliang 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*, pages 25–36. ACM, 2013.

[Jia13] Yajin Zhou Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[LLW+12] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[LLW+14] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 424–439. IEEE, 2014.

[MORK13] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 259–268. ACM, 2013.

[MRK14] Collin Mulliner, William Robertson, and Engin Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 459–470. ACM, 2014.

[NVdVL+14] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter sandbox: Android sandbox comparison. *arXiv preprint arXiv:1410.7749*, 2014.

[PFB+14] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[PVA+14] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[RCJ13] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[SFE+13] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[SuS] Susi. `https://github.com/secure-software-engineering/SuSi`. Accessed: 2016-02-27.

[TKFC15] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[VC14] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[Xpo] Xposed framework. `https://repo.xposed.info`. Accessed: 2016-02-27.

[XSA12] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 539–552, 2012.

[ZAG+15] Yury Zhauniarovich, Maqsood Ahmad, Olga Gadyatskaya, Bruno Crispo, and Fabio Massacci. Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 37–48. ACM, 2015.

[ZPW+15] Yajin Zhou, Kunal Patel, Lei Wu, Zhi Wang, and Xuxian Jiang. Hybrid user-level sandboxing of third-party android apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 19–30. ACM, 2015.

[ZZD+12] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104. ACM, 2012.