2-2023

FA3: Fine-Grained Android Application Analysis

Yan LIN

Weng Onn WONG
Singapore Management University, joshuawong@smu.edu.sg

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

FA^3: Fine-Grained Android Application Analysis

Yan Lin*
Jinan University
Guangzhou, China
yanlin@jnu.edu.cn

Joshua Wong
Singapore Management University
Singapore
joshuawong@smu.edu.sg

Debin Gao
Singapore Management University
Singapore
dbgao@smu.edu.sg

ABSTRACT

Understanding Android applications' behavior is essential to many security applications, e.g., malware analysis. Although many systems have been proposed to perform such dynamic analysis, they are limited by their applicable analysis environment (on device vs. emulator), transparency to subject apps, applicable runtime (Dalvik vs. ART), applicable system stack, or granularity. In this paper, we propose FA^3 (Fine-Grained Android Application Analysis), a novel on-device, non-invasive, and fine-grained analysis platform by leveraging existing profiling mechanisms in the Android Runtime (ART) and kernel to inspect method invocations and control-flow transfers for both Java methods and third-party native libraries. FA^3 embeds its tracing capability in multiple layers of the Android system stack to not only capture fine-grained application behaviors but ensure even non-conventional or malicious tricks of loading and executing OAT/ELF binaries cannot escape our monitoring. We carefully evaluated FA^3 using real-world malware. Experimental results showed that FA^3 successfully analyzes sophisticated malware samples and provides a comprehensive view of their behavior.

ACM Reference Format:
Yan Lin, Joshua Wong, and Debin Gao. 2023. FA^3: Fine-Grained Android Application Analysis. In *The 24th International Workshop on Mobile Computing Systems and Applications (HotMobile '23), February 22–23, 2023, Newport Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3572864.3580338

1 INTRODUCTION

It is essential for analysts to fully understand behaviors of Android applications, especially for malware. Many systems have been proposed for the objective of *statically* analyzing Android apps, but most of them can be evaded by advanced techniques in one way or the other. For instance, malware can evade static analysis tools that inspect the Dalvik bytecode by using various obfuscation techniques to raise the bar of code comprehension [13, 19], implementing malicious activities in native libraries [7, 18], and leveraging packing techniques to hide malicious payloads [26, 32].

Similar sophisticated techniques could also make *dynamic* analysis systems ineffective for a number of reasons. First, malicious applications usually exhibit their behavior across multiple system layers (e.g., Android Runtime and native library), failing in capturing any one of which could render the dynamic analysis incomplete and result in malicious behaviors escaping analysis and detection. However, the majority of dynamic analysis systems [12, 25, 31] lack capability of cross-layer inspection. For example, CopperDroid [25] monitors malware behaviors mainly through the trace of system calls, making it hard to expose the execution details for methods in Android Runtime. A more critical failure resulted from this lack of cross-layer and comprehensive analysis attributes to the non-conventional and malicious way of loading and executing OAT/ELF binaries that were not included in the app's APK file [21], leading to its behavior escaping radar of the dynamic analysis system.

Second, anti-debug and anti-emulator techniques [15, 16] further limit the usage of many dynamic analysis systems, most of which [27, 30] rely on emulators or instrumentation tools to monitor malware behaviors. Such limitation demands for not only *on-device* dynamic analysis that are friendly with end-user handsets, but *non-invasive* properties such that the original behavior exhibited by the app is not affected by the monitoring platform.

Third, existing systems are not effective in capturing executions in a fine-grained manner. Most existing systems only monitor the method invocation, and lacks support of tracing intra-procedural control-flow transfers. However, control-flow transfer information is important in understanding the app's behavior [20], especially for (malicious) apps that apply control-flow obfuscation and when there are return-oriented programming (ROP) executions [22].

In this paper, we propose FA^3, a novel on-device, non-invasive, and fine-grained platform which captures the app's execution on multiple layers in a fine-grained manner. Moreover, FA^3 does not need to statically instrument apps' bytecode or native library.

FA^3 embeds its monitoring capability in multiple Android system layers. It records the invocations of Java methods, including framework APIs and methods in apps, and captures stealthy behaviors such as dynamic code loading and JNI invocations in the Android Runtime. It monitors control-flow transfers in Java methods, which helps to understand the behavior of apps protected by control-flow obfuscation [10]. FA^3 also monitors control-flow transfers in third-party native libraries and system calls. Last but not the least, FA^3 supports (efficiently) dumping memory contents and register values for fine-grained analysis. For example, we can dump the stack memory when a return instruction gets executed and check whether there is a potential return-oriented programming attack [22] by combining the control-flow transfer information we collected. Table 1 provides a detailed comparison between FA^3 and state-of-the-art analysis tools.

It is probably not surprising that the above features come at a cost of added runtime overhead, which is likely an important reason why existing approaches did not venture into such a comprehensive

*This work was done when she was a Research Scientist at Singapore Management University.

HotMobile '23, February 22–23, 2023, Newport Beach, CA, USA

ACM ISBN 979-8-4007-0017-0/23/02…$15.00
https://doi.org/10.1145/3572864.3580338

Table 1: Comparison of FA^3 with state-of-the-art analysis tools.

Tool	On device	Non invasive	Support ART	Multiple layer	Fine grained	Overhead
TaintDroid [12]	✓	✓	×	Ⓕ Ⓐ	×	18%
CopperDroid [25]	×	✓	✓	Ⓕ Ⓝ	×	32%
TaintART [24]	✓	×	✓	Ⓕ Ⓐ	×	14%
ARTist [9]	✓	×	✓	Ⓕ Ⓐ	×	31%
DroidScope [30]	×	✓	×	Ⓕ Ⓐ Ⓝ	×	3400%
Ndroid [27]	×	✓	✓	Ⓕ Ⓡ Ⓝ	×	1000%
Malton [29]	×	✓	✓	Ⓕ Ⓐ Ⓡ Ⓝ	×	3600%
FA^3	✓	✓	✓	Ⓕ Ⓐ Ⓡ Ⓝ	✓	428.5%

Ⓕ, Ⓐ, Ⓡ and Ⓝ indicate that the tool can capture behaviors for **f**ramework APIs, Java methods in the **a**pp, methods in Android **R**untime, and **n**ative libraries. Shading denotes partial/full support. For example, Ⓕ Ⓐ of TaintART suggests that it can monitor partial framework behaviors and all Java methods.

solution. FA^3 is able to overcome this with reasonable overhead by leveraging existing profiling mechanisms in the Android Runtime and kernel, as well as a novel dynamic instrumentation mechanism. We perform our on-device evaluation of FA^3 on Pixel 5 devices, and report moderate overhead while all subject apps in our test run smoothly and successfully.

FA^3 can be used by different parties. End users can use it to understand the behavior of an app; corporations can use it to perform quality-check on an app before it's rolled out to customers or employees. The Android Play Store can also use FA^3 to see whether an app uploaded has hidden behaviors. Moreover, FA^3 can also be integrated into Mobile Device Management (MDM) framework to monitor the Android device and apps automatically.

2 RELATED WORK AND BACKGROUND

In this section, we first discuss related work on existing dynamic Android application analysis tools, and then cover some background of Android Runtime (ART) and profiling system that are essential to the design of FA^3, finally we discuss the attack model.

2.1 Related Work

This section focuses on the related dynamic and hybrid Android application analysis techniques .

TaintDroid [12] performs dynamic taint analysis to detect information leakage by modifying the Dalvik virtual machine (DVM). It trusts the native libraries and does not capture the behaviors for them. Although TaintDroid can track taint propagation for Dalvik bytecode, it neither monitors the runtime behaviors nor supports ART. In order to support ART, TaintART [24] and ARTist [9] modify `dex2oat` to insert taint propagation instructions into the compiled code. However, they only propagate taint information for Java code and do not support the taint propagation through JNI or native code. Moreover, they cannot handle packed applications because such apps dynamically load the Dalvik bytecode directly without triggering the invocation of dex2oat.

DroidScope [30] reconstructs OS-level and Java-level semantics based on Qemu [11]. It does not monitor JNI and therefore cannot capture the complete behaviors for Java code. CopperDroid [25] is also built on top of Qemu and records system call invocations. However, only a limited number of behaviors can be monitored by it. NDroid [27] tracks information leakage in multiple layers, but it also relies on Qemu to perform the instrumentation.

Malton [29] probably has the most similar design with our system FA^3. It inspects Android malware from different system layers — recording sensitive framework APIs and concerned methods of malware in the framework layer, capturing stealthy behaviors such as dynamic code loading and JNI reflection in the runtime layer, and monitoring library APIs and system calls in the system layer. However, tracking of system calls is based on Valgrind [17], resulting in its lack of support for on-device monitoring. Moreover, FA^3 can monitor in a finer-grained manner, such as monitoring control-flow transfers, register values, and memory content.

2.2 The ART Runtime

ART is the new runtime introduced in Android version 4.4, and it becomes the default runtime from version 5.0 onward. ART uses ahead-of-time (AOT) compilation, and starting in Android 7.0, it uses a hybrid combination of AOT, just-in-time (JIT) compilation, and profile-guided compilation. Specifically, an app is initially installed without any AOT compilation, but a new file in the OAT format (extended ELF) is generated with only Dalvik bytecode included. When the device is idle and charging, a compilation daemon AOT-compiles frequently used code to native instruction based on a profile generated during the first few runs, and inserts it into the OAT file. FA^3's capability is based on the monitoring and tracing of these native instructions.

ART provides mechanisms to trace method executions since Android 5.0, and is used by the Android Profiler tools [6]. Its original intent is to assist understanding of CPU, memory, and network usage by using the Debug class to instrument a process of a running app. However, its working assumes availability of source code and does not typically work for third-party apps[1]. It also fails on third-party native libraries since it uses Simpleperf [4] which requires debugging information embedded. FA^3 realizes its monitoring capability by leveraging and modifying this embedded tracing mechanism. We will discuss more details in Section 3.

2.3 Attack Model

The main purpose of FA^3 is for benign agents/users to make use of it to understand the behavior of an application. Since FA^3 is integrated into Android OS (of the end-user devices), it indicates that a hacker cannot use it to infer private data on an end-user device except the device is lost or stolen. Apps might detect FA^3 via side channels to evade the analysis.

3 ARCHITECTURE

Figure 1 illustrates the overall architecture of FA^3. To track execution of an Android app at multiple layers, FA^3 introduces four main modules. We discuss each module in detail.

3.1 Dalvik Bytecode Tracer

The first module in our tool is responsible for tracing Java methods executed in the interpreter mode. To do this, we examine the

[1] https://developer.android.com/studio/profile/cpu-profiler

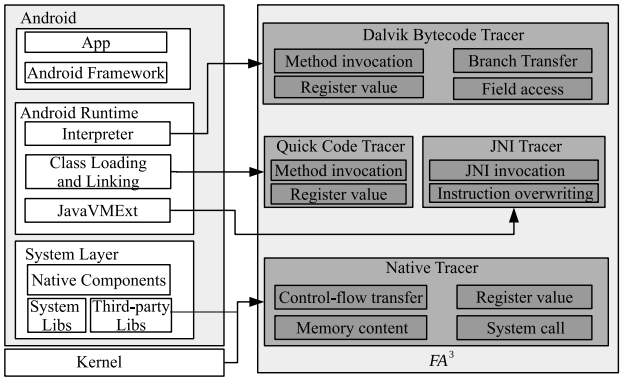

Figure 1: Overview of FA^3 (shaded components)

different handlers in Android interpreter, and find corresponding handlers that process all branch transfer instructions, including `HandleGoto`, `HandleIf`, `PACKED_SWITCH`, and `SPARSE_SWITCH`. FA^3 modifies these handlers to record the source and target addresses of the branch transfer instructions. Since these native handlers are executed for every branch transfer, FA^3's intercepting on them manages to achieve finer-grained Android app analysis with good performance overhead.

Similarly, FA^3 modifies the handlers for `iget` and `iput` (Dalvik's field access instructions) to provide low-level analysis on field related information, including the Java method in which the field is accessed, the type of the field, and the value of the field.

FA^3 additionally supports analysis of vregisters, which are stored in the shadow stack frame memory, at method enter/exit as well as branch transfers. Specifically, FA^3 first finds the shadow frame for the method that is currently executed by walking through the stack, and then obtains the register values by calling `ShadowFrame::GetVReg()` (API provided by ART).

3.2 Quick Code Tracer

The second module tracks the execution for Java methods which have been compiled into native instructions[2] using AOT compilation. When ART wants to execute such native instructions, it needs to first locate the entrypoint which is done in the Java class linking and OAT file loading process. FA^3 modifies these loading and linking procedures to redirect the entrypoint of compiled Java methods to a stub called `art_quick_instrumentation_entry`, which performs fine-grained app analysis including logging native register values and method invocation information (e.g, method ID) before calling the actual compiled method. In addition, the stub sets the value of the intended link register (lr) to the address of another stub named `art_quick_instrumentation_exit` so that when the compiled method finished its execution, the control returns to the stub to continue the execution. FA^3 additionally performs native register and method exit logging in this stub, too.

To obtain native register values, FA^3 saves each register (`x0-x30` for arm64) onto the stack at the beginning of `art_quick_instrumentation_entry` and `art_quick_instrumentation_exit`, and passes the current stack pointer to a function called `GetRegisterValue()` to read register values from the stack.

[2]We use the term "native instruction" to differentiate from native code compiled from C/C++ code.

```
1 6241|smali|com.example.hellolibs.MainActivity.onCreate(
      Landroid/os/Bundle;)V| .enter |
2 6241|native|libhello-libs.so| JNI 0x7c0
```

Figure 2: Example of JNI invocation analysis result

3.3 JNI Tracer

The Android runtime allows apps to call native code (the classical shared object file format `.so` extension) from Java code via JNI. Our JNI tracer enables analysis in this Java-native interface, while the native tracer (discussed in Section 3.4) supports analysis in the execution of native code.

FA^3 modifies the native library loading and address searching process to perform JNI tracing. Specifically, ART uses function `SharedLibrary::FindSymbol()` to find the address of the native method going to be invoked through JNI. FA^3 can easily record the source and target addresses for the JNI invocation by modifying the `SharedLibrary::FindSymbol()` function. Figure 2 shows that the Java method `onCreate` in thread 6241 calls the native method at offset $0x7c0$ in the third-party native library `libhello-libs.so` through JNI. Once the target address for a native method is found, FA^3 will perform an on-demand instrumentation of control-flow transfer instruction in the shared library to help us analyze its behavior; see Section 3.4.

3.4 Native Tracer

Native libraries could typically make system calls and are frequently used to implement malicious behaviors [7, 18, 23]; therefore, it is important to include it in the multi-layer fine-grained analysis.

One possible solution of tracking the execution of native library is to make use of process trace (ptrace). However, ptrace cannot attach to apps with anti-debugging capability, such as apps that provide government digital services [28].

Our idea is to replace control-transfer instructions in the native library with system calls, so that the execution would trap into the kernel, enabling the kernel to perform fine-grained analysis. Such a solution works especially well on an ARM architecture due to its RISC nature. To obtain a balanced performance, our dynamic rewriting of control-transfer instructions is performed on-the-fly for each basic block. The handler in the kernel performs three main tasks — calculating the target of the control-flow transfer, logging the control-flow transfer information, and on-the-fly binary rewriting for the control-transfer target. To calculate the actual target of the control transfer, FA^3 stores a copy of the original native library into another read-only memory for look-up purposes. The dynamic rewriting is performed by temporarily setting the code page to be writable.

Figure 3 shows an example of native library `liblemon.so` with three branch transfer instructions. Assuming that instruction **A** is the first control-transfer instruction executed after a JNI call, **A** is replaced with a system call instruction (`svc #07`; ①) and the replacement is performed by our instrumented `SharedLibrary::FindSymbol()` function in the `JavaVMExt` class.

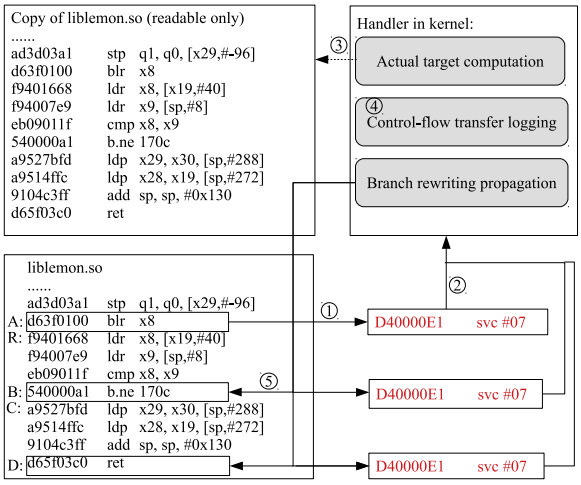

Figure 3: Example of native library rewriting

After the system call (②), our handler in the kernel refers to the copy of the original library to compute the actual target (③), which turns out to be a `blr` instruction with the actual target stored in register `x8`. FA^3 therefore sets the program counter (PC) to the value of `x8`, and sets the value of link register (lr) (which stores the return address when a subroutine call is made) to the address of instruction **R**). Meanwhile, FA^3 logs the source and target addresses of this control transfer to a log buffer in the user space (④). Lastly, on-the-fly control-transfer instruction rewriting will be performed on the next basic blocks (that pointed to by `x8` and **B**).

3.5 User Interface

To provide a convenient experience, FA^3 consists of a third-party application called AppTracer to allow end users to choose which application to analyze. Upon selecting the application, end users are presented with the list of third-party native libraries, classes, fields, and registers that are present in the application code to analyze. Once selected, they will be saved into a configuration file for FA^3 to consult at runtime. Figure 4 shows a few screenshots of AppTracer[3]. We plan on releasing the source of FA^3 after our paper publication.

4 EVALUATION

We first compare the traces collected by FA^3 with other existing instrumentation tools to demonstrate that the trace collected by FA^3 is complete. We further evaluate FA^3 with two real malware (i.e., gta3 and QR Scanner) downloaded from Koodous[4] to show its capability in detecting malicious behavior. These two types of malware are chosen because they used packing, JNI reflection, and obfuscation to make the analysis more difficult. Moreover, we evaluate the performance of FA^3 using GeekBench-5 [3]. All evaluations are done on unrooted Pixel 5 running Android 11. Note that we don't need the phone to be rooted when FA^3 is enabled.

[3]More screenshots can be found in https://www.dropbox.com/s/82nn2p1bsgc9poi/AppTracer.jpg?dl=0
[4]https://koodous.com/

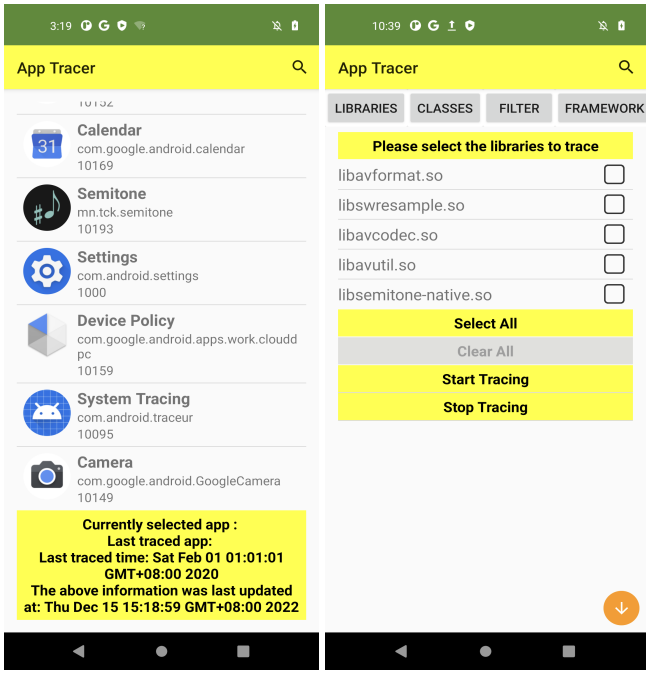

Figure 4: Screenshot of AppTracer

4.1 Completeness of FA^3

We downloaded 11 open-source applications from F-Droid [5] across different categories, and compared the traces of Java methods collected using the *AspectJ instrumentation framework [2]* and FA^3. AspectJ is an Aspect-Oriented Programming (AOP) extension for Java and can be used in Android for instrumenting mobile applications [8]. We choose AspectJ, a source-level instrumentation tool, to ensure that the basis of comparison is complete and correct.

The details about these apps and the comparison result can be found in Table 2. Result shows that the Java methods collected by AspectJ are 100% present in traces from FA^3 when obfuscation is disabled in compiling the apps. One interesting thing we notice is that FA^3 captures the execution of all Java methods including constructors, while AspectJ cannot. For ProGuard obfuscated applications and methods, e.g., GreenTooth, AspectJ reports the original method names before obfuscation while FA^3 outputs the obfuscated method names. This is because FA^3 works on the apk without source-level information; nevertheless, the one-to-one correspondence in the trace comparison shows that FA^3's output is complete.

4.2 Malicious Behavior Detection

(1) *gta3*[6] is a malicious app that sends SMS messages in a stealthy manner, which adopts Java/JNI reflection to hide its malicious behaviors with the class and method names obfuscated/encrypted. At runtime, instead of calling the methods directly, it takes a string previously encrypted and decrypts it using a lookup table before using reflection to find the method that bear the decrypted name.

We use the Dalvik bytecode tracer and quick code tracer components in FA^3 to trace the invocations of framework APIs and Java

[5]https://f-droid.org/
[6]MD5: dd40531493f53456c3b22ed0bf3e20ef

Table 2: Comparison results for FA^3 and AspectJ

App's Name	Description	Result
A Photo Manager	Manage local photos	Y
Specie	Currency conversion	Y
Compass	Compass with a range of styles	Y
Neidum	Read Medium articles without limits	Y
Smart EggTimer	Cook egg timer	Y
Baby Phone	Pretend phone app for little people	Y
Metronome	Metronome with presets and tap tempo	Y
SimpleTextEditor	Simple Text Editor	Y
QR Scanner	Privacy Friendly QR Scanner	Y
HashEasily	Calculate hash checksums of files	Y
GreenTooth	Automatic Bluetooth disabler	N

Y means FA^3 can capture all method executions obtained by AspectJ
N means the captured method executions in FA^3 and AspectJ are different

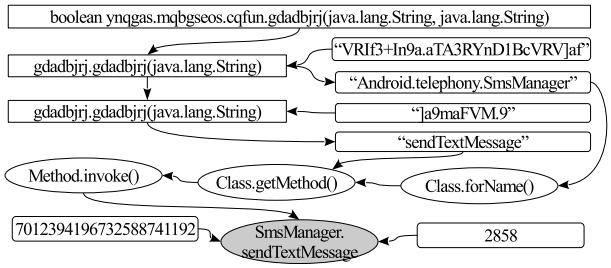

Figure 5: FA^3 detects Java reflection of the gta3 malware. Ellipses refer to framework method. Round corner rectangles represent data. Other rectangles indicate method in the app.

methods in the app. As shown in Figure 5, the string used in the reflection call is encrypted and needs to be decrypted by calling function `gdadbjrj.gdadbjrj`. FA^3 reveals that the malware obtains the object of `android.telephony.SmsManager` class through Java reflection method `Class.forName()` by using the decrypted result `android.telephony.SmsManager` as the input. It then retrieves the method object of `sendTextMessage()` using Java reflection method `Class.getMethod()`. Finally, it calls the framework API `sendTextMessage()` by sending message "701239419673258874119 2" to number 2858. Since we also record the register values, we can easily find out the content of the message.

(2) *QR Scanner*[7] is a malicious app that checks the simcard operator code and visits a site to subscribe for a premium service. It is packed by Tencent's Legu [1] and the malicious activities are performed in the background without the user's knowledge by downloading three different payloads. In this experiment, we configure FA^3 to enable the method invocation and register value tracing in Dalvik bytecode tracer and quick code tracer, JNI tracer, and control-flow transfer tracing in native tracer to gain a comprehensive understanding of the malware. The JNI tracer reveals that the Java code uses the native code in the third-party native library to load the real Dalvik bytecode that is encrypted by the packer. The native tracer records how the native code decrypts and loads the real Dalvik bytecode into the memory.

Figure 6 shows the log file generated by FA^3 in monitoring execution of the packed malware.[8] Once the malware is started, class `com.easyqr.scannertool.MyWrapperProxyApplication` is loaded for preparing the real payload (Line 1). Then, the Android framework API `android.app.Application.attach()` is invoked (Line 4) to set the property of the app context. After that, the malware calls the Java method `System.loadLibrary()` to load its native component `libshell-super.2019.so` at Line 6. After initialization, the JNI tracer finds that the malware calls the JNI method `JNI_OnLoad()` to decrypt and load the hidden Dalvik bytecode that is encrypted into memory. More specifically, the control-flow transfer tracing in the native tracer finds that it opens the file `0OO00l111l1l` that has the encrypted Dalvik bytecode and decrypts it (Line 9-11). Java method `installDexes()` is called through JNI reflection by the native code to create a new Dex file and the decrypted Dalvik bytecode is written into it using `memcpy` (Line 12-18). Finally, the Dalvik bytecode tracer finds that the app loads and initializes the class `com.kitkats.mike.MainActivity` (Line 20), following which the malicious behavior gets executed, including the downloading of three Dex files and invoking sensitive APIs.

The result demonstrates that FA^3 has the capability to detect malware with powerful attacking techniques due to the fine-grained and multiple-layer tracing.

4.3 Performance Overhead

To understand the overhead introduced by FA^3, we run the benchmark tool GeekBench-5 [3] 15 times under a number of setting:

- stock Android with FA^3 disabled (Base);
- FA^3 enabled to monitor Java method invocations, with (MR) and without (M) register values recording;
- FA^3 enabled to monitor method invocations and branch transfers for Java code, with (MBR) and without (MB) register values recording;
- FA^3 enabled to monitor field access (MF);
- FA^3 enabled to monitor method invocations and branch transfers for Java and third-party native libraries, with (MBNR) and without (MBN) register values recording.

The overall result is shown in Figure 7. The y-axis is the running score of GeekBench-5. Specifically, GeekBench runs a series of tests on a processor and times how long the processor takes to complete the tasks. The quicker the CPU completes the tests, the higher the GeekBench score is.

The results show that FA^3 introduces between 21.4% and 34.3% slowdown when monitoring Java method invocation, while monitoring branch transfers result in an additional 5.5% slowdown. Higher runtime overhead comes from the control-flow transfer tracing for third-party native libraries at around 48.2% slowdown compared with stock Android. Note that these results are obtained by monitoring *all* method invocations and control-flow transfers; while in more realistic application scenarios, FA^3 can be configured to monitor specific methods only with smaller overhead.

We also measured the overall system performance overhead introduced by FA^3 by using a macrobenchmark that exercises 1,000 top-downloaded apps from the Google Play Store via the Android UI/Application Exerciser Monkey[9]. We configured Monkey to exercise apps and generate the exact same sequence of events on the

[7] MD5: 3bbf45eab9796a2781e640393fae7423
[8] A more complete version can be found in https://www.dropbox.com/s/j04nnx4vmwz2am1/packer_tracing.png?dl=0
[9] https://developer.android.com/studio/test/monkey.html

```
java.lang.ClassLoader.loadClass("com.easyqr.scannertool.MyWrapperProxyApplication")
com.easyqr.scannertool.MyWrapperProxyApplication.<init>()
com.wrapper.proxyapplication.WrapperProxyApplication.<init>()
android.app.Application.attach()                    // Internal framework API
android.content.ContextWrapper.attachBaseContext() // Set the base context for this ContextWrapper.
    System.loadLibrary("shell-super.2019")     // Load native library libshell-super.2019.so
    /* JNI invocation: decrypt the actual dex file in the assets */
    JNI_OnLoad()
        0x2613c()                              // Open the real Dex file and decrypt it.
            open("0OO00l111ll1",0x2)            // Open real Dex file
            mmap()
        0x6a44()
            /* JNI reflection, create a new Dex file */
            com.wrapper.proxyapplication.MultiDexForTinker.installDexes()
        0x17a38()
            do{
                memcpy()        // Copy the decrypted Dalvik bytecode into the memory
            }while
/* Initialize the new activity arg: Activity="com.kitkats.mike.MainActivity" */
android.app.Instrumentation.newActivity()
/*hidden Dex file loading*/
dalvik.system.DexClassLoader.<init>("/data/user/0/com.easyqr.scannertool/files/xia", , ,)
dalvik.system.DexClassLoader.<init>("/data/user/0/com.easyqr.scannertool/files/changeone", , )
dalvik.system.DexClassLoader.<init>("/data/user/0/com.easyqr.scannertool/files/kakghkg", , ,)
/* hidden behavior */
android.telephony.TelephonyManager.getSimOperator()
android.telephony.TelephonyManager.getSimOperatorNumeric()
android.telephony.TelephonyManager.getLine1Number() // Returns the phone number string
android.telephony.TelephonyManager.getSubId()
```

Figure 6: Method invocation collected by FA^3. We only show the arguments that can help to understand the behavior due to space limitation. Note that these argument values are obtained by logging the vregister values.

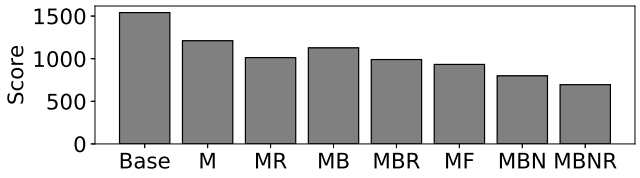

Figure 7: Performance overhead of FA^3

Pixel 5 smartphone when running both the stock Android OS and the modified version of Android with FA^3 enabled.

The experimental results show that the average recorded system-wide performance overhead is 428.5% when measuring the additional time required by FA^3 to collect all class (method) executions. That said, the minimal overhead is only around 2.4% and the maximum overhead is about 11540.0%. We comment that: (1) all exercises run successfully on FA^3 without any events severely delayed or blocked due to the overhead, which makes FA^3 suitable for reliably monitoring and tracing of Android apps; and (2) the overhead could be substantially reduced when the monitoring is configured to focus on specific classes and methods.

5 DISCUSSION

The current implementation of FA^3 focuses on monitoring the execution of Java Code, native code, native libraries, and the JNI invocation to enable fine-grained understanding of the apps' execution. FA^3 is not designed to draw a line between malicious and benign software. We further discuss the limitations of FA^3.

First, the code coverage is a concern for all dynamic analysis platforms, including FA^3. To address this, we can use Monkey [5] or other UI automation frameworks [14] to generate events with a better code coverage.

Second, FA^3 does not support native libraries involving self-modifying code at the moment. In order to address it, we can check whether the memory content is changed once the control-flow traps into the kernel by looking up the *vm_area_struct* structure.

Last but not the least, hybrid apps leverage the advantages of both traditional apps written in Java and web apps using various web techniques (e.g., JavaScript, HTML) to speed up the development for multiple platforms. FA^3 currently does not support tracking information flows going through the JavaScript context switch.

6 CONCLUSION

We propose FA^3, a novel on-device, non-invasive, and fine-grained analysis platform to capture application execution on multiple layers at a fine-grained manner. FA^3 generates logs containing the information of method invocations and control-flow transfer for Java and native code. We have implemented a prototype of FA^3 and evaluated it with real-world malware samples and benign applications. The experimental results show that FA^3 successfully analyzes them and provides a comprehensive view of their behaviors.

ACKNOWLEDGEMENT

We thank our shepherd Tauhidur Rahman and the anonymous reviewers for their valuable comments and suggestions. This research / project is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme, National Satellite of Excellence in Mobile Systems Security and Cloud Security (NRF2018NCR-NSOE004-0001). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] Baidu. http://legu.qcloud.com/, 2015-12-12.

[2] Aspectj. https://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html, 2017-11-01.

[3] Geekbench 5. https://www.geekbench.com/, 2017-12-18.

[4] Simpleperf. https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md, 2019-04-04.

[5] Android monkey. https://developer.android.com/studio/test/monkey, 2020-08-25.

[6] Android profiler. https://developer.android.com/studio/profile/android-profiler, 2021-07-28.

[7] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65, 2017.

[8] S. Arzt, S. Rasthofer, and E. Bodden. Instrumenting android and java applications as easy as abc. In *International Conference on Runtime Verification*. Springer, 2013.

[9] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*. IEEE, 2017.

[10] V. Balachandran, D. J. Tan, V. L. Thing, et al. Control flow obfuscation for android applications. *Computers & Security*, 61, 2016.

[11] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41. Califor-nia, USA, 2005.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2), 2014.

[13] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[14] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014.

[15] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[16] D. Kirat and G. Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6), 2007.

[18] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On tracking information flows through jni in android applications. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014.

[19] 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*, 2016.

[20] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, volume 96435, 2013.

[21] M. Sun, X. Li, J. C. Lui, R. T. Ma, and Z. Liang. Monet: a user-oriented behavior-based malware variants detection system for android. *IEEE Transactions on Information Forensics and Security*, 12(5), 2016.

[22] M. Sun, J. C. Lui, and Y. Zhou. Blender: Self-randomizing address space layout for android apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[23] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014.

[24] M. Sun, T. Wei, and J. C. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[25] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: automatic reconstruction of android malware behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[26] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu. Adaptive unpacking of android apps. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE, 2017.

[27] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3), 2018.

[28] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au. Happer: Unpacking android apps via a hardware-assisted approach. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[29] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu. Malton: Towards on-device non-invasive mobile malware analysis for {ART}. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[30] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[31] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *International Conference on Security and Privacy in Communication Systems*. Springer, 2015.

[32] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*. Springer, 2015.