# ADRENALIN-RV: Android Runtime Verification using Load-time Weaving

Haiyang Sun, Andrea Rosà, Omar Javed, and Walter Binder
Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland
Email: first.last@usi.ch

***Abstract*—Android has become one of the most popular operating systems for mobile devices. As the number of applications for the Android ecosystem grows, so is their complexity, increasing the need for runtime verification on the Android platform. Unfortunately, despite the presence of several runtime verification frameworks for Java bytecode, DEX bytecode used in Android does not benefit from such a wide support. While a few runtime verification tools support applications developed for Android, such tools offer only limited bytecode coverage and may not be able to detect property violations in certain classes. In this paper, we show that ADRENALIN-RV, our new runtime verification tool for Android, overcomes this limitation. In contrast to other frameworks, ADRENALIN-RV weaves monitoring code at load time and is able to instrument all loaded classes. In addition to the default classes inside the application package (APK), ADRENALIN-RV covers both the Android class library and libraries dynamically loaded from the storage, network, or generated dynamically, which related tools cannot verify. Evaluation results demonstrate the increased code coverage of ADRENALIN-RV with respect to other runtime validation tools for Android. Thanks to ADRENALIN-RV, we were able to detect violations that cannot be detected by other tools.**

## I. Introduction

Android has become the dominant operating system for mobile devices. The increase in popularity has led to a rapid growth of the number of applications for Android, as well as their complexity. As applications become more complex, advanced testing techniques and runtime verification are fundamental to mitigate buggy and malicious applications. While testing is used during development, runtime verification is typically used to monitor a program after deployment, making them complementary.

Our work focuses on runtime verification on the Android platform. While there is a large body of runtime verification frameworks suitable for Java bytecode [1]–[4], such tools cannot be readily applied to DEX bytecode used by Android applications, as the Java and DEX bytecode formats are different. As a result, there are few runtime verification tools for Android. Moreover, the bytecode coverage of existing tools is rather limited. For example, RV-Droid [5] and RV-Android [6] can be used to monitor several properties of Android applications, but are not able to instrument classes of the core Android library. In addition, they rely on static weaving, which prevents the instrumentation of classes loaded dynamically, including third-party libraries downloaded from a remote server or classes generated dynamically. Violations of properties in such classes will remain undetected.

In this paper we tackle the limited code coverage of existing tools by introducing ADRENALIN-RV (AnDRoid-ENabled Aspect-oriented Load-time INstrumnetation for Runtime Verification), our new runtime verification tool for the Android platform. Differently from related tools based on static weaving, ADRENALIN-RV is based on load-time weaving and is able to instrument every class loaded by the runtime environment. In addition to the default classes inside the application package, ADRENALIN-RV covers both classes of the Android library and dynamically loaded classes, including libraries downloaded from the network and dynamically generated. Our evaluation results show that ADRENALIN-RV offers a considerably increased code coverage with respect to related tools, and that it is able to detect violations that cannot be detected with static weaving.

This work makes the following contributions. We present ADRENALIN-RV, our new runtime verification tool for the Android platform. We show that ADRENALIN-RV achieves more code coverage than RV-Droid and RD-Android on selected Android applications. Moreover, we show that ADRENALIN-RV can find violations that cannot be found by related tools.

This paper is organized as follows. Section II provides an overview of the Android platform. Section III describes ADRENALIN-RV and details its architecture. Section IV outlines the main challenges in developing ADRENALIN-RV. In Section V we present our evaluation results. Section VI summarizes lessons learned in implementing and evaluating ADRENALIN-RV. Finally, we present related work in Section VII and give our concluding remarks in Section VIII.

## II. Background: The Android Platform

Android is a Linux-based operating system. Applications running on Android execute in a dedicated sandbox: each application executes in a separate process and has access to only its own files. Android applications are written in Java and are built from interconnected components. Each component has a different role and can serve as an entry point for the application.

While written in Java, Android applications do not execute on the Java Virtual Machine (JVM). Depending on the version of Android, they execute either on the Android Runtime (ART, since Android 5), or on the Dalvik Virtual Machine (DVM,

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.61

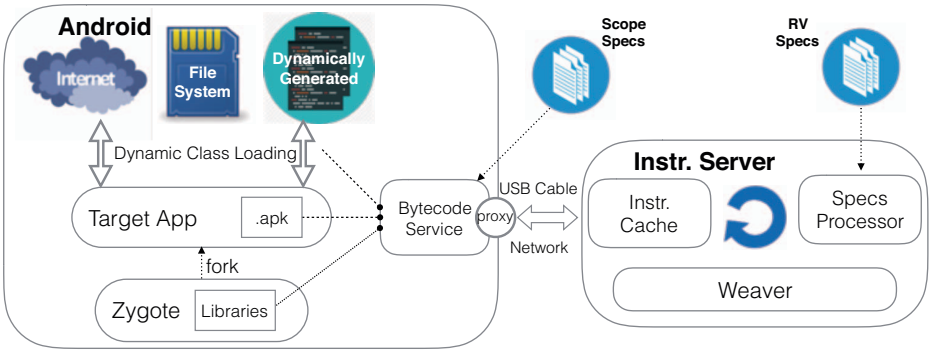



Fig. 1. ADRENALIN-RV architecture. The figure shows only one of the monitored applications on the Android platform. Other running applications are omitted for clarity.

before Android 5).[1] Due to the limited resources on mobile devices, Android uses DEX bytecode instead of Java bytecode. Application classes must be converted from Java bytecode to DEX bytecode before deployment.

Components of the same application execute in a single VM instance by default. However, any component can be configured to execute in a separate process, and thus in a separate VM. Creating a new VM instance involves considerable overhead due to VM bootstrap and initialization of the core libraries. As process start and termination occur frequently, the overhead of VM initialization would cause a serious performance degradation. To mitigate such issues, Android starts a special process called Zygote early during system boot, which only bootstraps the VM and initializes the core classes. Hence, Zygote becomes a live snapshot of a newly initialized VM. That snapshot can be efficiently duplicated when needed thanks to the copy-on-write implementation of the fork() system call: when a new VM instance is needed, Zygote is simply forked, yielding a child VM which can readily execute application code.

For the purpose of this paper, application classes can be classified into three groups, according to when they are loaded in the VM during application execution. The first group is composed of classes contained in Android Application Package (APK) files. Each application is shipped in an APK file, a compressed package containing the main classes of the application (in DEX format) and other resources. Classes in the package are loaded when the application is launched. The second group consists of library classes. Such classes are loaded by Zygote during bootstrap and are shared among all application processes. The third group is composed of classes loaded dynamically by applications through the DexClassLoader API.[2] This API is frequently used to load classes not included in the APK file of the application, such as third-party libraries stored in the file system or on a remote server. It is also used by some malicious applications to hide their behavior [7].

[1]For the purpose of the paper, the specific runtime environment used by Android does not matter. We use the term VM to refer to either ART or DVM indiscriminately.

[2]https://developer.android.com/reference/dalvik/system/DexClassLoader.html

## III. ADRENALIN-RV

In this section we describe the architecture of ADRENALIN-RV. First, we detail the process of instrumenting Android classes with monitoring code. Then, we describe how our tool can intercept and weave all classes loaded on Android. Figure 1 depicts the high-level architecture of ADRENALIN-RV.

### A. Instrumentation

ADRENALIN-RV relies on DiSL [8], a dynamic program analysis framework based on Java bytecode instrumentation. DiSL instruments classes on a separate server, here called *instrumentation server*, according to specified runtime-verification specifications (henceforth called *RV specs*). ADRENALIN-RV has several built-in RV specs that can be readily verified on Android applications—Section V showcases some of them.

The instrumentation server runs outside the Android platform. The classes to be instrumented are sent to the instrumentation server over the network or via a USB cable, and instrumented classes are sent back over the same medium. USB support is necessary to instrument classes when the network is not available on the device (e.g., during the boot phase before the network module has been loaded). In this way, ADRENALIN-RV can instrument the system class libraries since the beginning of the boot phase.

Often, users wish to monitor different sets of properties for different applications. To ease the process of setting the properties of interest for a given application, the user can map properties to applications in the RV specs. In particular, users can define properties of interest for a given bytecode file. RV specs will be processed by a custom component of the instrumentation server, the *specs processor*, before starting

the instrumentation process. The processor ensures that only the monitoring code corresponding to the desired properties for a class is woven.

Another frequent need when monitoring applications is to repeat the analysis multiple times, without changing the specifications. In this scenario, the instrumented bytecode does not change between different runs. To avoid unnecessary instrumentation, we introduce an *instrumentation cache* in the server, which stores the last version of the instrumented bytecode and validates whether the instrumentation for a given bytecode remains the same. In this case, the instrumentation process is avoided, and the bytecode stored in the cache will be used at runtime.

### B. Load-Time Weaving

To guarantee full bytecode coverage on the Android platform, it is fundamental to intercept and instrument every class loaded by an application. This implies that all three groups of application classes (see Section II) must be intercepted at load-time and instrumented with monitoring code. In contrast to related tools relying on static weaving, ADRENALIN-RV enables load-time weaving for Android. This makes it possible to instrument any class loaded by the VM, in contrast to static weaving which enables one to instrument only classes in the application APK file.

To enable load-time instrumentation, we modify the VM to hook class loading. This allows ADRENALIN-RV to monitor all classes, including shared libraries loaded by Zygote and any dynamically loaded class, such as classes downloaded from a remote server, loaded from a third-party library on the file system, or dynamically generated by the application. In addition, we add an internal API to Android for sending and receiving classes to/from the instrumentation server. To communicate with the outside server, we add *proxy* service in Android, listening to all observed VM instances.

In principle, each class loaded by the VM can be passed to the instrumentation server. However, users may be interested in verifying properties of interest only in selected classes. Users can specify classes of interest through specifications (*scope specs*). These specs will be parsed by a *bytecode service*, a custom component implemented in C++ which uses Binder—the inter-process-call library in Android—to communicate with the bytecode loading processes. The bytecode service ensures that only classes of interest are sent to the instrumentation server. The addition of this component avoids unnecessary slow-downs which can degrade the performance of the platform.

## IV. Technical Challenges

Increasing the bytecode coverage for Android applications requires a great design effort in solving limitations of the underlying platform, which does not allow to readily instrument all loaded classes. Here, we outline the main challenges in instrumenting classes on Android, and present how we solve those issues in ADRENALIN-RV.

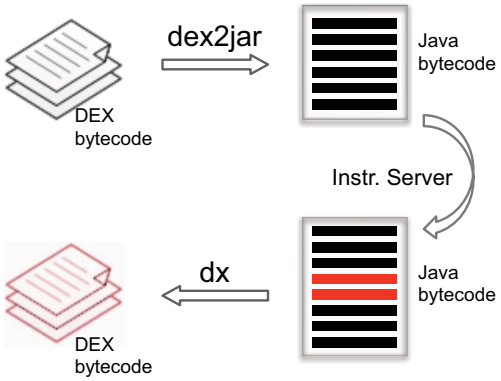



Fig. 2. Bytecode conversion for instrumenting application classes.

### A. Intercepting Class Loading

The difficulties of instrumenting Android classes upon loading stem from the lack of interfaces allowing external tools to be notified about specific events. For example, class loading can easily be intercepted on the JVM thanks to the JVM Tool Interface (JVMTI), which is integrated in the JVM and allows an external agent to replace a class with the instrumented version before the class is loaded and linked. Unfortunately, the Android VM does not offer interfaces akin to the JVMTI. As a result, it is not possible to instrument classes upon their loading without modifying the VM.

To enable full bytecode coverage in ADRENALIN-RV, we modify the VM class-loading process to instrument classes before they are mapped to memory. Before loading a DEX file into memory, the modified VM sends the file through a proxy to the instrumentation server, which instruments the encoded class and sends back another DEX file which is then loaded into memory. This ensures that both application classes and the associated libraries are instrumented.

### B. Core Libraries

When the Android system starts, Zygote loads and initializes the core classes. As subsequent VM instances are obtained by forking Zygote, they all share the code of the core libraries. If the user is interested in monitoring core classes in one application, such classes will be instrumented, resulting in a single instrumented version of the core libraries shared among all applications, including those not being targeted by the user. It is technically impossible to instrument only the core libraries for the applications that are being monitored.

To ensure that code in the code libraries is monitored only when used by the target application, the instrumentation used by ADRENALIN-RV relies on a bypass functionality [9]. The instrumentation is only enabled in the monitored application and bypassed with minimum overhead in other applications.

### C. Bytecode Conversion

Android classes need to be translated from Java bytecode to DEX bytecode for execution. Manipulating DEX bytecode is

an added burden on developing runtime verification tools without any technical merit, as DEX bytecode has been developed primarily to avoid licensing issues and is officially produced only by conversion from Java bytecode. Given the extensive support of frameworks for manipulating Java bytecode, there is little interest in manipulating DEX bytecode directly.

ADRENALIN-RV uses existing bytecode conversion tools to translate between the two representations as necessary, and relies on DiSL to instrument Java bytecode. In particular, ADRENALIN-RV extracts the classes to be instrumented from the corresponding DEX file, and uses dex2jar[3] to convert them from DEX bytecode to Java bytecode. Then, our tool passes the converted Java bytecode to the instrumentation server, which weaves monitoring code into the converted class. ADRENALIN-RV converts the instrumented Java bytecode back to DEX bytecode through dx.[4] Finally, the tool repackages the class into a DEX file, which is sent back to the Android platform. The bytecode conversion process is shown in Figure 2.

## V. Evaluation

In this section we evaluate the increased bytecode coverage of ADRENALIN-RV for two use cases. We start by comparing the bytecode coverage of ADRENALIN-RV with related tools on Android applications. Then, we show how ADRENALIN-RV can detect property violations that related tools cannot detect, thanks to load-time weaving. In detailing our evaluation results, we also show the properties that ADRENALIN-RV can monitor, and how the user can write new monitoring code in ADRENALIN-RV.[5]

### A. Code Coverage

ADRENALIN-RV relies on load-time weaving to insert monitoring code in Android applications, differently from related tools. For example, RV-Droid and RV-Android both use static weaving. Here, we show that load-time weaving yields a significantly extended bytecode coverage wrt. static weaving. As a result, ADRENALIN-RV can monitor more code than RV-Droid and RV-Android.

We conduct our evaluation on two Android applications of different nature. The first application is Google Mobile Services (GMS)[6], which includes several services from Google as well as popular APIs. The second application is a malware[7] which uses dynamically loaded code to obfuscate its behavior.

To compare ADRENALIN-RV with RV-Droid and RV-Android, we evaluate a set of properties that can be monitored by all three tools. In particular, we choose multiple well-known JavaMOP properties shown in Table I. For each property, we collect the number of join point[8] shadows, i.e., locations in the source code that at run-time produce a join point, and the number of join point executions. Finally, we differentiate the result according to the class categorization introduced in Section II, i.e., in 1) APK classes, 2) shared libraries, and 3) dynamically loaded classes. While static weaving can only instrument APK classes, load-time weaving can instrument all loaded classes. As a result, RV-Droid and RV-Android can intercept join points in the first group, whereas ADRENALIN-RV can intercept join points in all groups.

TABLE I
JavaMOP properties evaluated.

| Property | Description |
|---|---|
| *HasNext* | Program should always call hasNext() before next() on an iterator. |
| *SafeEnum* | Collection (with an associated enumeration) should not be modified while the enumeration is in use. |
| *SafeSyncMap* | Synchronized collection should always be accessed by a synchronized iterator, and the iterator should always be accessed in a synchronized manner. |
| *UnsafeIterator* | When the iterator associated with a collection is accessed, the collection should not be updated. |
| *UnsafeMapIterator* | Like *UnsafeIterator*, with differences related to the creation of iterators. |

Figure 3 depicts our results for GMS. For all JavaMOP properties considered, several join point shadows refer to shared libraries or dynamically loaded classes, as shown by Figure 3(a). On average, ∼7% of join point shadows are contained in shared libraries, while ∼13% of them are located in dynamically loaded code. *SafeEnum* follows a different behavior, with ∼50% of join point shadows than can be detected only by load-time weaving. However, the total number of join point shadows for this property is quite limited (only 121 in GMS), whereas the mean number of join point shadows for other properties is around 30000. On average, ∼22% of join point shadows cannot be detected with static weaving.

Figure 3(b) show the number of joint point executions in GMS. Here, results vary considerably among different properties. While in *SafeEnum* and *SafeSyncMap* a high percentage of join point executions occurs in APK classes, this holds for only 52% of the join point executions in *UnsafeIter*, and ∼40% of join point executions in *HasNext* and *UnsafeMap*. In these properties static weaving can cover less than half of the join point executions. Therefore, employing load-time weaving is fundamental to guarantee a full bytecode coverage in such properties.

The importance of using load-time weaving is even more remarked by Figure 4, which shows our results on the malware. For this application, static weaving can cover only a single

[3]https://github.com/pxb1988/dex2jar

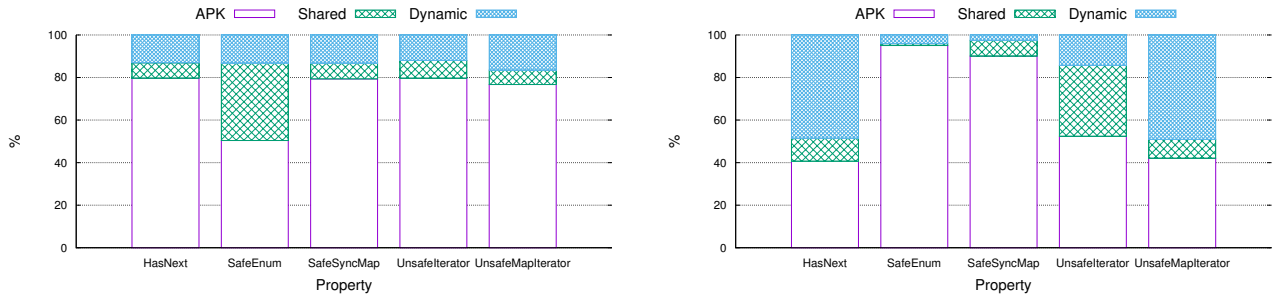
[4]dx is one of the tool contained in the Android Software Development Kit (SDK).

[5]All evaluation results presented in this section have been obtained on Android 4.4r_1 running on a Nexus 5 with 2GB RAM. We use DiSL 2.0. The instrumentation server is deployed on quadcore Intel Core i7 (2.5GHz, 16GB RAM) and runs under Java 8.

[6]https://www.android.com/gms/

[7]https://github.com/ashishb/android-malware/tree/master/Android.Malware.at_plapk.a

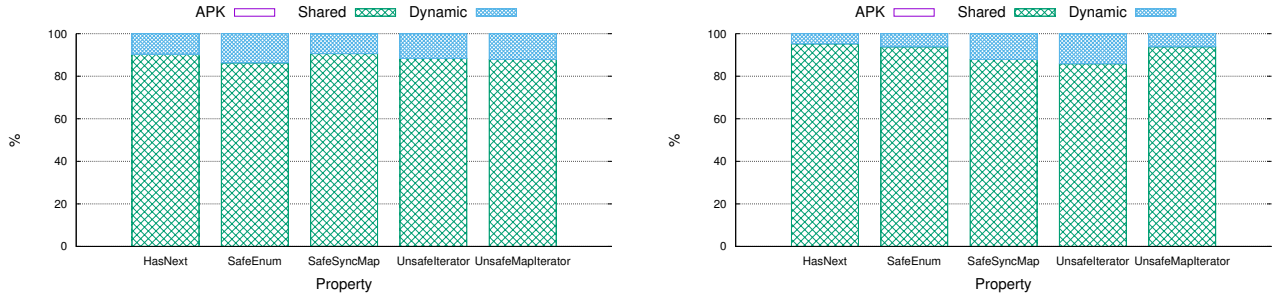
[8]In Aspect-Oriented Programming (AOP), the term *join point* refers to any identifiable execution point in a system. As all three tools rely on AOP, we use this terminology in the paper.




(a) Join point shadows.

(b) Join point executions.

Fig. 3. Percentage of join points for GMS, broken down by bytecode group, for the five considered properties of JavaMOP. In the legend, *APK*, *Shared* and *Dynamic* refer to join points in APK classes, system libraries and dynamically loaded classes, respectively.




(a) Join point shadows.

(b) Join point executions.

Fig. 4. Percentage of join points for the malware, broken down by bytecode group, for the five considered properties of JavaMOP.

TABLE II
JOIN POINT SHADOWS AND EXECUTIONS.

| | Join Point Shadows | | | Join Point Executions | | |
|---|---|---|---|---|---|---|
| **Property** | **APK** | **Shared** | **Dynamic** | **APK** | **Shared** | **Dynamic** |
| **(GMS)** | | | | | | |
| HasNext | 11632 | 1051 | 1910 | 113 | 30 | 135 |
| SafeEnum | 61 | 44 | 16 | 3768 | 34 | 162 |
| SafeSyncMap | 40503 | 3775 | 6772 | 4027 | 329 | 113 |
| UnsafeIterator | 21277 | 2221 | 3200 | 296 | 188 | 81 |
| UnsafeMapIterator | 19087 | 1682 | 4109 | 162 | 34 | 189 |
| **(Malware)** | | | | | | |
| HasNext | 0 | 1051 | 112 | 0 | 739 | 37 |
| SafeEnum | 0 | 44 | 7 | 0 | 940 | 62 |
| SafeSyncMap | 1 | 3775 | 390 | 0 | 12173 | 1683 |
| UnsafeIterator | 0 | 2221 | 293 | 0 | 1193 | 196 |
| UnsafeMapIterator | 0 | 1682 | 232 | 0 | 1141 | 75 |

join point shadow of the *SafeSyncMap* property, which is not executed at runtime.

As shown by Figures 4(a) and 4(b), ∼90% of join points are contained and executed in shared libraries, while the remainder refers to dynamically loaded code. In such kind of applications, static weaving is not able to cover any code, being unable to identify any property violation. Detailed numerical results on joint point shadows and executions can be found in Table II.

Overall, our results show that employing static weaving (as done in RV-Droid and RV-Android) results in limited coverage of the monitoring code, while load-time weaving—a core part of ADRENALIN-RV—guarantees that all executed bytecode is monitored. The extended bytecode coverage enables ADRENALIN-RV to identify property violations that cannot be detected by RV-Droid and RV-Android, as shown in the next section.

### B. Violation Detection

In this section, we show how ADRENALIN-RV can unveil a property violation that cannot be detected by related tools. We take the Android malware introduced above as monitored application. As for the violation, we focus on information leaks, i.e., unauthorized transmissions of private data (such as device ID, contacts, messages, etc.) to an untrusted part. Information leaks represent one of the major security concerns of Android applications.

We define the *InformationLeak* property as follows: no private data from a pre-defined set of Android APIs must be sent to a third party. To monitor violations of such property, we define two events of interest: *DataSource* (i.e., information about private data is obtained) and *DataSink* (i.e., data is sent over a transmission channel). For this use case, we focus in particular on the transmission of the device ID over the network.

Figure 5 shows snippets of DiSL code[9] defining the two events, which can be added as RV specs in ADRENALIN-RV.

[9]Intuitively, DiSL uses an AOP notation to express instrumentation code. A description of the DiSL language is not in the scope of this paper. We refer the reader to [8] for related information.

The first snippet (lines 2–7) detects all calls to TelephonyManager.getDeviceId() (which allows retrieving the ID of an Android device) inside application code by instrumenting all invocations with such method as parameter.[10] Hence, this snippet makes it possible to monitor the event *DataSource* referring to the device ID. When such an event is triggered, the instrumentation code stores the device ID to check at a later time whether such information is sent over the network.[11] Thanks to load-time weaving, ADRENALIN-RV can detect the event in any class, including dynamically loaded classes. This is particularly important for this target application, as it generates many libraries at runtime, loading classes from such libraries dynamically. In the presence of such a behavior, tools relying on static weaving are likely to detect no violations, as they cannot monitor most of the classes loaded by the application.

The second code snippet (lines 10–15) refers to instrumentation code for the event *DataSink*. In particular, ADRENALIN-RV instruments the method sendTo of class libcore.io.IoBridge (used for writing data over files), and checks whether 1) data is sent over a socket, and 2) the information sent is the device ID stored by the first code snippet.[12] As the name suggests, libcore.io.IoBridge is a library class, shared by all Android applications. As such, it cannot be instrumented by related tools. Without the possibility to instrument shared libraries, the instrumentation would have to detect all callsites of libcore.io.IoBridge and verify whether the communication occurs through sockets. Still, such callsites may be included in dynamically generated classes, requiring load-time weaving to be detected.

Intuitively, violations of the *InformationLeak* property occur if a *DataSink* is observed after a *DataSource*. By monitoring the malware with ADRENALIN-RV, we were able to find a violation of the property. Figure 6 shows a portion of the method call trace related to the events *DataSink* and *DataSource*.[13] The figure shows that the device ID is retrieved in com.jfdlplapk.by.b() (event *DataSource* detected), and is sent over a socket in com.jfdlplapk.au.a() (event *DataSink* detected). The property violation occurs in dynamically generated (and loaded) classes. As a result, related tools such as RV-Droid and RV-Android are not able to detect such a violation.

## VI. Lessons Learned

While developing and evaluating ADRENALIN-RV, we learned several lessons. We summarize them in the following text.

[10]DeviceIdGuard, here not shown, specifies that only calls to TelephonyManager.getDeviceId() must be detected.

[11]Details of the instrumentation code are not of interest in this paper. We represent instrumentation code with the class Monitor and its methods, and omit further details for clarity.

[12]Note that ADRENALIN-RV can also instrument invocations to libcore.io.IoBridge done via reflection.

[13]The call trace has been generated by ADRENALIN-RV by inserting monitoring code at every method enter and exit to identify methods in which *DataSink* and *DataSource* occurs.

```
1  // DataSource event
2  @Before(marker=BytecodeMarker.class,
3      guard=DeviceIdGuard.class,
4      args = "invokevirtual")
5  public static void getDeviceId() {
6      Monitor.newDataSource("DeviceId");
7  }
8
9  // DataSink event
10 @AfterReturning(marker=BodyMarker.class,
11     scope="libcore.io.IoBridge.sendto(...)")
12 public static void sendto(...){
13     // Some implementation details omitted
14     Monitor.newDataSink("NetworkSend", paraInfo);
15 }
```

Fig. 5. DiSL code snippets for the *DataSource* and *DataSink* properties.

```
1  com/jfdlplapk/g.run
2      com/jfdlplapk/f.b
3          com/jfdlplapk/by.e
4              com/jfdlplapk/by.b
5              * DataSource *                <-----
6              com/jfdlplapk/by.c
7              com/jfdlplapk/bn.c
8                  com/jfdlplapk/by.b
9                      com/jfdlplapk/by.a
10             com/jfdlplapk/by.a
11             com/jfdlplapk/by.a
12             com/jfdlplapk/bo.b
13                 com/jfdlplapk/bo.a
14             com/jfdlplapk/bn.b
15                 com/jfdlplapk/by.b
16                     com/jfdlplapk/by.a
17     com/jfdlplapk/au.a
18         com/jfdlplapk/au.a
19         * DataSink *                      <-----
```

Fig. 6. Portion of the method call trace related to events *DataSource* and *DataSink*.

### A. Static weaving is limited

Our evaluation results show that static weaving can cover only a limited set of join points for several evaluated properties. Therefore, tools relying on static-weaving are not able to fully monitor an Android application. This limitation is removed with load-time weaving, which is able to monitor system libraries[14] and dynamically loaded classes, in addition to standard APK classes. While it is easier to use static weaving, runtime verification tools for Android should rely on load-time weaving to increase their code coverage.

### B. Load-time weaving is challenging

While offering an extended code coverage wrt. static weaving, it is not straightforward to develop tools based on load-time weaving. We encountered several challenges in implementing load-time weaving on the Android platform, which required modifications to the Android runtimes to hook class loading. Moreover, obtaining full coverage in Android necessarily requires the ability to instrument system libraries that are

[14]While shared libraries could in principle be instrumented with static weaving, this would be inconvenient, as it would require to rebuild the system at each instrumentation.

shared among all applications. Still, the analysis needs a way to exclude code of shared libraries executed by applications not being monitored, such as the bypassing mechanism used by ADRENALIN-RV. In general, enabling load-time weaving on Android requires careful design decisions.

### C. Monitoring dynamically loaded classes is important

When evaluating ADRENALIN-RV, we encountered several applications that load bytecode dynamically for different needs. In some applications, the bytecode needed for execution is very large, and exceeds the maximum allowed bytecodes in a single method of a DEX class (64K bytecodes). One solution to this problem is to store extra bytecodes in the installation package and load them at runtime. Some applications rely on third-party libraries that download bytecodes from a remote server. For example, the AdMob[15] advertisement library from Google follows this behavior, and is often used by Android applications. Another case when classes are dynamically loaded consists in application updates, which require the application to load the new bytecode dynamically. In addition, some malicious applications may load classes dynamically to avoid being detected by anti-virus tools using static checking. Overall, runtime verification tools should not overlook dynamically loaded classes, as they are frequently present in several Android applications and can contain violations that would remain undetected otherwise.

## VII. Related Work

In the mobile-phone market, a dominant share is composed of Android applications. This fact has stimulated the development of malicious applications, particularly targeting device integrity and user privacy. To combat this threat, researchers have developed tools aimed at identifying malicious software on the Android platform using runtime verification [10], [11]. Such tools rely on the AspectJ instrumentation framework [12] to weave monitoring code in application classes. However, it has been shown that AspectJ offers only limited bytecode coverage [13], particularly being unable to instrument the core class library. In contrast, ADRENALIN-RV does not present this limitation, as it relies on DiSL for instrumentation and on a custom modification of the Android VMs for intercepting class loading.

The aspectbench compiler (abc) [14] is an extensible AspectJ compiler that makes it possible to add new features. It uses the Polyglot [15] framework as its front-end and the Soot framework [16] as its back-end for improving code generation. It can be applied on Android applications to detect security and privacy violations [11]. However, like AspectJ, abc also suffers from the limitations outlined above.

RV-Droid [5] is a runtime verification tool for Android built upon an in-house version of AspectJ suited for Android instrumentation. The tool is based on JavaMOP [17]—a runtime verification tool for the JVM—to produce monitoring libraries. Apart from security properties, it enables one to monitor a general set of properties suited for correct implementation, debugging, statistics, etc. RV-Android [6] is another tool targeting safety properties in Android applications. Similarly to RV-Droid, it uses AspectJ for instrumentation. Both tools are based on static weaving, restricted to classes contained in application APK files. In contrast, ADRENALIN-RV employs load-time weaving, extending the coverage to core libraries and to dynamically loaded code.

Monitor-Me [18] identifies software threats through a first-order logic abstraction of malware behavior. It uses an Android interception tool to capture any system-level event triggered by an application. For intercepting the system calls, Monitor-Me uses a custom kernel module to probe the calls. In contrast, our approach is based on bytecode instrumentation and does not require any modification to the Android kernel (only the VM need to be modified).

Raindroid [19] uses a combination of static analysis and runtime verification to identify communication patterns susceptible to malicious attacks. StaDynA [20] complements static and dynamic analysis techniques by computing inter-procedural call graphs that can be used by external tools to monitor malware. While StaDynA focuses on building method call graphs, ADRENALIN-RV weaves monitor at runtime to verify safety properties.

In prior work, we presented a bytecode instrumentation framework for Android [9]. The framework provides a high-level programming model and abstractions for developing dynamic program analyses for the Android platform. ADRENALIN-RV focuses on runtime verification, providing a set of properties that can be readily verified on Android applications. While the framework analyzes the application on an external machine, ADRENALIN-RV monitors the application inside the target VM, resulting in a lower overhead than our previous framework, at the cost of a higher memory footprint.

We also developed a compiler that translates runtime-verification aspects written in AspectJ to DiSL [13]. The compiler makes it possible to use existing, unmodified runtime verification tools on top of the DiSL framework to bypass the limitations of AspectJ. Unfortunately, the support offered by the compiler is limited to a subset of AspectJ constructs (e.g., around advice and inter-type declarations are not supported). Hence, existing AspectJ-based tools cannot be fully adapted for the Android platform. ADRENALIN-RV is not based on our previous compiler, it implements monitoring code in DiSL.

## VIII. Concluding Remarks

In this paper, we have presented ADRENALIN-RV, a new runtime verification tool for Android. In contrast to related tools, ADRENALIN-RV is based on load-time weaving, which enables to instrument all loaded classes, including classes in the core library, dynamically generated classes, and classes downloaded from remote servers. Evaluation results show that ADRENALIN-RV offers an increased code coverage with respect to other runtime validation tools for Android. ADRENALIN-RV is currently in a research prototype state. As part of our future work, we plan to gradually expand

[15] AdMob is a Google mobile advertising platform specifically designed for mobile apps. More details can be found at https://www.google.com/admob/.

ADRENALIN-RV by adding more properties. The tool and more information can be found at our website.[16]

The current version of ADRENALIN-RV presents some limitations, that we discuss in the following text. When monitoring some applications, the bytecode retargeting tools used by ADRENALIN-RV (i.e., dex2jar and dx) can fail during the bytecode transformation. Due to this error, it is not possible to fully monitor some applications. We note that this issue does not arise from bugs in ADRENALIN-RV, but from third-party software. Our tool requires modification to the Android runtimes to intercept class loading. While this modification enables load-time weaving on Android, it requires the possibility to apply a patch and recompile the operating system, which might be not possible for some users. Finally, while the support for the DVM is solid, the support for the ART is currently experimentally. We plan to make our project open source and improve support for ART in the near future.

## ACKNOWLEDGMENT

The work presented in this paper was supported by Oracle (ERO project 1332), by the Swiss National Science Foundation (project 200021_141002), by the European Commission (contract ACP2-GA-2013-605442), and by the Federal Commission for Scholarships for Foreign Students (Swiss Government Excellence Scholarship, ESKAS No. 2015.0989).

## REFERENCES

[1] C. Colombo, G. J. Pace, and G. Schneider, "LARVA—safer monitoring of real-time Java programs (tool paper)," in *SEFM*, 2009, pp. 33–37.

[2] G. Reger, H. C. Cruz, and D. Rydeheard, "MARQ: monitoring at runtime with QEA," in *TACAS*, pp. 596–610.

[3] C. Xiang, Z. Qi, and W. Binder, "Flexible and extensible runtime verification for java (extended version)," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 09n10, pp. 1595–1609, 2015.

[4] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-mac," *Electronic Notes in Theoretical Computer Science*, pp. 218 – 235, 2001.

[5] Y. Falcone, S. Currea, and M. Jaber, "Runtime verification and enforcement for Android applications with RV-Droid," in *RV*, 2013, pp. 88–95.

[6] P. Daian, Y. Falcone, P. Meredith, T. F. Şerbănuţă, S. Shiriashi, A. Iwai, and G. Rosu, "Rv-android: efficient parametric android runtime verification, a brief tutorial," in *RV*, 2015, pp. 342–357.

[7] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of malicious applications on Android OS," in *IWCF*, 2010, pp. 138–149.

[8] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "DiSL: A Domain-specific Language for Bytecode Instrumentation," in *AOSD*, 2012, pp. 239–250.

[9] H. Sun, Y. Zheng, L. Bulej, A. Villazón, Z. Qi, P. Tůma, and W. Binder, "A programming model and framework for comprehensive dynamic analysis on Android," in *MODULARITY*, 2015, pp. 133–145.

[10] Y. Falcone and S. Currea, "Weave Droid: Aspect-oriented programming on android devices: fully embedded or in the cloud," in *ASE*, 2012, pp. 350–353.

[11] S. Arzt, S. Rasthofer, and E. Bodden, "Instrumenting Android and Java applications as easy as abc," in *RV*, 2013, pp. 364–381.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP*, 2001, pp. 327–353.

[13] O. Javed, Y. Zheng, A. Rosà, H. Sun, and W. Binder, "Extended code coverage for AspectJ-based runtime verification tools," in *RV*, 2016, pp. 219–234.

[14] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "abc: An extensible AspectJ compiler," in *AOSD*, 2005, pp. 87–98.

[15] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for Java," in *CC*, 2003, pp. 138–152.

[16] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the soot framework: is it feasible?" in *CC*, 2000, pp. 18–34.

[17] D. Jin, P. O. N. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *ICSE*, 2012, pp. 1427–1430.

[18] J.-C. Küster and A. Bauer, "Monitoring real Android malware," in *RV*, 2015, pp. 136–152.

[19] B. Schmerl, J. Gennari, J. Cámara, and D. Garlan, "Raindroid: A system for run-time mitigation of Android intent vulnerabilities [poster]," in *HotSos*, 2016, pp. 115–117.

[20] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications," in *CODASPY*, 2015, pp. 37–48.

[16] http://haiyang-sun.github.io/tool/intro.html