Runtime Verification and Enforcement for Android Applications with RV-Droid[⋆]

Yliès Falcone, Sebastian Currea, and Mohamad Jaber

Laboratoire d'Informatique de Grenoble, UJF Université Grenoble 1, France
`FirstName.LastName@ujf-grenoble.fr`

Abstract. RV-Droid is an implemented framework dedicated to runtime verification (RV) and runtime enforcement (RE) of Android applications. RV-Droid consists of an Android application that interacts closely with a cloud. Running RV-Droid on their devices, users can select targeted Android applications from Google Play (or a dedicated repository) and a property. The cloud hosts third-party RV tools that are used to synthesize AspectJ aspects from the property. According to the chosen RV tool and the specification, some appropriate monitoring code, the original application and the instrumentation aspect are woven together. Weaving can occur either on the user's device or in the dedicated cloud. The woven application is then retrieved and executed on the user's device and the property is runtime verified. RV-Droid is generic and currently works with two existing runtime verification frameworks for (pure) Java programs: with JavaMOP and (partially) with RuleR. RV-Droid does not require any modification to the Android kernel and targeted applications can be retrieved off-the-shelf. We carried out several experiments that demonstrated the effectiveness of RV-Droid on monitoring (security) properties.

1 Introduction

Android [1] has risen as one of the most popular mobile operating systems. As the popularity of Android increases so is the need for validation techniques. A huge number of applications is available and an exhaustive/satisfactory validation process is missing. With this success has emerged bugged applications (because of complex life-cycle) and malwares that could seriously hinder devices' integrity and users' privacy [2].

Monitoring the behavior of Android applications appear as a candidate solution to circumvent these problems [3, 4]. Runtime verification (RV) and enforcement (RE) are increasingly popular and effective dynamic validation techniques aiming at checking and ensuring the correct behavior of systems, respectively. These techniques consist in synthesizing a *monitor* from a high-level specification language, instrument the system and then integrate the monitor at relevant locations. At runtime, the monitor observes and possibly corrects the system's execution. In most of the runtime verification frameworks, instrumentation is automatic and relies on efficient and effective frameworks, e.g., aspect-oriented programming [5] and AspectJ (www.eclipse.org/aspectj/) its implementation for Java.

However, for the Android platform such an effective instrumentation technique did not exist until quite recently [6]. Consequently, previously proposed approaches [3,

[⋆] This work was funded in part by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project.

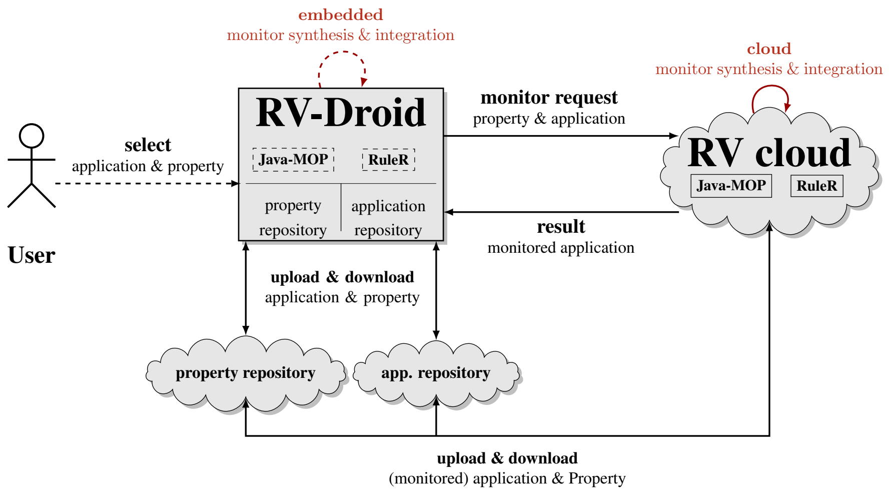


Fig. 1: RV-Droid context

4] had to modify, to different extents, the Android system to be able to log security-sensitive events. The downsides are limited portability between platforms and incompatibility with future releases of the operating system.

Contributions We propose a framework that gets closer to the principles of runtime verification. Based on an in-house version of the AspectJ compiler [6] for the Android platform, we propose RV-Droid, a framework for "traditional" and user-friendly RV and RE that is compatible with state-of-the-art tools. RV-Droid is a stand alone Android application that does not require any modification to any part of Android devices. RV-Droid takes credit from Java-MOP [7] and RuleR [8] hence allowing efficient monitoring of various expressive specification formalisms. Because Android applications use a unified API where most of the sensitive operations go through a clearly identified set of methods, it becomes easy to write properties and monitors that work with any application. We propose several examples of such requirements. Finally, the architecture behind RV-Droid can be seen as a basis that can be further extended into more specialized implementations.

Paper Organization Due to space reason, we do not provide an overview of Android, as literature abound on the subject and we believe that the architecture of RV-Droid and monitored properties are self-intelligible. Section 2 presents RV-Droid and its architecture. Experimentation and evaluation of monitoring properties with RV-Droid is done in Section 3. Related work is discussed in Section 4, while Section 5 draws some conclusions and present future developments.

2 An overview of RV-Droid

RV-Droid is an Android application that interacts closely with a dedicated cloud (see Fig. 1). We provide a description of its features and some insights about its internal architecture. It represents approximately 4,200 LLOC (libraries and third-party tools excluded): 3,000 for the Android application and 1,200 for the cloud. RV-Droid allows user-friendly runtime verification of Android applications. RV-Droid takes as input an existing Android application with a property and then:

1. it synthesizes a monitor for the property, and
2. it integrates the monitor inside the application in a transparent way for the user.

RV-Droid works with Android Froyo 2.2 or higher, and does not require any modification to neither the Android kernel nor any part of the targeted device. Applications can be retrieved *off-the-shelf* from a personal (local or remote) repository or Google Play. Properties are available in a repository and are selected by the user according to an abstract description (informal requirement), the events involved in the property, and the formalization of the requirement. Monitor synthesis and runtime monitoring rely on third-party RV tools such as (for now) Java-MOP [7] and RuleR[1] [8]. Mainly, two operations are performed by RV-Droid: monitor synthesis and monitor integration. Monitor synthesis consists in taking as input a property and generate some monitoring code, i.e., a decision procedure for this property. Monitor integration consists in instrumenting the target application to observe the relevant events that will trigger the monitoring code. For this purpose, RV-Droid relies on the aspect technology and an in-house version of the AspectJ compiler.

Monitor integration and aspect-oriented programming RV-Droid supports two monitor integration (and aspect weaving) modes: embedded or in the cloud. Support of Aspect-Oriented Programming (AOP) on Android is ensured by Weave Droid, an in-house version of the AspectJ compiler [6]. One of the challenges faced by RV-Droid is to circumvent the current limitations to use AOP on Android applications that seriously hinder the mobility of the device and forbids "standard" runtime verification. For a description of the previously existing issues in using AOP on Android, the reader is referred to [6]. In a nutshell, the issues stem from the incompatibility of existing aspect-compilers with the Android `.apk` files (Android target binary file format). From an abstract point of view, our weaving process is achieved in several stages that mainly are: de-compile the application, weave the classes, convert the classes again in a format that Android can execute, and sign the application. These steps rely partly on third-party tools such as dex2jar (http://code.google.com/p/dex2jar/), Android dx tool (http://developer.android.com), and Zipsigner (http://code.google.com/p/zip-signer/).

Some code is shared between Weave Droid and RV-Droid, but Weave Droid has been re-implemented since then to make it more generic, and, to use indifferently aspects or specifications used by runtime verification tools.

Using third-party runtime verification tools Based on the previously described process, monitor synthesis and monitor integration become possible using third-party runtime verification tools. Java-MOP provides facilities for monitor synthesis by generating aspects that query monitoring code in a library. RuleR does not provide aspect synthesis facilities and one has to provide a specification together with the suitable aspect that will query the RuleR engine in a third-party library. We had to modify the third-party monitoring libraries because of some initial incompatibility with the Android system. From

[1] Courtesy of Howard Barringer and Klaus Havelund who offered a pre-release version.

Table 1: Benchmarks for properties over Java data structures – Galaxy Tab 10.1

Property	B1 (3.134) (s)		B2 (524.4) (ms)		B3 (489.3) (ms)	
	mon (s)	ovhd (%)	mon (ms)	ovhd (%)	mon (ms)	ovhd (%)
HasNext	3.439	9.732	552.1	5.282	547.8	11.956
UnsafeIterator	3.182	1.532	568.3	8.371	498.7	1.921
SafeEnum	3.189	1.755	591.3	12.757	512.3	4.701
SafeFileWriter	4.171	33.089	632.0	20.519	540.1	10.382
SafeSyncColl.	3.141	0.223	544.9	3.909	525.7	7.439
HashSet	3.142	0.255	574.8	9.611	549.8	12.365
UnsafeMapIterator	3.251	3.733	563.1	7.380	548	11.997
SafeSyncMap	3.152	0.574	540.4	3.051	553.7	13.162

Table 2: Benchmarks for properties over Java data structures – Galaxy Gio S5660

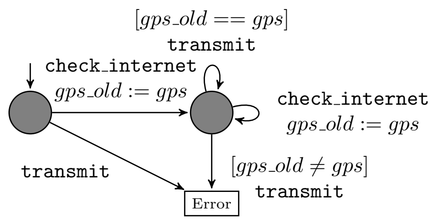
Property	B1 (19.65) (s)		B2 (1346) (ms)		B3 (2092) (ms)	
	mon (s)	ovhd (%)	mon (ms)	ovhd (%)	mon (ms)	ovhd (%)
HasNext	20.898	6.315	1567	16.359	3013	43.99
UnsafeIterator	21.115	7.419	2462	82.87	3121	49.15
SafeEnum	19.966	1.570	2476	83.894	2989	42.84
SafeFileWriter	20.532	4.454	2399	78.169	3569	70.56
SafeSyncColl.	20.623	4.939	2305	71.189	3035	45.04
HashSet	21.512	9.440	2292	70.2	2842	35.82
UnsafeMapIterator	20.775	5.689	2431	80.575	2895	38.35
SafeSyncMap	20.25	3.015	2416	79.46	3254	55.50

an abstract point of view, these libraries call some Java classes that are not provided by the Android runtime. Thus, we had to redirect these calls to a customized version of the Java runtime library. Note that the aforementioned modifications are transparent to the user who, in all cases, has only to download and install an Android application.

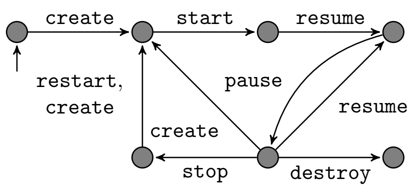
The remote processes are implemented as a web service queried by RV-Droid using the Simple Object Access Protocol (SOAP) and the web service client library kSoap (http://ksoap2.sourceforge.net/). The web service is deployed in the Glassfish application server. The two repositories execute on an SSH file transfer protocol server (SFTP).

3 Experimentation and Evaluation

Verifying correct usage of Java data structures To evaluate RV-Droid, and assess the performance of state-of-the-art RV tools on recent Android devices, we carried out performance evaluation of Java-MOP monitors on three benchmarks with usual properties (available at Java-Mop's website). The first device is a Samsung Galaxy Tab 10.1, a tablet, with processor NVIDIA Tegra 2 dual core 1GHz and 1GB of RAM running on Honeycomb 3.1. The second device is a Samsung Galaxy Gio S5660, a mobile phone, with a 800 MHz processor and 278MB of RAM, running on Froyo 2.2.1. The considered benchmarks were Linpack (http://www.greenecomputing.com, B1), BenchmarkPi



(a) Checking internet connection



(b) Activity life-cycle

Fig. 2: Some properties inspired from the developer's guide

(http://androidbenchmark.com, B2), and DaCapo-xalan (http://dacapobench.org, B3). Linpack provides a general evaluation of the performance of the Dalvik virtual machine. BenchmarkPi provides an evaluation of the processor power of the device. DaCapo is the traditional benchmark used in RV that makes intensive use of Java data structures. Linpack and BenchmarkPi were taken off-the-shelf. However, using DaCapo required tweaking the original code, and, based on code analysis, we discovered that it is possible for only 6 of the 14 applications inside the benchmark.

Performance results are shown in Tables 1 and 2 for the tablet and mobile phone, respectively. On the first line, for each benchmark, the execution time without monitor is indicated. For each property, the entries mon and ovhd indicate the average time for 10 executions of the monitored application and the induced overhead, respectively.

Verifying Android programming good practices. We monitored properties indicating whether Android's programming guidelines [1] are respected on some of the most popular games. Due to space reasons, an abstract monitor is given only for P1 and P2.

P1 *Before transmitting any data, it must be ensured that the device is connected to internet. And, it should be checked again each time the device is moved.* An abstract representation of the monitor used for this property is represented in Fig. 2a.

P2 *All methods involved in the activity life-cycle should be overridden.* To check whether the developer has followed this requirement, we can write an aspect that instruments those methods and tracks the (simplified) application life-cycle represented in Fig. 2b. If the method has been overridden by the developer, an event (corresponding to the method name) will be emitted by the monitored program. If, in a state, an unexpected event is emitted, it means that there is at least one method not overridden by the developer.

P3 *The device rotation facility should not be disabled.*

P4 *Only one dialogue window should be poped-up.*

P5 *In the restricted-memory mode, an application should start at most one service and end it, and not let the Dalvik virtual machine kill it.*

Preventing security issues through runtime enforcement. Among the 27 security findings discovered in [9], we wrote a monitor for 19 of them to either detect the vulnerability or even prevent it by disabling malicious method calls. The 8 remaining findings were related to too general concepts (e.g., "some developers toolkits probe for permissions through customized methods"). Being able to write a monitor to prevent security issues mostly depends on whether the referred sensitive data is retrieved through method calls. Method calls are caught by monitors and the data (passed as parameter) is then analyzed (e.g., an URI or string containing a premium-rate phone number).

4 Related Work

Both static and dynamic methods already exist to validate Android applications.

Static validation techniques Verification of Android applications has been mostly investigated in relation with Android permissions [10]. At installation time, the user is asked whether the downloaded application is allowed to access security-relevant parts of the API. Stonaway [10] is a static analysis tool that determine whether applications disobey the principle of least privilege. Stonaway compares the permissions required by the calls made to Android's API to the permissions requested by the application. ComDroid [11] similarly analyses inter-application communication by examining emissions and receptions of intents (i.e., more or less messages) between applications to prevent information disclosure.

Dynamic analysis of Android applications TaintDroid [3] is a framework for information-flow analysis of Android applications. It is based on information tainting and log collecting to determine whether sensitive information flows between applications.

Even closer to our work is a framework where a monitor runs on an Android device as a stand-alone application [4]. The "light" version of this approach modifies two files of the Android system to get notifications about security-sensitive events. This mild modification comes at the price of not being able to observe some low-level, potentially security-sensitive, operations. To circumvent this problem and get information about more events, the authors propose an in-house kernel module that has to load during boot. It is thus an *out-line* monitoring approach based on permission requests seen as events. Moreover, monitored properties are specified in an LTL variant and monitored using progression (i.e., formula rewriting).

Comparison with our approach RV-Droid falls in the category of dynamic-analysis approaches. In contrast with existing approaches, RV-Droid is based on aspect-oriented programming for instrumentation. RV-Droid performs *in-line/on-line* monitoring, and, it features the following novelties and advantages. RV-Droid does not modify Android architecture, which, in our opinion, greatly favors usability, portability, and compatibility with next releases of Android. Monitors can be expressed using any event observable through AspectJ. RV-Droid is not restricted to security properties, and, more general properties (e.g., correct implementation, debugging, statistics, etc) can be considered. Moreover, RV-Droid takes credit from Java-MOP and RuleR which are complementary in terms of expressiveness and efficiency and offer several input formalisms. While [4] is based on progression that can cause the size of the monitored formula to augment with the length of the trace, our monitors have been tested by running monitored applications for more than an hour without noticeable overhead. Also, RV-Droid permits runtime enforcement by e.g., disabling dangerous method calls. Finally, with a reasonable effort, RV-Droid can be extended to also support off-line monitoring.

5 Conclusion & Future Work and Developments

RV-Droid widens the interest of runtime verification to a large set of potential applications on mobile devices. Our framework is in the line of traditional of RV frameworks:

(i) applications are seen as black boxes, (ii) applications are taken off-the-shelf, and (iii) the execution platform does not need to be instrumented. Our tool is still a prototype but will be released soon on Google code and Google Play.

We plan several conceptual extensions. Enforcement on method calls as presented in this paper can be extended to ensure the good usage of interfaces [12]. We also plan to design more elaborated aspects to be able to prevent intent-based attack surfaces [11] that requires to analyze the manifest data.

In the roadmap of RV-Droid, we plan to propose i) repositories of debugging and security monitors (aspects synthesized from properties), ii) integration with complementary RV tools (e.g., LARVA [13]), iii) customized application installer and donwloader where applications are automatically augmented with monitors after download, iv) repositories with sanitized (monitored) applications.

References

1. Google Inc.: Android developer site (2012) http://developer.android.com.
2. Nouveau, T.: The Rise of Android Malware (Nov 2011) TG Daily.
3. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Arpaci-Dusseau, R.H., Chen, B., eds.: OSDI, USENIX Association (2010) 393–407
4. Bauer, A., Küster, J.C., Vegliach, G.: Runtime verification meets Android security. In Goodloe, A., Person, S., eds.: NASA Formal Methods. Volume 7226 of LNCS., Springer (2012) 174–180
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
6. Falcone, Y., Currea, S.: Weave Droid: Aspect-Oriented Programming on Android Devices – Fully Embedded or in the Cloud. In: ASE'12: the 27th IEEE/ACM International Conference on Automated Software Engineering. (2012) To appear. Preprint available online.
7. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14** (2012) 249–289
8. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. J. Log. Comput. **20** (2010) 675–706
9. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX conference on Security. SEC'11, Berkeley, CA, USA, USENIX Association (2011) 21–21
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In Chen, Y., Danezis, G., Shmatikov, V., eds.: ACM CCS, ACM (2011) 627–638
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys'11, ACM (2011) 239–252
12. Hallé, S., Villemaire, R.: Browser-based enforcement of interface contracts in web applications with BeepBeep. In Bouajjani, A., Maler, O., eds.: CAV. Volume 5643 of LNCS., Springer (2009) 648–653
13. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java programs (tool paper). In Hung, D.V., Krishnan, P., eds.: SEFM, IEEE Computer Society (2009) 33–37