VeriDroid: Automating Android Application Verification

Yepang Liu
Dept. of Comp. Sci. and Engr.
The Hong Kong Univ. of Sci. and Tech.
Kowloon, Hong Kong, China
andrewust@cse.ust.hk

Chang Xu*
State Key Lab for Novel Soft. Tech.
Dept. of Comp. Sci. and Tech.
Nanjing University, Nanjing, China
changxu@nju.edu.cn

ABSTRACT
Smartphone applications' quality is vital. Many smartphone applications, however, suffer from various defects. One major reason is that developers lack viable techniques to expose potential defects in their applications. This paper presents a tool VeriDroid to help automatically verify Android applications. We built VeriDroid by extending Java PathFinder (JPF), a widely-used verification framework for general Java programs. Our extension addresses two technical challenges. First, Android applications are event-driven and lack explicit calling relationships between event handlers for verification. Second, Android applications closely hinge on different framework libraries, whose implementations are platform-dependent. To address these challenges, we derive event handler scheduling policies from Android documentations, and encode them to guide JPF to realistically execute Android applications. Besides, we model side effects for a critical set of Android APIs such that one can conduct verification precisely. By doing so, our VeriDroid can verify Android applications in a fully automated manner. We implemented a prototype checker on VeriDroid and applied it to detect null-pointer dereference and resource leak defects in Android applications. Our experiments with five large-scale and popularly-downloaded subjects showed that VeriDroid can effectively detect real defects and provide actionable information to facilitate program debugging.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging.

General Terms
Design, Verification.

Keywords
Android application, dynamic analysis, functional defects

1. INTRODUCTION
The market of smartphone applications is expanding at an unprecedented rate. As of July 2013, over one million Android applications on Google Play store have received 50 billion downloads [1]. Users rely on such applications for different purposes such as daily task assistance, entertainment, socializing or even financial management. As such, the software quality of these applications is of vital importance. Developers should extensively test their applications before shipping them.

Middleware 2013 Doctoral Symposium, December 10, 2013, Beijing, China.

Unfortunately, the reality is not optimistic. Many applications suffer from different kinds of defects. A notorious example is that Android SMS application intermittently sent meaningless messages to random recipients [4]. The pervasiveness of defects in smartphone applications is attributable to two major reasons. First, smartphone applications are typically developed by small teams without dedicated quality assurance. It is not realistic for developers to perform a thorough testing of their applications on different devices. In fact, many Android applications such as K-9 Mail [13], an email client with millions of downloads, do not even have a well-designed test suite. Second, unlike their desktop counterparts, smartphone platforms have a short history. Developers lack mature industrial-strength tools to help analyze their applications and expose defects. Existing tools like Robotium [14], although powerful, require non-trivial manual effort to provide certain models (e.g., GUI models) or write test cases to achieve an effective analysis. Thus, automated quality assurance tools for smartphone applications are desirable.

To facilitate automated defect detection, we in this paper present a tool VeriDroid, which is designed to help Android developers automatically verify their applications. VeriDroid is built by extending JPF, a widely-used verification framework for general Java programs [12][22]. This extension is a difficult task. Specifically, our earlier work [16] and related studies [18] identified two major technical challenges in extending JPF to analyze an Android application. The challenges are:

Lack of explicit control flows. Android applications follow an event-driven programming paradigm, which hides an application's program control flows in the canned machinery of the Android framework. Developers specify an application's logic in a set of loosely-coupled event handlers. At runtime, these event handlers are implicitly called by the Android system. For example, the onStart() lifecycle handler of an activity component is called after the onCreate() lifecycle handler (see Section 2.1 for details), but such calling order is never explicitly specified in the program code. This causes trouble for dynamic analysis tools like JPF as they are designed to execute and analyze programs whose control flows are explicitly stated.

Heavy reliance on native libraries. Android exposes more than 8,000 public APIs to developers [10]. Many of them rely on Android system functionalities or native libraries whose implementations are platform-specific (e.g., thread manipulation and GUI-related APIs). Related code is written in system-native languages (e.g., C), and thus not suitable to be executed in JPF's Java virtual machine. However, the side effect of such code must be considered, as otherwise JPF may encounter various unexpected problems when it executes and analyzes Android applications.

To address the first challenge, we derived event handler scheduling policies from Android documentations, and formulated these

* Corresponding Author.

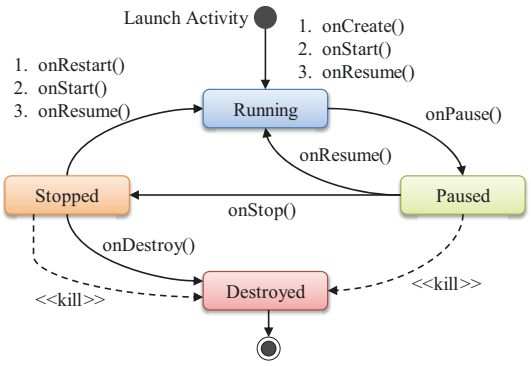

Figure 1. Activity lifecycle

policies as an Application Execution Model (AEM model). This model captures application-generic temporal rules that specify the calling relationships between event handlers. Then enforcing these rules at runtime can guide JPF to call event handlers at appropriate time. To address the second challenge, we identified a critical set of Android APIs that rely on native libraries, and properly modeled their side effects using JPF's listener and native peer mechanisms (see Section 2.2 for details). Such API modeling involves non-trivial effort as we will show later. By addressing these two challenges, JPF would be able to execute Android applications for verification.

To validate the effectiveness and usefulness of VeriDroid, we implemented and integrated a defect checker into it to detect null-pointer dereference and resource leak defects. In our evaluation, we applied VeriDroid to five large-scale and popularly-downloaded Android applications across four different categories. Encouragingly, VeriDroid successfully detected seven real defects in these applications, and the computational resource consumption is affordable. This work, together with our earlier work[2] [16], demonstrates that it is feasible to extend JPF for detecting both functional and non-functional defects in Android applications.

The rest of this paper is organized as follows. Section 2 gives the background of Android applications and JPF. Section 3 describes the design and technical details of VeriDroid. Section 4 evaluates VeriDroid using real-world Android application subjects, and Section 5 concludes this paper.

2. BACKGROUND

2.1 Android Programming

Android [3] is a widely-adopted Linux-based smartphone platform. Its applications are written in Java language with special enforcements. Typically, an Android application contains four types of components: *activity*, *service*, *broadcast receiver*, and *content provider* [2]. Activities are the only type of components that contain Graphical User Interfaces (GUIs). An application may comprise multiple activities that work together to provide a cohesive user experience. Services are components that run at background for conducting long-running tasks. Broadcast receivers define how an application responds to system-wide broadcast messages sent from other components, applications or the Android system. Content providers manage shared application data persisted in file systems, databases or network locations and provide an interface for data queries or modifications.

[2] Our earlier work extended JPF to detect energy problems (non-functional) in Android applications. This work further extends JPF to detect functional problems.

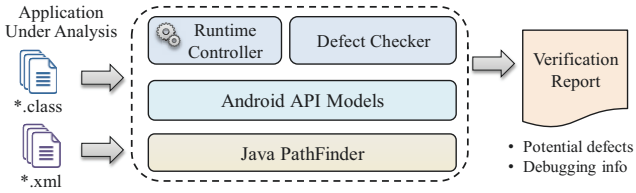

Figure 2. Approach overview

Each application component is required to follow a prescribed lifecycle that defines how this component is created, used and finally destroyed. For example, Figure 1 gives the lifecycle for an activity component. It starts with a call to onCreate() handler, and ends with a call to onDestroy() handler. An activity's foreground lifetime (i.e., the "Running" state) starts after calling onResume() handler, and lasts until onPause() handler is called, when another activity comes to foreground. An activity can interact with its user only when it is at foreground. When it goes to background and becomes invisible (i.e., the "Stopped" state), its onStop() handler would be called. When its user navigates back to a paused or stopped activity, that activity's onResume() or onRestart() handler would be called, and the activity would come to foreground again. In exceptional cases, a paused or stopped activity can be killed for releasing memory to other applications with higher priorities. Such prescribed lifecycles as well as other related handler scheduling policies need to be encoded in JPF for realistically executing Android applications.

2.2 Java PathFinder

JPF [22] is a highly customizable execution environment designed for verifying general Java programs. It takes as input the Java bytecode instructions of an application under analysis, and systematically executes the application in its Java virtual machine (JVM). The execution starts from an entry point, e.g., the main function of a conventional Java program. During execution, JPF checks for all violations of pre-specified properties. In theory, given an infinite amount of computational resource, JPF is able to explore the whole state space of an application for verification purposes. However, due to the "software state explosion" problem [8] and the limits of computational resource, we need to restrict JPF's state space exploration in practice.

JPF provides two major extension points: *listeners* and *native peers*. Listeners monitor JPF's internal executions and interact with JPF (recall that JPF executes an application in its own JVM). When there are special events happening in JPF's JVM (e.g., a certain bytecode instruction has been executed), the corresponding listeners that have been registered will be notified, and the actions defined in the listeners will be triggered. The second way of extending JPF is via native peers. Like Java Native Interface mechanism [15], the native peer mechanism in JPF supports delegating the execution of some methods to the host JVM (i.e., the JVM in which JPF runs). For example, when JPF encounters a native method that it is not able to execute, it will delegate the execution of this method to the host JVM. If a native peer of this method is defined, the host JVM will execute this peer such that some critical side effects of the native method will not be ignored.

3. APPROACH

3.1 Overview

Figure 2 gives an overview of VeriDroid. It takes two inputs: (1) the Java bytecode instructions of an Android application under analysis, and (2) the configuration files of this application (e.g., for learning GUI models). After verification, it generates a de-

tailed report including all detected defects and relevant debugging information. Conceptually, VeriDroid consists of two components, i.e., a runtime controller and a defect checker, and adds an Android library modeling layer on top of JPF. The runtime controller guides JPF to execute an Android application, and the defect checker detects certain types of defects (e.g., null-pointer dereference and resource leak defects). This overview looks intuitive. However, there are several challenges: (1) how to schedule event handlers in Android applications, and (2) how to model Android APIs using JPF's extension mechanisms. In the following, we discuss how we addressed such challenges and enabled VeriDroid to detect functional defects.

3.2 Event Handler Scheduling

Application execution model. An Android application starts with its main activity, and ends after all its components are destroyed. It keeps handling received events by calling their handlers according to Android specifications. Each call to an event handler may change the application's state by modifying its components' local or global program data. To realistically execute Android applications in JPF's JVM, we manually derived event handler scheduling policies from Android specifications, and organized them as an Application Execution Model (AEM).

Our AEM model is a collection of temporal rules that specify the calling relationships between event handlers. They are generic to all Android applications. Formally, we define our AEM model as follows (unary temporal connective *G* means "always"):

$$AEM \coloneqq G \bigwedge_i R_i$$

Each temporal rule is expressed in the following form:

$$R_i \coloneqq [\psi],[\phi] \Longrightarrow \lambda$$

In a rule R_i, ψ and λ are two temporal formulae expressed in linear-time temporal logic [9], and refer to the past and future, respectively. ϕ is a propositional logic formula referring to the present. ψ describes what has happened in an execution, ϕ evaluates the current situation (what event is received), and λ describes what should be done in the future. Then the whole rule can be interpreted as: *If both ψ and ϕ hold, λ should be executed next*.

We list two example rules in the following. The propositional connectives $\wedge$, $\Rightarrow$, and $\neg$ in these example rules follow their traditional interpretations, and the meaning of the temporal connectives is explained as follows. Unary temporal connective *X* means "next", and its past time analogue X^{-1} means "previously". Binary temporal connective *S* means "since". Specifically, a temporal formula "$F_1\ S\ F_2$" means that F_2 held at some time in the past, and since then F_1 always holds.

- Rule example 1: When to call the lifecycle handler onStart() of an activity component *act*?

  $$[\boldsymbol{X^{-1}}\ act.onCreate()], [\neg ACT\_FINISH\_EVENT] \Longrightarrow \boldsymbol{X}\ act.onStart()$$

- Rule example 2: When to call a button-click GUI event handler onClick()?

  $$[(\neg act.onPause()\ \boldsymbol{S}\ act.onResume()) \wedge (\neg btn.reg(null)\ \boldsymbol{S}\ btn.reg(listener))], [BTN\_CLICK\_EVENT] \Longrightarrow \boldsymbol{X}\ listener.onClick()$$

The first example rule states that the onStart() handler should be called after the onCreate() handler completes as long as the concerned activity does not finish. The second rule requires a button-click event handler to be called if: (1) the button is clicked, (2) its enclosing activity is at foreground (i.e., the activity's onPause() handler has not been called since the last call to onResume() handler), and (3) its click event listener is properly registered. More rule examples can be found in our earlier work [16] and technical report [17].

```
add main activity to activity stack and start it
while(application not stopping){
  get the stack top activity act
  if(act ready for interaction){
    randomly generate one GUI event
  } else{
    generate act's corresponding lifecycle event
  }
}
finish all active activities
finish all running services
```

Figure 3. Main scheduler's event generation algorithm

AEM model enforcement and handler scheduling. To enforce AEM model at runtime, we encoded it in the main scheduler of our runtime controller. All temporal rules in AEM model are converted to a decision procedure. This procedure helps the main scheduler to decide which event handler to call next according to the application's execution history and its newly received events. The events come from two sources: (1) those actively generated by the main scheduler, including all GUI events (e.g., button clicks), main activity's start event, activity components' lifecycle events and component destroying events; (2) other events (e.g., the event to start a service) passively monitored during the application's execution. We can observe that the main scheduler is sensitive to an application's execution history. So it needs to track the following information:

- **A stack of active activities.** Our main scheduler maintains active activities in a stack. Each active activity is associated with a GUI model, describing what UI widgets and GUI event listeners are defined in this activity. This model is obtained by analyzing the activity's layout configuration file [17].
- **A list of running services.** Running services are maintained in a list. Particularly, if a service is launched by binding from other application components, it will be associated with a collection of such components that are bound with it.
- **A list of registered broadcast receivers.** The main scheduler also tracks a list of broadcast receivers. Each registered broadcast receiver is associated with a filter specifying its interested message types and permissions.

The scheduler serves as the analysis entry point for JPF. Figure 3 gives the pseudo code for the main scheduler's event generator (a core function of the scheduler). It first starts the application by launching its main activity. After that it continuously generates corresponding events to simulate user interactions, until the application is terminated. In addition to the event generator, the main scheduler also contains a monitor to listen to all events (including all actively generated and passively monitored events) and handle them on the fly by querying the decision procedure (i.e., the encoded AEM model) and calling corresponding event handlers.

3.3 Android API Modeling and Abstraction

As mentioned earlier, many Android APIs leverage system level functionalities or rely on native libraries. This causes a big trouble to JPF. First, the bytecode instructions of these Android APIs are typically not available for analysis [18]. Second, even if their bytecode instructions are obtained (e.g., by building the Android framework to get the non-stub version of library classes), JPF will still fail to execute them because it has no idea about how to handle the transitively called native methods. Due to these reasons, we need to properly model such APIs and their side effects. We note that completely and precisely modeling all APIs requires

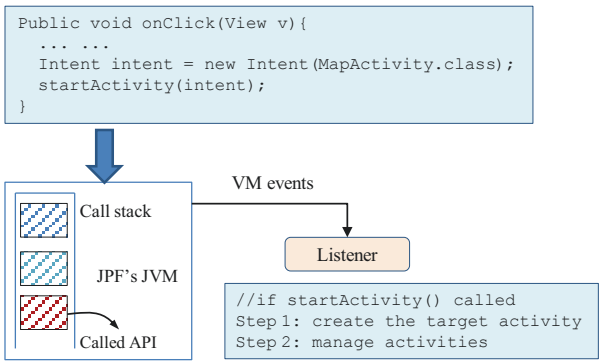

Figure 4. Modeling the startActivity() API

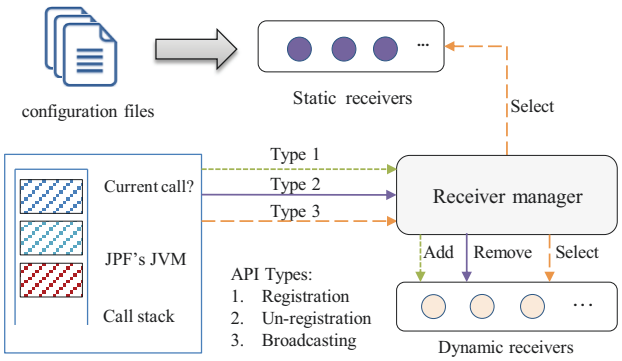

Figure 5. Broadcast receiver API modeling

enormous engineering effort. As such, in our work, we took a pragmatic approach by modeling a critical set of 76 APIs, which are commonly called in Android applications [17]. We give some concrete examples below for illustration.

Activity and service API modeling. Android applications can start new activities by calling several APIs (e.g., startActivity()). After completing certain tasks, an active activity can also be finished in several ways by calling corresponding APIs. Here, we introduce the modeling of startActivity() API as an example. The major effect of this API is to switch the current foreground activity to background, launch the new activity and put it to foreground. In order to model such effects, we guide JPF to conduct the following tasks[3], as illustrated in Figure 4:

- **API call interception.** We closely monitor an application's execution, and intercept each call to the startActivity() API.
- **Side effects abstraction.** The startActivity() API leverages system level functionality (e.g., thread manipulation), and relies on native code (e.g., native activity manager). We ignore its real implementation, and abstract its side effects by defining a stub native peer method in JPF.
- **Activity management.** After call interception and native peer creation, we can manage the activity stack to switch the current foreground activity to background and put the new activity to foreground (i.e., modeling the critical side effects).

The first two tasks can be done by registering a listener to monitor the JPF's JVM state and creating a stub native peer for the startActivity() API. The third task can be done by directly manipulating the call stack of JPF's JVM to invoke certain lifecycle event handlers of the concerned activities (e.g., pushing the event handlers onto call stack). Manipulating the call stack of JPF's JVM is needed because an application under analysis runs in JPF's JVM instead of the host JVM where JPF and its listeners run. Example code snippet for such direct call stack manipulation can be found in our technical report [17].

Similar to activities, other application components can start or stop a service by calling certain APIs. The modeling of these APIs resembles the modeling of activity APIs. We do not make further elaborations in this paper.

Broadcast receiver API modeling. In Android applications, broadcast receivers can be statically registered in application configuration files or dynamically registered in other components (typically, activities or services) at runtime by calling registration APIs. A dynamically registered receiver can be un-registered by calling un-registration APIs. Each broadcast receiver is associated with a message filter and permission (both can be optional), specifying its interested messages and required permission. After proper registration, other components can broadcast a message for handling at any time by calling the broadcasting APIs. As discussed earlier, our main scheduler maintains and manages a list of registered broadcast receivers (static receivers are considered as being always registered). Figure 5 illustrates how such management is done, as well as how the concerned APIs are modeled. As shown in the figure, the list of statically registered broadcast receivers is obtained by analyzing an application's configuration files. The dynamically registered broadcast receivers are managed by monitoring their registration and un-registration API calls. When a dynamic broadcast receiver is registered (i.e., a registration API is called), our receiver manager will add this receiver to the dynamic receiver list. Similarly, when a dynamic broadcast receiver is unregistered (i.e., a un-registration API is called), it will be removed from the dynamic receiver list. By maintaining both static and dynamic broadcast receivers, we can then properly pick up the appropriate receivers to handle broadcasted messages (i.e., when a broadcasting API is called) by checking the messages' details (e.g., message type and permission). Similar to the activity modeling, all management methods (e.g., receiver adding method) are called by directly manipulating JPF's JVM.

GUI-related API modeling. Graphical user interfaces (GUIs) play a central role in Android applications. However, their construction and manipulation highly rely on native code. This complicates the analysis of Android applications using JPF. Specifically, JPF will not be able to continue its execution when a GUI-related API is called. Therefore, we need to properly model GUI-related APIs. In the following, we discuss the modeling of a commonly called API findViewById() as an example. The modeling of other GUI-related APIs is similar. The findViewById() API traverses the UI element tree of the current activity's GUI (such a tree is constructed by parsing the layout configuration file of the concerned activity), and locates a UI element with a specific ID. Then the application can perform any legitimate operations on the retrieved UI element. For example, the following code snippet locates a button and registers a click event listener for it.

```
Button btn = (Button) findViewById(R.id.btn);
btn.setOnClickListener(myListener);
```

If we do not model any GUI-related API, the abovementioned UI element tree would not be available when findViewById() is called. This is because JPF cannot execute the GUI construction code in native libraries. Therefore, due to the prevalence of GUI-related code in Android applications, modeling such APIs is necessary. Figure 6 illustrates how we model the findViewById() API. It consists of two parts: a static part and a dynamic part. In the static part, we pre-analyze an application's configuration files

[3] The modeling of other APIs generally follow this three-step approach.

Table 1. Experimental subject information and detected defects

Application name	Category[1]	Downloads[1]	Revision no.	Size (LOC)	Source availability	Detected defects	
						NULL	LEAK
Ushahidi [21]	Communication	10K[2] ~ 50K	5750c01	20.4K	GitHub	0	1 (issue 100)
c:geo [7]	Entertainment	1M ~ 5M	44ee89c	27.0K	GitHub	1 (issue 124)[3]	0
Omnidroid [20]	Productivity	1K ~ 5K	727	11.7K	Google Code	1 (issue 77)	2 (issue 46, 103)
AnySoftKeyboard [5]	Tools	500K ~ 1M	6a1a580	19.3K	GitHub	1 (issue 80)	0
OI File Manager [19]	Productivity	5M ~ 10M	f41b141	6.7K	GitHub	1 (issue 30)	0

[1]: The category and download information is obtained from Google Play store [11]; [2]: 1K = 1,000 and 1M = 1,000,000

[3]: The detailed information of detected defects can be retrieve from corresponding issue tracking system using our provided IDs.

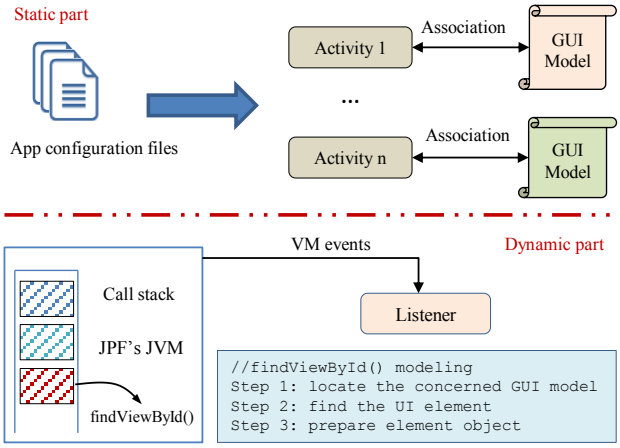

Figure 6. Modeling findViewById() API

to learn the GUI model of each activity component. The GUI model contains key information such as each UI element's type (e.g., a button), ID, and its associated text if any. Then, when the application is executed by JPF, we register a listener to monitor the call to findViewById(). When it is called, we would identify the current activity, and its GUI model. By doing so, we would obtain all necessary information about the UI element under search. Finally, we can create a corresponding object in JPF's JVM for the UI element if it has not been constructed.

In the above, we discussed the modeling of some representative Android APIs. We can see from the discussion that such modeling tasks are labor-intensive. It took us several months to model the 76 Android APIs. In practice, to save manual effort, one can choose to ignore the side effects of some APIs if these effects are not relevant to the verification target. This "partial modeling" helps reduce the state space that JPF needs to explore.

3.4 Defect Detection

With the two technical challenges addressed, VeriDroid is now able to execute an Android application. It continuously generates GUI events to simulate user interactions[4] so as to explore different application states for exposing defects. To study whether VeriDroid can help detect real defects, we implemented a checker for detecting null-pointer dereference and resource leak defects [6]. We in the following briefly discuss the detection algorithms.

[4] In this work, we do not study how to generate environmental inputs such as sensory data. If such inputs are needed for application execution, VeriDroid will randomly select a value from a pre-specified data pool.

Detect null-pointer dereference defects. To detect null-pointer dereference defects, VeriDroid actively monitors each dereference operation during the execution of an Android application. Specifically, VeriDroid intercepts the execution of a subset of Java bytecode instructions that involves object reference resolving. For example, the instruction *invokevirtual* invokes a virtual method on an object *obj*. After bytecode interception, VeriDroid checks if the concerned object reference (e.g., *obj*) equals null or not. If yes, VeriDroid reports a warning.

Detect resource leak defects. Resource leaks happen when an application fails to release computational resources (e.g., file handles and database cursors) it acquired from the operating system before it exits. Such defects are common in large applications [23] and can cause system performance degradation if the leak is serious, because computational resources are finite. To detect such defects, VeriDroid tracks the resource acquisition and releasing by monitoring certain API calls (e.g., the open() API call on a file object means a file handle is acquired from system). It maintains a list of acquired but not released computational resources. When an application exits, VeriDroid checks whether the list is empty. If not, it means certain resources are not properly released. Then VeriDroid will report warnings accordingly.

4. PRELIMINARY EVALUATION

To evaluate VeriDroid, we selected five real-world open-source Android applications. Table 1 lists the basic information of these applications, including (1) application name, (2) category, (3) number of downloads, (4) the revision we selected for experiments, (5) size of the selected revision, and (6) source code availability. As we can observe from the table, the selected applications are large-scale (up to 27 thousand lines of code), and popularly-downloaded (up to 10 million downloads). Besides, they cover four different application categories. We then built these application subjects and applied VeriDroid to verify them. For experimental purposes, we controlled VeriDroid to generate user event sequences whose length does not exceed six. This suffices for VeriDroid to explore thousands of different application states. All our experiments were run on a dual-core machine with Intel Core i5 CPU and 8GB RAM, running Windows 7 Professional SP1. We report our experimental results below.

Defect detection capability. Encouragingly, we found that VeriDroid successfully detected seven defects in our selected five application subjects. The last two columns of Table 1 report the information of these defects. For example, VeriDroid detected one null-pointer dereference defect and two resource leak defects in the Omnidroid application. These detected defects have been confirmed by developers and fixed in later revisions. This demonstrates that VeriDroid is capable of verifying Android applications and detect real defects. In addition, VeriDroid can also report

Table 2. Verification overhead

Application name	Verification time (Seconds)	Memory consumption (MB)
Ushahidi	29	129.1
c:geo	169	374.6
Omnidroid	151	141.7
AnySoftKeyboard	97	227.4
OI File Manager	22	112.3

actionable information (e.g., handler invocation sequences) to help developers debug these defects.

Computational resource consumption. Table 2 reports the resource consumption details when VeriDroid verifies our five application subjects. We can observe that even for the largest subject c:geo, the verification can finish within three minutes, with memory consumption less than 374.6 MB. Such verification overhead is well-supported by modern PCs. This suggests that developers can run tools like VeriDroid on their workstations to detect potential defects in their applications and fix these defects before releasing their applications to market.

Limitations and discussion. Our current implementation of VeriDroid has two major limitations. First, it can generate false alarms, especially when detecting null-pointer dereference defects. We analyzed corresponding warnings and realized the false alarms mostly arise from the incomplete and imprecise modeling of Android APIs. Although related studies [18] modeled certain APIs using simple stubs (e.g., randomly returning a value at call boundaries), our experience from the evaluation with real-world subjects suggests that the quality of API models can seriously affect certain analysis. So we believe these false alarms can be removed with more complete and precise modeling of Android APIs. Second, VeriDroid cannot measure the code coverage (e.g., branch coverage) of its verification. We are designing a coverage measurement component for VeriDroid. This can give developers more control on how to set parameters such as the length limit of generated user event sequences during verification.

5. CONCLUDING REMARKS

In this paper, we have presented a program verification tool, VeriDroid, for Android applications. VeriDroid is built by extending JPF, a widely-used Java program verification framework. We discussed in details how we enabled JPF to verify Android applications. Specifically, we addressed two challenges: (1) scheduling of event handlers, and (2) modeling of Android APIs. To validate the effectiveness of VeriDroid, we implemented and integrated a null-pointer dereference and resource leak defect checker into VeriDroid. We applied VeriDroid to five large-scale and popularly-downloaded Android applications. VeriDroid successfully detected seven real defects in these applications, and only consumed a reasonable amount of computational resources. This demonstrates the practical usefulness of VeriDroid.

In future, we are going to extend VeriDroid to detect more types of defects and address its major limitations. We wish our work can help improve the software quality of smartphone applications. This can benefit millions of smartphone users.

6. ACKNOWLEDGMENTS

This work was supported in part by National High-tech R&D Program (863 Program; Grant No. 2012AA011205), and National Natural Science Foundation (Grant Nos. 61100038, 91318301, 61361120097) of China, and by Research Grants Council (611912) of Hong Kong. Chang Xu was also partially supported by Program for New Century Excellent Talents in University, China (Grant No. NCET-10-0486). Besides, we also wish to thank our advisor S.C. Cheung for his kind guidance during this work and anonymous reviewers for their valuable comments on earlier versions of this paper.

7. REFERENCES

[1] "Google Play." http://en.wikipedia.org/wiki/Google_Play

[2] "Android developers." http://developer.android.com/

[3] "Android platform." http://www.android.com/

[4] "Android issue 9392." http://code.google.com/p/android

[5] "AnySoftKeyboard." https://github.com/AnySoftKeyboard

[6] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," *ACM TOSEM*, vol. 21, pp. 2:1-2:35, 2011.

[7] "c:geo." https://github.com/cgeo

[8] E.M. Clarke, K. William, N. Miloš, and Z. Paolo. "Model checking and the state explosion problem." In *Tools for Practical Software Verification*, 2012, pp. 1-30.

[9] K. Etessami, and T. Wilke, "An until hierarchy for temporal logic," In *Proc. IEEE Symp. Logic in Comp. Sci*. 1996, pp. 108-117.

[10] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permission demystified," In *Proc. ACM Conf. Computer and Communications Security*, 2011, pp. 627-638.

[11] "Google Play Store." https://play.google.com/store

[12] "Java PathFinder." http://babelfish.arc.nasa.gov/trac/jpf

[13] "K-9 Mail on Google Play." https://play.google.com/store/apps/details?id=com.fsck.k9

[14] "Robotium". http://code.google.com/p/robotium/

[15] "The JVM Specification." http://docs.oracle.com/

[16] Y. Liu, C. Xu, and S.C. Cheung, "Where has my battery gone? Finding sensor related energy black holes in smartphone applications," In *Proc. 11th IEEE Int'l Conf. Pervasive Computing and Communications*, 2013, pp. 2-10.

[17] Y. Liu, C. Xu, and S.C. Cheung, "Verifying Android apps using Java PathFinder," Technical Report HKUST-CS12-03. The Hong Kong Univ. of Sci. and Tech., September 2012.

[18] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *SIGSOFT Softw. Eng. Notes,* vol. 37, pp. 1-5, 2012.

[19] "OI File Manager." https://github.com/openintents/

[20] "Omnidroid." https://code.google.com/p/omnidroid

[21] "Ushahidi." https://github.com/ushahidi/

[22] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," In *Proc. Int'l Conf. Automated Soft. Engr.*, 2000, pp. 3-11.

[23] W. Weimer, and G. C. Necula, "Finding and preventing run-time error handling mistakes," In *Proc. ACM Conf. Object-oriented Programs, Sys., Lang., and Apps.*, 2004, pp. 419-431.