6-2018

Towards model checking Android applications

Guangdong BAI

Quanqi YE

Yongzheng WU

Heila BOTHA

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Citation

BAI, Guangdong; YE, Quanqi; WU, Yongzheng; BOTHA, Heila; SUN, Jun; LIU, Yang; DONG, Jin Song; and VISSER, Willem. Towards model checking Android applications. (2018). *IEEE Transactions on Software Engineering*. 44, (6), 595-612.
Available at: https://ink.library.smu.edu.sg/sis_research/4849

Author

Guangdong BAI, Quanqi YE, Yongzheng WU, Heila BOTHA, Jun SUN, Yang LIU, Jin Song DONG, and Willem VISSER

Towards Model Checking Android Applications

Guangdong Bai, Quanqi Ye, Yongzheng Wu, Heila Botha,
Jun Sun, Yang Liu, Jin Song Dong, and Willem Visser

Abstract—As feature-rich Android applications (*apps* for short) are increasingly popularized in security-sensitive scenarios, methods to verify their security properties are highly desirable. Existing approaches on verifying Android apps often have limited effectiveness. For instance, static analysis often suffers from a high false-positive rate, whereas approaches based on dynamic testing are limited in coverage. In this work, we propose an alternative approach, which is to apply the *software model checking* technique to verify Android apps. We have built a general framework named DroidPF upon Java PathFinder (JPF), towards model checking Android apps. In the framework, we craft an executable mock-up Android OS which enables JPF to dynamically explore the concrete state spaces of the tested apps; we construct programs to generate user interaction and environmental input so as to drive the dynamic execution of the apps; and we introduce Android specific reduction techniques to help alleviate the state space explosion. DroidPF focuses on common security vulnerabilities in Android apps including sensitive data leakage involving a non-trivial flow- and context-sensitive taint-style analysis. DroidPF has been evaluated with 131 apps, which include real-world apps, third-party libraries, malware samples and benchmarks for evaluating app analysis techniques like ours. DroidPF precisely identifies nearly all of the previously known security issues and nine previously unreported vulnerabilities/bugs.

Index Terms—Software model checking, security verification, android application

1 Introduction

In recent years, Android has gained an astonishing popularity. According to a recent study, it has reached nearly 87 percent smartphone market share [1]. The popularity of Android could be partially attributed to its well-evolved application ecosystem and the active developer community. Take Google Play, the official app market, as an example—so far, it has hosted more than 2,800,000 Android apps, with 60,000 new ones joining per month on average [2]. Besides, various alternative markets also host numerous apps. These apps are readily accessible and provide feature-rich functionalities for the mobile end users (simply *users* hereafter).

Android apps have been extensively used in security-sensitive scenarios. The users heavily rely on them to handle personal data (e.g., contacts, financial data and geographic location) and consume premium services (e.g., online banking, online shopping and sending SMS messages). Moreover, Android apps are playing an increasingly important role in enterprise, government and military bureaus. Nonetheless, various security issues of Android apps are continually being discovered and discussed, ranging from sensitive data leakage [3], [4], [5], to privilege escalation [6], [7], [8]. To enjoy the benefit of apps while preserving security, verifying them ahead of releasing and installation becomes imperative by app market operators and users.

Most of the prior studies rely on static analysis and dynamic testing for security analysis, for instance, detecting sensitive data leakage [9], [10], [11], [12] and analyzing capability leakage [6], [13]. These approaches may have their limitations. Static analysis may generate false alarms due to its inherent limitations in capturing runtime context (e.g., actual parameters and index of arrays) and tackling late-binding programming paradigms such as polymorphism and reflection. For instance, points-to analysis (which yields an over-approximation) is often used so that all potential violations are identified. In addition, the effectiveness of static analysis is restricted due to the distinctive features of Android's programming paradigm. Android apps are based on the Android OS, which can be regarded as a giant set of libraries containing both Java code and native code (so far, Android has consisted of more than 13 million lines of code [14]). The API calls to Android OS usually make a large part of the OS relevant to the verification and often lead to path explosion. To make the problem even worse, Android depends heavily on the callback mechanism (due to its event-driven and rich-interaction nature), which makes the API calls ubiquitous in the apps. Contrary to static analysis, dynamic testing only executes selected program paths and thus can precisely identify property violations [15], [16], but never proves their absence.

In this work, we seek an alternative approach for verifying Android apps against security properties. A potential technique is the software model checking [17], [18], which proposes an automatic way to verify properties of a finite-state

- G. Bai is with the Singapore Institute of Technology, Singapore 138682. E-mail: Guangdong.Bai@SingaporeTech.edu.sg.
- Q. Ye and J.S. Dong are with National University of Singapore, Singapore 119077. E-mail: {yequanqi, dcsdjs}@u.nus.edu.
- Y. Wu is with Huawei. E-mail: wu.yongzheng@huawei.com.
- J. Sun is with Singapore University of Technology and Design, Singapore 487372. E-mail: sunjun@sutd.edu.sg.
- Y. Liu is with Nanyang Technological University, Singapore 639798. E-mail: yangliu@ntu.edu.sg.
- H. Botha and W. Visser are with Stellenbosch University, Stellenbosch 7599, South Africa. E-mail: {hvdmerwe, wvisser}@cs.sun.ac.za.

Manuscript received 26 July 2015; revised 22 Jan. 2017; accepted 19 Mar. 2017. Date of publication 24 Apr. 2017; date of current version 20 June 2018.
Recommended for acceptance by Z. Hu.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2697848

system. The advantages of software model checking, compared with the above mentioned approaches, include that it does not generate false alarms if the model checker (e.g., Java Pathfinder (JPF) [17] for Java programs) actually executes the program under investigation and that it is capable of proving the absence of violations with an exhaustive state-space search. Inspired by software model checking, we started an effort to build an Android model checker named DroidPF upon JPF. The core technique of software model checking we use in this work is the dynamic *state-space exploration* which runs through the possible executions of an app. In the exploration engine, we also implement a taint-style system to track sensitive information flow for privacy property checking, and additional reachability checking for the privilege properties (e.g., blocking incoming SMS messages).

Using the state-of-the-art model checker JPF allows us to benefit from well-developed techniques of JPF, as well as the various evolving features from JPF's active developer community. Nonetheless, the challenge on applying JPF to verify Android apps is at least threefold. First, unlike ordinary Java programs, Android apps are tightly coupled with the Android OS which consists of a set of libraries containing both Java and native code, and complex inter-process communications. Although written in Java, the apps are compiled to bytecode that only runs on the Dalvik virtual machine instead of the traditional Java virtual machine (JVM). Therefore, it is hard for JPF to execute apps without using a real device or an emulator, not to mention storing and recovering program states. Second, due to the asynchronicity and event-driven execution paradigm, an app can have many entry points, whereas JPF allows only one entry at a time. One way to solve this problem is to construct a driver program that enumerates all possible event permutations, which would then activate all possible paths within the app. However, this approach could lead to false positives because some of the paths may never appear in practice. Lastly, there is the infamous state space explosion problem for model checking.

In order to tackle the first challenge, DroidPF includes an extensible mock-up Android OS that abstracts the Android OS using ordinary Java programs so that analysis techniques/tools developed for ordinary Java can be employed. There are techniques available to automatically generate mock-ups of the environment [19], [20], but since the dependencies within the environment are complex, these techniques are not mature enough [21]. Thus, we create the mock-ups manually, which is the same approach used by other tools that require a mock-up of their environment for model-checking purpose [17], [22], [23], symbolic execution purpose (e.g., KLEE [24] and S2E [25]) and so on. Mocking up an entire OS manually is a major effort, and DroidPF has supported a range of functionalities of Android OS which allows us to verify a number of small- and intermediate-scale (from 1 to 30 K lines of *smali* code) apps. Furthermore, DroidPF provides an extensible framework so that analysts can incrementally develop the mock-up. Our experience suggests that, based on the mock-up provided by DroidPF, it takes an analyst who has basic knowledge on Android OS a few days to incrementally develop the necessary mock-up for a given intermediate-scale app.

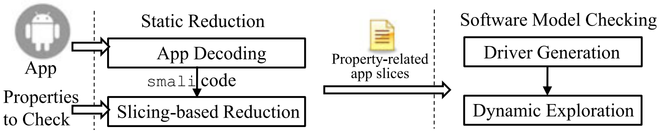


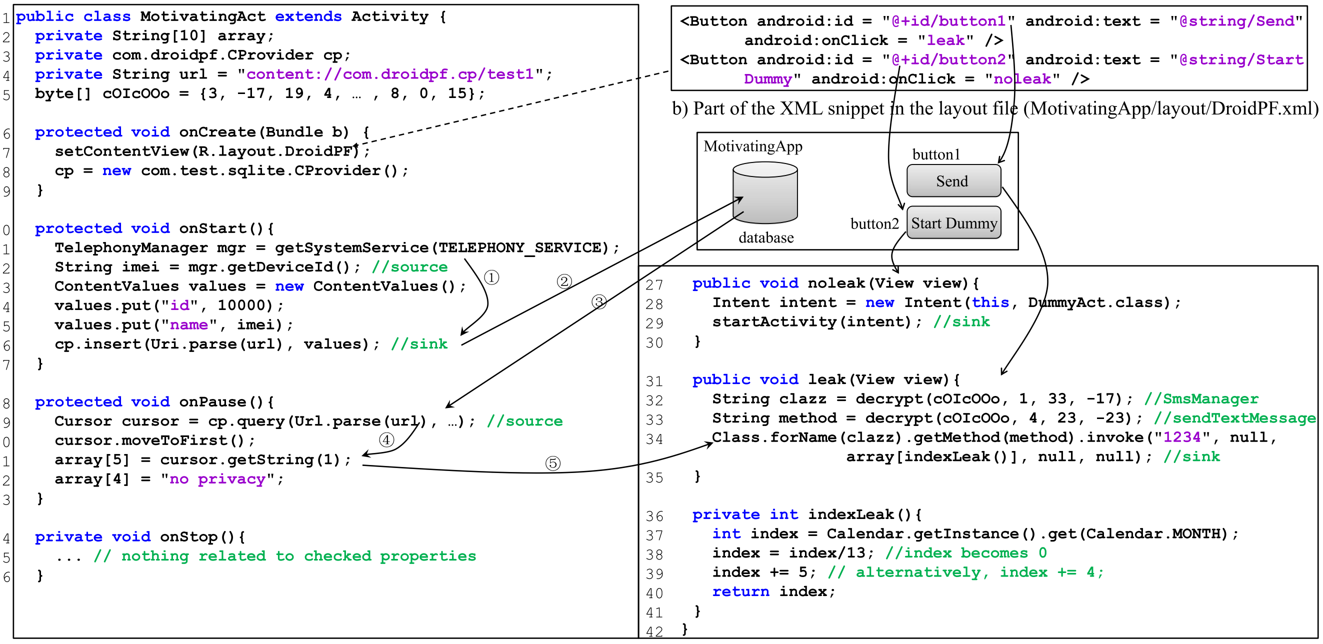
Fig. 1. High-level overview of our approach.

In order to tackle the second challenge, DroidPF generates driver programs which simulate and schedule event occurrences to drive the execution of the tested apps. We develop a technique called *dependency-constrained event permutation*. This technique reduces the event permutations by excluding impossible event sequences based on the dependency among the events. The details are in Section 6.1.

Lastly, in order to cope with the problem of state-space explosion, DroidPF applies static analysis first to reduce the app and then model check (i.e., dynamically explore) only the remaining parts of the app which are relevant to the checked properties. Fig. 1 shows the high-level work flow of our approach, which consists of two stages. First, we obtain an over-approximation of the program paths which might lead to property violations. We remark that though it is over-approximation (and therefore no precise points-to analysis is required), DroidPF does not produce false alarms as bugs are reported only after actually executing the app. Afterward, the approximation is used to reduce the program by removing the assured safe parts of the app and we then model check the reduced app against the properties. As a result, the efficient but imprecise static analysis reduces the state space, whereas the precise dynamic exploration pinpoints the actual violations of the properties.

To summarize our main contributions, we propose DroidPF, a verification framework which employs the state-of-the-art software model checking techniques to Android apps. It enables JPF-based dynamic state-space exploration, by including an extensible mock-up Android OS that provides the major OS functionalities. To the best of our knowledge, DroidPF is the first attempt to apply software model checking to verify security properties of Android apps.

We acknowledge that the effectiveness of DroidPF is limited by not only the underlying JPF engine, but also the correctness and completeness of the mock-up OS. However, verifying the equivalence of the mock-up and the Android OS, or the correctness of the mock-up is a far more challenging task, and therefore is not discussed within the scope of this paper. Nonetheless, we empirically show that DroidPF not only can find bugs but also verify apps in a number of cases. We evaluate DroidPF using three sets (a total of 131) of apps: thirteen real-world apps (including two ad libraries embedded by them) downloaded from Google Play, F-Droid and Anzhi app markets, seven known malware samples and the DroidBench [10] which itself is a comprehensive benchmark (including 62 apps of version 1.0 and 49 of version 2.0) built to evaluate information leakage analysis. DroidPF detects nearly all (except those with implicit information flow) of the known security issues from the malware samples, five previously unreported data leakages from three apps and both ad libraries, a *use-after-free* bug from the benchmark, a deadlock from a real-world app, and two software bugs leading to app crash.



```
public class MotivatingAct extends Activity {
  private String[10] array;
  private com.droidpf.CProvider cp;
  private String url = "content://com.droidpf.cp/test1";
  byte[] cOIcOOo = {3, -17, 19, 4, … , 8, 0, 15};

  protected void onCreate(Bundle b) {
    setContentView(R.layout.DroidPF);
    cp = new com.test.sqlite.CProvider();
  }

  protected void onStart(){
    TelephonyManager mgr = getSystemService(TELEPHONY_SERVICE);
    String imei = mgr.getDeviceId(); //source
    ContentValues values = new ContentValues();
    values.put("id", 10000);
    values.put("name", imei);
    cp.insert(Uri.parse(url), values); //sink
  }

  protected void onPause(){
    Cursor cursor = cp.query(Url.parse(url), …); //source
    cursor.moveToFirst();
    array[5] = cursor.getString(1);
    array[4] = "no privacy";
  }

  private void onStop(){
    ... // nothing related to checked properties
  }
```

```
<Button android:id = "@+id/button1" android:text = "@string/Send"
      android:onClick = "leak" />
<Button android:id = "@+id/button2" android:text = "@string/Start
      Dummy" android:onClick = "noleak" />
```

b) Part of the XML snippet in the layout file (MotivatingApp/layout/DroidPF.xml)

```
  public void noleak(View view){
    Intent intent = new Intent(this, DummyAct.class);
    startActivity(intent); //sink
  }

  public void leak(View view){
    String clazz = decrypt(cOIcOOo, 1, 33, -17); //SmsManager
    String method = decrypt(cOIcOOo, 4, 23, -23); //sendTextMessage
    Class.forName(clazz).getMethod(method).invoke("1234", null,
                      array[indexLeak()], null, null); //sink
  }

  private int indexLeak(){
    int index = Calendar.getInstance().get(Calendar.MONTH);
    index = index/13; //index becomes 0
    index += 5; // alternatively, index += 4;
    return index;
  }
}
```

a) Activity component of the illustrative app

Fig. 2. The code snippets of the running example. (The numbered arrows stand for the data flow through the app.)

2 Background

In the following, we present an overview of the Android framework and exemplify the difficulties in verifying Android apps.

2.1 Overview of Android

Android apps consist of four types of app components: *activity*, *service*, *content provider* and *broadcast receiver*. An activity implements the foreground logic, such as Graphical User Interface (GUI); a service runs in the background; a content provider is devoted to data management, which can use files, SQLite databases or the web as its back-end persistent storage, and a broadcast receiver handles the broadcasts sent by the OS and other components. The app components interact through inter-component communication (or ICC), and the exchanged messages are called *intents*. The components included in an app can be either statically defined in its *manifest* file (`AndroidManifest.xml`) or dynamically created at runtime.

Unlike a traditional Java program with a `main` method as its only entry point, Android adopts an *event-driven* execution mechanism, where an app implements a set of callbacks as event handlers (which can be registered/unregistered either statically or dynamically). Whenever a particular event occurs (e.g., launching an app or clicking a button), the corresponding callback methods are invoked. Therefore, an app usually has multiple entry points and event handlers (*entry points* and *event handlers* are used interchangeably). The events can be categorized into three types.

- *Lifecycle Events.* A component's *lifecycle* consists of multiple stages. As the user launches, pauses and resumes an app, its components of the app change their stages accordingly. When a component enters a new stage, the corresponding callback that is pre-implemented by the app is invoked. For instance, an activity has multiple stages such as `launched`, `running`, `paused` and `stopped`. When the user clicks the app's launcher icon, the activity enters the `launched` stage. Accordingly, the method `onCreate()` is invoked first and afterwards `onStart()`.
- *GUI Events.* GUI events occur when the user interacts with the apps, which include two types: data inputs from the user and actions taken by the user.
- *ICC Events.* The ICC events are mostly used for communication among apps/components. Besides, when some particular events occur, the OS broadcasts a message or invokes the callback methods implemented by the app. For example, when there is an incoming SMS message, the OS broadcasts an intent containing the SMS message encoded in PDU format to notify the apps, and the `onReceive()` of the registered broadcast receivers will be invoked with the intent as a parameter. As another example, when the system is running low on memory, the `onLowMemory()` which is pre-implemented by the apps is invoked.

2.2 A Running Example

In the following, we present a simple app and illustrate the challenges in verifying Android apps. Fig. 2 shows a made-up app which combines features of three apps in the DroidBench [10] and a malware sample named Pincer. In this app, there exists a path through which the IMEI of the device is leaked, which is commonly seen in real-world malware samples [3]. Although we show source code in this section, DroidPF directly works with the Java bytecode of the apps.

This app contains three components, i.e., two activity components `MotivatingAct` and `DummyAct`, and a content provider component `CProvider`. The code of `DummyAct` and `CProvider` is omitted in the figure to save space. `DummyAct` does nothing but just displays an empty canvas (i.e., not security-sensitive). Besides, two buttons are defined in the `MotivatingAct`'s `xml` layout file and their click handlers are set as `leak` and `noleak`, respectively (Fig. 2b). When `MotivatingAct` is started, it reads the IMEI by invoking `getDeviceID()` (line 12) and stores it in the database maintained by `CProvider` (line 16). When `MotivatingAct` is paused (`onPause()` is invoked), the IMEI is read from `CProvider` (line 19-21). After that, once `button1` is clicked, the IMEI is sent to a particular phone number through an SMS message (line 34). Here the names of the class and method are obfuscated using reflection to obstruct security analysis. The concrete names are decrypted from a byte array at runtime (line 32-33).

This example demonstrates several technical challenges that may lead to imprecision in app verification.

- *Dependency on the OS.*The apps heavily depend on the Android OS—even a small app contains multiple Android API calls. These API calls are relevant to the app's behaviors such as the event handlers, and thus cannot be simply ignored during the analysis. For instance, if an analyzer does not examine the logic in `setContentView` (where the `xml` layout in Fig. 2b is registered), the binding between the button and the click handler would be missed. As a result, the data leakage would not be detected because it is activated by the `onClick` handler. An intuitive remedy would be to analyze the part of the OS which becomes relevant given an API call. However, due to the complexity of the OS, even a simple API call would imply that a big portion of the OS must be analyzed.
- *Data Management with Native Code.*Android apps rely on the OS to manage the data. Unfortunately, data management in the OS is mostly implemented in native code. For example, after `MotivatingAct` requests `CProvider` for data management (line 16&19), `CProvider` uses SQLite which is implemented in native code to manage the database. To precisely track the data flow, the internals of the database management must be taken into consideration. Nonetheless, analyzing native code is highly non-trivial.
- *Asynchronous Event Occurrences.* Events in Android may asynchronously occur. The order of these events is unpredictable and nondeterministic, but may be relevant to the checked properties. For example, in the running example, `onCreate`→`onStart`→ `button1.onClick` is safe but `onCreate`→`onStart`→ `onPause`→`button1.onClick` causes data leakage.
- *Obfuscation.* To obstruct malware detection, malware developers may employ obfuscation techniques [26], [27]. This example shows an obfuscation technique similar to what Pincer uses. It stores seemingly random and meaningless bytes into the byte array `cOIcOOo` (line 5), and calculates the name of the method to be invoked at runtime using an intentionally complicated function *decrypt* based on three delicately designed indices of the byte array (lines 32-33). We demonstrate more details on this technique later in Section 7.1.
- *Dynamic Valuation.*The API `sendTextMessage` is invoked to send an SMS message (line 34). To decide whether this call leaks sensitive information (i.e., the IMEI), the actual values of the parameters have to be examined. However, line 37-40 make it difficult to statically determine the value of `index`.

3 Problem Definition and Overview

In this section, we define the problem of model checking Android apps (e.g., what is the model and what is the property) and present an overview on how DroidPF solves the problem.

3.1 Goal and Scope

The goal of DroidPF is to provide a framework for dynamically and systematically exploring the state space of Android apps. During the exploration, DroidPF identifies malicious behaviors (e.g., abusing and misusing permissions and sensitive data). In this work, we focus on application-level threats and exclude those in the OS, since according to our study of the vulnerabilities impacting Android which are listed in CVE [28], most (86.8 percent of 387) security threats on the Android platform reside in the apps.

In our threat model, the attacker is able to develop and release malicious apps, or embed his malicious bytecode into benign apps (e.g., via the repackaging attack). The attacker can use obfuscation techniques, such as complicating code flow, inserting extraneous code blocks or using reflection. In addition, similar to related work in the literature [10], [29], DroidPF cannot handle hybrid apps that include native code (in the app itself not from the Android OS) and JavaScript code. In this work, we also do not consider implicit information leakage (discussed in Section 8).

3.2 The Model

In order to model check an app, we first formalize its behaviors. The semantics of an app can be defined as a transition system $\mathfrak{L}_{app} = (S, init, Tran)$, where S is the set of states; $init \in S$ is an initial state; and $Tran \subseteq S \times S$ is a transition relation.

A state in S is a *program state* of the app, which is a snapshot of the execution status. It consists of the status of the heap (e.g., the values of the program variables), the status of files and databases that the app possesses, the program counter and threads states. We remark a program state here is the same as a state in JPF. We refer the readers to [17] for details on how program states are represented and compared in JPF. A *transition* is a state change caused by executing an atomic sequence of instructions (i.e., a *block*).

Given the above transition system semantics of an app, we then define relevant terms in the standard way. The execution of an app is formalized as *runs*, which are sequences of states in the transition system $\tau = \langle s_0, s_1, \ldots \rangle$, and a state s_n is *reachable* if there exists a run τ such that $s_0 = init$ and $(s_i, s_{i+1}) \in Tran$ for all $i < n$. Furthermore, given a property (e.g., a temporal logic formula), the truth value of the property is defined based on the above-defined transition system in the standard way (discussed soon in Section 3.3).

As an example, Fig. 3 shows part of the transition system generated from our running example. Each state contains the values of the global variables and status of the database (other information such as the program counter is not shown). At each state, there are multiple choices for

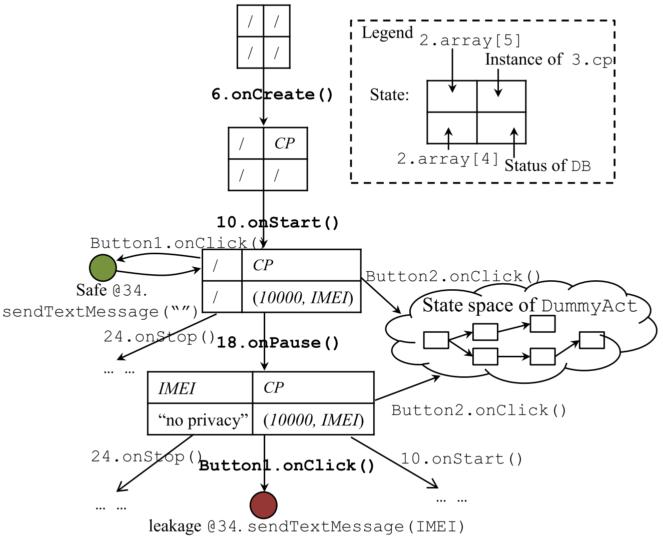


Fig. 3. Partial state space of the running example. (The numbers before the instructions/variables stands for the line numbers in Fig. 2; *CP* stands for a valid instance of the content provider; the emphasized path (in bold) shows the shortest path leading to a violation of privacy property.)

different behaviors. The different choices correspond to different thread scheduling and different event occurrences.

3.3 The Property

DroidPF focuses on checking the so-called privilege properties and privacy properties, which, according to the research of Android Malware Genome Project [3] (shown in Fig. 4), cover most of the behaviors of the existing malware. For instance, *privilege escalation*, *remote control* and *financial charges* require invocation of particular APIs, violating the privilege property; *information stealing* violates the privacy property.

A privilege property is related to the use of the sensitive permissions, e.g., stealthily sending SMS messages, deleting contacts and blocking incoming SMS messages. A privilege property is violated if there is a finite path in $\mathfrak{L}_{app}$ which leads to an invocation of the high-privileged APIs. In order to model check privilege properties, we extend $\mathfrak{L}_{app}$ with information on whether any high-privileged API is invoked or not. That is, for each state, we introduce an auxiliary Boolean variables *escalated* whose truth value tells whether a high-privileged API is invoked during the transition that leading to the current state. For each transition (s_i, s_{i+1}) in $\mathfrak{L}_{app}$, *escalated* is true at s_{i+1} if and only *escalated* is true at s_i or the transition is due to invocation of a high-privileged API. We remark that DroidPF includes a set of APIs which are commonly considered high-privileged (discussed soon in Section 4.2), and the analysts can customize this setting. It can be seen that the problem of model checking privilege properties is effectively reduced to a reachability analysis, i.e., whether a state where *escalated* is true is reachable.

A privacy property is related to the actions of disclosing the private information, e.g., IMEI, GPS location and contacts. A private property is violated if there is a finite path in $\mathfrak{L}_{app}$ from a private information *source* (e.g., contacts) to an information *sink* (e.g., the invocation of messaging sending API), and furthermore, the data at the sink must be tainted by the data at the source, i.e., there is a data dependency between them.

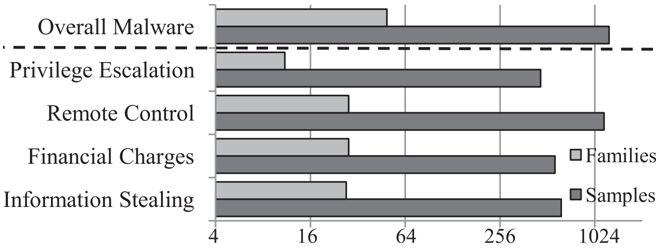


Fig. 4. The distribution of behaviors of existing malware. (The statistics is based on the Android malware genome project data set [3].)

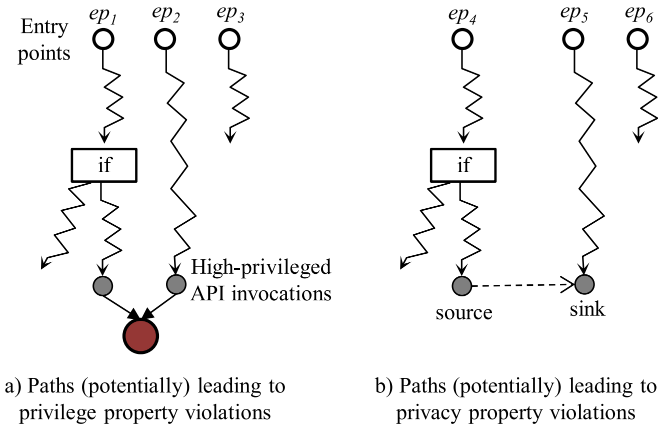
Privacy properties are often checked with the so-called taint analysis [10], [30]. The taint analysis can also be reduced to reachability checking. Similarly, in order to model checking privacy properties, we extend $\mathfrak{L}_{app}$ with information on data dependency. That is, for each data variable in the app, we introduce an auxiliary Boolean variable *tainted* whose truth value tells whether the variable in a state $s \in S$ is tainted. Initially, *tainted* is false. For each transition (s_i, s_{i+1}) in $\mathfrak{L}_{app}$, *tainted* is true at s_{i+1} if and only if the transition is caused by executing an introduction which is consider an information source (e.g., an API call for getting the GPS location), or there is a data-dependency between the data variable at s_{i+1} to a tainted variable at s_i. It can be seen that the problem of model checking private properties is effectively reduced to a reachability analysis, i.e., whether a state where the program counter points to a sink (e.g., an API call for sending a message) is reachable with the *tainted* of the variable to be used being true.

3.4 Overview of DroidPF

Based on the above discussion, the problem of model checking an Android app is reduced to reachability analysis and thus can be solved using model checkers like JPF, which supports reachability analysis, among other things. JPF works by systematically and dynamically enumerating the states of a Java program. However, as discussed previously, there are three challenges that we need to solve before JPF can be applied. In the following, we present an overview of how DroidPF solves the challenges and leave the details in the following sections.

The high-level workflow of DroidPF is shown in Fig. 1. First, given a privilege property or privacy property, DroidPF performs a static analysis (i.e., program slicing) to remove the program paths which are assured to *never* lead to property violation. For instance, in Fig. 3, the state space contains some parts which are obviously irrelevant and can be identified statically, such as the state space of `DummyAct` and the partial space led by `onStop()`. Once the irrelevant parts are identified, DroidPF uses this information in the later exploration step, such that they are pruned and thus avoided during dynamic exploration. The details of this step is presented in Section 4.

Second, DroidPF relies on JPF to model check the reduced apps. DroidPF provides a mock-up of the Android OS so that JPF can dynamically execute the app. The details on how the mock-up OS in DroidPF is constructed is in Section 5. In addition, in order to systematically explore the state space, we need an environment which generates not only different scheduling but also possible valid event sequences. The former is solved by relying on JPF, which



a) Paths (potentially) leading to privilege property violations

b) Paths (potentially) leading to privacy property violations

Fig. 5. Paradigm of entry-point-wise static slicing. (After static reduction, DroidPF avoid exploring the states led by ep_3 and ep_6. Note that at static reduction step, DroidPF does not consider whether the data flow (the doted arrow in (b)) between source and sink is feasible or not.)

implements a backtrackable JVM that provides non-deterministic choices and control over thread scheduling. The latter (specific to the event-driven nature of Android apps) is solved by constructing driver programs which generates the relevant event sequences. On this step, we exclude the event sequences that are impossible to happen in reality, e.g., `onStart` → `onCreate` in the app shown in Fig. 2. Notice that these infeasible event sequences rely on domain knowledge to reduce and cannot be reduced by those general reduction techniques used in JPF. The details on how the driver program is constructed are presented in Section 6.

4 Static App Reduction

Recall that the problem of model checking a privilege property or a privacy property is reduced to the problem of checking whether certain bad state is reachable or not. The problem can be solved using JPF by exploring program paths. The trouble is of course there may be too many program paths. Thus, DroidPF performs a preprocessing step to eliminate the program paths that are guaranteed to be "safe"-executing which never leads to any bad state. To do this, we introduce a static app reduction based on static backward program slicing.

4.1 Overview of Slicing-Based Reduction

Program slicing [31] is a popular tool for program debugging, testing, abstraction and verification [32]. It takes as input a program and a *slicing criterion*, i.e., a particular program point of interest and a set of variables at that program point, and identifies the part of the original program which influences the slicing criterion, i.e., the part of program which the slicing criterion has a dependency on (including control and data dependence).

In [33], it has been shown that static program slicing is effective in reducing the size of the finite-state transition system generated from a program. In theory, only the part of program identified by program slicing is relevant to the property and the rest are safe and thus can be pruned. In practice, it is not that simple because standard program slicing typically may not guarantee to generate an executable slice, while DroidPF requires the reduced program to be executable (for dynamic exploration). To achieve this, DroidPF does not remove the irrelevant instructions from the original app. Instead, it only records the irrelevant event handlers and prevents them from being invoked during subsequent dynamic exploration. In this sense, program slicing used in DroidPF is coarser than traditional slicing algorithm. Recall that an Android app typically has more than one entry points. After the reduction, DroidPF only needs to execute the rest of entry points in the dynamic exploration. Fig. 5 illustrates how this works.

How effective this reduction can be depends on whether a large part of the app is irrelevant to the violation of verified property. Furthermore, in practice, given the presence of challenges like reflection, aliasing and polymorphism, the program slice is often an over-approximation, i.e., it may contain part of program which is in fact irrelevant to the slicing criteria. Nonetheless, by a simple argument, it is clear that as long as it is over-approximation, a program path relevant to the violation of the property is never pruned.

4.2 Slicing Criteria in DroidPF

Since DroidPF focuses on privilege and privacy properties, which can be violated only through invocation of a set of particular APIs, DroidPF sets the slicing criteria to be set of program instructions which contain those API calls. In addition, for conservativeness, it also includes the program instructions which may invoke those potentially sensitive APIs. In summary, depending on what the property is, the slicing criteria used in DroidPF may include the following categories.

- *High-privileged APIs.* Invocations of APIs that require privilege equal or higher than the Android's `dangerous` level may violate a privilege property, for example, `abortBroadcast()` which may block incoming SMS. Thus, we include the APIs that are mapped to Android's permission levels of *dangerous*, *signature* and *signatureOrSystem*, similar to the previous related studies [34], [35].
- *Taint Sources and Sinks.* In order to model check a privacy property, we need to examine the program paths which involve a taint source and a taint sink. Reaching a taint sink (e.g., an invocation of the system function to send an SMS) would potentially violate a privacy property and thus the taint sinks are included. In addition, we include taint sources in the set so that we can tell whether a path reaching a sink goes through a source. Identifying sources and sinks has been well researched in the literature [10], [36], [37] and thus DroidPF simply uses those defined by them, which include file/network/database I/O APIs (e.g., line 12, 16&19 in Fig. 2) and ICC APIs (e.g., line 29).
- *Reflection APIs.* Since the exact class/method/field accessed through reflection is hard to decide statically, reaching an API for reflection, such as `java. lang. reflect.Method.invoke()`, might lead to violation of the property and thus they are included (e.g., line 34 in Fig. 2).
- *APIs for Dynamic Registration.* Android allows an app to register components and event handlers both statically and dynamically. For example, Fig. 2b shows how to register the button and its `onClick` listener statically through a layout file. Equivalently, the

```
Button btn = new Button(this);
btn.setText("send");
btn.setOnClickListener(new View.OnclickLister(){public void
      onClick(View v){ leak(v); } });
```

Fig. 6. Dynamic event handler registration.

code snippet in Fig. 6 registers a button at runtime. Due to this flexibility, it is hard to predicate statically whether the dynamic registration would lead to a property violation and thus they are included.

4.3 Workflow

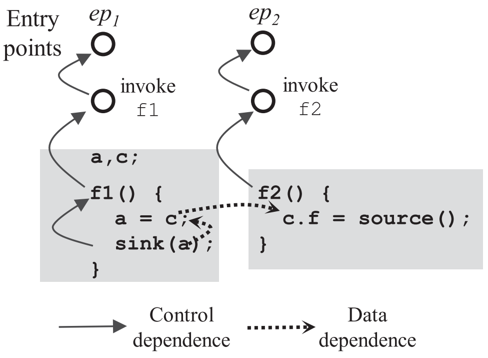
Our reduction works by starting the backward slicing from each slicing criterion in the set. It tracks both intra- and inter-component data and flow dependency. Since program slicing is a relatively mature technique, in the following, we merely brief the workflow of DroidPF.

Intra-Component Backward Slicing. For intra-component backward slicing, DroidPF searches for *use-def chains* and *call chains*. The former captures data flow relations while the latter captures call relations. Fig. 7 demonstrates these two relations in the intra-component backward slicing. First, based on the use-def chains, DroidPF identifies 1) registers and fields that have data dependency with the criteria, and 2) methods in which the depended data are modified (we call them relevant methods). For instance, tracking the parameter `array` at line 34 of Fig. 2 leads to identification of the method `onPause`. Second, based on the call chains, DroidPF identifies the paths which start from an event handler and lead to the invocation of any relevant method.

The more precise the slicing is, the more we can prune statically from the app. Nonetheless, because DroidPF relies on dynamic execution afterwards to find actual problems, we can afford to over-approximate (without worrying about false alarms) when a precise points-to analysis is expensive. In particular, the following strategies are adopted in DroidPF for efficiency. First, once an element of an array/string becomes relevant, the whole array/string becomes relevant. Second, once a field of an object becomes relevant, the whole object becomes relevant. Third, for polymorphism and overriding, whenever we cannot decide precisely, methods with the same signature are included.

Aliasing. For each variable contained in the slice, DroidPF searches backwards for its aliases. Once aliases detected, DroidPF adds them into workqueue as new criteria for further tracking. Fig. 8 demonstrates this process.

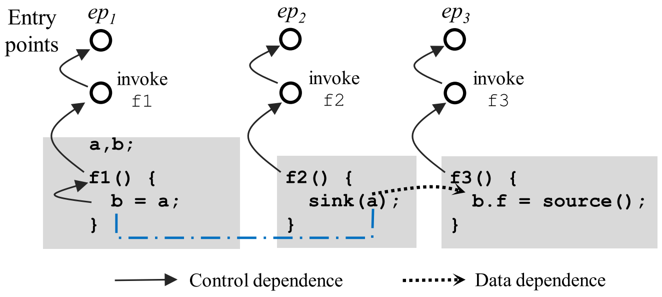
Reflection. As discussed in Section 4.2, the invocation sites of reflection APIs are taken as slicing criteria by DroidPF, but there may be some whose caller cannot be identified if multi-layer reflection calls are used. One of such examples is shown in Fig. 9. After tracking back to `f2` from the reflection API, DroidPF fails to find an entry point leading to `f2`, since `f2` is invoked through reflection. Under this circumstance, there is no need for DroidPF to approximate the caller who invokes `f2`. Because the reflection APIs are included in the slicing criteria, the functions which invokes `f2` using reflection, including `f1`, will be reached by the dynamic exploration.



```
a,c;
f1() {
  a = c;
  sink(a);
}
```

```
f2() {
  c.f = source();
}
```

Fig. 7. Demonstration of intra-component slicing.



```
a,b;
f1() {
  b = a;
}
```

```
f2() {
  sink(a);
}
```

```
f3() {
  b.f = source();
}
```

Fig. 8. Processing aliasing.

*Inter-Component Dependency.*DroidPF tracks inter-component flow as well. After identifying relevant methods within components, the next step is to collect the relevant components. The approach is to over-approximate the receivers of each ICC. Although it is sometimes feasible to statically identify the exact receivers of each ICC (e.g., using the approach in [38] which requires string analysis), for the same reason above, we approximate the ICC by treating the components which receive intents and include at least a sink (i.e., the components which are likely to send out the sensitive data it has received from ICC) to be relevant.

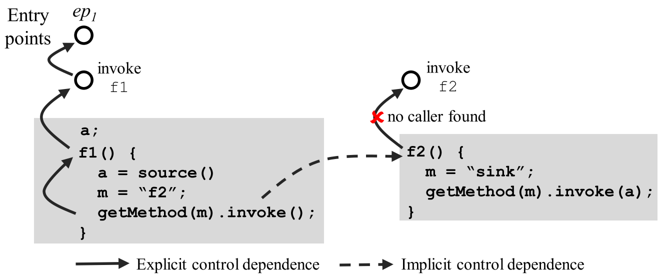
We extract the components and entry points from the slices and only execute them in the dynamic exploration step. Taking the running example for instance, after static reduction, the paths starting from `onStop` and the state space of `DummyAct` can be excluded.

5 Dynamic State Space Exploration

After static reduction, DroidPF obtains the app slices (in terms of relevant components and events) to be explored dynamically so as to check whether a property violation can indeed occur. In this section, we introduce the platform for dynamic exploration and property checking in DroidPF.

5.1 Overview of Dynamic Exploration

Fig. 10 shows the overall design of dynamic exploration. We use JPF as our exploration engine. It explores an app's state



```
a;
f1() {
  a = source()
  m = “f2”;
  getMethod(m).invoke();
}
```

```
f2() {
  m = “sink”;
  getMethod(m).invoke(a);
}
```

Fig. 9. An example of two-layer reflection.

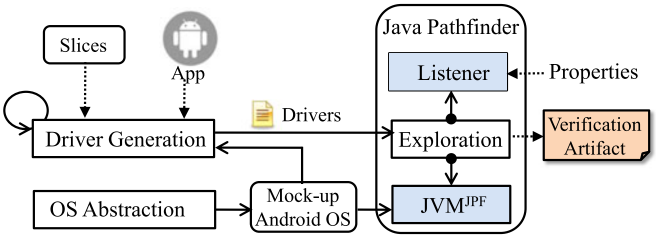


Fig. 10. Overview of dynamic exploration.

space by dynamically executing possible paths that are activated by *choices*. At a program point where two or more choices become possible, e.g., scheduling points, human interaction and random system events, JPF records current *state* and executes possible choices individually; after completing one choice, it *backtracks* to the previous point and restores the state. During the exploration, JPF calls back analyst-defined *listeners* which can be used to check whether a property is violated given a particular state of $\mathfrak{L}_{app}$. The details are presented in Section 5.2.

As discussed before, the execution of Android apps heavily relies on the underlying OS, which poses a challenge for DroidPF. Our solution is to provide an abstract and executable mock-up Android OS in Java to support the dynamic execution of the apps. We present the details on the OS abstraction in Section 5.3.

The exploration strategy in JPF only handles programs with single entry point, but an app usually contains multiple event handlers as entry points. The solution of DroidPF is to generated drivers to simulate Android's event-driven execution mechanism by directly invoking the event handlers in a scheduled order. We leave the detail of driver generation to Section 6 and in the remaining of this section, we focus on our platform for supporting dynamic exploration and property check.

5.2 Security Property Checking in JPF

During exploration conducted by JPF, a notification mechanism is provided to enable dynamic observation and interaction with the *listeners*. This notification mechanism is based on callbacks, which enables us to implement the property checkers for privilege and privacy properties.

As discussed in Section 3.3, the privilege and privacy properties can be encoded as *reachability* analysis. A privilege property is violated if a state where an app conducts an over-privileged behavior is reachable, such as sending SMS messages and accessing camera/microphones. DroidPF listens on the `methodEntered` callback to check whether the over-privileged APIs are invoked.

Although privacy property checking can be reduced to readability analysis, it is slightly more complicated because DroidPF needs to track the *tainted* variables. DroidPF uses the standard dynamic taint analysis [30], [39] for privacy property checking. It tracks the flow of the sensitive information within the app, and raises an alert if the information is exfiltrated. The information tracker consists of three aspects. First, DroidPF listens on the `methodExited` callback to detect the invocations to the sources and then labels the return values of those calls as *tainted*. Second, it listens on the `instructionExecuted` callback to track the taint flow. Third, it listens on the `methodEntered` to check whether the parameters of the invocations to the sinks contain *tainted* values.

5.3 Mocking Up Android OS

In this section, we tackle the challenge of executing Android apps in a traditional JVM, which is necessary for JPF. A straightforward way might be to use an Android emulator as a *stub* and interact with DroidPF through JPF's *native peer* mechanism. However, DroidPF would not be able to track the execution in the stub since from its perspective, the stub is a black box. As a result, the logic in the OS which might be relevant to the checked property would be lost. One such example, in the content provider (`CProvider`) of our running example, is the sensitive data flow through the database file. Without tracking the behaviors of the OS, the data flow would become untraceable for DroidPF.

DroidPF's solution is to develop a set of mock-ups of OS functionalities in Java. The main challenge of OS abstraction is to balance scalability and correctness. For instance, while it might be feasible to mock-up native code in the Android OS, it may not be easy to ensure that the mock-up is correct, due to lack of specification of the native code. Rather, we choose to develop the mock-up at such a level that the semantics relevant to the property are easy to understand. For instance, instead of mocking up only the native code in the database component in Android (which contains both naive code and Java code), we mock up the entire database component instead (by implementing a HashMap essentially). In addition, instead of building the mock-up OS from scratch, we reuse the source code of the Android OS as much as possible.

Table 1 lists the main modules of the Android OS that we have mocked up. DroidPF intercepts the invocations to the native code and mocks-up their expected behaviors in Java. Some mock-ups are straightforward. For example, the resource managers (e.g., location manager and SMS manager), can be substituted with dummy ones which simply return faked values without invoking the real managers in native code. Some are rather complicated, including the GUI and the I/O management. We have thus devoted most of our efforts to those.

GUI. GUI plays a crucial role in Android apps. However, it relies on enormous native code to render images and handle the user's interaction, which makes it difficult, if not impossible, to implement the entire GUI system in JVM. In addition, it cannot be simply ignored since it determines the control flow of the apps.

GUI events include interaction events (e.g., clicking a button) and data inputs (for example, typing texts through a `TextEdit`). DroidPF mocks them up in different ways.

- *Interaction events.* For an action event, DroidPF directly invokes the event handler from the drivers.
- *Data inputs.* For the data inputs, DroidPF relies on the analyst to configure the range of the input values with a *choice generator*. Since most of the GUI events are related to the actions in the mobile application domain, we consider this approach practical.

  Other approaches such as symbolic execution can be used to generate data input in our future work. We acknowledge that deciding the data input is a

TABLE 1
Main Modules Modified in DroidPF

Types	Modules
Components	android.app.(Activity|Service|ContextImpl|Dialog), android.content.(BroadcastReceiver|ContextWrapper), android.content.pm.PackageManager, android.os.(AsyncTask|Bundle|Looper), android.content.res.AssetManager
GUI	android.widget.(TextView|Button|CheckBox|EditText|ImageView|LinearView|Toast|LinearLayout), android.opengl.GLSurfaceView, android.view.(View|Surface|WindowManagerImpl|MenuItem|Menu), android.graphics.Bitmap, android.webkit.WebView
ICC	android.content.Context.(startActivity|sendBroadcast|registerReceiver|unregisterReceiver|startService |stopService|bindService|unbindService), android.content.Intent, android.os.CountDownTimer
Resource Manager	android.location.LocationManager, android.net.ConnectivityManager, android.media.AudioManager android.telephony.(TelephonyManager|SmsManager), android.hardware.SensorManager
I/O	java.io.*, android.database.sqlite.*, android.content.SharedPreferences

complicated problem. Currently, DroidPF can facilitate in figuring out the data inputs that activate different branches, by offering a callback whenever a branch condition is evaluated. In Section 7.2, we show how this approach is useful for identifying commands that activate hidden behaviors of malware.

I/O Management. As shown in the running example, the sensitive information can flow through the database and files, which may become untraceable. To address this challenge, we simulate the database and files I/O with in-memory data structures. When the app performs a `write` operation to store a data item into an external file, the item is written into an in-memory buffer (under control of DroidPF). Similarly, file-based SQL databases are simulated using in-memory tables. With the interception and proper implementation of a set of I/O APIs, such as `read`, `cursor`, `uri` and `SQLiteOpenHelper`, the mock-up I/O is completely transparent to the apps.

Limitation. Mocking up the Android OS indeed requires significant engineering effort, as we have experienced. The current basic mock-up costs six months effort of the first three authors. The same approach is used by other tools that require a mock-up of their environment, such as KLEE [24] and S2E [25]. Having said that, mocking up the whole OS before analyzing any app is perhaps not smart. Thus, in our work, besides mocking up commonly used components like GUI and I/O management, we always start with static app reduction and then focus on mocking up only the relevant components. As has been learned from our experience with the tested apps in this work, it takes an analyst who has basic knowledge on Android OS a few days to adapt the mock-up for a given intermediate-scale app, and a significant portion is spent on modelling the objects returned from the mock-up OS because if the mock-up does not match the app's requirement (e.g., *non-null*), the app would simply crashes.

6 Driver Generation

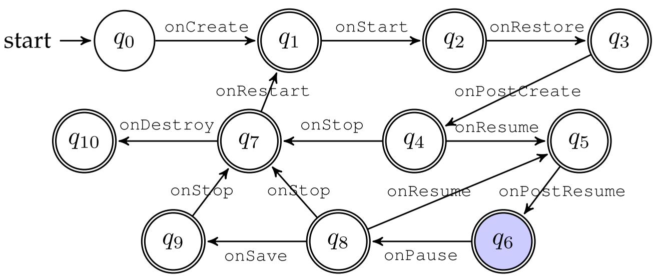
Unlike ordinary Java programs, Android apps are driven by user interaction (e.g., through GUI) or environmental input (e.g., calls for service from other apps). The goal of driver generation is to construct a driver program which allows JPF to systematically explore the state space of the app. Meanwhile, for the sake of precision, DroidPF must be designed to generate only feasible event sequences relevant to the app.

6.1 Dependency-Constrained Event Permutation

One challenge for the driver generation is that events can occur asynchronously. Enumerating all permutations of the events would be complete but would at the same time result in many infeasible sequences. Therefore, we introduce a dependency-constrained event permutation, which exploits the dependency relations among the events so as to reduce the event sequences.

We use deterministic finite automata (DFA) to specify the dependency relations among the events. An event sequence is valid iff it is accepted by the DFA. A DFA is a 5-tuple: $(Q, \Sigma, \Delta, q_0, S_a)$, where Q is a finite set of states; Σ is a finite set of events (alphabet); Δ is a transition relation; q_0 is the start state and S_a is the set of accepting states. We start with defining a *full* lifecycle DFA (L-DFA) for each of the four types of app components, based on the official documentations of Android [40]. The alphabet of the full L-DFA includes all of the lifecycle event handlers. As an example, Fig. 11 shows the full L-DFA of the activity component.

Often, not all of the lifecycle events are relevant to the checked properties. Therefore, at the second step, DroidPF projects the full L-DFA with respect to the app's alphabet using Algorithm 1. We call the obtained DFA a concrete L-DFA. This algorithm works as follows. For each transition $q_i \xrightarrow{e} q_j$ in the full L-DFA, if e is in the app's alphabet, this transition is preserved in the concrete L-DFA. Otherwise,



$q0$: start state; $q1$: activity is launched; $q2$: activity is restoring saved state; $q3$: saved state is restored; $q4$: activity is becoming visible; $q5$: activity is resuming; $q6$: activity is active in the foreground; $q7$: activity is finishing; $q8$: activity is not visible; $q9$: system is saving activity state; $q10$: activity is being destroyed by the system. `onSave` stands for `onSaveInstanceState` and `onRestore` for `onRestoreInstanceState`

Fig. 11. Full L-DFA of activity component.

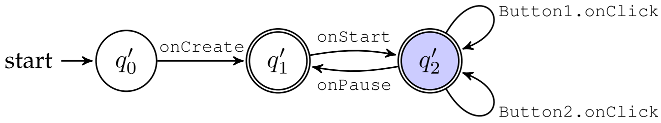


Fig. 12. DFA of the `MotivatingAct` in Fig. 2.

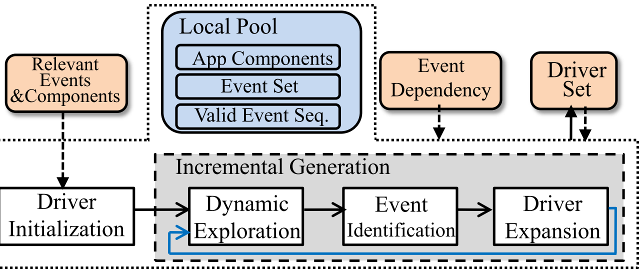


Fig. 13. Driver generation process.

the algorithm first combine q_j into q_i (line 3); then transitions starting from q_j are changed to start from q_i (line 6-7), and transitions destinating to q_j are changed to q_i (line 8-9).

Algorithm 1. L-DFA Projection Algorithm

```
   Input: $dfa$—full L-DFA, $EvntSet$—lifecycle event set
   Output: $cdfa$—concrete L-DFA
 1 foreach $t = q_i \xrightarrow{e} q_j$ in $dfa.\Delta$ do
 2   if $e \notin EvntSet$ then
 3     $q_i \leftarrow combine(q_i, q_j)$;
 4     $dfa.Q \leftarrow dfa.Q - q_j$;
 5     $dfa.\Delta \leftarrow dfa.\Delta - t$;
 6     foreach $t' = q_j \xrightarrow{e'} q_k$ do
 7       $dfa.\Delta \leftarrow dfa.\Delta \cup \{q_i \xrightarrow{e'} q_k\}$;
 8     foreach $t' = q_k \xrightarrow{e'} q_j$ do
 9       $dfa.\Delta \leftarrow dfa.\Delta \cup \{q_k \xrightarrow{e'} q_i\}$;
10 $dfa.\Sigma \leftarrow EventSet$;
11 return $dfa$
```

After obtaining the concrete L-DFA, DroidPF extends it to incorporate those relevant GUI and ICC events identified in static reduction. Our key insight is an invariant that an activity only handles GUI events and ICC events when it is in *active* state (q_6 in Fig. 11). Therefore, we add the permutations of these events to the active state (availability of a GUI item is checked before invoking its handlers, which ensures that it is not disabled). As an example, Fig. 12 shows the final DFA specifying the dependency relations of events in `MotivatingAct`.

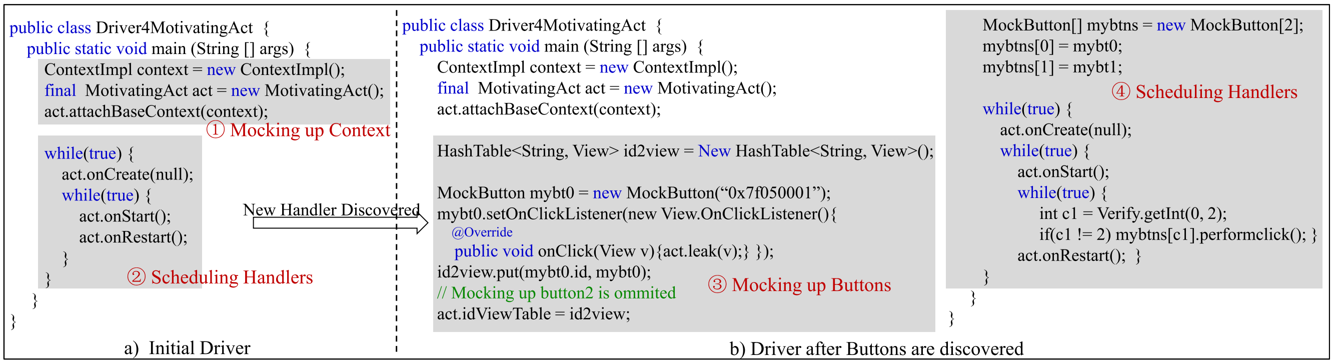
Based on the final DFA, DroidPF generates legitimate event sequences. We remark even if the number of the sequences is infinite due to the loops in the DFA (e.g., $q'_1 q'_2 q'_1$ in Fig. 12), the state exploration by JPF would terminate as long as there are finitely-many states only (not event sequences).

6.2 Incremental Driver Generation

Fig. 13 shows an overview of the driver generation process. Given the relevant components and events, DroidPF creates an initial driver which only includes a set of relevant lifecycle event sequences of the app's main activity component (i.e., the first component invoked when the user launches the app). By dynamically executing the created driver, DroidPF is able to reach more relevant components and events. When a relevant event is reached, DroidPF constructs new event sequences which are accepted by the DFA, and generates new drivers to drive the next-round execution. When a relevant component is reached, DroidPF includes its lifecycle event sequences. By iteratively executing each of the drivers, DroidPF gradually expands the driver set until no more components and events can be reached, i.e., a fixed point is reached.

We take our running example (Fig. 2) to illustrate the generation process. Initially, DroidPF generates a driver which only includes the three lifecycle events of `MotivatingAct` (Fig. 14a): `onCreate`, `onStart` and `onPause` . It then dynamically executes the driver. When executing the `setContentView` (line 7 in Fig. 2), it reaches two relevant buttons which are registered in the layout `xml` file. It thus includes the `onClick` events of the buttons into the new driver (Fig. 14b). Later, it reaches the content provider `CProvider` which is dynamically registered in line 8. It then includes the lifecycle events of `CProvider` and generates another driver which contains the present components and events. We remark that when executing line 29, DroidPF parses the intent and identifies the invoked component as `DummyAct`. Because `DummyAct` has been found irrelevant in the static analysis step, DroidPF does not include it into the driver.

Algorithm. Algorithm 2 details our driver generation algorithm. The inputs of the algorithm include the app, the relevant components/events and a DFA. The output of the algorithm is a set of drivers. The algorithm consists of two steps: driver initialization (line 1-3) and expansion process (line 4-10). The *drvInit* method (line 1) first identifies the main component by parsing the app's



```
public class Driver4MotivatingAct  {
   public static void main (String [] args)  {
      ContextImpl context = new ContextImpl();
      final  MotivatingAct act = new MotivatingAct();
      act.attachBaseContext(context);

      while(true) {
         act.onCreate(null);
         while(true) {
            act.onStart();
            act.onRestart();
         }
      }
   }
}
```

① Mocking up Context
② Scheduling Handlers
New Handler Discovered

a) Initial Driver

```
public class Driver4MotivatingAct  {
   public static void main (String [] args)  {
      ContextImpl context = new ContextImpl();
      final  MotivatingAct act = new MotivatingAct();
      act.attachBaseContext(context);

      HashTable<String, View> id2view = New HashTable<String, View>();

      MockButton mybt0 = new MockButton(“0x7f050001”);
      mybt0.setOnClickListener(new View.OnClickListener(){
        @Override
         public void onClick(View v){act.leak(v);} });
      id2view.put(mybt0.id, mybt0);
      // Mocking up button2 is ommited
      act.idViewTable = id2view;
      MockButton[] mybtns = new MockButton[2];
      mybtns[0] = mybt0;
      mybtns[1] = mybt1;

      while(true) {
         act.onCreate(null);
         while(true) {
            act.onStart();
            while(true) {
               int c1 = Verify.getInt(0, 2);
               if(c1 != 2) mybtns[c1].performclick(); }
            act.onRestart();  }
      }
   }
}
```

b) Driver after Buttons are discovered

Fig. 14. Driver generation for `MotivatingAct` in Fig. 2.

TABLE 2
Events Supported by DroidPF

	Class Name	Handlers
lifecycle	android.app.Activity	onCreate, onDestroy, onPause, onStop, onPostResume, onRestart, onResume, onPostCreate, onRestoreInstanceState, onSaveInstanceState, onStart
	android.app.Service	onBind, onCreate, onStart, onDestroy
	android.app.ContentProvider	onCreate
	android.app.Application	onCreate, onTerminate, ActivityLifecycleCallbacks
	android.preference.PreferenceActivity	onCreate, onDestroy, onStop, onPreferenceTreeClick
GUI	android.app.Activity	onCreateOptionsMenu, onPrepareOptionsMenu
	android.widget.EditText	getText, setText
	android.widget.Button	performClick
	android.view.View.onClickListener	onClick
	android.view.View.onTouchListener	onTouch
	android.preference.Preference.OnPreferenceChangeListener	onPreferenceChange
	android.preference.Preference.OnPreferenceClickListener	onPreferenceClick
	android.view.GestureDetector.OnGestureListener	onDown, onFling, onLongPress, onSingleTapUp, onScroll
ICC	android.app.Activity|Service	onLowMemory
	android.content.BroadcastReceiver	onReceive
	android.app.Application	onConfigurationChanged, onLowMemory, onTrimMemory
	android.location.LocationListener	onLocationChanged, onProviderEnabled, onProviderDisabled

`AndroidManifest.xml` file. After obtaining the main component, DroidPF searches its lifecycle event handlers from the relevant event set. The identified events are taken as inputs to the *drvGen* which permutates the events and generates a set of event sequences. Each of the sequences (denoted by *seq*) is then used to generate a driver, which invokes the event handlers in the order of *seq*.

Algorithm 2. Driver Generation Algorithm

```
  Input: $App$—app, $RCmpSet$—relevant components,
         $REvntSet$—relevant events, $ED$—dependency DFA
  Output: A driver set
1  $EvntSet \leftarrow drvInit(App, RCmpSet, REvntSet, ED)$
2  $CmpSet \leftarrow EMPTY$
3  $DrvSet \leftarrow drvGen(EvntSet, CmpSet, ED)$
4  $DrvSet_{old} \leftarrow EMPTY$
5  while $DrvSet \neq DrvSet_{old}$ do
6      $DrvSet_{old} \leftarrow DrvSet$
7      foreach $drv$ in $DrvSet$ do
8          $(EvntSet, CmpSet) \leftarrow dynExplore(drv)$
9      $(EvntSet, CmpSet) \leftarrow filter(EvntSet, CmpSet, RCmpSet, REvntSet)$
10     $DrvSet \leftarrow drvGen(EvntSet, CmpSet, ED)$
11 return $DrvSet$
```

In the expansion step (line 4-10), DroidPF expands the initial driver set. It starts from a dynamic exploration (line 7-8). The *dnyExplore* method executes each of the drivers in the driver set *DrvSet*. During the execution, it identifies dynamically-registered components and GUI elements.

- *Components.*DroidPF considers two types of registration: *instantiation* and *invocation by ICC*. First, once a component is instantiated dynamically, such as `CProvider` in our running example (line 7, Fig. 2), DroidPF adds it into *CmpSet*. Second, when the program invokes a new component using ICC, such as `startActivity()` and `startService()`, the invoked component is included.
- *GUI Elements.*For each GUI element that is registered dynamically, DroidPF adds its event handlers into *EvntSet*.

After identifying new components and events, DroidPF selects those relevant ones for driver generation (line 9-10).

7 Implementation and Evaluation

DroidPF has been implemented with approximately 20 K lines of Java code, in addition to various libraries that we employ. Most of our engineering efforts on implementing DroidPF are spent on driver generation and mocking up the methods in Android OS. The static app reduction is implemented based on SAAF [36]. We use apktool [41] to translate the DEX code of apps into *smali* format, on which the static slicing is performed. The events that are taken into consideration by driver generation are listed in Table 2. The mock-up Android OS is developed based on the Framework of Android 4.0. We have to decode DEX code (i.e., *.dex* file format) of apps into Java bytecode (i.e., *.class* file format) compatible with JPF in order to execute the code. Currently we rely on dex2jar [42] (which has 99 percent retargeting success rate) and Dare [43] (which has 99.99 percent retargeting success rate) for this purpose.

In the following, we evaluate DroidPF in terms of its effectiveness and accuracy. In particular, we investigate the following four research questions.

- *RQ1* Effectiveness: can DroidPF detect security and privacy property violations in real-world apps, or show their absence?
- *RQ2* Precision: can DroidPF achieve a correct and precise analysis on the benchmark apps?
- *RQ3* Given that DroidPF has the mock-up OS to enable the execution of Android apps on JPF's JVM, is it possible to support JPF's diverse set of property checkers for non-security properties?

Our evaluation subjects include the following three sets. The first set consists of eleven small to intermediate scale real-world apps downloaded from Google Play, Anzhi market and F-Droid, including an ebook app (denoted by

TABLE 3
Statistics in Our Experiments

Subjects		Static Reduction			Dynamic Exploration			
App name	LOC	#Components	#Events	Time(S)	Exploration Time(S)	Memory (MBytes)	#Backtracked States	Violation/Bug detected?
Ebook (main app)	7.0 K	2/2	7/9	10.6	2 (4)	188 (321)	1,428 (3,900)	✘
Ebook Youmi (main view)	15.6 K	1/1	1/5	10.6	2 (13)	185 (590)	73 (5,257)	✓(UR)
Calculator (main Activity)	6.5 K	3/4	3/26	3.4	1 (AC)	119 (AC)	8 (AC)	✓(UR)
GPSShare	1.2 K	1/3	8/17	2.4	3 (172)	366 (409)	2,792 (49,556)	✘
A2DP Volume	24.5 K	6/14	23/113	16.8	10 (-)	382 (-)	674 (-)	✓(UR)
GPS optimisation	854	2/2	7/7	1.0	1 (-)	174 (-)	352 (-)	✘
GPS Booster	1.0 K	3/3	7/9	1.0	524 (-)	943 (-)	224,671 (-)	✘
MYPosition	3.1 K	2/3	9/12	6.8	4 (-)	422 (-)	174 (-)	✓(UR)
GPS Compass	7.2 K	1/1	7/7	46.8	121 (-)	3,237 (-)	160,690 (-)	✓(UR)
cn.waps.AppOffer	10.9 K	1/1	5/5	6.3	1(-)	160(-)	305(-)	✓(UR)
TinyClock	1.0 K	3/3	4/4	3.6	1 (-)	72 (-)	4 (-)	✘
SMS Spammer	340	1/1	3/3	5.9	2 (-)	72 (-)	32 (-)	✓(UR)
Battery level	8.2 K	1/1	8/8	6.3	235 (-)	565 (-)	18,521 (-)	✘
InsecureBank†	2.3 K	4/5	11/12	3.0	183 (184)	596 (596)	132,307 (133,809)	✓
ZitMo	576	1/3	3/4	2.3	6 (6)	111 (117)	393 (394)	✓
Geinimi	13.0 K	4/6	6/21	5.9	OM (OM)	OM (OM)	2.5 K (OM)	✓
Spitmo	704	1/1	1/1	2.6	4 (4)	78 (78)	66 (66)	✓
Zsone‡	29.4 K	2/4	3/10	30.5	3 (AC)	61 (AC)	113 (AC)	✓
Obad	80.5 K	10/11	23/28	9.6	1 (-)	AC (-)	AC (-)	✘
Pincer	4.3 K	11/12	12/15	4.0	29 (-)	959 (-)	1,466 (-)	✓
PrivateDataLeak1	211	1/1	3/3	8.6	2 (2)	78 (78)	38 (38)	✓
Button2	201	1/1	4/4	9.0	2 (2)	78 (78)	41 (41)	✓
AnonymousClass1	157	1/1	3/6	8.6	2 (2)	78 (78)	20 (177)	✓
LocationLeak2	171	1/1	3/6	8.7	2 (2)	78 (78)	20 (177)	✓

LOC: lines of code in smali*, including third-party libraries; #Components: number of relevant/overall components; #Events: number of relevant/overall events; OM: out of memory; AC: app crashed; UR: previously unreported violations/bugs. Those numbers in the brackets stand for the statistics in the exploration without static reduction.*
† When fed with infinite event sequence, the exploration of InsecureBank did not terminate. The statistics in this row was obtained by setting the length of sequence as 16.
‡ In the experiment of full permutations, Zsone crashed due to a null reference when its `onResume()` *is called.*

ebook), a scientific calculator app and a set of apps which requests for the location or SMS permission, such as a location sharing app named GPSShare [44] which allows users to share locations through SMS. These apps utilize a variety of core functionalities supported by Android, such as accessing locations, accessing the IMEI and sending SMS messages. We select the ebook app and the calculator app because we find that they request privileges that are not necessary for their functionality; for example, the ebook app requests the permissions of `INTERNET` and `READ_PHONE_STATE`. We also select the apps requesting the location and SMS permission because we are interested in whether the user's private data are leaked by them. These apps are used to investigate whether DroidPF can identify malicious behaviors which abuse these permissions. During the experiments, we find that some of these apps embed third-party ad libraries, we thus also test two ad libraries embedded by them.

The second set consists of malware samples. We test four known malware samples released by Malware Genome Project [3] in 2013, namely ZitMo, Geinimi, Spitmo and Zsone, and two relatively new samples obtained from Contagio mobile, namely Obad and Pincer. These samples violate the privilege property by sending premium SMS messages (Geinimi, Zsone, Obad and Pincer) and blocking incoming SMS messages (ZitMo, Spitmo and Pincer), and privacy property by stealing the incoming SMS messages (ZitMo and Spitmo) and IMEI (ZitMo and Pincer). Our subject set also includes a vulnerable open source app called InsecureBank [45], which is embedded with harmful API calls and various behaviors of leaking information.

The third set of apps is a comprehensive benchmark called DroidBench [10], which has been created to evaluate information flow analysis. It includes 111 open source apps, some of which lead to violations of privacy properties. It contains a suite of challenges for analysis tools to check both false negatives and false positives, such as locations in arrays and lists, callbacks, field and object sensitivity, ICC, obfuscation, reflection and implicit flows.

7.1 Effectiveness of DroidPF

In our experiments, we check both privilege and privacy properties. For privilege properties, we focus on the sensitive behaviors that are not initiated by the GUI events (i.e., stealthy behaviors), such as blocking incoming SMS messages and sending SMS messages. For privacy properties, we check whether the sensitive data, including device ID, location and contacts are leaked through the network, SMS and logs. Table 3 lists the statistics of our experiments (for the sake of brevity, we omit most statistics of the DroidBench experiments). Our experiments were conducted on a PC with Intel Core 2 DUO CPU E6550 at 2.33 GHz and 4 GB RAM. The verification results using DroidPF (i.e., the counterexamples or the correctness claim) have been confirmed by manually analyzing the `smali` code.

```java
public class CCOIoll extends Activity {
  private static final byte[] oCIlCll = { -12, 17, -47,
   46, 0, 5, 1, 1, -7, 11, 8, -45, 43, -11, 3, 12, -5,
   1, -15, 20, 17, 2, -9, 7, -5, 15, -8, 16, -1, -4, -3,
   …,
   11, 8, -68, 68, -1, -61, 36, 19, 4, 10, -8, 8, 0, -22,
   22, 15, -11, 8, 0, 15 };

  private static String oCIlCll(int paramInt1,
         int paramInt2, int paramInt3) {
    byte[] arrayOfByte1 = oCIlCll;
    int i = paramInt3 + 33;
    int j = paramInt2 + 96;
    byte[] arrayOfByte2 = new byte[i];
    int k = 0;
    int m;
    if (arrayOfByte1 == null) {
      m = i;
    }
    for (int n = paramInt1;; n = arrayOfByte1[paramInt1]){
      paramInt1++;
      j = -2 + (m + n);
      arrayOfByte2[k] = ((byte)j);
      k++;
      if (k >= i) {
        return new String(arrayOfByte2, 0);
      }
      m = j;
    }
  }
  protected void onCreate(Bundle b) {
    …
    String model =
     Class.forName(oCIlCll(64, 1, -17)/*android.os.Build*/)
          .getField(oCIlCll(126, -19, -28)/*MODEL*/);
    …
  }
```

Fig. 15. A code snippet extracted from the main activity of Obad. It uses reflection to obtain the Android OS model.

Real-World Apps. As shown in last column of Table 3, DroidPF successfully verifies that six of the real-world app samples are free of property violation, and detects violations or bugs from the remaining five. DroidPF detects in GPSShare that the location information is taken as a parameter to invoke Andoid's SMS activity, we do not regard it as a "sink" since SMS activity prompts the user and will not send out the SMS messages without the user's consent. DroidPF detects that a library in the calculator app connects the server located at `http://58.221.57.115:81` to download app packages. Location leakage is detected from both A2DP Volume (through publicly accessible files) and MYPosition (through log). SMS Spammer is detected to send SMS messages at background. GPS Compass is reported to have a deadlock in the app.

From the ad library GPS Compass uses (named AppOffer), DroidPF reports the leakage of IMSI and device information to a server located at `http://app.wapx.cn/action/notify/click`. Similar leakage is also found in the ebook app, in which DroidPF reports leakage of the device ID. Our investigation reveals that its main components do not leak the device ID. Instead, the leakage occurs in an embedded advertising SDK named Youmi. These two experiments show that third-party libraries may overprivilege the benign apps. Studies of isolating suspicious libraries from the main apps have been conducted, such as AdSplit [46], AdDroid [7] and DroidVault [47].

Malware/Vulnerable Samples. For the malware samples, DroidPF identifies data leakage from four of them.

- Zsone. DroidPF detects a trace which is initiated by the `onCreate` lifecycle event of its main activity and leads to the invocation of `sendTextMessage`, which sends seemingly meaningless SMS messages (e.g., `aAHD`) to four numbers (e.g., `10626213`). After further investigation of the phone numbers, we find that those messages are used to register premium services from the Chinese mobile networks.
- Spitmo and ZitMo. DroidPF identifies that one of Spitmo's broadcast receivers (i.e., `SMSReceiver`) blocks incoming SMS messages by invoking `abortBroadcast`. It also forwards the messages to a phone number stored in the file `asset/settings.xml`. In ZitMo, DroidPF detects a similar trace.
- Geinimi. DroidPF does not terminate on Geinimi and we stop the exploration when the machine is out of memory. After our manual investigation, we find that Geinimi blocks on a loop of sending encrypted string which contains sensitive data like IMEI and location. The loop may cause JPF to create new states, which leads to the out-of-memory exception.
- Pincer. DroidPF detects that Pincer sends the device id and the phone number to its C&C server located at `198.211.118.115:9081/Xq0jzoPa/g_L8jNgO.php` and a phone number `+447937281444` once it is started. It waits for the commands to conduct other behaviors. We use DroidPF to assist in recovering these commands, and we report this in Section 7.2.
- Obad. In order to obstruct static analysis, Obad uses a reflection-based obfuscation technique. In particular, it substitutes its API calls with reflection calls. At run time, the names of the APIs are calculated in an intentionally complicated way. Fig. 15 demonstrates how it uses this method to obtain the Android OS model. DroidPF successfully identifies the obfuscated API calls, although the tested malware sample crashes due to incompatibility with Java 7's class verifier.
- InsecureBank. From the InsecureBank, DroidPF identifies the leakage of sensitive data (e.g., phone number and input data) through the channels of HTTP, system logs and SD card.

To investigate the effect of the static reduction, we test the main components of some apps without applying the reduction (i.e., the numbers are in brackets in Table 3). As shown in the table, when we take the full permutations of the events as input to the apps, the time and space efficiencies decrease significantly. Nonetheless, the full exploration may detect issues irrelevant to the checked properties. In the experiment of the calculator app, DroidPF detected a bug. The app does not validate the input before parsing a value of type double, and this causes the crash of the app when DroidPF clicks button "=" after clicking button ".".

7.2 Applying DroidPF on Malware Behavior Analysis

In Pincer case, the malware sample waits for commands from the C&C server to activate its hidden behaviors. These behaviors cannot be observed currently by DroidPF if the C&C server is no longer alive. To explore these behaviors, the analyst has to feed the commands from the mock up OS, yet the challenge is to figure out those commands. We have made an attempt to apply DroidPF to identify these commands. Our idea is to borrow techniques from symbolic execution. In particular, DroidPF monitors the branch statements and prompt the branch condition whenever a branch statement is executed. The analyst then can manually examine the relation between the condition and data input. In this way, the analyst can gradually figure out the commands.

TABLE 4
Results on DroidBench

App Name	Result	App Name	Result	App Name	Result	App Name	Result
Arrays, Lists and HashMaps		Callbacks		Field and Object Sensitivity		General Java	
ArrayAccess1		AnonymousClass1	✓	FieldSensitivity1&2&4	☾	Loop1&2	✓
ArrayAccess2		Button1-5	✓	FieldSensitivity3	✓	SourceCodeSpecific1	✓
ListAccess1		LocationLeak1-3	✓	InheritedObjects1	✓	StaticInitialization1-3	✓
HashMapAccess1		MethodOverride1	✓	ObjectSensitivity1&2		Exceptions1&2&4	✓
ArrayCopy1	✓	MultiHandlers1		Lifecycle		Exceptions3	✓
ArrayToString1	✓☾	Unregister1		BroadcastReceiverLifecycle1&2	✓	UnreachableCode	
MultidimensionalArray1	✓	RegisterGlobal1&2	✓	ActivityLifecycle1-4	✓	FactoryMethods1	✓
Inter-Component Communication		Inter-App Communication		ServiceLifecycle1&2	✓	Serialization1	✓
IntentSource1	✓	Echoer	✓	ApplicationLifecycle1-3	✓	StringFormatter1	●
IntentSink1&2	✓	SendSMS	✓	Ordering1		StringToOutputStream1	✓
ActivityCommunication1-8	✓	StartActivityForResult1	✓	FragmentLifecycle1&2	●	VirtualDispatch1	✓↻
BroadcastTaintAndLeak1	✓	Miscellaneous Android-Specific		SharedPreferenceChanged1	✓	VirtualDispatch2	✓
EventOrdering1	✓	PrivateDataLeak1-3	✓	AsynchronousEventOrdering	✓	VirtualDispatch3&4	●
ServiceCommunication1	✓	DirectLeak1	✓	EventOrdering1		StringPatternMatching1	○
ComponentNotInManifest1		InactiveActivity		Threading		StartProcessWithSecret1	✓
Singletons1	✓	LogNoLeak		AsyncTask1	✓	Implicit Flow	
SharedPreferences1	✓	Library1&2	✓	Executor1	✓	ImplicitFlow1-4	○
UnresolvableIntent1	✓	PublicAPIField1&2	✓	JavaThread1	✓	Aliasing	
Emulator Detection		ApplicationModeling1	✓	JavaThread2	✓	Merge1	●
ContentProvider1	●	Parcel1	○	Looper1	✓		
IMEI1	✓	Obfuscation		Reflection			
PlayStore1	●	Obfuscation1	✓	Reflection1-4	✓		

✓= correct alarm, ✘= false alarm, ○ = missed leak, ● = incompatible, ☾ = bug, ↻ = non-termination, empty cell: no leaks expected and none reported

As a case study, we have applied this approach on analyzing Pincer. In this section, we show the usefulness of DroidPF in facilitating malware analysis by reporting part of Pincer's behaviors that we obtain with DroidPF. Pincer uses a listener to obtain the incoming SMS messages. If an incoming message contains string “"command"” or if the sender matches the phone number stored under the `numbers_to_sms_divert` item in the shared preference file `diverter_pref_key.xml`, it blocks the message using `abortBroadcast` to prevent it from being observed by other SMS receivers. Then, it extracts the concrete commands from the message. One of the commands is a ping message in the JSON format `["command":"ping", "result":"true"]`. Receiving this, the malware will send the message `["action":"pong"]` to `+447937281444` to report its liveness. Another command is in the format of `["command":"send_sms", "result":"true", "phone_number":NUMBER, "message_text":MESSAGE]`. Receiving this, the malware will send an SMS with `MESSAGE` as the body to `NUMBER` (this is likely used to conduct a DDOS attack). This case study shows that DroidPF is useful in identifying not only the concrete commands but also the precise message format (or the protocol) between the malware client and server.

In this type of analysis, DroidPF relies on the analyst to provide data input to satisfy the condition in each branch. Advanced techniques such as symbolic execution and predicate abstraction (J2BP [48]) could be applied to enhance this analysis towards fully automation (discussed in Section 8).

7.3 Experiment on DroidBench

We use the DroidBench to evaluate DroidPF in terms of precision, which is the most critical criteria for a verification tool. Table 4 summarizes the test results. Overall, DroidPF achieves precise results. The dynamic feature of DroidPF makes it capable in addressing the Android-specific event-driven execution feature. In particular, DroidPF can precisely bind event handlers, track dynamic (un)registrations and order the occurrence of events. In addition, DroidPF adopts a fine-grained taint tracking on composite data types, such that it is precise in detecting taint flowing through these data types.

During the exploration of DroidBench, the app `FieldAndObjectSensitivity_FieldSensitivity1` is crashed due to one event sequence. Our manual inspection confirms that the crash is caused by a *use-after-free* vulnerability. We have reported this vulnerability to the author of DroidBench, who acknowledged our finding and rectified it in the DroidBench 2.0. The app `ArrayToString1` crashes due to an array index out of bound exception. DroidPF fail to execute six apps (labeled as • in Table 4) due to unmocked `Fragment`. In the app `Parcel1` which serializes the tainted object, DroidPF captures the value which is sent out through SMS, but it loses the taint tag when the object is serialized using native code `nativeWrite*()`. DroidPF does not terminate on `VirtualDispatch1`, because each time when a button is clicked, a variable is changed, leading to infinite number of states.

7.4 Experiments on Non-Security Properties

Given that DroidPF enables the execution of Android apps on JPF, we have also explored whether various property checkers of JPF can be enabled. We apply DroidPF on two case studies conducted by JPF-Android, a general tool that verifies Android applications using JPF, including an app containing deadlock and a calculator app. Since this experiment does not target the detection of security errors, we directly apply driver generation and exploration without static reduction of the application code. DroidPF successfully detects the deadlock which has been reported by JPF-Android. It also detects an unreported bug in the calculator

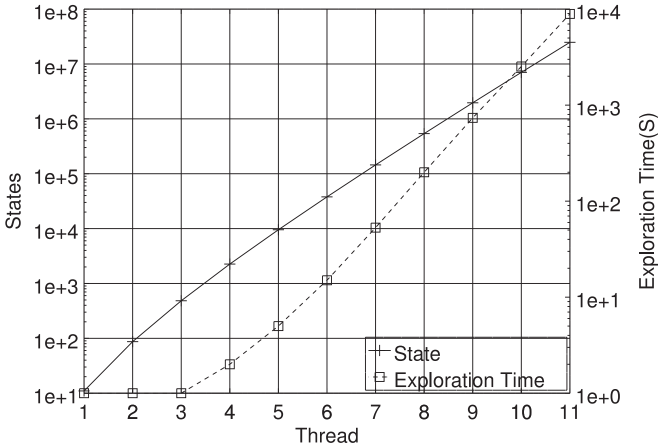


Fig. 16. Number of states and exploration time grow exponentially as number of threads increases. (The statistics is collected by adjusting the number of threads in DroidBench's `Threading_JavaThread1`, without explicitly applying any reduction techniques like BDD and abstraction which JPF may support.)

leading to an app crash. The bug is caused by an unchecked array index, which occurs when the input box is empty and DroidPF "presses" the `backspace` button.

8 Limitations and Discussion

We acknowledge that verifying Android apps is an extremely challenging task. DroidPF is only a step towards that goal and it has its limitations. Alleviating them will be important future work for eventually ubiquitous app verification.

Limitation in Applying Model Checking. DroidPF is based on model checking and thus it has the limitation of model checking, i.e., the app (after static reduction) must have only finitely-many states and the number of states is not beyond the capability of current model checking techniques. Since DroidPF leaves the task of state exploration and state comparison (to avoid exploring the same state more than once) to the underlying JPF, DroidPF can verify an app only if the app, after reduction performed in DroidPF, can be verified by JPF.

Mock-Up OS. Mocking up the underlying OS takes significant effort in our approach. The security analyst also has to have background of Android OS in order to incrementally develop the mock-up to support his/her apps. This is the major challenge that hinders DroidPF from verifying large-scale real-world apps and hybrid apps with native and JavaScript code, and verifying apps in a mass manner. However, it would be helpful to alleviate this challenge somewhat if the analyst becomes familiar with the app to be verified such that he/she can simplify the mock-up while preserving the semantics relevant to the app.

Implicit Information Flow. Similar to most information flow analysis [30], [39], DroidPF cannot identify the implicit leakage through control flow dependency; for example, the value of `x` is leaked implicitly in this statement: `if(x==1) y=1; else y=0;`. A straightforward solution is to propagate the taint to `y`, but this may lead to over-tainting and false-propagation problems. For future work, DroidPF can employ more advanced solution such as DTA++ [49].

Data Inputs. DroidPF is perhaps not effective if the app contains behavior which is triggered by data inputs and which cannot be pruned through static reduction, such as the attack behaviors enabled on a specific date or by the IMEI of a specific manufacturer, and our Pincer case studies which launch attacks based on received commands. In theory, we could always enumerate all possible values for primitive data inputs, but would often result in state space explosion, and for data input of type string or float, this becomes undecidable. Currently, DroidPF relies on the human interaction for deciding the possible values of the input, like what we show in the Pincer case study in Section 7.2. For future work, techniques like symbolic execution [50], [51], [52], [53], [54] can be considered to alleviate this problem.

```
int globalCounter = 0;
onClick() {
    ...
    globalCounter++;
    ...
    if (globalCounter == N) violationBehavior();
}
```

Fig. 17. An event whose occurrence always leads to a new state.

Multithreading. The state explosion is an inherent problem of explicit state model checking. As shown in Fig. 16, DroidPF becomes inscalable as the number of threads grows. Once the number of threads becomes greater than 10, the number of the states becomes the order of 10^7, and it takes hours for DroidPF to explore the state space. To cope with this problem, advanced techniques such as partial order reduction [55] and interface reduction [56] can be applied.

Event Sequences. As shown in the `VirtualDispatch1` case, DroidPF can easily end up with nontermination if there is an event which always leads to a new state once it happens. This case can be exemplified as Fig. 17. To address this problem, abstraction is required. In particular, further analysis should be added to figure out the dependency between the variables and the property-related behavior. If there is no dependency, the variables should be removed from the state.

9 Related Work

DroidPF is inspired and related to multiple groups of research.

Model Checking Implementations. Model checking techniques have been used to detect security properties from software for more than three decades. Traditional model checkers require the analysts to manually specify the analyzed systems in particular languages or logics. However, manual specification is often expensive and can be error-prone. Therefore, model checking implementation-level software has become active recently. Implementation-level model checkers [17], [56], [57], [58], [59], [60] directly work on the implementations of the software system by dynamically exploring the state space.

Model Checking Android Apps. JPF-Android [23], [61] is a general tool that verifies Android applications using JPF. JPF-Android makes use of JPF's class modeling and native method modeling features to bound the environment of the application created by the Android core libraries. To drive the execution of the application, users script nondeterministic event sequences and can also set the state of the environment. JPF-Android supports detection of deadlock, race conditions and runtime errors using the listeners provided by JPF. It does not target security errors in Android apps but allows users to specify properties in the form of *Checklists* to verify that the application executes specific event sequences [61].

Android Application Analysis. There have been several approaches using program analysis to analyze the security and privacy properties of Android apps [6], [10], [12], [51], [62], [63], [64], [65]. FlowDroid [10] is one of the most advanced approaches, which proposes a taint analysis featuring in addressing Android's ubiquitous callbacks. Pegasus [62] checks apps for the properties that can be specified in Linear Temporal Logic (LTL). Similar to Pegasus, AppIntent [51] aims to detect malicious behaviors by identifying whether the suspicious behaviors are initiated by the user. CHEX [6], SCandroid [12] and IccTA [66] are mainly devoted to inter-component flow analysis. Although program analysis is mature and has been proven powerful in detecting vulnerabilities, its precision is limited by points-to analysis [67]. There are several dynamic analysis approaches to test the apps, which is either by instrumenting the Android OS [30], [68] or based on virtualization [69], [70]. TaintDroid [30] and VetDroid [68] dynamically track the sensitive data flow through the OS and apps. HARVESTER [71] uses an *explore-after-slicing* approach to obtain the constant strings in the malware samples.

The precision of DROIDPF depends on activating the application behaviors. Its dependency-constrained event permutation approach aims to cover possible valid event sequences. There are a few studies also attempting to solve this challenge. SmartDroid [72] and ORBIT [73] combines static and dynamic analysis to trigger UI events. *SwiftHand* [74] uses an *abstract-refinement* approach to generate sequences of test inputs. AppIntent [51] and [70] use symbolic analysis to identify inputs to drive the analysis further. Dynodroid [75] uses an *observe-select-execute* approach which selects event inputs based on observed states to improve the coverage. A^3E [15] explores app components and mimics user actions based on the strategy learned from the control flow graphs.

10 CONCLUSION

We present DROIDPF, which provides a framework for verifying Android apps against security properties based on targeted software model checking. We have made efforts to address the main problems in verifying Android apps, such as multiple entry points/event-driven execution, GUI testing and path explosion. DROIDPF shows that it is feasible to model check the software implemented in high-level language like Java and running on a complicated OS. We hope DROIDPF can inspire future research that brings the cutting-edge model checking techniques from the specifications to the implementations.

ACKNOWLEDGMENTS

This research is partially support by Singapore NRF grant RGNRF1501 and South African NRF under Grant 88210. This research is also supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30) and administered by the National Cybersecurity R&D Directorate. This work was partially done when Guangdong Bai and Yongzheng Wu were with National University of Singapore and Singapore University of Technology and Design, Singapore, respectively. Guangdong Bai is the corresponding author.

REFERENCES

[1] IDC, "Smartphone OS market share, 2016 Q3," (2017). [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] AppBrain, "Number of available Android applications," (2017). [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps

[3] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

[4] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on Android)," in *Proc. 4th Int. Conf. Trust Trustworthy Comput.*, 2011, pp. 93–107.

[5] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2012, pp. 101–112.

[6] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. 19th ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.

[7] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, 2012, pp. 71–72.

[8] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[9] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy*, 2013, pp. 209–220.

[10] S. Arzt, et al., "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 259–269.

[11] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. 5th Int. Conf. Trust Trustworthy Comput.*, 2012, pp. 291–307.

[12] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," Tech. Rep. CS-TR-4991, Nov. 2009.

[13] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012.

[14] "Open source 360," (2017). [Online]. Available: https://www.openhub.net/p/android/analyses/latest/languages_summary

[15] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2013, pp. 641–660.

[16] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2014, pp. 204–217.

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[18] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2001, pp. 203–213.

[19] O. Tkachuk, M. Dwyer, and C. Pasareanu, "Automated environment generation for software model checking," in *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.*, 2003, pp. 116–127.

[20] M. Ceccarello and O. Tkachuk, "Automated generation of model classes for Java PathFinder," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[21] H. van der Merwe, O. Tkachuk, B. van der Merwe, and W. Visser, "Generation of library models for verification of Android applications," in *Proc. Java Pathfinder Workshop*, 2014, pp. 1–5.

[22] P. Mehlitz, O. Tkachuk, and M. Ujma, "JPF-AWT: Model checking GUI applications," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 584–587.

[23] H. van der Merwe, B. van der Merwe, and W. Visse, "Verifying Android applications using Java PathFinder," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, 2012.

[24] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[25] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for In-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 265–278.

[26] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM Symp. Inf. Comput. Commun. Secur.*, 2013, pp. 329–334.

[27] C. Ionescu, "Obfuscating embedded malware on android," (2012). [Online]. Available: http://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android

[28] "CVE," (2015). [Online]. Available: http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Android+

[29] M. D. Ernst, et al., "Collaborative verification of information flow for a high-assurance app store," in *Proc. 21st ACM Conf. Comput. Commun. Secur.*, 2014, pp. 1092–1104.

[30] W. Enck, et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 393–407.

[31] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[32] F. Tip, "A survey of program slicing techniques," *J. Program. Languages*, vol. 3, pp. 121–189, 1995.

[33] J. Hatcliff, M. Dwyer, and H. Zheng, "Slicing software for model construction," *Higher-Order Symbolic Comput.*, vol. 13, no. 4, pp. 315–353, 2000.

[34] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 627–638.

[35] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228.

[36] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing droids: Program slicing for smali code," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 1844–1851.

[37] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *Proc. 21st Netw. Distrib. Syst. Secur. Symp.*, 2014.

[38] D. Octeau, et al., "Effective inter-component communication mapping in Android: An essential step towards holistic security analysis," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 543–558.

[39] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 317–331.

[40] "Process lifecycle," (2015). [Online]. Available: http://developer.android.com/reference/android/app/Activity.html#ProcessLifecycle

[41] "apktool," (2015). [Online]. Available: http://code.google.com/p/android-apktool/

[42] "dex2jar," (2015). [Online]. Available: http://code.google.com/p/dex2jar/

[43] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," in *Proc. 20th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 6.

[44] "GPS share," (2015). [Online]. Available: https://play.google.com/store/apps/details?id=com.kkinder.sharelocation&hl=en

[45] Paladion, "Insecurebank," (2015). [Online]. Available: http://www.paladion.net/downloadapp.html

[46] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating smartphone advertising from applications," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, Art. no. 28.

[47] X. Li, H. Hu, G. Bai, Y. Jia, Z. Liang, and P. Saxena, "Droidvault: A trusted data vault for Android devices," in *Proc. 19th Int. Conf. Eng. Complex Comput. Syst.*, 2014, pp. 29–38.

[48] P. Parízek and O. Lhoták, "Predicate abstraction of Java programs with collections," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2012, pp. 75–94.

[49] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. 18th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[50] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. 20th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 59.

[51] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing sensitive data transmission in Android for privacy leakage detection," in *Proc. 20th ACM Conf. Comput. Commun. Secur.*, 2013, pp. 1043–1054.

[52] C. Cadar, et al., "Symbolic execution for software testing in practice: Preliminary assessment," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1066–1071.

[53] A. Baars, et al., "Symbolic search-based testing," in *Proc. Int. Conf. Automated Softw. Eng.*, 2011, pp. 53–62.

[54] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic execution of Java bytecode," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2010, pp. 179–180.

[55] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2005, pp. 110–121.

[56] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 265–278.

[57] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *ACM Trans. Comput. Syst.*, vol. 24, pp. 393–423, 2006.

[58] J. Morse, L. Cordeiro, D. Nicole, and B. Fischer, "Model checking LTL properties over ANSI-C programs with bounded traces," *Softw. Syst. Model.*, vol. 14, pp. 1–17, 2013.

[59] G. Bai, J. Hao, J. Wu, Y. Liu, Z. Liang, and A. Martin, "Trustfound: Towards a formal foundation for model checking trusted computing platforms," in *Proc. 19th Int. Symp. Formal Methods*, 2014, pp. 110–126.

[60] Q. Ye, G. Bai, K. Wang, and J. S. Dong, "Formal analysis of a single sign-on protocol implementation for Android," in *Proc. 20th Int. Conf. Eng. Complex Comput. Syst.*, 2015, pp. 90–99.

[61] H. van der Merwe, B. van der Merwe, and W. Visser, "Execution and property specifications for JPF-Android," *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[62] K. Z. Chen et al., "Contextual policy enforcement in Android applications with permission event graphs," in *Proc. 20th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2013.

[63] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1036–1046.

[64] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating android permission gaps with static and dynamic analysis," in *Proc. IEEE Global Commun. Conf.*, 2015, pp. 1–6.

[65] G. Bai, et al., "All your sessions are belong to us: Investigating authenticator leakage through backup channels on Android," in *Proc. 20th Int. Conf. Eng. Complex Comput. Syst.*, 2015, pp. 60–69.

[66] L. Li, et al., "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 280–291.

[67] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 2001, pp. 54–61.

[68] Y. Zhang, et al., "Vetting undesirable behaviors in Android apps with permission use analysis," in *Proc. 20th ACM Conf. Comput. Commun. Secur.*, 2013, pp. 611–622.

[69] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, pp. 29–29.

[70] 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.

[71] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016.

[72] C. Zheng, et al., "SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications," in *Proc. 2nd ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2012, pp. 93–104.

[73] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proc. 16th Int. Conf. Fundam. Approaches Softw. Eng.*, 2013, pp. 250–265.

[74] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2013, pp. 623–640.

[75] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. 9th Joint Meet. Found. Softw. Eng.*, 2013, pp. 224–234.

Guangdong Bai received the bachelor's and master's degrees in computing science from Peking University, China, in 2008 and 2011, respectively and the PhD degree in computing science from National University of Singapore (NUS), in 2015 and worked as a postdoctoral research fellow there. He has been a faculty member of Singapore Institute of Technology (SIT) since 2016. His research interests include mobile security, protocol verification, and formal methods on security.

Quanqi Ye is working toward the PhD degree at the University of Singapore working on applying formal method on Android applications security. His research interests include model checking, program analysis, and system security.

Yongzheng Wu received the bachelor's and PhD degrees in computer science from National University of Singapore, in 2004 and 2011, respectively. He worked as a postdoctoral research fellow with Singapore University of Technology and Design. He currently works in Huawei as a researcher. His research interests include system security and operating systems.

Heila Botha is working toward the PhD degree at Stellenbosch University, South Africa working on verification of Android applications. Her research interests include model checking, environment generation, and program analysis.

Jun Sun received the bachelor's and PhD degrees in computing science from National University of Singapore (NUS), in 2002 and 2006, respectively. He is currently an associate professor with Singapore University of Technology and Design (SUTD). In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member of SUTD since 2010. He was a visiting scholar at MIT from 2011-2012. His research interests include software engineering, formal methods, program analysis, and cyber-security. He is the co-founder of the PAT model checker.

Yang Liu received the bachelor of computing from National University of Singapore (NUS) and the PhD degree from NUS, in 2010 and continued with his post doctoral work in NUS. Since 2012, he joined Nanyang Technological University as an assistant professor. His research focuses on software engineering, formal methods and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, Process Analysis Toolkit.

Jin Song Dong received the bachelor's (hon I) and PhD degrees in computing from the University of Queensland, in 1992 and 1996, respectively. From 1995 to 1998, he was research scientist with CSIRO Australia. Since 1998, he has been in the School of Computing, National University of Singapore (NUS) where he received full professorship in 2016. He is on the editorial board of the *ACM Transactions on Software Engineering and Methodology* and the *Formal Aspects of Computing*.

Willem Visser is a professor in computer science with Stellenbosch University, South Africa. Before joining Stellenbosch in 2009, he spent 8 years at NASA Ames Research Center, where he was one of the research leads for the Java PathFinder project. His research interests include model checking, testing, symbolic execution and model counting. He has been co-chair of ASE in 2008 and ICSE in 2016. He is also currently on the steering committee for ICSE and SPIN, on the executive committee of ACM SIGSOFT and is a member of the editorial board of the *ACM Transactions on Software Engineering and Methodology*.