2015 IEEE 29th International Conference on Advanced Information Networking and Applications

ConDroid: Targeted Dynamic Analysis of Android Applications

Julian Schütte, Rafael Fedler, Dennis Titze
Fraunhofer AISEC
Garching near Munich
Email: *{julian.schuette,rafael.fedler,dennis.titze}@aisec.fraunhofer.de*

***Abstract*—Recent years have seen the development of a multitude of tools for the security analysis of Android applications. A major deficit of current fully automated security analyses, however, is their inability to drive execution to interesting parts, such as where code is dynamically loaded or certain data is decrypted. In fact, security-critical or downright offensive code may not be reached at all by such analyses when dynamically checked conditions are not met by the analysis environment.**

To tackle this unsolved problem, we propose a tool combining static call path analysis with bytecode instrumentation and a heuristic partial symbolic execution, which aims at executing interesting calls paths. It can systematically locate potentially security-critical code sections and instrument applications such that execution of these sections can be observed in a dynamic analysis. Among other use cases, this can be leveraged to force applications into revealing dynamically loaded code, a simple yet effective way to circumvent detection by security analysis software such as the Google Play Store's Bouncer. We illustrate the functionality of our tool by means of a simple logic bomb example and a real-life security vulnerability which is present in hunderd of apps and can still be actively exploited at this time.

I. Introduction

Up to date, Android is by far the most widespread mobile operating system. Its tremendous popularity and openness has fostered both the development of increasingly sophisticated malware, as well as research on security analysis of Android applications as a response. In recent years, researchers have developed various tools capable of detecting several types of application layer vulnerabilities which are increasingly integrated into a variety of fully automated security analysis frameworks [9].

Early tools relied mostly on static analysis which is, however, severely limited under a number of circumstances, for example when an application's behavior depends on user or remote site interaction, when control flow conditions cannot be resolved statically, when code dynamically creates or recombines data, or when code is loaded dynamically and thus not present at the time of analysis. These circumstances can introduce significant difficulties for static analysis.

These conditions can also be induced on purpose for bypassing security tests, for example by loading code over the Internet at runtime which is then not present during static analysis. Such code can introduce security issues into an application or be intentionally malicious, e.g., stealing user data or executing exploit code to gain control over a device. This way malicious applications can also easily pass the *Bouncer* security test system which is in charge of checking applications in the Google Play Store [10]. Due to these reasons, prior work addressed dynamic loading and execution of native code which poses special risks to devices [5].

The deficits of static analysis gave rise to dynamic analysis techniques. While dynamic testing tools like TaintDroid [4] have proven their effectiveness, they rely on manual interaction of the user with an application or simple record-and-replay methods, limiting their efficiency. This also bars many dynamic analysis techniques from being integrated into fully automatic test tools such as Google's Bouncer and frameworks [9], [14], [15], and prevents effective large-scale automated testing. To overcome this limitation, approaches for automated UI testing have been proposed, either using slicing of UI elements [17] or concolic execution, an analysis technique combining symbolic and concrete execution, to analyze UI interaction dependent code [1].

However, the challenge with both fully and semi-automated dynamic analysis is to control execution in such a way that the actual critical execution paths are invoked and can be observed. Simple fully automated testing tools as mentioned above may be suited for continuous GUI testing, but not for systematic testing of previously unknown applications. In contrast to that, more formal approaches like symbolic execution suffer from state space explosion and are not practicable for average size applications. Additionally, limitations of current approaches result in incomplete dynamic analysis results. This applies especially when dynamic conditions do not only depend on user input but also, for example, on the execution environment or communication with a remote site – cases which cannot be covered by UI event generation. Generating the correct sequence of user inputs may not be easy (or possible) too, leaving many potentially security-critical code sections undiscovered.

Addressing the aforementioned issues and strengthening the stance of dynamic analysis, we present *ConDroid*, a test framework for locating critical, interesting, or dangerous code and enforcing its execution and observability. We achieve this by means of concolic execution and targeted instrumentation of applications to steer control flow. Furthermore, while taking user input into account, our approach is also capable of reaching code dependent on other factors.

Our framework has a number of advantages over existing approaches. In contrast to existing dynamic analysis tech-

1550-445X/15 $31.00 © 2015 IEEE
DOI 10.1109/AINA.2015.238

niques which take UI events into account, we are neither limited to pre-recorded UI event sequences, nor in fact to UI events at all. Instead, we can discover all paths in an application and their dynamically tested conditions independently of whether they require user input, sensor or remote site input, or certain environmental conditions to be met. Our framework, by instrumenting an app under test, can force execution to follow paths leading to "interesting", i.e., potentially security-critical code sections.

The rest of the paper is structured as follows: In Section II, we review existing work in the field of security-related dynamic analysis of Android applications. Section III introduces an example motivating our work and sketches the main analysis steps of our framework. They are explained in detail in subsequent sections, namely a preparatory code bytecode instrumentation in Section IV and the concolic execution in Section V. In Section VI we illustrate how ConDroid works in action by means of the afore introduced example and highlight its practical relevance by a critical and wide-spread vulnerability found by it. A discussion of limitations and potential future improvements is provided in Section VII, before Section VIII concludes the paper.

II. Related Work

Research on dynamic analysis of Android applications has become a hot topic as malware is getting increasingly sophisticated and uses techniques like dynamic code loading in order to avoid detection by static analysis. As pointed out in [10], especially the issue of dynamic code loading is critical, as it is not only difficult to observe and analyze in an automated way. It also introduces new attack vectors and paves the way for side-loading malicious code into wide-spread applications. The analysis carried out in [10] relies on a purely static heuristic approach, which already delivers great insights. But it also becomes obvious that static heuristics soon reach their limits, e.g., if locations to load code from are dynamically crafted.

In contrast to that, dynamic analyses have the potential to observe malicious behavior at runtime.

One of the first and most prominent tools for dynamic analysis of Android applications is TaintDroid [4], which applies a dynamic taint analysis in order to trace data leaks as they occur during the execution. TaintDroid does not modify an application's control flow, but rather observes its behavior which makes it not suited for fully automated dynamic analysis. In [11], we introduced an approach to inject such dynamic analysis directly into the application, thereby reducing overhead and the need of modified system images. Mulliner et al. propose a dynamic instrumentation of the Dalvik virtual machine [8] which allows function hooking without the need to break the code signature of the original application. These and other dynamic analysis approaches have in common that they require the user to drive execution of the application to the respective code section to observe, i.e., they are not suited for fully automated tests.

A first approach to tackle this problem by applying a symbolic execution has been proposed in [7]. The authors adapt the Android framework implementation to be supported by JavaPathfinder [2] and propose respective drivers to simulate the user's behavior. Symdroid [6] in contrast, proposes a dedicated virtual machine capabable of running a subset of dalvik bytecode in a symbolic fashion. Acteve [1], a concolic execution tool for Android applications, attempts to overcome the practical limitation of symbolic execution by proposing a hybrid symbolic and concrete (*concolic*) execution of Android apps. While the general approach of injecting control flow enforcement into the application is similar to ours, the goal of Acteve is to achieve the greatest possible code coverage by generating user inputs in the form of touch events.

Acteve is closest related to our approach, as we also strive for a concolic execution which does neither require a modification of the underlying Android platform or the executing virtual machine. In contrast to Acteve however, we do not aim at achieving the greatest possible code coverage, but rather at driving execution towards specific target sections to observe, whereas we also consider branches whose execution may depend not only on user input but on any condition.

III. Overview

The analysis framework proposed herein combines a static control flow analysis with hybrid concrete and symbolic execution (*concolic execution*) in order to observe an execution path which leads to a specific target section containing "interesting" code, e.g., dynamic code loading [10] or invocation of native methods [5]. Without loss of generality, we regard the target section as a single statement.

A. Illustrative example

To illustrate our approach, consider the method in Listing 1 which acts as a simple logic bomb. We assume the method is part of an Android application and is registered as a BroadcastReceiver for incoming text messages.

```
public class Mallory extends BroadcastReceiver {
 public void onReceive(Context ctx, Intent i) {
  ...
  if (System.currentTimeMillis()>1483228800) {
   if (android.os.Build.BRAND!=null) {
    if (!android.os.Build.BOARD.contains("goldfish")
    ){
     DexClassLoader dcl = new DexClassLoader(libpath
     ,
                dexOutDir.getAbsolutePath(),
                null,
                ClassLoader.getSystemClassLoader());
     Class<?> clazz = dcl.loadClass("SomeClass");
    }
   }
  }
 }
}
```

Listing 1. Time-based logic bomb with subsequent dynamic code loading

The Android framework will invoke the method whenever a text message arrives. Only if the current date is after 2017/01/01 and the code does not run in an emulated environment, indicated by the *goldfish* kernel, the payload will dynamically be loaded and executed.

Hence, execution of the dynamic code loading does not only depend on user input but on conditions of the environment. Fuzzing user inputs or slicing UI elements will never be able to trigger and observe the code loading. We therefore strive for an approach for controlling conditions over any API calls and constants so as to drive execution along the path to the dynamic code loading. Yet, our approach works by only instrumenting the application under test and does not require any modification of the underlying Android framework.

B. Analysis Framework

Our approach consists of multiple steps. The first is a preparatory static analysis step to identify target sites and paths leading to them, which we will detail in Section IV. The subsequent steps are done by a concolic execution to force control flow onto the desired paths, explained in Section V. Combined, our approach proceeds as follows:

1) Preparatory static analysis to find call paths to "interesting" code parts from all entry points, including lifecycle, input event, and external caller induced entries.
2) Adaptive concolic execution:
   a) Instrumentation of the application along these paths which allows us to overwrite registers at runtime
   b) Instrumentation of the application along these paths which dumps the path conditions
   c) A solver which takes the path condition, negates its last condition and solves the resulting constraint problem. This new solution to the condition before a branch ensures that the second execution path behind a conditional branch is followed as well.
   d) The set of register values which will be used for the next execution, enforcing the execution of a different path
   e) A heuristic which will select additional call paths to invoke before the actual one, in order to initialize null references.

Figure 1 illustrates the overall workflow of the analysis framework.

IV. Static Call Graph Analysis

The static analysis step is concerned with finding entry points of the application and call paths to target statements. In contrast to standalone programs, Android applications do not have a single main method, but rather provide various callback methods which are mostly invoked by the Android framework in an event-driven manner, or even directly by other applications. Further, each application may directly load and execute code from other applications using `DexClassLoader` with the `CONTEXT_IGNORE_SECURITY` flag – in this case, however, execution takes place in the process of the calling application. Thus there is no overall correct set of entry points and choosing an appropriate set depends on the analysis purpose. In general, for concolic execution all application methods which may be called by the Android framework in the context of the application's process are relevant. However, for practical reasons it must be considered that each additional entry point will increase the number of executions by a multitude and thereby increase the analysis time significantly. We therefore limit the set of entry points to those we consider most relevant and extend the call graph of the application so as to make these entry points reachable from the default entry point, which is the default Activity's `onResume` method.

a) Android lifecycle methods: First, we are interested in all methods triggered by Android lifecycle events. That is, the call graph will be extended by edges from the default Activity's onResume methods to methods of all Activities, Services, Receivers, and Providers (ASRP), implementing the `ActivityLifecycleCallbacks` interface.

b) References to views: Android UIs consist of *Views*, which are either declared in XML files or created programmatically. They are *inflated* when a screen is rendered, depending on screen resolution, rotation, etc. Classes inheriting from *android.view.View* are instantiated by the `LayoutInflater` just before they are inflated. While the exact time of inflation cannot be determined from the application code, we approximate it by adding edges to the call graph whenever a component calls `findViewById(int)`, which is the standard way of getting a reference to the instance of that view. Thus, the additional edges overapproximate the actual execution, but they are correct in the sense that upon invocation of `findViewById(int)` it is guaranteed that the constructor of the view has been called.

c) UI event handlers: An Android application may register handlers for a wide number of UI events such as clicking or dragging. Handlers registered for such events can advance execution into code sections which would not be reached without user interaction. Thus, the application is extended by artificial calls to these handlers to ensure that the respective code parts are reached, even without user interaction.

d) Intents: Further, we extend the call graph to take into account intra-application Intents.

Intents can either be sent *explicitly* to a component by addressing its class name or *implicitly* in a publish-subscribe-manner. We consider only explicit intents and identify the receiver's class name. Whenever an explicit intent to a method $X.x$ is found to be sent from a method $Y.y$, a respective edge $Y.y \rightarrow X.x$ is inserted into the call graph. Again, the edge reflects an incorrect context but it is correct in terms of the execution sequence.

Figure 2 depicts a call graph with the artificially inserted edges as dashed lines.

Finally, we trace back the execution paths from the target statement to all entry points and consider these for the following instrumentation and concolic execution.

V. Concolic execution

In the next step, the application is prepared for concolic execution. Concolic execution is a hybrid of a plain execution of the application with concrete register values, and symbolic execution which represents register values symbolically. The idea of symbolic execution is to trace symbolic registers at

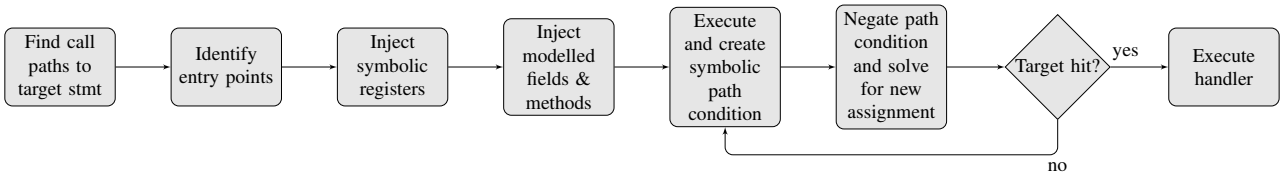

Figure 1. Sequence of steps performed by ConDroid

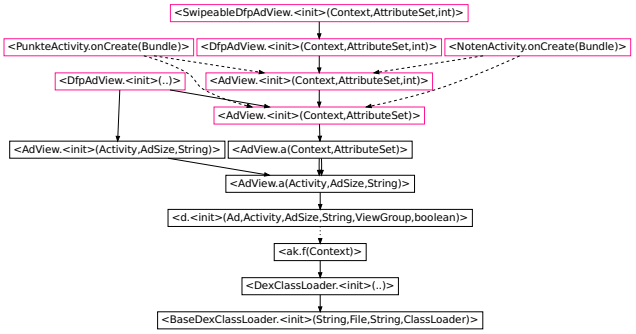

Figure 2. Call graph extension (potential entry points E in pink, artificial calls dashed)

each conditional statement in order to build *path conditions* for specific execution traces. Consider the following code snippet. Symbolic execution will treat the three registers as symbolic values x', y', and z' and create path conditions $PC_1 =< ((x' + y')\,50)$ and $PC_2 = not(< ((x' + y')\,50))$, corresponding to the execution of PC1 and PC2, respectively. For a more detailed discussion on symbolic execution we refer to [12].

```
public void test(int x, int y) {
  int z = x + y;
  if (z<50) {
    //PC1
  } else {
    //PC2
  }
}
```

Listing 2. Conditional statment leading to two path conditions

Concolic execution aims to address the notorious state explosion problem by executing the program with concrete values and tracing symbolic counterparts of only certain registers in parallel. That is, the program is executed in its normal context and conditional branches are evaluated using concrete register values. In addition, however, symbolic shadow copies of registers are maintained and the path condition of the current execution trace is dumped.

In order to enforce a new execution path, a new path condition is created by negating the condition of the last executed branch, i.e., $PC' = c_1, c_2, ...\neg c_k$. Using a constraint solver, a set of concrete register values is found, which would lead to the execution of the new path when injected before the respective conditional statements in the program.

In the following, we describe the preparation of an Android application for concolic execution, the procedure of iterative execution of the application and the actual concolic execution semantics.

A. Preparatory instrumentation

Symbolic execution is usually implemented at the virtual machine level, i.e., specific virtual machines trace symbolic variables and path conditions, independent from the concrete execution. Examples for this approach are S^{2}E and KLEE for native code, or JPF [2] for Java bytecode.

With concolic execution, however, the application is mostly executed normally, while only some variables have to be traced symbolically. Hence, it is more sensible to inject symbolic tracing into the original application and run it in an unmodified execution environment. Acteve [1] chooses this approach and instruments the application and the Android framework, rather than modifying the VM.

We thus followed the latter approach and based our implementation on the Acteve prototype, which we extended by an upstream static analysis and the ability to inject solution values for arbitrary method calls and fields in order to manipulate conditions along the call path.

1) Selectively jump to entry points: In a normal execution, the Android framework manages events such as user inputs or received messages and calls the corresponding entry point of an application. Our goal is however to immediately observe an interesting execution path, no matter which sequence of input events is required to trigger it. Therefore, the application is instrumented in such a way that all entry points mentioned in section IV are immediately called from the application default entry point. In detail, the instrumentation proceeds as follows:

The application's default entry point is the `onResume` method of its default Activity, or of its Application component, if defined. At the end of the default entry method, the Android lifecycle methods of all further application components are invoked. In addition, the initial lifecycle handlers of all components along the call path are extended by direct calls to UI event handlers which are used in the component's layout. For example, an Activity using a layout with several Buttons and TextViews will have calls to the `onClick()` and `onTextChanged()` methods of the registered handler classes injected into its `onResume` method. Listing 3 shows an instrumented entry point method.

```
public void onResume() {
 AdRequest ar = new com.google.ads.AdRequest();
 AdView av = this.adView;
```

```
 av.loadAd(ar);
 android.content.Context cont = this.
     getApplicationContext();

 //Invocation of further Activities along the CP
 Intent in = new Intent(cont, com.google.ads.
     AdAcitivity.class);
 this.startActivity(in);
 //Invocation of UI event handlers:
 View.onClickListener ocl1 = someOnClickListener;
 ocl1.onClick(someView);
 return;
}
```

Listing 3. Instrumented Android lifecycle entry point

Adding calls to UI event handlers and other components only at the end of the initial lifecycle method is a heuristic which aims at enforcing all selected call paths but still allowing the application to initialize as usual. This is important as applications often require initialization of databases or file structures at first start or set static fields which are needed at a later time during execution. In theory, suitable object instances for these fields could be generated by symbolic execution. CUTE [13], for example, a tool for concolic execution of C code, attempts to solve this problem by using the constraint solver's solutions to create `memory graphs` for objects which are required to be non-null. This, however, is not feasible in practice where complex objects like `android.content.Context` would have to be efficiently generated. We therefore rely on the existing initialization routines of the application and inject all modifications only after them. While ConDroid is not able to generate any object instance as needed, many typically required objects can be generated on the fly. For example, when an `android.content.Context` object is not present in a method we can acquire one by retrieving it from an injected call to `getApplicationContext()` in the application's main entry method and storing it in a publicly accessible static variable which can be accessed from all over the application. Other object instances may require further generation heuristics and where we are not able to create appropriate instances, we omit insertion of the method calls requiring instances as arguments.

2) Symbolic registers and path conditions: All methods along the chosen call path will have symbolic complements added to their registers. That is, for each local register r, a symbolic register r' is introduced and for instructions writing to r, a symbolic counterpart is added. These are initially limited to arithmetic operations over primitive types, but can be extended by symbolic models of any method.

Further, at each conditional branch, the respective conditional expression over the symbolic registers is created and will be dumped to the console.

For registers of boolean type, the path condition is not immediately dumped, but ConDroid rather attempts to trace back the register value to a function supported by the SMT solver and use it instead of the original conditional operation. For example, consider the following code for comparing two strings.

```
$z0 = $r0.java.lang.String: bool contains("X")
if ($z0 == 0) goto label A
```

Instead of using $eq(\$z_0' \ 0)$ as a path condition, the use-def-chain of $z0 leads to the method call to `String.contains` which is in fact supported by the constraint solver. Thus, the path condition is turned into $contains(\$r_0' \ "X")$ which will lead to the semantically richer solution for $r0 instead of one for the boolean variable which is merely used in conditional. Such backtracing is necessary, as it allows to retrieve solutions for actual complex objects, rather than for the result of comparison methods such as `equals`.

3) Models of methods and fields: Finally, we allow to overwrite return values of individual method calls and field values by a *solution*, i.e., concrete values computed by the constraint solver. This way, not only user inputs can be modeled, but also any Android API method can be replaced by a symbolic counterpart. As a result, potential code coverage of the concolic execution is expanded to blocks whose reachability does not depend on mere user input, but on any condition on the execution environment.

This is an essential feature when it comes to analyzing potential malware, as it allows to solve conditions of typical logic bombs or countermeasures such as attempts to detect an emulated execution environment in order to hide malicious behavior from dynamic analysis tools.

B. Concolic execution semantics

By instrumenting the original application we slightly change its execution semantics. Mainly, the way how method invocations and assignments referring to modeled methods and fields are treated is changed so as to replace the original assignment with the injected solutions. In the following, we detail the changed semantics.

We regard an Android application A as a program consisting of classes c and methods $c.m \in Meths$, made up of statements $stmt$. During execution a program counter pc points to the next statement to execute, which is retrieved by $insAt(pc)$. Each statement denotes a transition of program configurations $C = \langle S, H, SF \rangle$. Here, a static heap S denotes the set of static fields, a heap denotes a set of non-static fields, and a stack SF denotes the vector of stack frames at the current point of execution. Each stack frame $sf \in SF$ is denoted by $s = \langle m, pc, R \rangle$ where m and pc refer to the current method and program counter and R is the set of local registers.
We further add symbolic representations of local registers, non-static fields, static fields, and specific methods, and denote them by Γ, while concrete values are represented by γ. Thus, $R.\gamma$ refers to concrete values of local registers and $R.\Gamma$ to their symbolic counterparts.

Operational semantics of Dalvik bytecode have been given in [16]. We instrument the application code to change the semantics of some statements along the call paths to the target statements, without requiring any modification of the actual Dalvik VM. For symbolically modeled methods we introduce an operation `call`, summarizing the `invoke-*` and

`move-result` operations for method invocation and retrieval of return values. Likewise, the `load-*` operations retrieves values from symbolically modeled static and non-static fields. In the following, we give the modified execution semantics of `call` and `load-*` which basically denotes the replacement of concrete return values and fields by injected solutions for the respective models. Note that the given semantics applies only to statements along the afore identified call paths. The rest of the application remains untouched.

C. Iterative execution

The thus instrumented application is now iteratively executed, where each run is configured with a set of symbolically modeled functions and a *solution*, i.e., a set of concrete register values to replace the actual return values from Android APIs. During the execution, conditions over symbolic registers are collected in a path condition. As soon as the execution deviates from the intended call path by branching to the wrong basic block, the last clause of the path condition is negated and a SMT solver is applied to generate a new solution of concrete register values leading to execution of the intended basic block. Due to the injected direct jumps to entry points, a single run will sequentially execute all entry points of the application, so that the number of executions will only depend on the length of the call path to observe, but not on the number of potential entry points, i.e., UI widgets and callback functions.

VI. Prototype evaluation

The framework can be easily adjusted to locate interesting code sections of a big variety. It will subsequently identify call paths to them and steer execution accordingly. To demonstrate its capabilities in practice, we chose the prevalent problem of dynamic code loading which poses significant risks to devices and manages to fool many security analysis tools easily and thus goes unnoticed.

A. Analysis of dynamic code loading

Dynamic class loading and method invocation via reflection is an often used pattern in Android applications. In fact, among the 10,000 most popular apps from the Google Play store, we found 7,923 occurrences of the dynamic class loading via `Class.forName()`. Around 45% account to Google Analytics and 22% to Google Ads. However, there are still 2,733 applications loading more critical APIs, such as `sun.misc.unsafe` or `android.os.Properties`, which allow direct memory manipulation and access to internal properties of the device platform.

As dynamically loaded bytecode is out of the scope of traditional static analysis and is not required to be signed or distributed through the normal app store, dynamic class loading is an attractive feature for malware authors and the need to analyze it in an automated way is obvious.

Once dynamic loading of classes can be observed at runtime, it is possible to dump the loaded bytecode and integrate it into the original application, as described in [3]. We thus illustrate how ConDroid can be used to automatically drive execution towards such critical statements, even if they would not be reachable under normal circumstances.

Referring back to the logic bomb example from Section III-A, we consider a conditional execution of dynamic code loading which would only become active at a specific time and on a specific platform. ConDroid will detect such conditional execution of malicious code, as follows:

- *Static call path analysis*
  In a first step, the `Mallory.onReceive` method will be identified as an application entry point and the class loading statement as a target. The call path thus comprises only the single method `onReceive`.
- *Call path modification*
  During the call graph modification, a direct call from the main activity's `onCreate` method to the `onReceive` handler from Listing 1 is inserted.
- *Solution injection and symbolic tracing*
  The first condition depends on an API call, which is consequently symbolically traced and its concrete value is overwritten by a solution.
- *Extending conditions to modeled predicates*
  The second condition is a boolean comparison. The use-def-chain of the involved register is traced back to the String `contains` method, which is supported by the SMT solver. Therefore, the original condition `$z0 == 0` is replaced by `$r2 contains "goldfish"`.

After this preparation, the iterative execution starts.

- *First execution with concrete values*
  In the first run, the solution set is empty and no replacement of concrete register values takes place. The initially dumped path condition is
  not `> ( currentTimeMillis 1483228800)`
- *Solving integer API condition*
  The last (and so far only) entry of the path condition is negated and solved using the Z3 SMT solver. The value (currentTimeMillis ↦1483228801) is written to the solution map and a second run of the app is started.
- *Solving string constant condition*
  Identically to the previous step, the next path conditions *not* `(Brand null)` and *contains* `(Board "goldfish")` are dumped, negated and added to the constraint system handed over to Z3. We use the Z3 extension Z3-Str[1] which provides an SMT string theory. In this case, the solutions (Brand ↦ "") and (Board ↦ "q") are found and written to the solution map.
- *Hitting the target*
  At another execution run, all three conditions are fulfilled and the call path to hit the dynamic code loading statement is executed:
  `Mallory.onReceive` is invoked due to the immediate entry point invocation, the return value of the `System.currentTimeMillis` function call, as well as the field values `android.os.Build.BRAND` and

[1] http://z3.codeplex.com/releases/view/95640

$$\texttt{call}\ \frac{m.instAt(pc) = invoke\text{-}*\, m' \qquad m.insAt(pc+1) = move\text{-}result\, x \qquad m' \in Meths}{A \vdash \langle S, H, \langle m, pc, R\rangle\rangle \Rightarrow \langle S, H, \langle m, pc+2, R\,[x \rightarrow Meths.\Gamma\,(m')]\rangle\rangle}$$

$$\texttt{load-static}\ \frac{m.instAt(pc) = sget\text{-}*\, y, x \qquad y \in R \qquad x \in S}{A \vdash \langle S, H, \langle m, pc, R\rangle\rangle \Rightarrow \langle S, H, \langle m, pc+1, R.\gamma\,[y \rightarrow S.\Gamma\,(x)]\rangle\rangle}$$

$$\texttt{store-static}\ \frac{m.instAt(pc) = sput\text{-}*\, x, y \qquad y \in R \qquad x \in S}{A \vdash \langle S, H, \langle m, pc, R\rangle\rangle \Rightarrow \langle S.\Gamma\,[x \rightarrow y]\,, H, \langle m, pc+1, R\rangle\rangle}$$

$$\texttt{load-instance}\ \frac{m.instAt(pc) = iget\text{-}*\, y, x \qquad y \in R \qquad x \in S}{A \vdash \langle S, H, \langle m, pc, R\rangle\rangle \Rightarrow \langle S, H, \langle m, pc+1, R.\gamma\,[y \rightarrow R.\Gamma\,(x)]\rangle\rangle}$$

$$\texttt{store-instance}\ \frac{m.instAt(pc) = iput\text{-}*\, x, y \qquad y \in R \qquad x \in S}{A \vdash \langle S, H, \langle m, pc, R\rangle\rangle \Rightarrow \langle S, H.\Gamma\,[x \rightarrow y]\,, \langle m, pc+1, R\rangle\rangle}$$

Figure 3. Excerpt of the operational semantics deviating from standard dalvik bytecode

`android.os.Build.BOARD` are overwritten by an injected solution. The target statement is executed and the actually loaded code can be observed. A simplified version of the method modified to reach the target statement is provided in Listing 4.

```
public class Mallory extends BroadcastReceiver {
 public void onReceive(Context ctx, Intent i) {
  ...
  int x = System.currentTimeMillis();
  x = Util.getSolution(0); //1483228801
  if (x>1483228800) {
   y = android.os.Build.BRAND;
   y = Util.getSolution(1); //""
   if (!y!=null){
    z = android.os.Build.BOARD;
    z = Util.getSolution(2); //"q"
    if (!z.contains("goldfish")){
     DexClassLoader dcl=new DexClassLoader(libpath,
                 dexOutDir.getAbsolutePath(),
                 null,
                 ClassLoader.getSystemClassLoader());
     Class<?> clazz = dcl.loadClass("SomeClass");
} } } } }
```

Listing 4. Method altered by ConDroid to enforce execution of target site

B. Application to real-world applications

In a second evaluation, we applied ConDroid to non-trivial applications from the Google Play store. As we will discuss in the following section, execution with the ConDroid prototype may lead to invalid execution contexts, so that it does not reliably drive execution along the desired call path in each and every application yet. Nevertheless, we were able to find a wide-spread vulnerability in applications based on the Adobe AIR framework[2]. Adobe AIR is a framework to compile Android applications from a Flash-based implementation. The boilerplate code of the AIR framework is kept in a separate application and is loaded by each AIR-based application at startup. As the loading takes place in a conditional basic block, depending on the availability of the AIR framework, it is not observable by a simple dynamic analysis in the Android emulator.

ConDroid automatically creates and injects a solution for the respective condition and executes the dynamic loading attempt which can consequently be observed. As there are no further integrity checks on the loaded code, an attacker would simply have to deploy malicious code with the package name of the Adobe AIR framework on the victim's phone to have it loaded and executed in the context of a benign AIR application. As a result, the attacker would be able to execute arbitrary code in the security context of the AIR application and get access to all data managed by it.

Among the 10,000 most popular applications from Google Play, 172 were prone to this vulnerability which we could automatically discover using ConDroid. Although in this case, the vulnerability is not deeply hidden in the code, even this simple example shows how powerful concolic execution can be for automatic discovery of previously unknown security vulnerabilities, which will not be revealed by current, simpler dynamic analysis techniques.

[2]The vulnerability has been reported to the vendor in July 2014

VII. Discussion

While our approach strengthens the stance of dynamic analysis, it still has a few limitations.

First, it is limited to bytecode only and cannot influence control flow within native code. The transition from bytecode to native code is a general hurdle in Android application analysis and further research on integrating both into a consistent model is needed.

Second, difficulties arise when there exist only execution paths which require the construction of complex objects. ConDroid is currently able to create primitive types, strings and complex objects which only require simple instantiation by the default constructor. A special case is the often used `android.content.Context` instance which is retrieved at start of the application and made available throughout it. Apart from that, the creation of complex objects with specific attributes and methods is currently not supported, although approaches similar to the creation of memory maps described in [13] would be applicable.

A third challenge are erroneous execution contexts caused by injected concrete instances for complex objects. Generation of concrete instances depends only on the path condition but not on any further assumptions which are not explicitly expressed in conditional statements but necessary for a correct execution. Incorrectly generated object instances can lead to unexpected behavior such as exceptions being thrown or control flow deviating from the intended path, due to side effects or because of non-initialized objects. One approach to tackle this problem is to infer assumptions on complex objects in a static analysis and to inject explicit checks for them into the code.

The three problems explained in this section are no specific limitations to our approach, but apply to concolic execution in general. However, as we have shown, ConDroid is already able to cope with real-life applications and to discover vulnerabilities in them. Extensions and heuristics for object instance generation will further increase the coverage of ConDroid.

VIII. Conclusion

In this paper, we introduced ConDroid, a new framework for targeted dynamic analysis of Android applications. Unlike previous dynamic analysis tools for Android applications, it allows for fully automatic analysis of applications while ensuring that all interesting code sites are reached to enable further analysis of such target sites and their effects. The definition of such interesting code sites can include, for example, dynamic loading of code, a prevalent problem for Android security and not yet satisfyingly analyzable by previous approaches. We can force applications under test to load and thus reveal code usually only loaded during real execution. Analysis supported by the techniques described in this paper can hence not be as easily evaded as current approaches which can be circumvented by only loading harmful code under conditions which are not met by an analysis environment. However, target specification is not limited to that and code of interest can be arbitrarily defined. ConDroid thus ensures the discovery of all code sites that are of interest to an analysis and enforces their execution such that their effects can be observed.

References

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, New York, New York, USA, November 2012. ACM Press.

[2] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250, 2011.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[5] R. Fedler, M. Kulicke, and J. Schütte. Native code execution control for attack mitigation on android. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, pages 15–20, New York, NY, USA, 2013. ACM.

[6] J. Jeon, K. K. Micinski, and J. S. Foster. SymDroid : Symbolic Execution for Dalvik Bytecode. *J Syst Softw*, 2012.

[7] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, November 2012.

[8] C. Mulliner, W. Robertson, and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 459–470, New York, NY, USA, 2014. ACM.

[9] S. Neuner, V. Van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl. Enter sandbox: Android sandbox comparison. In *Proceedings of the IEEE Mobile Security Technologies Workshop (MoST)*. IEEE, 5 2014.

[10] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *21st Network and Distributed System Security (NDSS) Symposium*, pages 23–26, February 2014.

[11] J. Schütte, D. Titze, and J. M. de Fuentes. AppCaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-14)*, July 2014.

[12] 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). *Security and Privacy SP 2010 IEEE Symposium on*, (7):317–331, 2010.

[13] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5):263, Sept. 2005.

[14] D. Titze, P. Stephanow, and J. Schütte. App-ray: User-driven and fully automated android app security assessment. Fraunhofer AISEC TechReport, Dec. 2013.

[15] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. Technical Report TR-ISECLAB-0414-001.

[16] E. R. Wognsen, H. S. n. Karlsen, M. C. Olesen, and R. R. Hansen. Formalisation and analysis of Dalvik bytecode. In *Science of Computer Programming (SCP)*, pages 1–47, 2013.

[17] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.