# Automated Execution Control and Dynamic Behavior Monitoring for Android™ Applications

Mike Ter Louw*, Marc Krull†, Tavaris Thomas*, Rebecca Cathey‡, Greg Frazier§ and Mike Weber‡

*LGS Innovations, Bell Labs
Florham Park, NJ, USA
{*mterlouw, tjthomas*}
@*lgsinnovations.com*

†‡BAE Systems
†Burlington, MA, USA
‡Arlington, VA, USA
{*marc.a.krull, rebecca.cathey,*
*michael.weber*}*@baesystems.com*

§Apogee Research
McLean, VA, USA
*glfrazier@apogee-research.com*

***Abstract*—We explore techniques for eliciting a behavioral description from an Android smartphone app in a controlled manner. A description of app behavior is useful for performing subsequent analysis such as model checking, for example to verify the app satisfies a set of desirable security properties. Our solution is to dynamically execute the app in a custom version of the Android SDK emulator, which provides many of an app's inputs as responses to invoked API calls. A more focused set of input values computed offline are then injected to the app via hooks introduced into the Android API implementation. To dynamically monitor app behavior, we instrument the app bytecode to record control and data flows during execution. We also instrument the Android API to record all of the app's inputs and outputs. We have used this technique on the DARPA Automated Program Analysis for Cybersecurity (APAC) program to reveal hidden, triggerable attacks in independently developed challenge apps. Our framework for extracting app behavior is part of Droid Reasoning, Analysis, and Protection Engine (DRAPE), an integrated, semi-automated app behavior analysis system capable of discovering hidden malware in Android apps.**

## I. Introduction

In this paper we explore techniques for eliciting a behavioral description from an Android smartphone app in a controlled manner. Our goal is an automated system that accepts a set of Android application inputs computed offline, and generates a detailed description of the behavior exhibited when an unknown app is subjected to those inputs. The behavior descriptions we elicit are useful for performing subsequent analysis such as model checking [1], [2], for example to verify the app satisfies a set of desirable security properties and thereby determining if the app constitutes malware [3], [4].

Generating a description of real app behavior when subjected to concrete inputs is an important component of concolic testing [5], [6], [7]. Concolic testing is useful for exploring the full range of behaviors that can be exhibited by an app, including hidden malicious functionality. One approach for obtaining a description of Android app behavior is static analysis of either its source code or Dalvik™ virtual machine (VM) bytecode [8], [9], [10]. For inputs that are unknown, static analysis approaches may symbolically evaluate code over a range of possible inputs, and will often produce uncertainty in the results. Therefore it is important to know as much about the inputs as possible, yet determining all of these inputs through static analysis can be difficult and tedious. A large portion of inputs are provided by the operating system in response to an app's Android API calls. Another significant source of statically-unknowable app inputs are those inputs controlled by external entities, such as network communications.

The same inputs that are hard to obtain statically can be readily available in dynamic analysis. Our solution is to dynamically execute the app in a custom version of the Android SDK emulator, which provides the inputs returned by API calls and external communication. A more focused set of input values computed offline are then injected to the app via hooks introduced into the Android API implementation.

We categorize app inputs as pull-inputs or push-inputs. *Pull inputs* are those which the app requests explicitly via an API call. Our system can be used to induce a control flow by providing values to pull inputs. For instance, the malicious behavior of an app with a time-bomb can be triggered by providing the right value in response to the `System.currentTimeMillis()` API call. *Push inputs* are invocations of an app's entry points, such as event handler methods. Our system can directly manipulate the app's GUI widgets and reactions to system events using push inputs.

To dynamically monitor app behavior, we instrument the app bytecode to record control and data flows during execution. We also instrument the Android API to record all of the app's inputs and outputs. In terms of overhead, instrumentation results in increased app file sizes, which is not a factor in the use cases we envision such as offline malware detection and analysis. App instrumentation also incurs a cost of slower runtime performance, which may prevent accurate characterization of app behaviors that are highly sensitive to timing. In our use of the system, the primary limiting performance factor has been the time required after app execution to return and store execution traces, which can grow very large.

We have used our technique on the DARPA Automated Program Analysis for Cybersecurity (APAC) program to reveal attacks in independently developed challenge apps where manual source code analysis had previously failed to discover an app's true nature. Our framework for discovering app behavior is part of Droid Reasoning, Analysis, and Protection Engine (DRAPE), an integrated, semi-automated app behavior analysis system capable of eliciting behavioral traces for Android apps

Approved for public release; distribution unlimited. Cleared for open publication on May 3, 2013. Android and Dalvik are trademarks of Google, Inc.

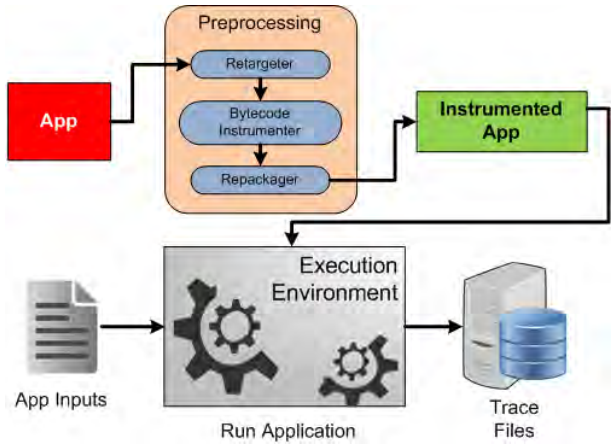


Fig. 1. Overview of app execution control and behavior monitoring system.

under various input conditions.

The rest of the paper is organized as follows. In Section II we describe our solution at a high level. We give details on our implementation in Section III, and evaluate the performance of our solution in Section IV. In Section V we highlight interesting challenges in our approach. We review related work in Section VI, and finally in Section VII we conclude.

## II. Overview

Figure 1 provides a high-level overview of our approach. We start with an Android app and a set of app inputs. The first step is to preprocess the app, which prepares it to run in our execution environment (EE) by instrumenting its bytecode and attaching our test harness. The bytecode instrumentation will cause the app to emit an execution trace as it runs. The test harness facilitates automated app control by manipulating the app's user interface and simulating system events. At the end of preprocessing, our solution yields an instrumented app that can be executed many times with varying inputs.

Next, the instrumented app is loaded into the EE and started. Our EE is based on a custom version of the Android SDK v4.2 emulator which has been instrumented to interpose on and record the app's input and output operations. As the app runs, the provided inputs can directly or indirectly control the app's execution path to exhibit behaviors of interest. During execution the EE monitors all instrumented activity and generates an execution trace. After the app's push inputs are exhausted, the EE terminates the app run. Finally, the EE outputs the execution trace, which is our representation of app behavior, for analysis.

## III. Implementation

This section describes our implementation in detail over three key areas: app preprocessing (§III-A), I/O hooking (§III-B), and the test harness (§III-C).

### A. App Preprocessing

Before an app is run within the EE, we must preprocess the app to instrument its bytecode and attach a test harness. However, we can not directly modify the app because the modification tools we require do not support Android file formats. Thus, our preprocessing approach is to (1) unpack the app, (2) convert its files into supported formats, (3) apply our changes to the app, (4) convert the files back into Android formats, and finally (5) repackage the app. We divide preprocessing into three major stages:

- *Retargeting.* Extract all app files from the app's `.apk` package file and convert them from Android formats to formats that are better supported by existing tools.
- *Bytecode Instrumentation.* Instrument the app's bytecode so that it emits an execution trace when run.
- *Repackaging.* Convert the modified app files back into Android formats and bundle them along with the test harness in a new `.apk` file.

The following sections provide details on each of these stages.

*1) Retargeting:* This stage begins by extracting the `.apk` file. We then use the `dex2jar` [11] program to retarget the app's bytecode. *Retargeting* converts bytecode written for one instruction set architecture to another. In our case, `dex2jar` retargets the app's Dalvik virtual machine (VM) bytecode to Java VM bytecode. Retargeting produces a `.jar` file, which we extract to obtain the app's JVM `.class` files. Finally, we use the `apktool` [12] program to convert the app's binary `AndroidManifest.xml` file into plain text.

*2) Bytecode Instrumentation:* Java class files can be viewed as binary instruction streams that operate in conjunction with a per-frame operand stack, per-frame local variable array, and per-class constant pool [13]. Events of interest within the instruction stream are instrumented by 1) detecting *event-signifying instructions*, 2) staging the operand stack appropriately to capture event metadata, and 3) inserting a static hook call to report the event. For a given class and for each bytecode event type, our instrumenter applies a transformation using the Apache Byte Code Engineering Library (BCEL) [14]. The general instrumentation approach is illustrated in Figure 2, and the event-signifying instructions, associated event types, and event attributes are listed in Table I.

Given that instrumentation is performed as post-compilation binary manipulation, it is necessary to ensure that operand stack adjustments neither (a) consume operands expected by existing instructions or method calls, nor (b) leave excess operands on the stack that conflict with existing instructions or method calls. Either scenario may cause load time verification exceptions or runtime errors in either the event hooking code or the existing code. In practice, a

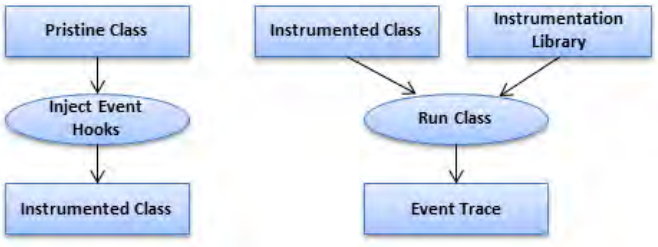


Fig. 2. Bytecode instrumentation involves an application modification phase followed by one or more application executions to generate trace files.

TABLE I. BYTECODE INSTRUCTIONS AND ASSOCIATED INSTRUMENTATION EVENTS

| Instruction | Event | Attributes |
|---|---|---|
| new | ObjectCreation | ClassName, Identifier |
| putstatic | FieldWrite | Identifier, FieldName, Value |
| putfield | FieldWrite | Identifier, FieldName, Value |
| getstatic | FieldRead | Identifier, FieldName, Value |
| getfield | FieldRead | Identifier, FieldName, Value |
| BCEL::method | MethodCalled | Identifier, MethodName, ArgumentValues[] |
| *return | MethodReturn | Identifier, MethodName, ReturnValue |
| invoke* | MethodInvoke | Identifier, MethodName, NumArguments, MethodSignature |
| *newarray | ArrayCreation | Type, Count, ArrayRef |
| *astore | ArrayWrite | ArrayRef, Index, Value |
| *aload | ArrayRead | ArrayRef, Index, Value |
| athrow | ExceptionThrow | InstanceIdentifier, NumFrames |
| BCEL::handler | ExceptionCatch | InstanceIdentifier, NumFrames |

TABLE II. ANDROID PERMISSIONS ADDED TO AN APP'S ANDROIDMANIFEST.XML FILE TO INTEGRATE WITH THE EE.

| Permission | Reason added |
|---|---|
| ACCESS_COARSE_LOCATION | Synthetic location updates |
| ACCESS_FINE_LOCATION | Synthetic location updates |
| ACCESS_MOCK_LOCATION | Synthetic location updates |
| INJECT_EVENTS | Synthetic touch events |
| INTERNET | Enable communication with EE controller |
| WRITE_EXTERNAL_STORAGE | Enable execution trace logging to files |
| WRITE_SMS | Synthetic SMS events |

common exception has been a stack imbalance wherein not enough operands are provided for a particular hook method.

As a concrete example, consider instrumentation of all instance field assignments in a given class. First, all instructions in all methods are sifted for the `putfield` (set field in object) instruction. When a `putfield` is found, the operand stack is rearranged from [objectref, value] to [objectref, value, objectref, value] by inserting a `dup2` in the instruction stream in front of the `putfield`. Any additional metadata is pushed onto the stack - in this case, the field name - leaving the operand stack as [objectref, value, objectref, value, fieldname]. Finally, the hook method invocation is inserted. When the hook is invoked at runtime, the instrumentation-related operands will be consumed, leaving the stack in its original state for the `putfield` to be invoked on [objectref, value] as expected. The resulting inline instrumentation for this example is illustrated in Figure 3.

*3) Repackaging:* Repackaging consists of adding our test harness to the app, altering the app's XML files so the app will run in our environment, converting the files back into Android formats, and then bundling them up as a single `.apk` file.

***Merging the test harness.*** Our test harness provides bootstrapping support for bytecode instrumentation and execution control for the app after it is started. It is implemented as several Java classes which we describe in §III-C. To enable the test harness to execute before the app is started, we add an `<instrumentation>` element to the manifest file (`AndroidManifest.xml`). We also copy all of the test harness's class files into the directory where the app's class files are stored. These class files will later be converted and merged into a single `classes.dex` file.

(a)
```
1 aload_0
2 iconst_0
3 putfield      #14  // Field lines:I
```

(b)
```
1 aload_0
2 iconst_0
3 dup2
4 ldc_w         #538 // String lines
5 invokestatic #169 // Method EventHooks.
6                   // ___hookFieldWrite:(...)V
7 putfield      #14  // Field lines:I
```

Fig. 3. Example bytecode instrumentation hook for `int lines = 0` field assignment. (a) Original `javap` disassembly (b) Hooked code.

***Test harness permissions.*** In addition to the changes involved in merging the test harness, the repackaging step adds content to the app's `AndroidManifest.xml` file to grant the app additional permissions as listed in Table II. This is to support our test harness, which requires permissions to perform its functionality but runs in the same security context as the app. For example, the test harness needs to send synthetic GPS location updates to the app upon command from the EE, and therefore requires the `ACCESS_MOCK_LOCATION` permission. The harness needs to log execution trace data to the file system, and thus requires the `WRITE_EXTERNAL_STORAGE` permission.

***File conversion and bundling the `.apk`.*** The final step of preprocessing is to convert the modified files back into Android-supported formats and bundle them into an `.apk` package file. For this we use tools that are part of the Android SDK. The `aapt` program converts the text XML files into binary formats and compiles them into a resource package file. The app and test harness class files are retargeted back to Dalvik bytecode using the `dx` program. We then bundle everything together using the `apkbuilder` tool, and sign the resulting `.apk` file with the `jarsigner` program. Optionally, we can optimize the package using the `zipalign` tool. At the end is a fully preprocessed app that can be run in the EE.

### B. I/O Hooking

Our solution requires instrumenting both the app code and the Android API framework code. As previously described, the goal of app bytecode instrumentation is to emit an execution trace as the bytecode executes. The goal of Android API framework instrumentation is to emit a trace of input and output events as the app runs, which we describe in this section. To enable correlation of method calls with input/output events, sequence numbers are synchronized across all app instrumentation and I/O trace files.

Each time the app reads in data from the API, we record an input event. The term we use for these events is *pull inputs*, since the app initiates the API call and "pulls in" a new value. Our solution also supports replacing the real pull input values with fake values to indirectly control the execution path of the app. We discuss pull input hooking in §III-B1. Each time the app writes data via the API, we record an output event. We discuss output hooking in §III-B2.

(a)
```
1 public double getLatitude() {
2   return mLatitude;
3 }
```

(b)
```
1 public double getLatitude() {
2   return new sys.drape
3     .ValuePuller<Double>(){}
4           .pull( 0, this, mLatitude );
5 }
```

Fig. 4. Example pull input hook. (a) Original source code for `android.location.Location.getLatitude()` API. (b) Hooked code.

*1) Pull inputs:* We manually instrument the Android API framework at the source code level. Figure 4 shows an example hook for the `getLatitude()` method of the `Location` API class, which an app can use to query the device's current location. The generic type `<Double>` at line (b).3 informs our test harness of the type of input being requested. In the call to `pull()` at line (b).4, the harness is provided this type along with other data, which it can use to select the value to be returned. This other data includes the class and method the hook is in (determined by `Class.getEnclosingMethod()`), the arguments to the method, and the original system-provided value that would be returned if no hook existed. If a fake input is not selected, the test harness returns the real input. The harness logs the input being pulled by the app, and whether the input value was provided by Android or the EE.

In this example, the `ValuePuller` support class is used at line (b).3. Alternatively, a `Puller` class is available which allows the harness to throw an exception (any `Throwable` object) as input.

*2) Outputs:* Output hooking is done in a manner very similar to pull inputs. However, rather than calling the `pull()` method, our test harness is invoked by a call to `logOutput()`. This method is provided with the value being output in addition to the same identifying data (hooked class, method, arguments, etc.) as pull input hooks. The harness receives this data and logs the output to the execution trace.

### C. Test Harness

Figure 5 diagrams our execution environment solution at runtime. The left side of the figure shows an EE application that executes and records the app's behavior using an Android emulator, which is shown on the right. Our *Test Harness* consists of two primary components:

1) *EE Controller.* Embedded in the EE Application and is the main interface to the execution environment. It manages the Android emulator, installs and interacts with the app, and retrieves the execution traces.
2) *Test Driver.* Runs on the emulator and manages the app in the Android environment. The Test Driver initializes the environment and provides inputs to the app as it executes.

***App control and execution trace generation.*** When a preprocessed app runs in the EE, the Dalvik VM executes our bytecode instrumentation hooks (⑥). The instrumentation records app activity in an execution trace log, and then returns to the app's original (i.e., pre-instrumentation) bytecode. When

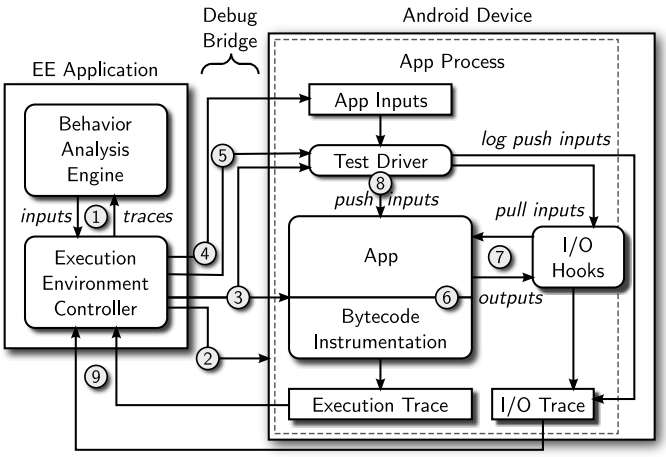


Fig. 5. Detail of app execution control and behavior monitoring process.

the app bytecode invokes an Android API call, it may trigger one of our I/O hooks (⑦). If it is a pull input hook, the value to be returned is requested from the test driver component, recorded in an I/O trace log, and then returned to the app. By injecting EE-provided pull inputs, we can indirectly control the app's behavior. If the API call is an output, the event and output value are logged to the I/O trace.

***Stimulating app behavior with push inputs.*** When the test driver is given the command to run the app, the first thing it does is enable our Android framework I/O hooks for the app process. These hooks are present for all Android Dalvik apps, but inactive until activated on a per-process basis by our test driver. The test driver then launches the app's starting *Activity* component, and waits for the completion of the `Activity.onResume()` method.

The driver then applies a series of *push inputs* to stimulate app behavior (⑧). These inputs are passed by the EE controller along with the pull inputs, and are essentially instructions to create and pass synthetic user interface and system event messages to the app. For instance, the push inputs can enter text, manipulate GUI widgets, provide GPS location updates, and simulate incoming SMS messages. In our current system, push inputs are initially identified via manual app analysis and then dynamically generated via a fuzzing framework. Ongoing work will integrate a constraint solver into the Execution Environment to select push inputs based on code coverage maximization and other metrics.

The push inputs are recorded in the I/O trace as they are applied to the app in sequence. After the push input series is over, the test driver waits a configurable delay (currently 1 s) for app activity to settle, and then kills the app.

***Execution environment control.*** The controller receives app inputs (①) and orchestrates the process of executing the app with those inputs, to ultimately return an app behavior description in the form of execution traces. The first part of this process is to start a new emulator instance (②) if one is not already running. The controller then installs the preprocessed app and test driver (③) using the Android SDK debug bridge `adb install` command. Next, the app inputs file is copied onto the emulator device (④) using `adb push`. At this point, the controller launches the test driver (⑤) using the `adb`

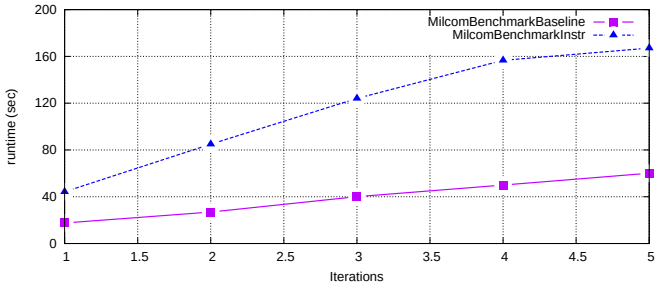


Fig. 6. Runtime performance analysis of Bytecode Instrumentation technique on Android/DVM

`shell am instrument` command. The test driver returns when the app has finished executing. Finally, the controller retrieves the execution and I/O traces using `adb pull` (⑨) and returns them (①).

## IV. Evaluation

We evaluated the overhead of our bytecode instrumentation approach by creating a benchmark Android application that performs file processing in a loop. The app exercises a majority of our instrumentation events. We evaluated the app in the EE on an Android v4.2 emulator by comparing the runtime performance of the instrumented app with a non-instrumented baseline version. As can be seen in Figure 6, the instrumented app runs anywhere from 151% to 215% slower than the baseline version.

With respect to file size, the instrumented `.apk` file grew from 29,204 bytes to 152,078 bytes, for an increase of 420%. File size overhead will differ based on the composition of instrumentation events in an app, as we apply a different transformation for each event type. Behaviorally, the event hooks never modify objects, variables, or any application state. It is worth noting, however, that this type of inline instrumentation would not be appropriate for analysis of applications with tight timing sensitivities.

## V. Discussion

Here we discuss several challenges that were addressed to get automated instrumentation and application execution working smoothly and reliably.

***Disabling Bytecode Verification.*** Several of our instrumentation events involve operand stack manipulation that fatally fails Dalvik bytecode verification. For example, instrumenting certain constructor calls can result in processing uninitialized object references, depending on where in the superconstructor call chain we are. To get around this, we disable bytecode verification in the Android emulator's Dalvik VM.

***Trampolining.*** When instrumenting method calls at the callee (`MethodCalled` event), we have the luxury of enumerating the object reference (if virtual) and any method arguments. We are often interested in reporting on API calls without having instrumented the called method body itself. We accomplish this by inserting a `MethodInvoke` event at the call site. However, by the time the `invokevirtual` is discovered in the instruction stream, the operand stack has been prepared, and there is no easy way to gain introspection into

(a)

```
1  public static void main(java.lang.String[]);
2    Code:
3  aload_1
4  bipush        19
5  invokevirtual #6 // Method foo:(I)I
```

(b)

```
1  public static void main(java.lang.String[]);
2    Code:
3  aload_1
4  bipush        19
5  invokestatic #344 // Method ___Test_foo:
6                    // (LTest;I)I
7
8  public static int ___Test_foo(Test, int);
9    Code:
10 aload_0
11 ldc_w         #338 // String foo
12 ldc_w         #339 // int 1
13 ldc_w         #340 // String (I)I
14 invokestatic #331 // Method EventHooks.
15                   // ___hookMethodInvoke:(LObject;
16                   // LString;ILString;)V
17 iload_1
18 invokestatic #147 // Method EventHooks.
19                   // ___hookMethodInvokeArg:(I)V
20 invokestatic #159 // Method EventHooks.
21                   // ___hookMethodInvokeEnd:()V
22 aload_0
23 iload_1
24 invokevirtual #6  // Method foo:(I)I
25 ireturn
```

Fig. 7. Example `MethodInvoke` trampoline for call to `public int foo(int x)` instance method. (a) Original `javap` disassembly of call site (b) Trampolined code and modified call site. Note that some type signatures have been abbreviated.

the arguments and object reference that have been pushed. We solve this problem by adding a synthetic static call that wraps the instrumented method invocation and *trampolining* to the original method. Since the static call mirrors the argument list and object reference, we can easily enumerate and hook all method attributes. On method return, we pass the return value appropriately to the original caller. To avoid instrumenting the trampolines themselves, we prepend the synthetic method name with a unique prefix that enables filtering. An example trampoline is illustrated in Figure 7.

## VI. Related Work

***Android app behavior monitoring and execution control.*** In [6], Anand et al. describe their concolic testing system for Android. Their technique uses bytecode instrumentation to record an app's field writes, which they use to generate a path constraint for the control flow induced by a given set of inputs. In addition to capturing field writes, our problem requires further bytecode modifications for generating a detailed execution trace, including method calls, exceptions, and other events. Both of our works instrument the Android framework API, but with different objectives and implementations. Whereas their purpose is to replace API calls with a symbolic evaluation, our purpose is to feed pull-inputs to the app for execution control and to log outputs for behavior monitoring.

In Dynodroid [15], the authors instrument the Android API for app input generation and execution control. Dynodroid uses instrumentation to identify *relevant* app inputs, which allows their solution to select inputs that are likely to transition

an app into subsequent states. Our work does not address the problem of input generation, and relies on an external component to provide app inputs. Their system provides push inputs exclusively, whereas our approach provides both push and pull inputs to influence app control flows. In addition to providing pull inputs, our API instrumentation is also used for app behavior monitoring.

Several approaches [16], [17], [18], [19] exist to provide a sandbox for executing potentially malicious Android apps and observing their behavior. TaintDroid [16] monitors apps using a multi-faceted model of behavior, including Dalvik VM operations, message passing, native code method calls, and file operations. In this work we do not treat native code as in scope. We chose to instrument the app bytecode, rather than the Dalvik bytecode interpreter, because good tools for bytecode instrumentation were readily available. TaintDroid monitors message passing at the kernel level and file operations via file system attributes, whereas our system uses Android framework API hooks for both.

***Bytecode instrumentation.*** Inline Java bytecode instrumentation has been leveraged to accomplish a variety of tasks including performance profiling, behavior restriction, policy enforcement, and aspect oriented programming. Here we discuss prior related work to enhance the security of mobile applications using inline binary instrumentation. [20] developed techniques that insert runtime tests into Java bytecode for the purpose of preventing certain mobile code behavior. Their approach differs from ours in that they are primarily interested in augmenting specific classes or methods with *sentinel* code to monitor and/or restrict functionality (e.g., control resource usage). [21] describe an in-vivo runtime monitoring tool chain involving application bytecode instrumentation. A primary difference here is that the runtime results are interpreted and processed on the device itself as an application is being used (e.g., policy enforcement and advertisement suppression). This contrasts with our concept of operation wherein a candidate application can be run many times with synthetic execution control and the trace results processed offline.

## VII. Conclusion

In this paper, we explored techniques for eliciting descriptions of Android app behavior via an automated execution environment. Our approach allows for execution control using push and pull inputs to exhibit the full range of app behavior, including hidden malicious functionality. We use bytecode and API instrumentation to generate a detailed description of app behavior, which is useful for analyzing potentially malicious apps. Instrumentation incurs costs of inflated app code size and slower runtime performance, which becomes increasingly important when using our approach to analyze behaviors that are highly sensitive to timing. Our system has been used to reveal attacks in independently developed challenge apps where manual source code analysis had previously failed to discover an app's true nature. Our framework is part of Droid Reasoning, Analysis, and Protection Engine (DRAPE), an integrated, semi-automated app behavior analysis system capable of eliciting behavioral traces for Android apps under various input conditions.

This material is based upon work supported by the United States Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-12-C-0097; Subcontract No. 793073.

## References

[1] K. Havelund and G. Roşu, "Testing linear temporal logic formulae on finite execution traces," Research Institute for Advanced Computer Science, Tech. Rep., May 2001.

[2] D. Lo, S.-C. Khoo, and C. Liu, "Mining past-time temporal rules from execution traces," in *Sixth International Workshop on Dynamic Analysis*, Seattle, WA, USA, Jul. 2008.

[3] R. Cathey, G. Frazier, and M. Weber, "Behavior analysis via execution path clustering," in *32nd Annual Military Communications Conference*, San Diego, CA, USA, Nov. 2013.

[4] F. Besson, T. Jensen, D. L. Métayer, and T. Thorn, "Model checking security properties of control flow graphs," *Journal of Computer Security*, vol. 9, no. 3, pp. 217–250, 2001.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, Jun. 2005.

[6] S. Anand, M. Naik, H. Yang, and M. J. Harrold, "Automated concolic testing of smartphone apps," in *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, NC, USA, Nov. 2012.

[7] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A concolic whitebox fuzzer for java," in *First NASA Formal Methods Symposium*, Moffett Field, CA, USA, Apr. 2009, pp. 121–125.

[8] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song, "Contextual policy enforcement in android applications with permission event graphs," in *20th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2013.

[9] J. Jeon, K. K. Micinski, and J. S. Foster, "SymDroid: Symbolic execution for Dalvik bytecode," Dept. of Computer Science, University of Maryland, College Park, Tech. Rep., Jul. 2012.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[11] dex2jar Developers, "dex2jar: Tools to work with Android .dex and Java .class files," http://code.google.com/p/dex2jar, Feb. 2013.

[12] apktool Developers, "apktool: A tool for reverse engineering Android apk files," http://code.google.com/p/android-apktool, Mar. 2013.

[13] G. B. Tim Lindholm, Franke Yellin and A. Buckley, "Java virtual machine specification," rl-http://docs.oracle.com/javase/specs/jvms/se7/html/index.html, 2013.

[14] Apache Commons, "BCEL: Byte code engineering library," http://commons.apache.org/proper/commons-bcel, Oct. 2011.

[15] A. MacHiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," Georgia Tech Program Analysis Group, Tech. Rep., Dec. 2012.

[16] 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 *9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, Canada, Oct. 2010.

[17] International Secure Systems Lab, "Andrubis: A tool for analyzing unknown Android applications," http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2, Jun. 2012.

[18] P. Lantz and K. Yang, "droidbox: Android application sandbox," http://code.google.com/p/droidbox, 2012.

[19] H. Lockheimer, "Android and security," Google Mobile Blog. olinkurlhttp://googlemobile.blogspot.com/2012/02/android-and-security.html, Feb. 2012.

[20] A. Chander, J. C. Mitchell, and I. Shin, "Mobile code security by Java bytecode instrumentation," in *In 2001 DARPA Information Survivability Conference & Exposition (DISCEX)*, 2001, pp. 1027–1040.

[21] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. L. Traon, "Improving privacy on Android smartphones through in-vivo bytecode instrumentation," *CoRR*, vol. abs/1208.4536, 2012.