# Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-Point Saturation

Shuying Liang, Andrew W. Keep, Matthew Might,
Steven Lyde, Thomas Gilray, and Petey Aldous
University of Utah
{liangsy,akeep,might,lyde,tgilray,petey.aldous}@cs.utah.edu

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

## ABSTRACT

Sound malware analysis of Android applications is challenging. First, object-oriented programs exhibit highly interprocedural, dynamically dispatched control structure. Second, the Android programming paradigm relies heavily on the asynchronous execution of multiple entry points. Existing analysis techniques focus more on the second challenge, while relying on traditional analytic techniques that suffer from inherent imprecision or unsoundness to solve the first.

We present Anadroid, a static malware analysis framework for Android apps. Anadroid exploits two techniques to soundly raise precision: (1) it uses a pushdown system to precisely model dynamically dispatched interprocedural *and* exception-driven control-flow; (2) it uses *Entry-Point Saturation* (EPS) to soundly approximate all possible interleavings of asynchronous entry points in Android applications. (It also integrates static taint-flow analysis and least permissions analysis to expand the class of malicious behaviors which it can catch.) Anadroid provides rich user interface support for human analysts which must ultimately rule on the "maliciousness" of a behavior.

To demonstrate the effectiveness of Anadroid's malware analysis, we had teams of analysts analyze a challenge suite of 52 Android applications released as part of the Automated Program Analysis for Cybersecurity (APAC) DARPA program. The first team analyzed the apps using a version of Anadroid that uses traditional (finite-state-machine-based) control-flow-analysis found in existing malware analysis tools; the second team analyzed the apps using a version of Anadroid that uses our enhanced pushdown-based control-flow-analysis. We measured machine analysis time, human analyst time, and their accuracy in flagging malicious applications. With pushdown analysis, we found statistically significant ($p < 0.05$) decreases in time: from 85 minutes per app to 35 minutes per app in human plus machine analysis time; and statistically significant ($p < 0.05$) increases in accuracy with the pushdown-driven analyzer: from 71% correct identification to 95% correct identification.


*SPSM'13,* November 8, 2013, Berlin, Germany.

ACM 978-1-4503-2491-5/13/11 ...$15.00.
http://dx.doi.org/10.1145/2516760.2516769.

## Categories and Subject Descriptors

D.2.0 [**SOFTWARE ENGINEERING**]: Protection Mechanisms; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—*Program analysis,Operational semantics*

## General Terms

Languages, Security

## Keywords

static analysis; taint analysis; abstract interpretation; pushdown systems; malware detection

## 1. INTRODUCTION

Google's Android operating system is the most popular mobile platform, with a 52.5% share of all smartphones [21]. Due to Android's open application development community, more than 400,000 apps are available with 10 billion cumulative downloads by the end of 2011 [20].

While most of these third-party apps have legitimate reasons to access private data, utilize the Internet, or make changes to local settings and file storage, the permissions provided by Android are too coarse, allowing malware to slip through the cracks. For instance, an app that needs to read information from only a specific website and access to GPS information must, necessarily, be granted full read/write access to the entire Internet, allowing it to maliciously leak location information. In another example, a note-taking application that writes notes to the file system can use the file system permissions to wipe out SD card files when a trigger condition is met. Meanwhile, a task manager that legitimately requires every permission available can be benign. To understand application behaviors like these before running the program, we need to statically analyze the application, tracking what data is accessed, where sensitive data flows, and what operations are performed with the data, *i.e.*, determine whether data is tampered with.

However, static malware analysis for Android apps is challenging. First, there is the general challenge of analyzing object-oriented programs where the state of the art is finite state based. Specifically, traditional analysis regimes like $k$-CFA [40] and its many variants implicitly or explicitly finitize the stack during abstraction. In effect, analyzers carve up dynamic return points and exception-handling points among a finite number of abstract return contexts. When two dynamic return points map to the same abstract context, the analyzer loses the ability to distinguish them.

Second, there is the domain-specific challenge from the Android programming paradigm, which is event-driven with multiple entry points and asynchronous interleaved execution. A typical strategy of existing static malware detection usually relies upon an existing analytic framework, *i.e.*, CHEX [31] depends on WALA [24], Woodpecker depends on Soot [41], and various *ad hoc* techniques proposed to deal with the problem: either by heuristic aborting of the paths unsoundly [31], or combining dynamic execution to eliminate paths not appearing at run time [44]. One problem with this strategy is the inherent imprecision of the underlying analytic framework means imprecise malware analysis on top of it. Another problem is that many static malware analyzers go unsound in order to handle the large number of permutations for multiple entry points for the sake of efficiency. This means the static analyzer can no longer prove the absence of behaviors, malicious or otherwise.

In this paper, we describe Anadroid, a generic static malware analyzer for Android apps. Anadroid depends on two integrated techniques: (1) it is the first static analysis using higher-order pushdown control-flow-analysis for an object-oriented programming language; (2) it uses *Entry-Point Saturation* (EPS) to soundly approximate interleaving execution of asynchronous entry points in Android apps. We also integrate static taint flow analysis and least permissions analysis as application security analysis. In addition, Anadroid provides a rich user interface to support human-in-the-loop malware analysis. It allows user-supplied predicates to filter and highlight analysis results.

To demonstrate the effectiveness of Anadroid's malware analysis, we evaluate a challenge suite of 52 Android apps released as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program. We compare a malware analyzer driven by finite-state control-flow-analysis with our model with respect to accuracy and analysis time. We found that pushdown-driven malware analysis leads to statistically significant improvements in both aspects over traditional static analysis methods (which use finite-state methods like $k$-CFA to handle program control [40]).

The remainder of this paper is organized as follows. We illustrate the challenges via a condensed example malicious application in Section 2. Then we present our solutions in the subsequent two sections, where Section 3 presents the foundational pushdown control-flow analysis and Section 4 details *Entry-point saturation* (EPS). Based on pushdown control-flow analysis and EPS, we integrate static taint flow analysis and least permissions analysis into our analytic framework, as presented in Section 5. Section 6 presents our tool and Section 7 evaluates the effectiveness of our technique by comparing our results with results from finite state-based analysis. Case studies summarizing the vulnerabilities we have found are described in Section 8. Section 9 presents related work and Section 10 concludes.

## 2. CHALLENGES OF STATIC MALWARE ANALYSIS FOR ANDROID

The program *Kitty.java* is adapted from an Android malware application developed and released by a red team on the DARPA APAC program. It exfiltrates location data from pictures stored on the users phone to a malicious site (on Lines 19 and 33) or posts the information (on Lines 36–38) via an Android Intent if the first attempt fails. This example helps illustrate the two challenges of static malware detection for Android programs.

```
1  public class KittyQuote extends Activity {
2    String img = "k155"; // kitty image
3    // other fields ...
4    public void onCreate(Bundle savedState) {
5      super.onCreate(savedState);
6      // build quote list and other initialization
7    }
8    public String getKitty() {
9      // access "DCIM/Camera" and use ExifInterface
10     //  to exfiltrate location
11   }
12   public void aboutButton(View view) {
13     // display normal information
14     String website = "http://www.catquotes.com";
15     startActivity(
16       new Intent(Intent.ACTION_VIEW,
17             Uri.parse(website)));
18     try {
19       new SendOut().execute(website);
20     } catch (Exception e) { }
21   }
22   public void nextButton(View view) {
23     img = getKittey(); // store loc info
24   }
25   public void prevButton(View view) {
26     img = getKittey();
27   }
28   public void kittyQuoteButton(View view) {
29     // display kitty quote as toast message
30     // ... and send out location info to a website
31     String url = "http://www.catquotes.com?" + img;
32     try {
33       new SendOut().execute(url);
34     } catch (Exception e) {
35       // if network fails, not giving up
36       startActivity(
37         new Intent(Intent.ACTION_VIEW,
38               Uri.parse(url)));
39     }
40   }
41   class SendOut extends AsyncTask {
42     protected Void doInBackground(URI... uris) {
43       HttpGet hg = new HttpGet(uris[0]);
44       try {
45         new DefaultHttpClient().execute(hg);
46       } catch (Exception e) { }
47       return null;
48     }
49     // ...
50   }
51 }
```

**Kitty.java**

*Fundamental challenge: imprecision induced by finite-based analytic model for object-oriented programs.*

Android apps are written in Java, and it can be difficult to statically produce a precise control flow graph of the pro-

gram, particularly in the presence of exceptions. Many existing static analyzers for Java programs are based on finite-state-machine analysis, *i.e.* $k$-CFA or $k$-object sensitivity, which have limited analytic power for analyzing dynamic dispatch or exceptions precisely. Analyzers built using these techniques have a greater rate of false positives and false negatives due to this imprecision.

For instance, in the example, both *aboutButton* and *kittyQuoteButton* create a non-blocking background thread via the class *SendOut* (Lines 19 and 33) wrapped inside a try block. In the *aboutButton* code, normal exception handling occurs at Line 20. In the *kittyQuoteButton* code, the exception handler attempts a second malicious action in Lines 36–38 using an Android web view activity to send out location information.

This illustrates one of the difficulties in making a precise analysis of the program, where the flow of the exception is difficult to track, leading to false positives or false negatives. Specifically, a finite-state-based analyzer cannot determine which exception handler block will be used when an exception is thrown at Line 19. It can (incorrectly) conclude that the handler block in Lines 36–38 will be invoked. This causes a false positive where Line 19 is identified as potentially leading to the malicious behavior in Lines 36–38. In the same way, a finite-state-based analyzer cannot determine which exception handler block will be used when an exception is thrown from Line 33. It may determine that the handler block on Line 19 is the appropriate exception handler block. This leads to a false negative, where the malicious behavior resulting from an exception at Line 33 is missed. The pushdown control-flow analysis model can precisely track exception handling, so our analyzer identifies the correct handler block (Line 20) where no malicious behavior exists.

In fact, the fundamental imprecision caused by spurious control flows due to exceptions has been reported before. Fu *et al.* [17] report this problem when testing Java server apps, and Bravenboer *et al.* [4, 5] describe the need to combine points-to and exception analysis to attempt to regain some of this precision. Unlike approaches using finite-state-based control flow analysis, the pushdown control-flow analysis used as the foundation for our malware analyzer can precisely match both normal and exception return flows achieving lower false positive and false negative rates.

### *Domain specific challenge: Permutations of Asynchronous multi-entry-points.*

The second challenge in analyzing Android apps is caused by the asynchronous multiple entry points into an Android application. The Android framework allows developers to create rich, responsive, and powerful apps by requiring developers to organize their code into components. Each component type serves a different purpose: (1) activities for the main user-interface, (2) services for non-blocking code or remote processes, (3) content providers for managing application data, and (4) broadcast receivers to provide system-wide announcements. Applications can register various component handlers, either explicitly in code or through a resource file (res/layout/*filename*.xml). Whenever an event occurs, the callbacks for the event are invoked asynchronously, potentially interleaving their execution with those in other components. Different apps can also invoke each other by exposing functionality via an *Intent* at both the application and component level[1]. Unlike an application with a single entry point, static analysis for an Android application must explore all permutations of these asynchronous entry points. Analyzing all permutations can greatly increase the expense of the analysis. As a result many analyzers use an unsound approximation that can lead to false negatives.

Our example illustrates these problems. First, not all of the callback methods are explicitly registered in code, like the *onCreate* method of *KittyQuote* and the *doInBackground* method of *SendOut*, *aboutButton*, *nextButton*, *prevButton*, and *kittyQuoteButton*[2] are registered in an XML layout resource file as follows.

```
<Button
        android:id="@+id/button2"
        ...
        android:onClick="prevButton" />
```

Second, malicious behaviors can be triggered from an entry point or series of entry points. For instance, if the *prevButton* or *nextButton* methods are called before the *aboutButton* or *kitteyQuoteButton* methods, the application will leak location data gathered from the Exif data of pictures on the device. In order to avoid missing malicious behavior, while still performing the analysis efficiently, we need a way to approximate the possible permutations of the asynchronous multi-entry points without loosing soundness. This means we cannot use heuristic pruning, but we also do not want to use a dynamic analysis, since we hope to analyze the program before we attempt to run it. This leads to our second contribution, Entry-Point Saturation (EPS), which can be used to analyze multi-entry points in a sound way.

To summarize the relationship between the challenges, an unsound approximation of asynchronous entry points can miss malicious behavior, while any imprecision in the underlying control-flow analysis can result in an analysis that misses malicious behavior in programs. This problem is further exacerbated by additional highly dynamic dispatched inter-procedural control flows caused by permutations of entry points. We need to address both challenges to ensure our analyzer does not miss malicious behavior in the program. The next two sections describe our solutions to these challenges.

## 3. PUSHDOWN FLOW ANALYSIS FOR OO

In this section, we describe our pushdown control-flow-analysis for object-oriented programs. We present the analysis as a small-step semantic analysis following the style of Felleisen *et al.* [12] and more recently Van Horn *et al.* [42]. We first define an object-oriented bytecode language closely modeled on Dalvik bytecode[3]. Our language dispenses with some of the bytecode size optimizations that the Dalvik bytecode uses to shrink the size of programs, allowing the analyzer to treat Dalvik instructions with similar functionality as a single instruction. Our language also includes explicit line number instructions to allow it to more easily relate malicious behavior in the bytecode to the original source code.

[1]Apps sharing the same Linux user ID are also able to access each other's files.
[2]In code, these are implemented with the *onClick* method in an *OnClickListener*.
[3]Java code in Android apps are compiled to Dalvik bytecode.

Then we develop our pushdown control-flow analysis for this language[4].

## 3.1 Syntax

The syntax of the bytecode language is given in Figure 1. Statements encode individual actions for the machine; Complex expressions encode expressions with possible side effects or non-termination; and atomic expressions encode atomically computable values. There are four kinds of names: Reg for registers, ClassName for class names, FieldName for field names, and MethodName for method names. There are two special register names: ret, which holds the return value of the last function called, and exn, which holds the most recently thrown exception.

The syntax is a straightforward abstraction of Dalvik bytecode, but it is worth examining statements related to exceptions in more detail:

- (throws *class-name* ...) indicates that a method can throw one of the named exception classes,
- (push-handler *class-name label*) pushes an exception handler frame on the stack that will catch exceptions of type *class* and divert execution to *label*, and
- (pop-handler) pops the initial exception handler frame off the stack.

With respect to a given program, we assume a syntactic metafunction $\mathcal{S} : \mathsf{Label} \to \mathsf{Stmt}^*$, which maps a label to the sequence of statements that start with that label.

## 3.2 Abstract Semantics

With the language in place, the next step is to define the concrete semantics of the language, providing the most accurate interpretation of program behaviors. However, the concrete semantics cannot be used directly for static analysis, since it is not computable. Fortunately, we can adapt the technique of abstracting abstract machines [42] to derive a sound abstract semantics for the Dalvik bytecode using an abstract CESK machine.

States of this machine consist of a series of statements, a frame pointer, a heap, and a stack. The evaluation of a program is defined as the set of *abstract* machine configurations reachable by an abstraction of the machine transition relation. Largely, abstract evaluation is defined as $\hat{\mathcal{E}} : \mathsf{Stmt}^* \to \mathcal{P}(\widehat{Conf})$, where $\hat{\mathcal{E}}(\vec{s}) = \left\{\hat{c} : \hat{\mathcal{I}}(\vec{s}) \leadsto^* \hat{c}\right\}$. Therefore, abstract evaluation is defined by the set of configurations reached by the reflexive, transitive closure of the $(\leadsto)$ relation, which shall be defined in Section 3.2.1.

### *Abstract configuration-space.*

Figure 2 details the abstract configuration-space for this abstract machine. We assume the natural element-wise, point-wise, and member-wise lifting of a partial order across this state-space.

To synthesize the abstract state-space, we force frame pointers and object pointers (and thus addresses) to be a finite set. When we restrict the set of addresses to a finite set, the machine may run out of addresses to allocate, and

[4]Although Android apps can include native code, our analyzer only handles Dalvik bytecode. Native code in the Android API is modeled directly in the analyzer.

$$
\begin{aligned}
\textit{program} &::= \textit{class-def}\ \ldots \\
\textit{class-def} &::= (\textit{attribute}\ \ldots\ \mathsf{class}\ \textit{class-name}\ \mathsf{extends}\ \textit{class-name} \\
&\qquad (\textit{field-def}\ \ldots)\ (\textit{method-def}\ \ldots)) \\
\textit{field-def} &::= (\mathsf{field}\ \textit{attribute}\ \ldots\ \textit{field-name}\ \textit{type}) \\
\textit{method-def} &::= (\mathsf{method}\ \textit{attribute}\ \ldots\ \textit{method-name}\ (\textit{type}\ \ldots)\ \textit{type} \\
&\qquad (\mathsf{throws}\ \textit{class-name}\ \ldots)\ (\mathsf{limit}\ n)\ s\ \ldots) \\
s \in \mathsf{Stmt} &::= (\mathsf{label}\ \textit{label}) \mid (\mathsf{nop}) \mid (\mathsf{line}\ \textit{int}) \mid (\mathsf{goto}\ \textit{label}) \\
&\mid (\mathsf{if}\ æ\ (\mathsf{goto}\ \textit{label})) \mid (\mathsf{assign}\ \textit{name}\ [æ \mid \textit{ce}]) \\
&\mid (\mathsf{field\text{-}put}\ æ_o\ \textit{field-name}\ æ_v) \\
&\mid (\mathsf{field\text{-}get}\ \textit{name}\ æ_o\ \textit{field-name}) \\
&\mid (\mathsf{push\text{-}handler}\ \textit{class-name}\ \textit{label}) \\
&\mid (\mathsf{pop\text{-}handler}) \mid (\mathsf{throw}\ æ) \mid (\mathsf{return}\ æ) \\
æ \in \mathsf{AExp} &::= \mathsf{this} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \mathsf{void} \\
&\mid \textit{name} \mid \textit{int} \\
&\mid (\textit{atomic-op}\ æ \ldots æ) \\
&\mid \mathsf{instance\text{-}of}(æ, \textit{class-name}) \\
\textit{ce} &::= (\mathsf{new}\ \textit{class-name}) \\
&\mid (\textit{invoke-kind}\ (æ \ldots æ)(\textit{type}\ \ldots)) \\
\textit{invoke-kind} &::= \mathsf{invoke\text{-}static} \mid \mathsf{invoke\text{-}direct} \\
&\mid \mathsf{invoke\text{-}virtual} \mid \mathsf{invoke\text{-}interafce} \\
&\mid \mathsf{invoke\text{-}super} \\
\textit{type} &::= \textit{class-name} \mid \mathsf{int} \mid \mathsf{byte} \mid \mathsf{char} \mid \mathsf{boolean} \\
\textit{attribute} &::= \mathsf{public} \mid \mathsf{private} \mid \mathsf{protected} \\
&\mid \mathsf{final} \mid \mathsf{abstract}
\end{aligned}
$$

*name* is an infinite set of frame-local variables or registers in the parlance of Dalvik byte code.

**Figure 1: An object-oriented bytecode adapted from the Android specification [34].**

$$
\begin{aligned}
\hat{c} \in \widehat{Conf} &= \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Kont} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \widehat{Val} \\
\hat{a} \in \widehat{Addr} &= \widehat{RegAddr} + \widehat{FieldAddr} \\
\widehat{ra} \in \widehat{RegAddr} &= \widehat{FramePointer} \times \mathsf{Reg} \\
\widehat{fa} \in \widehat{FieldAddr} &= \widehat{ObjectPointer} \times \mathsf{FieldName} \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^* \\
\hat{\phi} \in \widehat{Frame} &= \widehat{CallFrame} + \widehat{HandlerFrame} \\
\widehat{\chi} \in \widehat{CallFrame} &::= \mathbf{fun}(\hat{fp}, \vec{s}) \\
\widehat{\eta} \in \widehat{HandlerFrame} &::= \mathbf{handle}(\textit{class-name}, \textit{label}) \\
\hat{d} \in \widehat{Val} &= \mathcal{P}\left(\widehat{ObjectValue} + \widehat{String} + \widehat{\mathcal{Z}}\right) \\
\hat{ov} \in \widehat{ObjectValue} &= \widehat{ObjectPointer} \times \mathsf{ClassName} \\
\hat{fp} \in \widehat{FramePointer} &\text{ is a finite set of frame pointers} \\
\widehat{op} \in \widehat{ObjectPointer} &\text{ is a finite set of object pointers.}
\end{aligned}
$$

**Figure 2: The abstract configuration-space.**

when it does, the pigeon-hole principle will force multiple abstract values to reside at the same address. As a result, the range of the $\widehat{Store}$ becomes a power set in the abstract configuration-space. Crucially, in this machine, the stack is left unbounded, unlike the final step in the abstracting abstract machine approach. This enables us to faithfully model both normal function call and return and exception throw and catch handling intraprocedurally and interprocedurally. In the next section we detail the essence of the abstract CESK machine, which is one of the main contributions of this work.

### 3.2.1 Abstract transition relation

The machine relies on helper functions to evaluate atomic expressions and look up field values:

- $\hat{\mathcal{I}} : \mathsf{Stmt}^* \to \widehat{Conf}$ injects a sequence of instructions into a configuration:
$$\hat{c}_0 = \hat{\mathcal{I}}(\vec{s}) = (\vec{s}, \hat{fp}_0, [], \langle\rangle).$$
- $\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{Store} \rightharpoonup \widehat{Val}$ evaluates atomic expressions:
$$\hat{\mathcal{A}}(name, \hat{fp}, \hat{\sigma}) = \sigma(\hat{fp}, name) \quad \text{[variable look-up]}.$$
- $\hat{\mathcal{A}_{\mathcal{F}}} : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{Store} \times \mathsf{FieldName} \rightharpoonup \widehat{Val}$ looks up fields:
$$\hat{\mathcal{A}_{\mathcal{F}}}(æ_o, \hat{fp}, \hat{\sigma}, \textit{field-name}) = \bigsqcup \hat{\sigma}(\widehat{op}, \textit{field-name})$$
$$\text{where } (\widehat{op}, \textit{class-name}) \in \hat{\mathcal{A}}(æ_o, \hat{fp}, \hat{\sigma}).$$

The abstract transition relation $(\rightsquigarrow) \subseteq \widehat{Conf} \times \widehat{Conf}$ has rules to soundly model all possible concrete executions of a bytecode program. In the subsequent subsections, we illustrate the rules that involve objects, function calls, and exceptions, omitting more obvious rules to save space.

#### *New object creation.*

Creating a new object allocates a potentially non-fresh address and joins the newly initialized object to other values residing at this store address.

$$\overbrace{(\llbracket(\mathsf{assign}\ name\ (\mathsf{new}\ \textit{class-name})) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}}$$
$$\rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}'', \hat{\kappa}), \text{ where}$$
$$\widehat{op}' = \widehat{allocOP}(\hat{c})$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto (\widehat{op}', \textit{class-name})]$$
$$\hat{\sigma}'' = \widehat{initObject}(\hat{\sigma}', \textit{class-name}),$$

The helper function $\widehat{initObject} : \widehat{Store} \times \mathsf{ClassName} \rightharpoonup \widehat{Store}$ initializes the object's fields.

#### **Instance field reference/update.**

Referencing a field uses $\hat{\mathcal{A}_{\mathcal{F}}}$ to lookup the field values and joins these values with the values at the store location for the destination register:

$$(\llbracket(\mathsf{field\text{-}get}\ name\ æ_o\ \textit{field-name}) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa})$$
$$\rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}), \text{ where}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \hat{\mathcal{A}_{\mathcal{F}}}(æ_o, \hat{fp}, \hat{\sigma}, \textit{field-name})].$$

Updating a field first determines the abstract object values from the store, extracts the object pointer from all the possible values, then pairs the object pointers with the field name to get the field address, and finally *joins* the new values to those found at this store location:

$$(\llbracket(\mathsf{field\text{-}put}\ æ_o\ \textit{field-name}\ æ_v) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa})$$
$$\rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}) \text{ where}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\widehat{op}, \textit{field-name}) \mapsto \hat{\mathcal{A}}(æ_v, \hat{fp}, \hat{\sigma})]$$
$$(\widehat{op}, \textit{class-name}) \in \hat{\mathcal{A}}(æ_o, \hat{fp}, \hat{\sigma}).$$

#### *Method invocation.*

This rule involves all four components of the machine. The abstract interpretation of non-static method invocation can result in the method being invoked on a *set* of possible objects, rather than a single object as in the concrete evaluation. Since multiple objects are involved, this can result in different method definitions being resolved for the different objects. The method is resolved[5] and then applied as follows:

$$\overbrace{(\llbracket(\textit{invoke-kind}\ (æ_0 \ldots æ_n)\ (\textit{type}_0 \ldots \textit{type}_n))\rrbracket : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa})}^{\hat{c}}$$
$$\rightsquigarrow \widehat{applyMethod}(m, \vec{æ}, \hat{fp}, \hat{\sigma}, \hat{\kappa}),$$

where the function $\widehat{applyMethod}$ takes a method definition, arguments, a frame pointer, a store, and a new continuation and produces the next configuration:

$$\widehat{applyMethod}(m, \vec{æ}, \hat{fp}, \hat{\sigma}, \hat{\kappa}) = (\vec{s}, \hat{fp}', \hat{\sigma}', (\hat{fp}, \vec{s}) : \hat{\kappa}), \text{ where}$$
$$\hat{fp}' = \widehat{allocFP}(\hat{c})$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}', name_i) \mapsto \hat{\mathcal{A}}(æ_i, \hat{fp}, \hat{\sigma})].$$

#### **Procedure return.**

Procedure return restores the caller's context and *extends* the return value in the dedicated return register, ret.

$$(\llbracket(\mathsf{return}\ æ) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \mathbf{fun}(\hat{fp}', \vec{s}') : \hat{\kappa}) \rightsquigarrow (\vec{s}', \hat{fp}', \hat{\sigma}', \hat{\kappa}),$$
where $\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}', \mathsf{ret}) \mapsto \hat{\mathcal{A}}(æ, \hat{fp}, \hat{\sigma})]$.

If the top frame is an exception handler (**handle**) frame, the abstract interpreter pops until the top-most frame is a function call (**fun**) frame:

$$(\llbracket(\mathsf{return}\ æ)\rrbracket : \vec{s}, \hat{fp}, \hat{\sigma}, \mathbf{handle}(\textit{class-name}\ \textit{label}) : \hat{\kappa})$$
$$\rightsquigarrow (\llbracket(\mathsf{return}\ æ)\rrbracket : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}).$$

#### *Pushing and popping handlers.*

Pushing and popping exception handlers is straightforward:

$$(\llbracket(\mathsf{push\text{-}handler}\ \textit{class-name}\ \textit{label})\rrbracket : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa})$$
$$\rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}, \mathbf{handle}(\textit{class-name}\ \textit{label}) : \hat{\kappa})$$

[5]Since the language supports inheritance, method resolution requires a traversal of the class hierarchy. This traversal follows the expected method and is omitted here so we can focus on the abstract rules.

$$([\![(\textsf{pop-handler})]\!] : \vec{s}, \hat{fp}, \hat{\sigma}, \textbf{handle}(\textit{class-name label}) : \hat{\kappa}) \rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}).$$

*Throwing and catching exceptions.*

The throw statement pops entries off the stack until it finds a matching exception handler:

$$([\![(\textsf{throw } æ)]\!] : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow \widehat{handle}(æ, \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}),$$

where the function $\widehat{handle} : \textsf{AExp} \times \textsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Kont} \rightharpoonup \widehat{Conf}$ behaves like its concrete counterpart when the top-most frame is a compatible handler:

$$\widehat{handle}(æ, \vec{s}, \hat{fp}, \hat{\sigma}, \textbf{handle}(\textit{class-name}' \ \textit{label}) : \hat{\kappa}') = (\mathcal{S}(\textit{label}), \hat{fp}, \hat{\sigma} \sqcup [(\hat{fp}, \textsf{exn}) \mapsto (\widehat{op}, \textit{class-name})], \hat{\kappa}').$$

Otherwise, it pops a frame:

$$\widehat{handle}(æ, \vec{s}, \hat{fp}, \hat{\sigma}, \textbf{handle}(\_, \_) : \hat{\kappa}') = ([\![(\textsf{throw } æ)]\!] : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}')$$
$$\widehat{handle}(æ, \vec{s}, \hat{fp}, \hat{\sigma}, \textbf{fun}(\_, \_) : \hat{\kappa}') = ([\![(\textsf{throw } æ)]\!] : \vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}').$$

Executing the analysis consists of solving for the reachable control states of the implicit pushdown system in the abstract semantics. To compute the reachable control states of this pushdown system, we employ standard reachability algorithms from Reps' *et al.* [3, 26, 36, 37] work on pushdown-reachability.

# 4. ENTRY-POINT SATURATION (EPS)

The pushdown control-flow analysis described in the previous section provides the foundation for our object-oriented analysis. Now, we shift our focus to addressing the domain specific challenge: asynchronous multiple entry points using Entry-Point Saturation (EPS) and integrating it into pushdown control-flow analysis.

An entry point is defined as any point through which the system can enter the user application [19]. This means that any method that can be invoked by the framework is an entry point. Since there is no single "main" method, the static analysis must first identify the entry points in the program. Entry-point discovery is not a challenge, however, since they are defined by the Android framework. We briefly summarize possible entry-points here.

There are three categories of entry points, which we generalize as *unit*s. First, all the callback events of components defined by the Android framework are entry points. These entry points are designed to be overridden by application code and are invoked and managed by the framework for the purpose of component life cycle management, coordinating among different components, and responding to user events which are themselves defined to be asynchronous. Second, asynchronous operations that can be executed in the background by the framework are considered to be entry points. These include the *AsyncTask* class for short background operations, the *Thread* class for longer operations, and the *Handler* class for responding to messages. Finally, all event handlers in Android UI widgets, such as button, check box, etc. are entry points. Each UI widget has standard event listeners defined, where the event handler interface methods

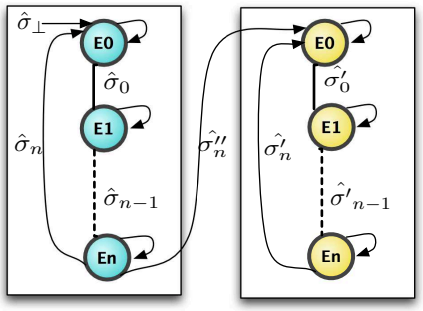


**Figure 3: Entry-Point Saturation**

are meant to be implemented by application code. Entry points from the first two categories are found by parsing Dalvik bytecode and organized into a set attached to the corresponding *unit*. Entry points from the final category can also be defined in resource layout files *res/layout/file-name.xml*, as illustrated in Section 2. These entry points can be obtained by parsing resource files before analysis.

After the entry point set for each unit is determined the real challenge begins. If we want to do a sound analysis, all the permutations of the entry points need to be considered. Complicating things further, entry points in the second category involve threads so interleaved execution of these entry points is possible. Other analyzers, such as CHEX, deal with this complexity using *app-splitting*, which is unsound and cannot model the threading case. Here, we present a sound and efficient technique which directly relies on the underlying pushdown analytic engine presented in Section 3. Figure 3 illustrates the process.

For each entry point $E_i$ in a *unit* (represented as a square), we compute the fixed point via pushdown analysis (refer to Section 3.2.1). After one round of computation the analysis returns a set of configurations. We then use the *configuration widening* technique from Might [33] on the set of configurations to generate a widened $\hat{\sigma}_i$. This abstract component will be "inherited" by the next entry point in the fixed point computation. The process repeats until the last entry point finishes its computation in a *unit*. In this way, the unit has reached a fixed point. The next step computes the fixed point between *unit*s. This is computed in a similar fashion to the intra-unit fixed point computation, so the widened result $\hat{\sigma}'_n$ from the previous unit (the left square) participates in the reachability analysis of the next unit (the right square).

EPS soundly models all permutations of entry points and their interleaving execution by passing the store resulting from analyzing one entry-point as the initial store for the next entry-point. This gives the "saturated" store for a given component. The saturated store for a component is similarly passed as the initial store for the next component. The final store models every execution path without needing to enumerate every combination.

EPS has several advantages:

1. It computes the fixed point from bottom up, intra-entry point, inter-entry point, and inter-unit, in an efficient manner and significantly simplifies the static analysis;

2. It not only computes all permutations of entry points, but also interleaving executions of them;

3. It is sound and easy to prove, because it is based on widening on the abstract configurations, where the soundness proof follows the same structure as shown by Might [33] and so we omit it here.

Applying this technique to the example in Section 2, we can see that all execution sequences where location information is read (*nextButton* and *prevButton*) to where it is leaked through the ACTION_VIEW Intent or the *doInBackground* method of the *SendOut* AsyncTask are covered and analyzed.

# 5. TAINT FLOW ANALYSIS AND LEAST PERMISSIONS ANALYSIS

As a whole, the previous two sections have described a solid foundation for static malware analysis. This section demonstrates two possible applications that build upon the foundation for specific security analyses.

## 5.1 Pushdown taint flow analysis with entry-point saturation

For any mobile application, one of the biggest concerns is leakage or tampering with private data. We can easily instrument our framework to perform taint flow analysis to detect these malicious behaviors. The idea is adapted from early work by Liang *et al.* [30], but has been enriched with Android sensitive data categories as taint values. The taint flow analysis within our pushdown-based analysis framework is done by modifying the abstract configuration in Figure 2 as:

$$\widehat{Conf} = \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{\boldsymbol{TaintStore}} \times \widehat{Kont}$$

and adding the definition for $\widehat{\boldsymbol{TaintStore}}$, which is a flat lattice across all taint values:

$$\hat{\sigma}^T \in \widehat{\boldsymbol{TaintStore}} = \widehat{Addr} \rightharpoonup \widehat{Val}$$

$$\hat{d} \in \widehat{Val} = \mathcal{P}\left(\widehat{ObjectValue} + \widehat{String} + \hat{\mathcal{Z}} + TaintVal\right)$$

$$\begin{aligned} TaintVal = {} & \text{Location} + \text{FileSystem} + \text{Sms} + \text{Phone} \\ & + \text{Voice} + \text{DeviceID} + \text{Network} + \text{ID} \\ & + \text{TimeOrDate} + \text{Display} + \text{Reflection} \\ & + \text{IPC} + \text{BrowserBookmark} + \text{SdCard} \\ & + \text{BrowserHistory} + \text{Thread} + \text{Picture} \\ & + \text{Contact} + \text{Sensor} + \text{Account} + \text{Media}. \end{aligned}$$

All the transition rules have an additional $\hat{\sigma}^T$ added, operations resemble the ones on $\hat{\sigma}$, except that the taint values are *monotonically* propagated through the abstract semantics. For example, in a function call, tainted values in arguments are bound to the formal parameters of the functions (using the $\sqcup$ operation in the taint store $\hat{\sigma}^T$), returning abstract taint values bound to a register address with ret, etc. The detailed formalism is omitted to save space.

The taint propagation analysis is not limited to detecting sensitive data leakage or tampering. As we discuss in Section 8, it can help analysts to find malicious behaviors, such as when SMS messages are blocked by a trigger condition or a limited resource is consumed, such as the local file system on the device being filled.

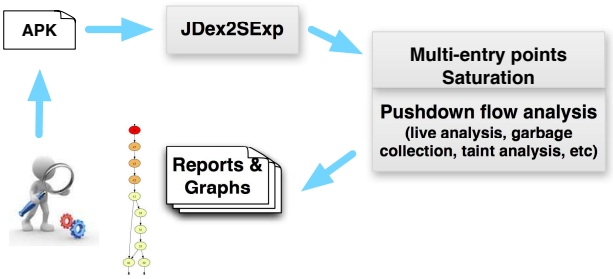


**Figure 4: Software Architecture**

## 5.2 Least permissions analysis

Our analysis framework can also detect malicious behaviors in the presence of zero-permissions, as the motivating example demonstrates. However, for apps that request permissions, it is highly desirable to analyze how the apps use the permissions, since over-privileged apps can easily be exploited by other apps in the Android framework. It is straightforward to determine this situation in our framework by instrumenting it with knowledge about permissions. Specifically, we use the data set from PScout [2] to annotate each API call with permissions that are required for usage. During reachability analysis, permissions are inferred and collected. The reached permissions are determined during the analysis. When the analysis finishes these permissions can be compared with the set of permissions requested in the manifest file. This allows us to statically determine whether an application is over-privileged.

# 6. THE TOOL: ANADROID

Anadroid is built on the principles illustrated in the previous sections. Figure 4 briefly sketches the software architecture.

Anadroid has the following features:

- It consumes off-the-shelf Android application packages (files with suffix .apk). In Figure 4 *JDexSExp* extracts the .dex file by invoking apktool [1] and then disassembles binaries and generates an S-expression IR based on the smali [16] format;

- It enables human-in-the-loop analysis, by providing a rich user interface for an analyst to configure the analyzer, *i.e.*setting the $k$ value, configuring abstract garbage collection to be used or not, specifying predicates over the state space, etc., to trade off precision and performance, as well as semantic predicates to search analysis results;

- It generates various reports and state graphs. The following three reports are included: (1) Least permissions presents which permissions are requested by an app and which permissions are inferred by Anadroid, reporting whether the app requests more permissions than it actually uses. (2) The information flow report presents triggers (mainly UI triggers) and tainted paths that lead from sources to sinks, with contexts such as class files, method names, and line numbers. (3) The heat map report shows rough profiling results of the analyzer, which can be used to help an analyst

understand where the analysis has focused its efforts and might indicate where an app developer has attempted to hide malicious behavior. Analysis graphs are presented with an SVG formatted file as a reachable control flow graph. It highlights suspicious source and sink states, as well as showing tainted paths between them. In addition, an analyst can click on any state node in the graph for detailed inspection of the abstract execution at this point in the graph.

In fact, the pushdown analysis based analyzer evolved from an older version with a similar user interface and output forms (reports and graphs). The previous version of Anadroid used traditional finite-state methods (mainly $k$-CFA and $k$-object-sensitivity) that are employed in analysis platforms such as WALA [24], Soot [41], etc.

The old analyzer enables us to compare the traditional finite-state based analysis with our new pushdown control-flow analysis to help us determine whether the new analysis is more effective and efficient. The following section details this comparison.

# 7. EVALUATION

We had two teams of analysts analyze a challenge suite of 52 Android apps released as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program. Both teams are composed of the same five people, a mixture of graduate students and undergraduate students, however, the applications are shuffled, ensuring the same app is not evaluated by the same analyst. The first team analyzed the apps with a version of Anadroid that uses traditional (finite-state-machine-based) control-flow-analysis used in many existing malware analysis tools; the second team analyzed the apps with a version of Anadroid that uses our enhanced pushdown-based control-flow-analysis. We measure the time the analyzer takes, the time human analysts spent reviewing the results, and the accuracy of the human analyst in determining if an app is malicious. Accuracy is measured by comparing the result of human-in-the-loop analysis with the results released by DARPA. All other factors being equal, we found a statistically significant ($p < 0.05$) decrease in time and a statistically significant increase ($p < 0.05$) in accuracy with the pushdown-based analyzer.

### *The challenge suite:.*

Among the apps, 47 are adapted from apps found on the Android market, Contagio [8], or the developer's source repository. A third-party within the APAC project injects malicious behavior into these apps and uses an anti-diffing tool on apps with larger code bases to make it difficult to simply diff the application with the original source code. The remaining five apps are variants of the original 47 apps with different malicious behaviors. For example, *App1* may leak location information to a malicious website while *App2* may not. The apps range in size from 18.7 KB to 10 MB, with 11,600 lines of source code in each app on average.

### *Experiment setup.*

The two teams of analysts were given instructions on how to use the tool (both versions of the tool use similar UIs and output forms) and some warm-up exercises on a couple of examples apps. Then they were given examples from the DARPA-supplied challenge suite. The analysts used Anadroid (deployed as a web application on our server) to analyze each app and then record the run time of the analyzer, the total time the human analysis spent investigating the results, and an indication if the analyst felt the app was malicious.

| | | mean | p-value |
|---|---|---|---|
| **Analyzer Time** | Finite | 994 sec | |
| | Pushdown | 560 sec | 0.003 |
| **Analyst Time** | Finite | 1.13 hr | |
| | Pushdown | 0.44 hr | **0.0** |
| **Accuracy** | Finite | 71% | |
| | Pushdown | 95% | **0.0005** |

**Table 1: Comparison of finite-state based vs. pushdown malware analysis: Pushdown malware analysis yields statistically significant improvements with $p < 0.05$ in both accuracy and analysis time over traditional static analysis method.**

We have made efforts to ensure that other factors remained unchanged so that the only difference was the tool the analyst used to detect malware. This restriction can help us gain insight into whether any improvement is made by our new analysis techniques.

Finally, we compare the analysts results with the DARPA supplied information on whether an app is malicious to check accuracy and run statistical analysis using one-way Analysis of Variance (ANOVA) to get mean value and p-value of analyzer time, analyst time, and accuracy. This allows us to see the statistical results of the experiment on finite-state-based-machine versus pushdown-based control-flow analysis. This is shown in Table 1.

We found that pushdown malware analysis yields statistically significant improvements with $p < 0.05$ in both accuracy and analysis time over traditional static analysis.

# 8. CASE STUDIES

In this section we report case studies of malware detected by Anadroid. The malicious behavior of the 52 apps provided as part of the APAC challenge are summarized in Table 2. We separate these behaviors into four categories: data leakage, data tampering, denial of service attacks, and other malicious behaviors[6].

*Data leakage* is one of most common, and concerning, malicious behaviors in Android apps [14, 31]. Sensitive data, including location information, an SMS message, or a device ID, is exfiltrated to a third-party host via an HTTP request or Android web component Intent, or to a predefined reachable local file via standard file operations. This kind of behavior is often embedded in a background Android service component, such as an AsyncTask or a thread, without interfering with the normal functionality of the app. Anadroid identifies this malicious behavior in 57% of the 42 malicious Android apps from our test suite that manifest malicious behavior. We found the taint flow analysis to be more useful than the least permissions analysis in identifying these behaviors, since half of these apps are designed to avoid requesting any permissions. For instance, instead of requesting the ACCESS_FINE_LOCATION an app can instead read locations from photos stored on the file system using Exif data and instead of requesting the INTERNET

[6]Some apps have more than one malicious behavior.

| Vulnerabilities | Percentile | Case examples |
|---|---|---|
| Data leakage | 57% | location, pictures, SMS, ID, etc. ex-filtrated to URL, intents, or predefined local file path. |
| Data tampering | 10% | fill local file system with meaningless data, (recursive) deletion of files |
| DoS attack | 11% | inode exhaustion via log, battery drainage (brightness, WiFi, etc.) |
| Other | 28% | random vibration, block or intercept SMS messages |

**Table 2: Vulnerabilities summarization**

the app can use the default Android web view through an ACTION_VIEW Intent, in both cases avoiding the need to explicitly request these permissions.

*Data tampering*, similar to data leakage, can be detected using static taint flow analysis by determining which operations are performed on data. Malware in this category might corrupt the local file system by overwriting file contents with meaningless data, recursively delete files from the SD card, or delete SMS messages. In these real-world apps, exceptions are frequently used, especially around IO operations. We found that finite-state-based analysis can lead to many spurious execution flows in the control graph when used with EPS. The pushdown-based model, on the other hand, produces more precise execution flows, which contributes to the sharp decline in analyst time when using Anadroid.

*DoS attack and other malicious behaviors*: Denial of Service (DoS) on mobile phones exhaust limited resources by intentionally causing the phone to use these resources in an inefficient manner. For instance, an app might drain the battery by setting brightness to maximum or keeping WiFi on at all times or exhaust file system space by logging every operation to a file.

*Other malicious behaviors* include those that do not leak or tamper with sensitive data but still do not behave the way the app was intended to behave. For example, a calculator that uses a random number in a calculation rather than the expected number, or blocks SMS messages in the *onReceive* method when a trigger condition is met. These two categories are more application-dependent and subject to human judgment, by determining if this functionality is too far outside the advertised functionality of the app. In these scenarios, it is important that the analyzer results not overwhelm analysts.

Anadroid cannot determine precisely whether the application is malicious or not by itself. Instead, it identifies suspicious application behavior and uses analyst-supplied predicates to help search analysis results for locations of interest to the analyst.

# 9. RELATED WORK

### *Static analysis for Java programs.*

Precise and scalable context-sensitive pointer analysis for Java (object-oriented) programs has been an open problem for decades. Remarkably, a large bulk of the previous literature focused on finite-state abstractions for Java programs, *i.e.* $k$-CFA, limited object sensitivity, and their variants. In work that addresses exception flows [38, 43, 39, 27], the analysis is often based on context-insensitivity or limited context-sensitivity, which means they cannot differentiate the contexts where an exception is thrown or precisely determine which handlers can handle an exception.

Spark [28] and Paddle [29] both use imprecise exception analysis. Soot [41] also uses a separate exception analysis implemented by Fu *et al.* [18] which is not based on pointer analysis and not integrated into the tool. Bravenboer and Smaragdakis [4] propose joining points-to analysis and exception flow analysis to improve precision and analysis run time in their Doop framework [5]. They have conducted extensive comparison of different options for polyvariance. It provides a more precise and efficient exception-flow analysis than Spark, Paddle, and Soot, with respect of points-to and exception-catch links with respect to the metric used in [17, 5]. IBM Research's WALA is a static analysis library designed to support different pointer analysis configurations. The points-to analyses of WALA can compute which exceptions a method can throw, but does not guarantee precise matches between exceptions and their corresponding handlers.

### *CFL- and pushdown-reachability techniques.*

Earl *et al.* [10] develop a pushdown reachability algorithm suitable for pushdown systems, which essentially draws on CFL- and pushdown-reachability analysis [3, 26, 36, 37]. We modify their traditional CESK machine to handle object-oriented programs and extend it to analyze exceptions. This allows us to apply their algorithm directly to our analysis.

CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [32] and type-based polymorphic control-flow analysis [35]. These analyses should not be confused with pushdown control-flow analysis, which is a fundamentally different kind of CFA.

### *Malware detection for Android applications.*

Several analyses have been proposed for Android malware detection.

Dynamic taint analysis has been applied to identify security vulnerabilities at run time in Android applications. TaintDroid [11] dynamically tracks the flow of sensitive information and looks for confidentiality violations. IPCInspection [15], QUIRE [9], and XManDroid [6] are designed to prevent privilege-escalation, where an application is compromised to provide sensitive capabilities to other applications. The vulnerabilities introduced by interapp communication is considered future work. However, these approaches typically ignore implicit flows raised by control structures in order to reduce run-time overhead. Moreover, dynamically executing all execution paths of these applications to detect potential information leaks is impractical. The limitations make these approaches inappropriate for computing information flows for all submitted applications.

Woodpecker [22] uses traditional data-flow analysis to find possible capability leaks. Comdroid [7] targets vulnerabilities related to interapp communications. However, it does not perform deep program analysis as Anadroid does, and this results in high false positive rates. SmartDroid [44] targets finding complex UI triggers and paths that lead to sensitive sinks. It addresses imprecision of static analysis by

combining dynamic executions to filter out infeasible paths at run time. CHEX [31] focuses on detecting *component hi-jacking* by augmenting existing analysis framework using app-splitting to handle Android's multiple entry points. Our tool takes a significantly different approach from it (and other finite-state-based static analysis tools) in three aspects. (1) We use pushdown flow analysis that handles traditional control-flow and exception flows precisely and efficiently. (2) Our Entry Point Saturation technique is sound, and we are able to detect interleaving execution of multiple entry points while CHEX handles only permutations of multi-entry points. (3) Our tool enables human-in-the-loop analysis by allowing the analyst to supply predicates for the analyzer allowing it to highlight inspection of deeply disguised malware.

Jeon *et al.* [25] proposes enforcing a fine-grained permission system. It limits access to resources that could normally be accessed by one of Android's default permissions. Specifically, the security policy uses a white list to determine which resources an app can use and a black list to deny access to resources. In addition, strings potentially containing URLs are identified by pattern matching and constant propagation is used to infer more specific Internet permissions. Grace *et al.* [23] have also identified unprivileged malicious apps that can exploit permissions on protected resources through a privileged agent (or app component in our test suite) that does not enforce permission checks. Anadroid can also identify this malicious behavior.

Stowaway [13] is a static analysis tool identifying whether an application requests more permissions than it actually uses. PScout [2] aims for a similar goal, but produces more precise and fine-grained mapping from APIs to permissions. Our least permission report uses the PScout permission map as Anadroid's database. However, they use a different approach, adapting testing methodology to test applications and identify APIs that require permissions, while our approach annotates APIs with permissions and statically analyzes all executable paths.

# 10. CONCLUSION

In this paper, we address two challenges in static malware detection for Android apps: the fundamental challenge of analyzing object-oriented programs and the Android domain specific challenge of asynchronous multi-entry points. We address the first challenge using pushdown control flow analysis (PDCFA) to precisely analyze both traditional control flows and exception flows. The second challenge is addressed via entry-point saturation (EPS) that when integrated with PDCFA serves as the basis for our analysis engine. We demonstrate a malware analyzer built on this engine, adding pushdown taint flow and least permissions analysis. We describe Anadroid, a generic analysis framework for Dalvik-bytecode that enables human-in-the-loop analysis by accepting user-supplied predicates to search analysis results for detailed inspection. We compare the new analyzer with a traditional finite-state analyzer using a test suite released by DARPA APAC project. We find that PDCFA together with EPS yields statistically significant improvements in both accuracy and analysis time over traditional static analysis methods. Our implementation is publicly available: `github.com/shuyingliang/pushdownoo`.

# 11. ACKNOWLEDGMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-12- 2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

# 12. REFERENCES

[1] APK TOOL. android-apktool, a tool for reverse engineering android apk files. `http://code.google.com/p/android-apktool/`.

[2] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. PScout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 217–228.

[3] BOUAJJANI, A., ESPARZA, J., AND MALER, O. Reachability analysis of pushdown automata: Application to Model-Checking. In *Proceedings of the 8th International Conference on Concurrency Theory* (London, UK, UK, 1997), CONCUR '97, Springer-Verlag, pp. 135–150.

[4] BRAVENBOER, M., AND SMARAGDAKIS, Y. Exception analysis and points-to analysis: Better together. In *ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ACM, pp. 1–12.

[5] BRAVENBOER, M., AND SMARAGDAKIS, Y. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), OOPSLA '09, ACM, pp. 243–262.

[6] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).

[7] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2011), MobiSys '11, ACM, pp. 239–252.

[8] CONTAGIO. Contagio. `http://contagiodump.blogspot.com`.

[9] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 23–23.

[10] EARL, C., SERGEY, I., MIGHT, M., AND VAN HORN, D. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 177–188.

[11] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. 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* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[12] Felleisen, M., and Friedman, D. P. A calculus for assignments in higher-order languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1987), ACM, pp. 314+.

[13] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 627–638.

[14] Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 3–14.

[15] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission Re-Delegation: Attacks and defenses. In *Security 2011, 20st USENIX Security Symposium* (Aug. 2011), D. Wagner, Ed., USENIX Association.

[16] Freke, J. Smali, an assembler/disassembler for Android's dex format. `http://code.google.com/p/smali/wiki/Registers`.

[17] Fu, C., Milanova, A., Ryder, B. G., and Wonnacott, D. G. Robustness Testing of Java Server Applications. *IEEE Trans. Softw. Eng. 31*, 4 (Apr. 2005), 292–311.

[18] Fu, C., and Ryder, B. G. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 230–239.

[19] Fundamentals, A. Bytecode for the Dalvik VM. `http://developer.android.com/guide/components/fundamentals.html`.

[20] Gartner. 10 billion Android Market downloads and counting. `http://googleblog.blogspot.com/2011/12/10-billion-android-market-downloads-and.html`.

[21] Gartner. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. `http://www.gartner.com/newsroom/id/1848514`.

[22] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).

[23] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).

[24] IBM. WALA. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[25] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 3–14.

[26] Kodumal, J., and Aiken, A. The set constraint/CFL reachability connection in practice. *SIGPLAN Not. 39* (June 2004), 207–218.

[27] Leroy, X., and Pessaux, F. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst. 22*, 2 (Mar. 2000), 340–377.

[28] Lhoták, O., and Hendren, L. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th international conference on Compiler construction* (Berlin, Heidelberg, 2003), CC'03, Springer-Verlag, pp. 153–169.

[29] Lhoták, O., and Hendren, L. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol. 18*, 1 (Oct. 2008), 3:1–3:53.

[30] Liang, S., and Might, M. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2012), PLAS '12, ACM.

[31] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 229–240.

[32] Melski, D., and Reps, T. W. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science 248*, 1-2 (Oct. 2000), 29–98.

[33] Might, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[34] Project, T. A. O. S. Bytecode for the Dalvik VM. `http://source.android.com/tech/dalvik/dalvik-bytecode.html`.

[35] Rehof, J., and Fähndrich, M. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2001), ACM, pp. 54–66.

[36] Reps, T. Program analysis via graph reachability. *Information and Software Technology 40*, 11-12 (Dec. 1998), 701–726.

[37] Reps, T., Schwoon, S., Jha, S., and Melski, D. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming 58*, 1-2 (Oct. 2005), 206–263.

[38] Robillard, M. P., and Murphy, G. C. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol. 12*, 2 (Apr. 2003), 191–221.

[39] Ryu, S. Exception analysis for multithreaded Java programs.

[40] Shivers, O. G. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[41] Soot. A Java optimization framework. `http://www.sable.mcgill.ca/soot`.

[42] Van Horn, D., and Might, M. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (2010), ICFP '10, ACM Press, pp. 51–62.

[43] wu Jo, J., mo Chang, B., Yi, K., and moo Choe, K. An uncaught exception analysis for Java, 2002.

[44] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. 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* (New York, NY, USA, 2012), SPSM '12, ACM, pp. 93–104.