Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 311 (2015) 3–14

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

AnaDroid: Malware Analysis of Android with User-supplied Predicates

Shuying Liang,[1,2] Matthew Might[1]

School of Computing, University of Utah
Salt Lake City, Utah, USA

David Van Horn[1]

College of Computer and Information Sciences
Northeastern University
Boston, Massachusetts, USA

Abstract

Today's mobile platforms provide only coarse-grained permissions to users with regard to how third-party applications use sensitive private data. Unfortunately, it is easy to disguise malware within the boundaries of legitimately-granted permissions. For instance, granting access to "contacts" and "internet" may be necessary for a text-messaging application to function, even though the user does not want contacts transmitted over the internet. To understand fine-grained application use of permissions, we need to statically analyze their behavior. Even then, malware detection faces three hurdles: (1) analyses may be prohibitively expensive, (2) automated analyses can only find behaviors that they are designed to find, and (3) the maliciousness of any given behavior is application-dependent and subject to human judgment. To remedy these issues, we propose semantic-based program analysis, with a human in the loop as an alternative approach to malware detection. In particular, our analysis allows analyst-crafted semantic predicates to search and filter analysis results. Human-oriented semantic-based program analysis can systematically, quickly and concisely characterize the behaviors of mobile applications. We describe a tool that provides analysts with a library of the semantic predicates and the ability to dynamically trade speed and precision. It also provides analysts the ability to statically inspect details of every suspicious state of (abstract) execution in order to make a ruling as to whether or not the behavior is truly malicious with respect to the intent of the application. In addition, permission and profiling reports are generated to aid analysts in identifying common malicious behaviors.

Keywords: static analysis, human analysis, malware detection

1 Introduction

Google's Android is the most popular mobile platform, with a share of 52.5% of all smartphones [10]. Due to Android's open application development community,

[1] Supported by the DARPA Automated Program Analysis for Cybersecurity Program.

[2] An extended report is available: `http://matt.might.net/a/2013/05/25/anadroid/`

http://dx.doi.org/10.1016/j.entcs.2015.02.002
1571-0661/© 2015 Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

more than 400,000 apps are available with 10 billion cumulative downloads by the end of 2011 [9].

While most of those third-party applications have legitimate reasons to access private data, the grantable permissions are too coarse: malware can hide in the cracks. For instance, an app that should only be able to read information from a specific site and have access to GPS information must necessarily be granted full read/write access to the entire internet, thereby allowing a possible location leak over the net. Or, a note-taking application can wipe out SD card files when a hidden trigger condition is met. Meanwhile, a task manager that requests every possible permission can be legitimately benign.

To understand fine-grained use of security-critical resources, we need to statically analyze the application with respect to what data is accessed, where the sensitive data flows, and what operations have been performed on the data (*i.e.*, determine whether the data is tampered with). Even then, automated malware detection faces three hurdles: (1) analyses may be prohibitively expensive, (2) automated analyses can only find behaviors that they are designed to find, and (3) the maliciousness of any given behavior is application-dependent and subject to human judgment.

In this work, we propose semantics-based program analysis with a human in the loop as an alternative approach to malware detection. Specifically, we derive an analytic engine, an abstract CESK* machine based on the design methodology of Abstracting Abstract Machines (AAM) [20] to analyze object-oriented bytecode. Then we extend the foundational analysis to analyze specific features: multiple entry points of Android apps and reflection APIs. Finally, we describe a tool that provides analysts with a library of semantic predicates that can be used to search and filter analysis results, and the ability to dynamically trade speed and precision. The tool also provides analysts the ability to statically inspect details of every suspicious state of (abstract) execution in order to make a ruling as to whether or not the behavior is truly malicious with respect to the intent of the application. Human-oriented, semantics-based program analysis can systematically characterize the behaviors of mobile applications.

Overview

The remainder of the paper is organized as follows: Section 2 presents the syntax of an object-oriented byte code, and illustrates a finite-state-space-based abstract interpretation of the byte code. Section 3 discusses analysis techniques to analyze Android-specific issues: multiple entry points and reflection APIs. Section 4 presents the tool implementation with user-supplied predicates. Section 5 discusses related work, and Section 6 concludes.

2 Semantic-based program analysis

Android apps are written in Java, and compiled into Dalvik virtual machine byte code (essentially a register-based version of Java byte code). In this section, we present how to derive an analysis for a core object-oriented (OO) byte code language

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{class\text{-}def}\ \ldots \\
\mathit{class\text{-}def} \in \mathsf{ClassDef} & ::= & (\mathit{attribute}\ \ldots\ \mathsf{class}\ \mathit{class\text{-}name}\ \mathsf{extends}\ \mathit{class\text{-}name} \\
 & & (\mathit{field\text{-}def} \ldots)\ (\mathit{method\text{-}def} \ldots)) \\
\mathit{field\text{-}def} & ::= & (\mathsf{field}\ \mathit{attribute}\ \ldots\ \mathit{field\text{-}name}\ \mathit{type}) \\
\mathit{method\text{-}def} \in \mathsf{MethodDef} & ::= & (\mathsf{method}\ \mathit{attribute}\ \ldots\ \mathit{method\text{-}name}\ (\mathit{type} \ldots)\ \mathit{type} \\
 & & (\mathsf{throws}\ \mathit{class\text{-}name} \ldots)\ (\mathsf{limit}\ n)\ s\ \ldots) \\
s \in \mathsf{Stmt} & ::= & (\mathsf{label}\ \mathit{label}) \mid (\mathsf{nop}) \mid (\mathsf{line}\ \mathit{int}) \mid (\mathsf{goto}\ \mathit{label}) \\
 & \mid & (\mathsf{if}\ æ\ (\mathsf{goto}\ \mathit{label})) \mid (\mathsf{assign}\ \mathit{name}\ [æ \mid \mathit{ce}]) \mid (\mathsf{return}\ æ) \\
 & \mid & (\mathsf{field\text{-}put}\ æ_o\ \mathit{field\text{-}name}\ æ_v) \mid (\mathsf{field\text{-}get}\ \mathit{name}\ æ_o\ \mathit{field\text{-}name}) \\
æ \in \mathsf{AExp} & ::= & \mathsf{this} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \mathsf{void} \mid \mathit{name} \mid \mathit{int} \\
 & \mid & (\mathit{atomic\text{-}op}\ æ \ldots æ) \mid \mathsf{instance\text{-}of}(æ, \mathit{class\text{-}name}) \\
\mathit{ce} & ::= & (\mathsf{new}\ \mathit{class\text{-}name}) \\
 & \mid & (\mathit{invoke\text{-}kind}\ \mathit{method\text{-}name}\ (æ \ldots æ)\ (\mathit{type}_0\ \ldots\ \mathit{type}_n)) \\
\mathit{invoke\text{-}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} \\
\mathit{type} & ::= & \mathit{class\text{-}name} \mid \mathsf{int} \mid \mathsf{byte} \mid \mathsf{char} \mid \mathsf{boolean} \\
\mathit{attribute} & ::= & \mathsf{public} \mid \mathsf{private} \mid \mathsf{protected} \mid \mathsf{final} \mid \mathsf{abstract}.
\end{array}
$$

Fig. 1. An object-oriented bytecode adapted from the Android specification [18].

based on Dalvik. After presenting this foundational analysis, we shall illustrate Android-specific analysis techniques in subsequent sections.

The first step is to define a syntax. Figure 1 presents the syntax of an OO byte code language that is closely modeled on the Dalvik virtual machine. Statements encode individual actions; atomic expressions encode atomically computable values; and complex expressions encode expressions with possible non-termination or side effects. There are four kinds of names: Reg for registers, ClassName for class names, FieldName for field names and MethodName for method names. The special register name ret holds the return value of the last function called. With respect to a given program, we assume a syntactic meta function $\mathcal{S} : \mathsf{Label} \rightarrow \mathsf{Stmt}^*$, which maps a label to the sequence of statements that start with that label.

Ordinarily, the next step toward an analyzer would be to derive a concrete machine to interpret the language just defined. The meaning of a program will be defined as the set of machine states reachable from an initial state. The purpose of static analysis is to derive a computable approximation of the concrete machine's behavior—of these states. We'll construct an abstract semantics to do that. Since the concrete and the abstract semantics are so close in structure, we will present only the abstract semantics of the byte code, while highlighting places that are different from its concrete counterpart to save space.

2.1 Abstract semantics

We define our abstract interpretation as a direct, structural abstraction of a machine model for the OO bytecode [20]. Because the structural abstraction creates an abstract machine nearly identical to the machine model itself (with exceptions that we explain), we don't provide the concrete machine model. The analysis of a program is defined as the set of *abstract* machine states reachable by an abstract

$$\begin{array}{rll}
\hat{c} \in \widehat{State} = \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{KontAddr} & & \text{[states]} \\
\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{Val} & & \text{[stores]} \\
\hat{a} \in \widehat{Addr} = \widehat{RegAddr} + \widehat{FieldAddr} + \widehat{KontAddr} & & \text{[addresses]} \\
\widehat{a_\kappa} \in \widehat{KontAddr} \text{ is a finite set of continuation addresses} & & \\
\widehat{ra} \in \widehat{RegAddr} = \widehat{FramePointer} \times \mathsf{Reg} & & \\
\widehat{fa} \in \widehat{FieldAddr} = \widehat{ObjectPointer} \times \mathsf{FieldName} & & \\
\hat{\kappa} \in \widehat{Kont} = \mathbf{fun}(\widehat{fp}, \boldsymbol{s}, \widehat{a_\kappa}) + \mathbf{halt} & & \text{[continuations]} \\
\hat{d} \in \widehat{Val} = \mathcal{P}\left(\widehat{ObjectValue} + \widehat{String} + \widehat{\mathcal{Z}} + \widehat{Kont}\right) & & \text{[abstract values]} \\
\widehat{ov} \in \widehat{ObjectValue} = \widehat{ObjectPointer} \times \mathsf{ClassName} & & \\
\widehat{fp} \in \widehat{FramePointer} \text{ is a finite set of frame pointers} & & \text{[frame pointers]} \\
\widehat{op} \in \widehat{ObjectPointer} \text{ is a finite set of object pointers} & & \text{[object pointers].}
\end{array}$$

Fig. 2. The abstract state-space.

transition relation $(\rightsquigarrow)$—the core of the abstract semantics. That is, abstract evaluation is defined by the set of states reached by the reflexive, transitive closure of the $(\rightsquigarrow)$ relation.

Figure 2 details the abstract state-space. We assume the natural element-wise, point-wise and member-wise lifting of a partial order $(\sqsubseteq)$ across this state-space. States of this machine consist of a tuples of of statements, frame pointers, heaps, and stack pointers. To synthesize the abstract state-space, we force frame pointers and object pointers (and thus addresses) to be a finite set. When we compact the set of addresses into a finite set during a structural abstraction, the machine may (and likely will) run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple abstract values to reside at the same (now abstract) address. As a result, we have no choice but to force the range of the $\widehat{Store}$ to become a power set in the abstract state-space: now each abstract address can hold multiple values.

2.1.1 Abstract transition relation

In this section, we provide major cases for the abstract transition relation. The abstract transition relation delegates to helper functions for injecting programs into states, and for evaluating atomic expressions and looking up field values:

- $\hat{\mathcal{I}} : \mathsf{Stmt}^* \to \widehat{State}$ injects an sequence of instructions into an initial state: $\hat{c}_0 = \hat{\mathcal{I}}(\boldsymbol{s}) = (\boldsymbol{s}, \widehat{fp}_0, [\widehat{a_{\kappa 0}} \mapsto \mathsf{halt}], \widehat{a_{\kappa 0}})$
- $\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{Store} \rightharpoonup \widehat{Val}$ evaluates atomic expressions (specifically for variable look-up): $\hat{\mathcal{A}}(name, \widehat{fp}, \hat{\sigma}) = \sigma(\widehat{fp}, name)$
- $\hat{\mathcal{A}_{\mathcal{F}}} : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{Store} \times \mathsf{FieldName} \rightharpoonup \widehat{Val}$ looks up fields:

$$\hat{\mathcal{A}_{\mathcal{F}}}(æ_o, \widehat{fp}, \hat{\sigma}, \mathit{field\text{-}name}) = \bigsqcup \hat{\sigma}(\widehat{op}, \mathit{field\text{-}name}) \text{ , where}$$
$$(\widehat{op}, \mathit{class\text{-}name}) \in \hat{\mathcal{A}}(æ_o, \widehat{fp}, \hat{\sigma}).$$

The rules for the abstract transition relation $(\leadsto) \subseteq \widehat{State} \times \widehat{State}$ describe how components of state evolve in light of each kind of statement. In subsequent paragraphs, we will illustrate the important rules that involve objects and function calls, omitting less important ones 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.

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

where the helper $\widehat{initObject} : \widehat{Store} \times \mathsf{ClassName} \rightharpoonup \widehat{Store}$ initializes 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:

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

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:

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

- *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[3] and then applied as follows:

$$\overbrace{(\llbracket(\mathit{invoke\text{-}kind}\ \mathit{method\text{-}name}\ (æ_0 \ldots æ_n)\ (\mathit{type}_0 \ldots \mathit{type}_n))\rrbracket : \boldsymbol{s}, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}}$$
$$\leadsto \widehat{applyMethod}(m, æ, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})$$

where the function $\widehat{applyMethod}$ takes a method definition, arguments, a frame

[3] 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.

pointer, a store, and a new stack pointer and produces the next states:

$$\begin{gathered}\widehat{applyMethod}(m, æ, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa}) = (\boldsymbol{s}, \hat{fp}', \hat{\sigma}'', \widehat{a_\kappa}'), \text{where} \\ \hat{fp}' = \widehat{allocFP}(\hat{c}), \quad \widehat{a_\kappa}' = \widehat{allocK}(\hat{c}), \\ \hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{a_\kappa}' \mapsto \{\mathbf{fun}(\hat{fp}, \boldsymbol{s}, \widehat{a_\kappa})\}], \quad \hat{\sigma}'' = \hat{\sigma}' \sqcup [(\hat{fp}', name_i) \mapsto \hat{\mathcal{A}}(æ_i, \hat{fp}, \hat{\sigma})].\end{gathered}$$

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

$$\begin{gathered}([\![(\mathsf{return}\ æ) : \boldsymbol{s}]\!], \hat{fp}, \hat{\sigma}, \widehat{a_\kappa}) \rightsquigarrow (\boldsymbol{s}', \hat{fp}', \hat{\sigma}', \widehat{a_\kappa}'), \text{ where} \\ \mathbf{fun}(fp', \boldsymbol{s}', a_\kappa') \in \sigma(a_\kappa) \text{ and } \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}', \mathsf{ret}) \mapsto \hat{\mathcal{A}}(æ, \hat{fp}, \hat{\sigma})].\end{gathered}$$

3 Analysis of reflection and multiple entry points in abstract CESK* machine

While the crux of the analysis for Android apps has been presented in the previous section, our abstract CESK* machine has to be extended to analyze multiple entry points in Android apps and some intricate APIs like reflection.

3.1 Fixed-point computation for multiple entry points

A typical static analysis only deals with one entry point of traditional programs (the **main** method). However, any Android application has more than one entry point, due to the event-driven nature of the Android platform. Intuitively, to explore the reachable states for all the entry points seems to require the exploration for all the permutation of entry points. But this can easily lead to state-space explosion. Related works like [13] prune paths for specific Android apps (but not soundly). We solve the problem in a sound but inexpensive way. Specifically, we iterate over all entry points that have been found. For each entry point, we compute its reachable states via the abstract CESK* machine. Then we compute a single widened store from those states using the widening techniques similar to the ones presented in [15]. The store then forms part of next state to continue the next entry point fixed-point computation. Obviously, the store is monotonic, which ensures a sound approximation of the effects introduced from all entry points. This diminishes precision slightly, but the gains in speed are considerable. The effects of similar technique are also noted in [15][19].

3.2 Reflection

This section presents how to extend the abstract CESK* machine to analyze one of the most commonly used dynamic features in Android—reflection.

Reflection enables programs to access class information to create objects and invoke methods at runtime. Type information involved is dynamically retrieved from strings. The strings can come from user input, files, network or hard-coded, literal strings. Literal strings are not infrequent in reflection. The following code snippet demonstrates a common case of reflection in Java:

```
Class<?> aeco = Class.forName("android.os.Environment");
Method externalDir = aeco.getMethod("getExternalStorageDirectory", (Class[])null);
(File)externalDir.invoke(null);
```

A class object is created in Ln.1 and the method object for *getExternalStorageDirectory* [4] is created in Ln.2. Finally, the method is invoked in Ln.3 via the method object *externalDir*. Since it is a static method with no arguments, the receiver object being invoked is null. Otherwise, the argument $aeco.newInstance()$ needs to be supplied in Ln.3.

To analyze such reflection, we can integrate string analysis into abstract interpretation of Java API calls. In the abstract CESK*, we need five additional transition rules, mainly for simple string analysis and the APIs involving creation of class object, method object, class instantiation and method invocation:

- *String:* Strings are objects in Java, and so string instantiation is a special case for the new rule (see Section 2.1) [5]:

  $$\overbrace{([\![(\textsf{const-string}\ name\ str) : \boldsymbol{s}]\!], \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}} \leadsto (\boldsymbol{s}, \hat{fp}, \hat{\sigma}'', \widehat{a_\kappa}), \text{where } \widehat{op} = \widehat{allocOP}(\hat{c}).$$

  $\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \{(\widehat{op}, \textsf{java/lang/String})\}]$, $\hat{\sigma}'' = \hat{\sigma}' \sqcup [(\widehat{op}, \textsf{value}) \mapsto \alpha(str)]$, Unlike the usual case for new rule, there is a field value paired with the string object pointer as field offset to store abstract string values. α is the abstraction function for string values. The simplest form is to construct a flat lattice for strings. Other string analysis such as Costantini *et al.* [4], Christensen *et al.* [3], *etc.* can be directly incorporated.

- *Class objects:* In byte code, the creation of a class object using Class.forName is an invoke-static statement, with the first argument referencing to string values. The rule will allocate a new class object on the heap, with the field offset class-name points to the string reference looked up by the address $(\hat{fp}, æ)$. In addition, the class object reference is stored into the ret address:

  $$\overbrace{([\![(\textsf{invoke-static java/lang/Class/forName}\ æ\ \textsf{java/lang/String}) : \boldsymbol{s}]\!], \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}}$$
  $$\leadsto (\boldsymbol{s}, \hat{fp}, \hat{\sigma}'', \widehat{a_\kappa}), \text{where } \widehat{op}_{\textsf{Cls}} = \widehat{allocOP}(\hat{c}),$$
  $$\hat{\sigma}' = \hat{\sigma} \sqcup [(\widehat{op}_{\textsf{Cls}}, \textsf{class-name}) \mapsto \hat{\sigma}(\hat{fp}, æ)],$$
  $$\hat{\sigma}'' = \hat{\sigma}' \sqcup [(\hat{fp}, \textsf{ret}) \mapsto (\widehat{op}_{\textsf{Cls}}, \textsf{java/lang/Class})]$$

- *Method objects:* Method objects are represented as method headers, including function name, arguments and their types, return values and exceptions that the method can throw. [6] A method object is resolved from a class object, whose class name can be obtained from the first argument $æ_0$. The second argument will be resolved as the method name. Arrays of argument types of the method object

[4] Since the method is a static method, so no instantiated object is needed, which is null.

[5] java/lang/StringBuilder is interpreted in similar way.

[6] Exceptions handling is omitted in the semantics.

are stored in the third register æ$_3$.[7]

$$\overbrace{([\![(\textsf{invoke-virtual java/lang/Class/getMethod}\ (æ_0\ æ_1\ æ_2)\ \mathit{types}_{args}) : \boldsymbol{s}]\!], \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}}$$
$$\rightsquigarrow \mathit{newMethodObject}(\widehat{op}_{\textsf{Method}}, \boldsymbol{m}, \boldsymbol{s}, \hat{fp}, \hat{\sigma}', \widehat{a_\kappa}), \text{ where}$$

$$(\widehat{op}_{\textsf{Cls}}, \textsf{java/lang/Class}) \in \hat{\sigma}(\hat{fp}, æ_0),\ (\widehat{op}_0, \textsf{java/lang/String}) \in \hat{\sigma}(\widehat{op}_{\textsf{Cls}}, \textsf{class-name}),$$
$$\mathit{class\text{-}name} \in \hat{\sigma}(\widehat{op}_0, \textsf{value}),\ (\widehat{op}_1, \textsf{java/lang/String}) \in \hat{\sigma}(\hat{fp}, æ_1)$$
$$\mathit{method\text{-}name} \in \hat{\sigma}(\widehat{op}_1, \textsf{value}),\ \widehat{op}_{\textsf{Method}} = \widehat{allocOP}(\hat{c})$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, \textsf{ret}) \mapsto (\widehat{op}_{\textsf{Method}}, \textsf{java/lang/Reflect/Method})].$$

Similarly like the transition rule for function call, the method resolution process is omitted here. The resolution process needs the information class-name and method-name. Also note that the resolution result is a set of public methods $\boldsymbol{m}$, rather than one. The helper function *newMethodObject* takes the newly allocated method object pointer, the set of method definitions in the domain MethodDef, the rest statements, the frame pointer, store, and the stack pointer and returns the successor states. Again, the method object value will be stored into the ret address.

- *Class instantiation:* The API call java/lang/Class/newInstance is used to instantiate a new object of a concrete class type (not an abstract class nor interface). The class type name can be resolved from the first argument æ of the instruction. Unlike the normal new statement, the class instantiation requires the invocation of default class constructor. Therefore, we first resolve class definitions by using a helper function $\mathcal{C} : \mathcal{P}(\textsf{ClassName}) \to \mathcal{P}(\textsf{ClassDef})$, and then use getDefaultConstructor : ClassDef → MethodDef to get the a constructor method. After that, the control is transferred to the constructor invocation via invoke-direct statement, which is inserted in front of the rest states $\boldsymbol{s}$.

$$\overbrace{([\![(\textsf{invoke-virtual java/lang/Class/newInstance}\ æ\ \textsf{java/lang/Class}) : \boldsymbol{s}]\!], \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}}$$
$$\rightsquigarrow (s' : \boldsymbol{s}, \hat{fp}, \hat{\sigma}'', \widehat{a_\kappa}), \text{where}$$

$$(\widehat{op}_{\textsf{Cls}}, \textsf{java/lang/Class}) \in \hat{\sigma}(\hat{fp}, æ),\ (\widehat{op}, \textsf{java/lang/String}) \in \hat{\sigma}(\widehat{op}_{\textsf{Cls}}, \textsf{class-name}),$$
$$\mathit{class\text{-}def} \in \mathcal{C}(\hat{\sigma}(\widehat{op}, \textsf{value})),\ \widehat{op}_{\textsf{Cls}} = \widehat{allocOP}(\hat{c}),$$
$$m = \textsf{getDefaultConstructor}(\mathit{class\text{-}def}),$$
$$s' = (\textsf{invoke-direct},\ m.\mathit{method\text{-}name}\ (æ_0 \ldots æ_n)\ (\mathit{type}_0 \ldots \mathit{type}_n)),$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, æ_0) \mapsto (\widehat{op}_{\textsf{Cls}}, \textsf{java/lang/Class})],$$
$$\hat{\sigma}'' = \hat{\sigma}' \sqcup [(\hat{fp}, \textsf{ret}) \mapsto (\widehat{op}_{\textsf{Cls}}, \textsf{java/lang/Class})].$$

- *Method invocation:* Reflection method invocation in byte code is achieved by invoking the API java/lang/reflect/Method/invoke via invoke-virtual on a method object, which can be obtained from the first argument. The second argument is

[7] We don't interpret the arrays of the types explicitly.

the receiver object that the method will be invoked on.[8] The third argument is an array of arguments.

$$\overbrace{(\llbracket(\textsf{invoke-virtual java/lang/reflect/Method/invoke }(æ_0\ æ_1\ æ_2)\ \mathit{types}_{args}) : \boldsymbol{s}\rrbracket, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa})}^{\hat{c}}$$
$$\leadsto \widehat{applyMethod}(m, æ, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa}).$$

Like the general rule for function call, we have to resolve the method m, and then by using $\widehat{applyMethod}$ we can get successor states.

4 The tool with user-supplied predicates

In this section, we first briefly presents the implemented analyzer, since the core of the analysis has been specified in Section 2 an 3. Then we illustrate the tool usage, particularly with respect to user-supplied predicates.

The analysis engine is a faithful rendering of the formal specification in Section 2 and 3. In addition, it incorporates previous techniques that boost precision and performance, including the abstract garbage collection [17], store-widening [16], and simple abstract domains to analyze strings [4]. Other constructs of the tool are:

- **Entry points finder:** This component discovers all the entry points of an Android application. Then the engine will explore reachable states based on the algorithm presented in Section 3.
- **Permission violation report:** It reports whether an application asks for more permissions than it actually uses or vice versa.
- **State graph:** This presents all reachable states with states-of-interest highlighted according to default predicates or those supplied by analysts.
- **API dumps:** This presents all the reachable API calls.
- **Heat map:** This reports analyzer intensity on per-statement basis.

The work flow of our human-in-the-loop analyzer—AnaDroid—is as follows: (1) an analyst configures analysis options and malware predicates; (2) AnaDroid presents a permission-usage report, an API call dump, a state graph and a heat map. The major parameters of the analyzer include call-site context-sensitivity—k, widening, abstract garbage collection, cutoffs, and predicates. An analyst can make the trade-off between runtime and precision of the analyzer with these parameters. In addition, the predicates enable analysts to inspect states of interests to detect malware.

4.1 Semantic predicates

To assist analysts, we provide a library of predicates for common patterns. The two major kinds of predicates in AnaDroid are: "State color predicate" renders matching states in a customized color; "State truncate predicate" optimizes the

[8] It will be null if the method to be invoked is a static method. Its transition rule can be easily adapted from the rule presented.

analysis exploration by allowing analysts to manually prune paths beginning at matching states.

Examples of usage of the predicates are listed as follows:

`uses-API?`: It is used to specify what color to render the state that uses the specified API call. The color is a string representing a SVG color scheme [11], *i.e.*"red, colorscheme=set312":

```
(lambda (state)
  (if (uses-API? state "org/apache/http/client/HttpClient/execute" st-attr)
      "red,colorscheme=set312"
      #f))
```

Note that `st-attr` is a specialized keyword used by our analyzer. `state` is the parameter of the predicate.

`uses-name?`: It is a variant of `uses-API?`, used to identify a state with the specific method name in code:

```
(lambda (state)
  (if (uses-name? state "org/ucomb/android/testinterface/RectanglePlus/getArea")
      "red,colorscheme=set312"
      #f))
```

An analyst can also use `cond` to specify multiple colors:

```
(lambda (state)
  (cond
    [(uses-API? state "org/apache/http/client/HttpClient/execute" st-attr )
         "red,colorscheme=set312"]
    [(uses-name? state "org/ucomb/android/testinterface /RectanglePlus/getArea")
         "8,colorscheme=set312"]
    [else #f]))
```

`truncate?`:

```
(lambda (state)
  (if (truncate? state "org/apache/http/client/HttpClient/execute")
      "12,colorscheme=set312"
      #f))
```

5 Related work

Stowaway [7] is a static analysis tool identifying whether an application requests more permissions than it actually uses. PScout [1] aims for a similar goal, but produces more precise and fine-grained mapping from APIs to permissions. Our least permission report uses the Stowaway permission map as AnaDroid's database. [9]

Jeon *et al.* [12] proposes enforcing a fine-grained permission system. It limits access to resources that could be accessed by Android's default permissions. Specifically, the security policy uses a white list to determine which resources an app can

[9] Our new version analyzer is upgraded to PScout data set.

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.

Dynamic taint analysis has been applied to identify security vulnerabilities at run time in Android apps. TaintDroid [6] dynamically tracks the flow of sensitive information and looks for confidentiality violations. QUIRE [5], IPCInspection [8], and XManDroid [2] 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.

The other approach to enforce security on mobile devices is delegating the control to users. iOS and Window User Account Control [14] can prompt a dialog to request permissions from users when applications try to access resources or make security or privacy-related system level changes. However, we advocate stopping potential malware from floating in the market beforehand via strict inspections. Our tool has designed with analysts in mind and can help them identify malicious behaviors of submitted applications.

6 Conclusion

In this work, we propose a human-oriented semantic-based program analysis for Android apps. We derive an abstract CESK* machine to analyze object-oriented bytecode. Then the foundational analysis is extended to analyze specific features: multiple entry points of Android apps and reflection APIs. We also describe a tool that provides analysts with a library of semantic predicates that can be used to search and filter analysis results, and the ability to dynamically trade speed and precision. It also provides analysts the ability to statically inspect details of every suspicious state of (abstract) execution in order to make a ruling as to whether or not the behavior is truly malicious with respect to the intent of the application. In addition, permission and profiling reports are generated to aid analysts in identifying common malicious behaviors. The technique can systematically, quickly and concisely characterize the behaviors of mobile applications, as demonstrated by case studies in the extended report.

References

[1] 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.

[2] 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).

[3] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *Proceedings of the 10th international conference on Static analysis* (Berlin, Heidelberg, 2003), SAS'03, Springer-Verlag, pp. 1–18.

[4] Costantini, G., Ferrara, P., and Cortesi, A. Static Analysis of String Values. In *Proceedings of the 13th International Conference on Formal Engineering Methods (ICFEM 2011)* (2011), S. Qin, Z. Qiu, S. Qin, and Z. Qiu, Eds., vol. 6991 of *Lecture Notes in Computer Science*, Springer, pp. 505–521.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] Graphviz. Brewer color schemes. http://www.graphviz.org/doc/info/colors.html#brewer.

[12] 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.

[13] 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.

[14] MicroSoft. What's user account control? http://windows.microsoft.com/en-us/windows-vista/what-is-user-account-control.

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

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

[17] Might, M., and Shivers, O. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2006), ACM, pp. 13–25.

[18] Project, T. A. O. S. Bytecode for the dalvik vm. http://source.android.com/tech/dalvik/dalvik-bytecode.html.

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

[20] Van Horn, D., and Might, M. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 51–62.