AnaDroid: Malware Analysis of Android with User-supplied Predicates

Shuying Liang[1] Matthew Might[2]

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

David Van Horn[3]

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, 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. 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] Email: liangsy@cs.utah.edu
[2] Email: might@cs.utah.edu
[3] Email: dvanhorn@ccs.neu.edu

1 Introduction

Google's Android is the most popular mobile platform, with a share of %52.5 of all smartphones [8]. 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 [7].

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. Or, a note-taking application can wipe out SD card files when a trigger condition is met. Meanwhile, a task manager that requests every possible permission can be legitimately benign.

To understand fine-grained use of granted permissions, 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 (e.g. whether the data has been tampered or not).

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. In particular, our analysis allows analyst-crafted semantic predicates to search and filter analysis results. We describe a tool that provides analysts with a library of such 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. Human-oriented, semantics-based program analysis can systematically, quickly and concisely 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 presents the tool's high-level architecture, followed by tool usage demonstrated in Section 4. Section 5 shares case studies of using our tool to precisely identify malware. Section 6 discusses related work; and Section 7 concludes.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{class\text{-}def}\ \ldots \\
\mathit{class\text{-}def} & ::= & (\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{throw}\ 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}\ (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 [17].

2 Semantic-based program analysis

In this section, we first define an object-oriented byte code language closely modeled on the Dalvik virtual machine to which Java applications for Android are compiled. Then we define an abstract interpretation for the language.

The syntax of the byte code language is given in Figure 1. Statements encode individual actions for the machine; atomic expressions encode atomically computable values; and complex expressions encode expression 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 $\mathcal{S} : \mathsf{Label} \to \mathsf{Stmt}^*$, which maps a label to the sequence of statements that start with that label.

With the language in place, the canonical step is to define its concrete semantics, which is the most accurate interpretation of program behaviors, but it is not computable. Fortunately, by adapting the technique of abstracting abstract machine[19], we derive a sound and computable abstract semantics for the Dalvik byte code: the finite-state based abstract CESK* machine.

In this machine, states of this machine consist of a series of statements, a frame pointer, a heap, and continuation address. Abstract evaluation of a program is defined as the set of *abstract* machine configurations reachable by an abstraction of the machine transitions relation, which is the reflexive,

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

Fig. 2. The abstract configuration-space.

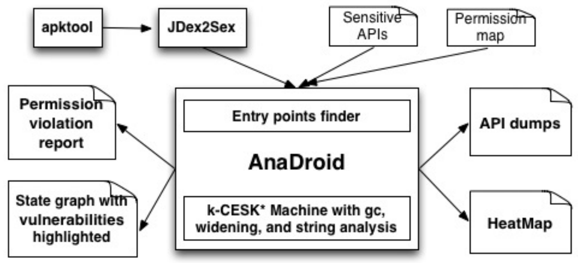
apktool
JDex2Sex
Sensitive APIs
Permission map
Permission violation report
Entry points finder
AnaDroid
k-CESK* Machine with gc, widening, and string analysis
API dumps
State graph with vulnerabilities highlighted
HeatMap

Fig. 3. Software organization and outputs

transitive closure of the $(\leadsto)$ relation.

Since the concrete semantics is nearly identical to the abstract semantics in presentation, we will illustrate abstract interpretation of the byte code only, while highlighting places that are different from its concrete counterparts to save space.

2.1 Abstract configuration-space

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

The set $\widehat{FramePointer}$ is the environmental component of the machine: by pairing the frame pointer with a register name, it forms the address of its value in the store.

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, the machine may run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple abstract values to reside at the same 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 configuration-space.

2.2 Abstract transition relation

The machine relies on helper functions for evaluating atomic expressions, and looking up field values:

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

$$\hat{\mathcal{A}_{\mathcal{F}}}(æ, \hat{fp}, \hat{\sigma}, \textit{field-name}) = \bigsqcup \hat{\sigma}(\widehat{op}, \textit{field-name}) \qquad \text{[field look-up]}$$
$$\text{where } (\widehat{op}, \textit{class-name}) \in \hat{\mathcal{A}}(æ, \hat{fp}, \hat{\sigma}).$$

The rules for the abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ models all the possible concrete execution of a byte code program in a sound way. In the subsequent subsections, we illustrate the important rules that involves objects and function calls, omitting less important ones to save space.

2.2.1 New object creation

Creating an object allocates a potentially non-fresh object pointer and joins the newly initialized object into that store location:

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

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

2.2.2 Instance field reference/update

Referencing a field uses $\hat{\mathcal{A}_{\mathcal{F}}}$ to evaluate the field values and join the store for destination register:

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

Updating a field first finds the abstract object values from the store, extracts its object pointer from each of all the possible values, then pairs this object pointer with the field name to get the field addresses, and finally *joins* the extensions to the store:

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

2.2.3 Method invocation

This rule involves all four components of the machine. In abstract interpretation of non-static method invocation, there can be a *set* of possible objects that are invoked, rather than only one as in its concrete counterpart. This also means that there could be multiple method definitions resolved for each object. For each such method m[4]:

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

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

$$\widehat{applyMethod}(m, æ, \hat{fp}, \hat{\sigma}, \widehat{a_\kappa}) = (\boldsymbol{s}, \hat{fp}', \hat{\sigma}'', \widehat{a_\kappa}'), \text{where}$$
$$\hat{fp}' = \widehat{allocFP}(\hat{c}), \qquad \widehat{a_\kappa}' = \widehat{allocK}(\hat{c}),$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{a_\kappa}' \mapsto \{\mathbf{fun}(\hat{fp}, \boldsymbol{s}, \widehat{a_\kappa})\}], \qquad \hat{\sigma}'' = \hat{\sigma}' \sqcup [(\hat{fp}', \mathit{name}_i) \mapsto \hat{\mathcal{A}}(æ_i, \hat{fp}, \hat{\sigma})].$$

2.2.4 Procedure return

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

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

3 High-level architecture of the tool

We have constructed our analysis engine based on Section 2. Its high level architecture is shown in Fig 3.

`Anadroid` is the analysis engine. It is a faithful rendering of the formal specification in Section 2. In addition, it incorporates previous techniques that boost precision and performance, including the abstract garbage collection [14], store-widening [13], and simple abstract domains to analyze strings [2]. Other constructs of the tool are:

[4] Since the language supports inheritance, method resolution requires a traversal of the class hierarchy. This traversal is not of interest, so we focus on the abstract rules.

- **Multi-entry points:** The component `Entry points finder` in `Anadroid` discovers all the entry points of an Android application. A typical static analysis only deals with one entry point of traditional programs (the **main** method). However, any Android application has one or more entry-points, 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 [11] prune paths for specific Android applications (but not soundly). While we solve the problem in a sound but not expensive way.

  Specifically, we iterate all the entry points. For each entry point, we compute its reachable states. Before exploring the next entry point, the store component of the next state "inherits" the *widened* store computed from states of last fixed point.
- **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 that or those supplied by analysts.
- **API dumps:** This presents all the reachable APIs calls.
- **Heat Map:** This reports analyzer intensity on per-statement basis.

4 Tool usage

This section presents the work flow of our human-in-the-loop analyzer: AnaDroid. We have deployed AnaDroid as a web service, accessible via: `http://pegasus.cs.utah.edu:8080/anadroid`.

The AnaDroid analysis process is as follows: (1) an analyst configures analysis options and malware predicates; (2) AnaDroid presents a permission-usage report, an API call dumps, 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 Predicates

To assist analysts, there are two kinds of predicates in AnaDroid: "State color predicate" renders matching states in a customized color; "State truncate predicate" optimizes the analysis exploration by allowing analysts to manually prune paths beginning at matching states. We provide a library of these kind of predicates for common patterns. 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 [9], e.g. "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 state with specific method name of from source code:

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

- `truncate?`:

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

5 Case studies

In this section, we illustrate two case studies [5] to demonstrate the effectiveness of our human-in-the-loop analyzer.

5.1 Case study 1: user location leaked to a malicious website

UltraCoolMap is advertised to be an application allowing users to browse an interactive map, mark, save, load marked locations, and get directions between marked locations. However, it is malicious. It exfiltrates users' private GPS data by means of an HTTP get request.

The *UltraCoolMapMaliciousCodeSnippet.java* lists the minimum malware code region identified by our tool. In the entry point function `onCreate`(in UltraCoolMapActivity.java), when a button is clicked, a typical path constructs the Google map address with location data in variable `real_badlyName` (com-

[5] The two applications are challenge applications from DARPA's APAC program.

mented out in line 6), but then from line 12-16, the app replaces part of the address with “maps.Google-com.cc” via the function `real_Bad_Name`. Lastly, the URI parameter, which is now the location data along with the modified malicious website, flows to the HTTP get request in line 21 in background via an instantiated asynchronous task in line 10.

Fig 4 is a screen snapshot of the related state graph generated by our tool, with vulnerabilities flagged in red and states with sensitive, private-data-bearing strings in orange. For any state, one can click into the node to inspect detailed state information, which includes (1) Specific API as well as correspondent permissions for the state; (2) Any sensitive values (normally for strings); (3) Current function frame pointer, time, and successive instructions; (4) The store environment.

An analyst typically starts from any highlighted node, e.g. 470, 453, 447, and then following forward/backward along the path to determine any vulnerabilities. In addition, one can find the direct trigger of a suspicious functionality by going to the first node of the path. Even though the orders of entry points are not deterministic, by inheriting the store environment for information propagation, we can track whether any sensitive value flows to a program point or not in a sound way, which proved to sufficient for finding malware in a red-team engagement in DARPA’s APAC program.

In the case of *UltraCoolMap*, an analyst can find that a HTTP request instantiated with some URI with sensitive values in the node, it is then executed in node 447, and the direct trigger is the `doInBackground` method. By checking paths across different entry points, one can make more sense of the application.

```
1  public void onCreate(Bundle realBadlyName) {
2      //...
3      real_badly_name.setOnClickListener(new OnClickListener() {
4          @Override
5          public void onClick(View v){
6             //      built part of maps.google.com in real_badlyName
7              try {
8                  URI realbadlyname = new URI(this
9                          .real_Bad_Name(real_badlyName));
10                 new ReallyBadName().execute(realbadlyname); }
11             catch (Exception realbadly_name){} }
12         private String real_Bad_Name(String really_bad_Name){
13             // string manipulation to replace with "maps.google-
                  com.cc"
14             really_BadName = really_BadName.concat("-");
15             really_BadName = really_BadName.concat(".cc");
16             return really_BadName;}});
17
18     private class ReallyBadName extends AsyncTask<URI,Void,Void>
          {
19         protected Void doInBackground(URI... uris){
20             HttpClient reallyBadName = new DefaultHttpClient();
```

```
            HttpGet reallybadName = new HttpGet(uris[0]);
            try { reallyBadName.execute(reallybadName);}
        catch (Exception really_bad_name){}
            return null; }}}
```

UltraCoolMapMaliciousCodeSnippet.java

5.2 *Case study 2: SD card files deleted*

TomDroid is a note taking system that lets users save and synchronize notes. Our experiment can precisely find suspicious behavior efficiently. Malicious code region identified by the tool is listed as follows:

```
 1 // called from onCreate method
 2 private void displayHundredRuns(){
 3     // update sdcard record to remind about 100 runs
 4     File sddir = new File("/sdcard");
 5     if(sddir.isDirectory() && sddir.canRead()) {
 6         //recurse is a recursively gathering files in  folder
 7         List<String> listing = recurse(sddir);
 8         if(listing != null && listing.size() > 0) {
 9             for(String entry : listing)
10                 updateContentOnRuns(100,new File(entry));}}}
11
12 public void updateContentOnRuns(long runCount, File filePath){
13     if (runCount < 100)
14         filePath.setLastModified(System.currentTimeMillis());
15     else {
16         if (runCount >= 0)
17             filePath.delete();
18         else
19             filePath.setLastModified(System.currentTimeMillis());
                }}
```

SDCardFilesDeleted.java

In the code snippet *SDCardFilesDeleted.java*, when activity's `onCreate` method is called by Android framework, it will call `displayHundredRuns` (defined in line 2-10), which traverses the folder and calls `updateContentOnRuns` method (line 10), where in line 17 deletes the iterating file object when `runCount` <100.

For this is kind of malware, our analyzer presents a "information flow" style graph to help analysts to precisely identify the malware functionality as shown in Fig 5.

The graph only presents reduced-size nodes and path. Node 48 on the left hand side signifies a file object is instantiated. However, with this information alone can not justify any malware behavior. By following the node 47-42, an analyst can see possible sensitive values ("sdcard") are used, and so we know that the app is trying to access to the external storage of the phone. The right

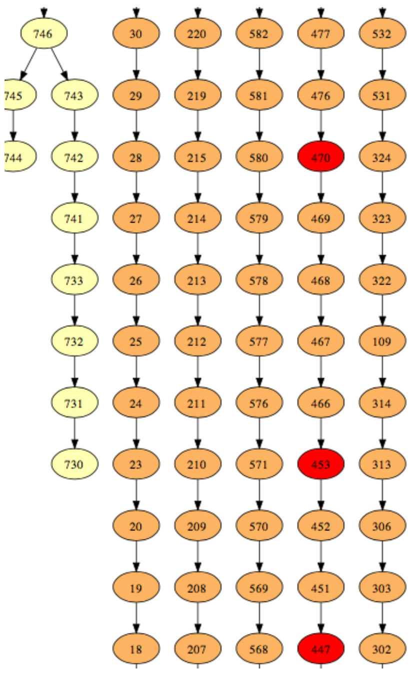

Fig. 4. Snapshot of state graph for UltraCoolMap

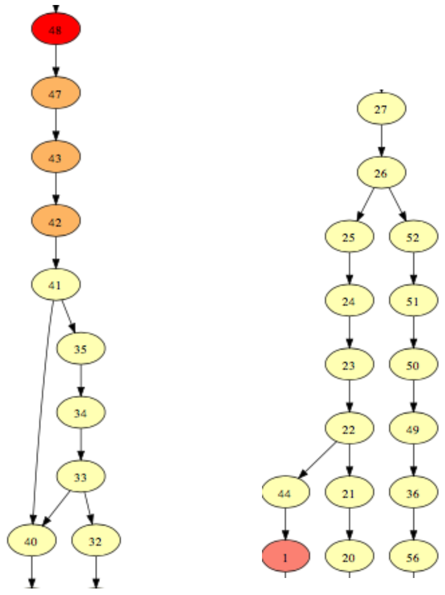

Fig. 5. Snapshot of state graph for case study 2

hand side is the continuous path of the graph. It only presents the neighboring nodes of the highlighted node 1. One can inspect into the node 1 for more detail (like Fig 4), knowing that the "delete" method is invoked on the file object along the path[6].

6 Related work

Stowaway [5] is a static analysis tool exploring whether an application requests more permissions than it actually uses. Our least permission report uses Stowaway's permission map as Anadroid's database. However, the main difference is that Stowaway adapted testing methodology to test applications and identify APIs that require permissions, while our approach annotates APIs with permissions and statically analyses all executable paths.

Language-based information flow [18][15] [16] allows developers to annotate variables with security attributes and compilers use the attributes to enforce information control flows. The main concern of this approach is that it imposes additional burdens on developers whose major focus and interest is business logic.

Recent techniques to enforce fined-grained permission system proposed by Jeon *et al.* [10] are also targeted developers (but can be retrofitted by sophisticated users). The idea of "fine-grained" permission is mainly achieved

[6] Again, the looping path is not showing in the figure due to space limit.

by limiting access to resources that could be accessed by Android's default permission system. Specifically, the security policy is designed to allow access to resources that are specified in white list while deny access to those in blacklist. In addition, sensitive strings (URL-like) are pattern matched and then traditional constant propagation is used to infer permissions.

Dynamic taint analysis has been applied to identify specific security vulnerabilities at run time in Android applications. TaintDroid [4] dynamically tracks the flow of sensitive information and looks for confidentiality violations. QUIRE[3], IPCInspection[6] and XManDroid [1]are designed to prevent privilege-escalation, where an application is compromised to provide sensitive capabilities to other applications. The vulnerabilities introduced by inter-app communication will be considered as our future work. However, these approaches typically ignore implicit flows raised by control structures in order to reduce runtime 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.

Another approach to enforce security control on mobile devices is delegating the control to users themselves. iOS and Window User Account Control [12] can prompt a dialog to request permissions from users when applications try to access resources or make security or privacy-related system level changes. Depending on users to enforce security control is putting users at their own risks. Notifications prompts by the tools usually provide no insights of how users' private sensitive data are used, and users tend to grant permissions in order to install the apps that want. Thus, it is desirable to stop 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 fast and efficiently.

7 Conclusion

In this work, we propose a human-oriented semantic-based program analysis with the ability to allow analyst-crafted semantic predicates to search and filter the result of analysis result. We describe our tool that provides analysts with a library of such 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. The technique can systematically, quickly and concisely characterize the behaviors of mobile applications, as demonstrated by our case studies.

References

[1] 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).

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] Myers, A. C. Jflow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), POPL '99, ACM, pp. 228–241.

[16] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol. 9*, 4 (Oct. 2000), 410–442.

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

[18] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21* (2003), 2003.

[19] 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.