SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications

Jinyung Kim, Yongho Yoon, and Kwangkeun Yi
Programming Research Laboratory
Seoul National University
Seoul, Korea
{jykim,yhyoon,kwang}@ropas.snu.ac.kr

Junbum Shin
SW R&D Center
Samsung Electronics
Suwon, Korea
junbum.shin@samsung.com

***Abstract*—Smartphone applications can steal users' private data and send it out behind their back. The worldwide Android smartphone market is growing, which raises security and privacy concerns. However, current Android's permission-based approach is not enough to ensure the security of private information.**

In this paper, we present SCANDAL, a sound and automatic static analyzer for detecting privacy leaks in Android applications. We analyzed 90 popular applications using SCANDAL from Android Market and detected privacy leaks in 11 applications. We also analyzed 8 known malicious applications from third-party markets and detected privacy leaks in all 8 applications.

***Keywords*-Android; privacy; static analysis; security;**

I. INTRODUCTION

A. Problem

Smartphone applications can steal users' private data and send it out behind their back [6], [7]. Smartphones store various personal data, such as phone identifiers, location information, and contacts. Third-party applications, which can be downloaded freely at markets, frequently access the data. Most of the applications do so to explore the fun and utility of smartphone technology. However, such accesses also raise concerns and issues of privacy risk.

Android's permission-based approach is not enough to ensure the security of private information. [3], [9]. Android requires application developers to declare the permissions so their applications can access users' private information. However, the permissions does not let you know the actual trace of private data. It is uncertain if an application only accesses private data locally, or sends the data out. Also, developers tend to request more permissions than what they need [10]. As a result, users also tend to care less about the permissions when they install applications.

B. Our Solution

We developed a static analyzer SCANDAL that detects privacy leaks in Android applications. SCANDAL determines if there exists any flow of data from an information source through a sink. SCANDAL is a sound analyzer. It covers all possible states which may occur when using the application. In other words, SCANDAL can detect every possible privacy leak in the application.

The following is a simple example of an interprocedural privacy leak SCANDAL detected. The example code is from the Dalvik bytecode of *Google Wallpaper 4.2.2*. This application sends a device ID, called IMEI, to the content server. At line 2, the application gets device ID by calling the `getDeviceId` API and stores it in a global variable. After that, in the `getLocale_version_IMEI_W_H` method, the IMEI is loaded and is appended to some other string values and returned. The returned string is passed to the `getSearchURL` method, and also manipulated and returned to `initTagWebView`. Finally, the string that contains IMEI is made into a URL and sent to the content server of the application.

```
1  Wallpapers.onCreate()
2      callv TelephonyManager.getDeviceId()
3      move-result r3
4      puts r3 eWallpaperConst.IMEI
5
6  XMLTools.getLocale_version_IMEI_W_H()
7      gets r5 eWallpaperConst.IMEI
8      callv StringBuilder.append(r4,r5)
9      move-result r4
10     callv StringBuilder.toString(r4)
11     move-result r4
12     return r4
13
14 XMLTools.getSearchURL()
15     calld getLocale_version_IMEI_W_H
16     move-result r2
17     callv StringBuilder.append(r1,r2)
18     move-result r1
19     callv StringBuilder.toString(r1)
20     move-result r0
21     return r0
22
23 SearchTagsActivity.initTagWebView()
24     calld XMLTools.getSearchURL(r1)
25     move-result r1
26     callv WebView.loadUrl(r1)
```

The following is another example of a privacy leak SCANDAL detected. The example code is from the Dalvik bytecode of *Bible Quotes*. At line 3, this application sends location information to an advertisement server. The application obtains the location information by calling the `getLatitude` API and stores it in an array. After that, the information passes various manipulation steps and is made as a part of an HTML code, which renders the advertisement. Finally, the advertisement is shown on the `WebView`.

```
1  protoFromLocation()
2      new-array r1, r1, Object
3      callv Location.getLatitude(r10)
4      move-result r3
5      mul-double r3, r7
6      double-to-long r3, r3
7      calld Long.valueOf(r3)
8      move-result r3
9      aput-object r3, r1, r2
10     calld String.format(r0, r1)
11     move-result r0
12     return r0
13
14 getLocationParam()
15     callv protoFromLocation(r6, r5)
16     move-result r1
17     const-string r5 "e1+"
18     callv StringBuilder.append(r0, r5)
19     callv StringBuilder.append(r0, r1)
20     ...
21     callv StringBuilder.toString(r0)
22     move-result r5
23     return r5
24
25 genearteHtml()
26     callv getLocationParam(r11)
27     move-result r4
28     calld AdSpec.Parameter.<init>(r11, r12, r4)
29     callv List.add(r5, r11)
30     ...
31     new r11, StringBuilder
32     calld StringBuilder.<init>(r11)
33     const-string r12,
34         ";\n</script>\n<script type=
35         'text/javascript' src='"
36     callv StringBuilder.append(r11, r12)
37     move-result r11
38     callv AdSpec.getAdUrl(r15)
39     move-result r12
40     callv StringBuilder.append(r11, r12)
41     move-result r11
42     const-string r12,
43         "'></script>\n</body>
44         \n</html>"
45     ...
46     calld AdUtil.generateJSONParameters(r5)
47     move-result r12
48     callv StringBuilder.append(r11, r12)
49     move-result r11
50     callv StringBuilder.toString(r11)
51     move-result r11
52     return r11
53
54 showAds()
55     callv generateHtml(r4, r5, r6)
56     move-result r0
57     callv WebView.loadData(r1,r0,r2,r3)
```

A brief video demo of the experiments is available at our website [16].

C. Organization

Sections 2 through 4 describe an overview of SCANDAL, Section 5 shows the experiment results, Section 6 discusses the limitations, Section 7 describes related work, and Section 8 concludes.

II. ANALYSIS TARGET

We build a static analysis framework for Android programs which takes Dalvik VM bytecode as an input to detect privacy leaks in Android applications.

A. Target: Privacy Leaks

SCANDAL determines if there exists a flow of data from an information source through a sink. We call such a flow a *privacy leak*.

1) Sources: API calls that return private information are considered as information sources. We track 3 types of private information:

- *Location Information*: Applications can send users' physical location to remote advertisement servers [7].
- *Phone Identifiers*: Applications can send phone identifiers to remote network servers [6], [7]. We track four phone identifiers: phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number).
- *Eavesdropping*: We track both audio (on microphone) and video (on camera) eavesdropping.

2) Sinks: API calls that can transfer data to the network, file or SMS are considered as sinks.

- *Network/File*: Relevant API calls in the `Webview`, `Outputstream`, `DataOutputStream`, and `HttpURLConnection` objects are considered as sinks.
- *SMS*: `sendTextMessage()` function in object `SmsManager` is also considered as a sink.

Whenever private data is sent out of the phone, SCANDAL considers it as a privacy leak. Such a leak might reasonably

be expected by users, depending on the functionality of the application. We do not try to understand the intention of the situations.

B. Target Language: Dalvik VM Bytecode

Dalvik is the register-based virtual machine in the Android operating system. Android developers commonly write an application in a dialect of Java, and the application is compiled to bytecode.

SCANDAL deals with the Dalvik bytecode, rather than translating the Dalvik bytecode to Java. We do this because known reverse engineering tools, such as dex2jar, fail in some cases. It is also possible for malicious developers to deliberately modify an application at the bytecode level, which makes the application hard to be decompiled. By dealing directly with the bytecode, we do not have to suffer from such issues.

III. HOW SCANDAL WORKS

In this section, we discuss the architecture of SCANDAL and how SCANDAL works.

We designed SCANDAL in the abstract interpretation framework [4], [5]. We designed Dalvik Core, an intermediate language for the analyzer. We defined the concrete and abstract semantics of Dalvik Core. Our analysis is path-insensitive, context-sensitive with a context depth 1, and adopts a hybrid of flow-sensitive and flow-insensitive analysis.

A. Front-end

SCANDAL takes a packaged Android application (.apk) file as an input. We can extract the bytecode of the application as a Dalvik Executable (.dex) file from the package.

Front-end consists of dumping and parsing of the Dalvik Executable file. For the dump part, we modified the existing reverse assembler DexDump. DexDump is included in the Android SDK [2]. The original DexDump is made for human readability, so the dumped result is not easy to be parsed directly. We modified the code of DexDump so that the dumped result could be parsed easily. Moreover, the original DexDump drops some key information such as switch tables or initial data of arrays. We also modified the code so it does not omit any important data.

B. Dalvik Core

We designed Dalvik Core, an intermediate language, for simple and efficient analysis. There are over 220 Dalvik instructions, but a lot of the instructions have semantics similar to some other instruction but differ only in detail. For example, there are more than 20 instructions that move a value into a register. The intermediate language consists of 15 instructions and can represent all of the original Dalvik instructions.

We define the syntax of Dalvik Core in Figure 2. A program is a tuple of instructions, method table, handler table, and subtype relation. Each instruction is labeled with the program counter, in an increasing order from 0. Method table stores entry points of the methods. Handler table stores exception handling stacks for every program points. The symbol $\star$ and $\diamond$ indicate various operators.

We define the concrete semantics of Dalvik Core in Figure 3. Collecting semantics of a program, which our analysis will safely approximate, is the set of all machine states occuring during the execution of the program for all inputs. This set is defined as the least fixed point of

$$\lambda S.S_0 \sqcup Next\ S \tag{1}$$

where S_0 is the set of all initial states and the $Next$ function does the one-step execution by applying the transition operation $\rightarrow$ to each state in T.

A machine state is a tuple of memory, environment, callback environment, program counter, and continuation. Memory models the heap memory while environment is for registers. Callback environment contains event-handlers which have been registered at the state. Continuation captures return points of the methods. State transition operation $\rightarrow$ is defined for every instructions. Function $\mathcal{V}$ defines the value of an expression under its environment. We used usual record notation for record values. We considered an array value as a record, but with an integer index as a field name. l_G indicates a reserved location for global variables. $Handler$ is a function that returns the exception handler which should be used to handle a given exception.

Our analyzer is sound with respect to Dalvik Core; anything that is not represented in the intermediate language could lead to unsoundness. For example, using Android APIs which are not manually encoded in the analyzer might lead to an unsound result. It is possible to design the analyzer sound even in those cases, but then the precision (true alarm ratio) will degrade drastically.

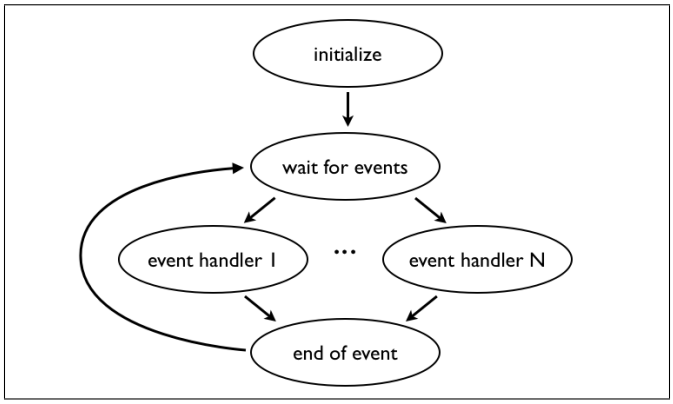

Figure 1: Dalvik Application's Execution Model

C. Translation

We designed and implemented a translator to convert Dalvik VM bytecode into Dalvik Core. The translator builds

$$
\begin{array}{rcrcl}
pgm & \in & Program & = & Instruction^{+} \times MethodTable \times HandlerTable \times Subtype \\
MT & \in & MethodTable & = & (Type \times ID) \xrightarrow{\text{fin}} PC \\
 & & HandlerTable & = & PC \xrightarrow{\text{fin}} (Type \times PC)^{*} \\
\preceq & = & Subtype & \subset & Type^{2}
\end{array}
$$

$$
\begin{array}{rcl}
ty & \in & Type \\
id & \in & ID \\
pc & \in & PC \\
instr & \in & Instruction \\
instr & ::= & \texttt{move}\ e\ e \\
 & | & \texttt{istype}\ e\ e\ ty \\
 & | & \texttt{new}\ e\ ty \\
 & | & \texttt{get}\ e\ e\ id \\
 & | & \texttt{put}\ e\ e\ id \\
 & | & \texttt{gets}\ e\ id \\
 & | & \texttt{puts}\ e\ id \\
 & | & \texttt{addcallback}\ id\ e \\
 & | & \texttt{call}\ ty\ id\ e^{*} \\
 & | & \texttt{vcall}\ id\ e^{+} \\
 & | & \texttt{return} \\
 & | & \texttt{throw}\ e \\
 & | & \texttt{jmpnz}\ e\ pc \\
 & | & \texttt{switch}\ e\ (e,\ pc)^{+} \\
 & | & \texttt{wait} \\
e & \in & E \\
e & ::= & r \mid i \mid \text{“}\texttt{str}\text{”} \mid ty \mid \star e \mid e \diamond e
\end{array}
$$

Figure 2: Syntax of Dalvik Core

control flow graph, adds hard-coded definitions of library calls, and create the main entry point according to our model of the whole execution of a Dalvik application.

Figure 1 describes the execution model of a Dalvik application. We divided an execution into two phases: initializing phase and event-handling phase. During the initializing phase, the application constructs the activity objects and initializes static variables. After initialized, the application waits for events, and calls the appropriate callback methods, called *listeners*, when an event occurs.

D. Abstract Interpretation

SCANDAL is an abstract interpreter. To detect all potential privacy leaks in Android applications, SCANDAL considers every machine state which may occur when executing applications. It computes a sound approximation of every machine state of the Dalvik Core program.

We define the abstract semantics of Dalvik Core in Figure 4. The abstract semantics follows conventional abstract interpretation designs. Abstract machine states are partitioned into sets of states by program point. The abstract semantics is the least fixed point of a function that transfers an abstract trace into next abstract trace. State transition operation $\hat{\rightarrow}$ is defined for every instruction. Formally, our analyzer computes a fixed point of the following abstract semantic function

$$\lambda \hat{S}.\hat{S}_0 \sqcup \hat{Next}\hat{S} \tag{2}$$

which is proven, within the abstract interpretation framework, a safe approximation of the collecting semantics (1). The $\hat{S}_0$ is a table from program points to their abstract initial states, and the $\hat{Next}$ function approximates the one-step execution by applying the abstract transtions operation $\hat{\rightarrow}$.

Most of the abstract domains are also defined conventionally. Integer domain is defined as the flat domain. Heap memory locations are abstracted by sets of allocation sites. All the abstract values of elements in an array are joined together. String values have a special domain called Prefix Domain, which keeps prefixes of strings. For example, in Google Wallpaper 4.2.2 we introduced in section 1.B, the leaked string which contains the device ID is represented as “http://www.imnet.us/api/wallpapers/photos/search_keywords?”*. A device ID follows the constant part of the string. Since the device ID is unique for each phone, it cannot be specified during static analysis. But SCANDAL can analyze prefixes of a string, which in this case helps identify the sink of the leak. Other values have power set

$$
\begin{array}{rcrcl}
 & & \mathit{Trace} & = & \mathit{State}^{+} \\
\mathit{st} & \in & \mathit{State} & = & \mathit{Mem} \times \mathit{Env} \times \mathit{CBEnv} \times \mathit{PC} \times \mathit{Conti} \\
 & & & & \\
M & \in & \mathit{Mem} & = & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Obj} \\
\sigma & \in & \mathit{Env} & = & \mathit{Reg} \xrightarrow{\text{fin}} \mathit{Val} \\
C & \in & \mathit{CBEnv} & = & 2^{\mathit{Id} \times \mathit{Val}} \\
K & \in & \mathit{Conti} & = & (\mathit{Env} \times \mathit{PC})^{*}
\end{array}
$$

$$
\begin{array}{rcl}
l & \in & \mathit{Loc} \\
r & \in & \mathit{Reg} \\
\mathit{obj} & \in & \mathit{Obj} = \mathit{Type} \times (\mathit{Id} \xrightarrow{\text{fin}} \mathit{Val}) \\
v & \in & \mathit{Val} \\
v & ::= & i \mid l \mid \text{“}\mathit{str}\text{”} \mid \mathit{ty}
\end{array}
$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{move}\ r\ e,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma\{r \mapsto \mathcal{V}\ \sigma\ E\},\ C,\ pc+1,\ K\rangle$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{istype}\ r_d\ r_t\ ty,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma\{r_d \mapsto 1\},\ C,\ pc+1,\ K\rangle}\ M(\sigma(r_t)).ty \preceq ty$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{istype}\ r_d\ r_t\ ty,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma\{r_d \mapsto 0\},\ C,\ pc+1,\ K\rangle}\ M(\sigma(r_t)).ty \not\preceq ty$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{new}\ r_d\ ty,\ K\rangle \hat{\rightarrow} \langle M\{l \mapsto \{\}_{ty}\},\ \sigma\{r_d \mapsto l\},\ C,\ pc+1,\ K\rangle}\ l \notin \mathit{Dom}(M)$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{get}\ r_d\ r_o\ id,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma\{r_d \mapsto M(\sigma(r_o)).id\},\ C,\ pc+1,\ K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{put}\ r_s\ r_o\ id,\ K\rangle \hat{\rightarrow} \langle M\{\sigma(r_o) \mapsto M(l) + \{id = \sigma(r_s)\}\},\ \sigma,\ C,\ pc+1,\ K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{gets}\ r_d\ id,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma\{r_d \mapsto M(l_G).id\},\ C,\ pc+1,\ K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{puts}\ r_s\ id,\ K\rangle \hat{\rightarrow} \langle M\{l_G \mapsto M(l_G) + \{id = \sigma(r_s)\}\},\ \sigma,\ C,\ pc+1,\ K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{addcallback}\ id\ r_a,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma,\ C \cup \{(id, \sigma(r_a))\},\ pc+1,\ K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{call}\ ty\ id\ r_a\ r_b\ ...,\ K\rangle \hat{\rightarrow} \langle M,\ \{r_0 \mapsto \sigma(r_a), r_1 \mapsto \sigma(r_b), ...\},\ C,\ MT(ty, id),\ (\sigma, pc); K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{vcall}\ id\ r_a\ r_b\ ...,\ K\rangle \hat{\rightarrow} \langle M,\ \{r_0 \mapsto \sigma(r_a), r_1 \mapsto \sigma(r_b), ...\},\ C,\ MT(M(\sigma(r_a)).ty, id),\ (\sigma, pc); K\rangle$$

$$\langle M,\ \sigma,\ C,\ pc : \texttt{return},\ (\sigma', pc'); K\rangle \hat{\rightarrow} \langle M,\ \sigma',\ C,\ pc'+1,\ K\rangle$$

$$\frac{(\sigma', pc', K') = \mathit{Handler}(\sigma,\ pc,\ K,\ M(\sigma(r)).ty)}{\langle M,\ \sigma,\ C,\ pc : \texttt{throw}\ r,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma'\{r_{ex} \mapsto \sigma(r)\},\ C,\ pc',\ K'\rangle}$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{jmpnz}\ e\ pc',\ K\rangle \hat{\rightarrow} \langle M,\ \sigma,\ C,\ pc',\ K\rangle}\ \mathcal{V}\ \sigma\ e \neq 0$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{jmpnz}\ e\ \_,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma,\ C,\ pc+1,\ K\rangle}\ \mathcal{V}\ \sigma\ e = 0$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{switch}\ e\ ST,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma,\ C,\ pc',\ K\rangle}\ (\mathcal{V}\ \sigma\ e,\ pc') \in ST$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{switch}\ e\ ST,\ K\rangle \hat{\rightarrow} \langle M,\ \sigma,\ C,\ pc+1,\ K\rangle}\ (\mathcal{V}\ \sigma\ e,\ \_) \notin ST$$

$$\frac{}{\langle M,\ \sigma,\ C,\ pc : \texttt{wait},\ K\rangle \hat{\rightarrow} \langle M,\ \{r_0 \mapsto l, r_1 \mapsto \mathit{input}(), ...\},\ C,\ MT(M(l).ty, id),\ (\sigma, pc); K\rangle}\ (id, l) \in C$$

Figure 3: Concrete Semantics

$$
\begin{array}{rcrcl}
 & & \hat{Trace} & = & PC \xrightarrow{\text{fin}} \hat{State} \\
\hat{st} & \in & \hat{State} & = & \hat{Mem} \times \hat{Env} \times \hat{CBEnv} \times PC \times \hat{Conti} \\
\hat{M} & \in & \hat{Memory} & = & \hat{Loc} \xrightarrow{\text{fin}} \hat{Obj} \\
\hat{\sigma} & \in & \hat{Env} & = & Reg \xrightarrow{\text{fin}} \hat{Val} \\
\hat{C} & \in & \hat{CBEnv} & = & 2^{Id \times \hat{Val}} \\
\hat{K} & \in & \hat{Conti} & = & 2^{PC} \\
\hat{l} & \in & \hat{Loc} & = & PC + \{l_G, null\} \\
\hat{obj} & \in & \hat{Obj} & = & Type \xrightarrow{\text{fin}} (Id \xrightarrow{\text{fin}} \hat{Val})
\end{array}
$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{move}\ r\ e,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r \mapsto \hat{\mathcal{V}}\ \hat{\sigma}\ e\},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{istype}\ r_d\ r_t\ ty,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r_d \mapsto 1\},\ \hat{C},\ pc+1,\ \hat{K}\rangle}\ \forall l \in \hat{\sigma}(r_t)\ \forall ty' \in Dom(\hat{M}(l))\ ty' \preceq ty$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{istype}\ r_d\ r_t\ ty,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r_d \mapsto 0\},\ \hat{C},\ pc+1,\ \hat{K}\rangle}\ \forall l \in \hat{\sigma}(r_t)\ \forall ty' \in Dom(\hat{M}(l))\ ty' \not\preceq ty$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{istype}\ r_d\ r_t\ ty,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r_d \mapsto \top\},\ \hat{C},\ pc+1,\ \hat{K}\rangle}\ otherwise$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{new}\ r_d\ ty,\ \hat{K}\rangle \hat{\to} \langle \hat{M}\{pc \mapsto \{\}_{ty}\},\ \hat{\sigma}\{r_d \mapsto \{pc\}\},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{get}\ r_d\ r_o\ id,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r_d \mapsto \bigsqcup_{l \in \hat{\sigma}(r_o)} \hat{M}(l).id\},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{put}\ r_s\ r_o\ id,\ \hat{K}\rangle \hat{\to} \langle \bigsqcup_{l \in \hat{\sigma}(r_o)} \hat{M}\{l \mapsto \hat{M}(l) + \{id = \hat{\sigma}(r_s)\}\},\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{gets}\ r_d\ id,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma}\{r_d \mapsto \hat{M}(l_G).id\},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{puts}\ r_s\ id,\ \hat{K}\rangle \hat{\to} \langle \hat{M}\{l_G \mapsto \hat{M}(l_G) + \{id = \hat{\sigma}(r_s)\}\},\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{addcallback}\ id\ r_a,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C} \cup \{(id, \hat{\sigma}(r_a))\},\ pc+1,\ \hat{K}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{call}\ ty\ id\ r_a\ r_b\ ...,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \{r_0 \mapsto \hat{\sigma}(r_a), r_1 \mapsto \hat{\sigma}(r_b), ...\},\ \hat{C},\ MT(ty, id),\ \{pc\}\rangle$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{call}\ ty\ id\ r_a\ r_b\ ...,\ \hat{K}\rangle \hat{\to} \langle \bot,\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{vcall}\ id\ r_a\ r_b\ ...,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \{r_0 \mapsto \hat{\sigma}(r_a), ...\},\ \hat{C},\ MT(ty, id),\ \{pc\}\rangle}\ l \in \hat{\sigma}(r_t), ty \in Dom(\hat{M}(l))$$

$$\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{vcall}\ id\ r_a\ r_b\ ...,\ \hat{K}\rangle \hat{\to} \langle \bot,\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{return},\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \{r_{ret} \mapsto \hat{\sigma}(r_{ret}), r_{ret+1} \mapsto \hat{\sigma}(r_{ret+1})\},\ \hat{C},\ pc'+1,\ \bot\rangle}\ pc' \in \hat{K}$$

$$\frac{(\hat{\sigma'},\ pc',\ \hat{K'}) \in \bigsqcup_{ty \in \hat{\sigma}(r)} \hat{Handler}(\hat{\sigma},\ pc,\ \hat{K},\ ty)}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{throw}\ r,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma'}\{r_{ex} \mapsto \hat{\sigma}(r_a)\},\ \hat{C},\ pc',\ \hat{K}\rangle}$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{jmpnz}\ e\ pc',\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc',\ \hat{K}\rangle}\ \hat{\mathcal{V}}\ \hat{\sigma}\ e \neq 0$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{jmpnz}\ e\ \_,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle}\ \hat{\mathcal{V}}\ \hat{\sigma}\ e = 0 \vee \hat{\mathcal{V}}\ \hat{\sigma}\ e = \top$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{switch}\ e\ ST,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc',\ \hat{K}\rangle}\ (\hat{\mathcal{V}}\ \hat{\sigma}\ e,\ pc') \in ST$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{switch}\ e\ ST,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc',\ \hat{K}\rangle}\ \mathcal{V}\ \sigma\ e = \top \wedge (\_,\ pc') \in ST$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{switch}\ e\ ST,\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc+1,\ \hat{K}\rangle}\ (\mathcal{V}\ \sigma\ e,\ \_) \notin ST$$

$$\frac{}{\langle \hat{M},\ \hat{\sigma},\ \hat{C},\ pc : \texttt{wait},\ \hat{K}\rangle \hat{\to} \langle \hat{M},\ \{r_0 \mapsto l, r_1 \mapsto \top, ...\},\ \hat{C},\ MT(ty, id),\ \{pc\}\rangle}\ (id, l) \in \hat{C},\ ty \in Dom(\hat{M}(l))$$

Figure 4: Abstract Semantics

domains.

To detect privacy leak, we collect information on where the value was created. When a value is created at an information source, we denote the program counter of the source and send it through the analysis. When a value is created from existing values, we denote the union of all sets of the program counters from existing values. By this, we can collect every values which could be created from information sources. If such values flow out through an information sink, SCANDAL detects it and consider it as a privacy leak.

SCANDAL runs both flow-sensitive and flow-insensitive analysis for different part of an application. For the initialize phase, SCANDAL runs flow-sensitive analysis for better precision. On the other hand, during the event-handling phase, the order of executions of methods can be random. So during the second phase, flow-sensitive analysis does not give us enough precision despite of analysis cost. Therefore, for the second phase, SCANDAL runs flow-insensitive analysis.

IV. Analysis of Dalvik's Challenging Features

A. Library Call

The analyzer has to know the semantics of library function calls to analyze precisely. There are about 3,000 API classes in the Android platform, and each class contains from several to dozens of methods. The source code of such classes and methods are not included in the application.

To maintain the precision, SCANDAL includes the definitions of popular API methods and classes. We chose 220 methods which are frequently used in the popular applications we collected for the experiments. The semantics of those methods are encoded in our core language. The front-end translator transforms the library method calls into the encoded core language instruction sequences.

B. Implicit Method Invocation

The analyzer should handle methods that are never explicitly invoked in the program. Such methods might be dead code, but it is also possible that they are intended to be invoked implicitly. Every Android application frequently includes methods that are not explicitly invoked, such as *Listeners*, *Threads*, and *Intents*.

SCANDAL handles implicit method invocation. We manually coded the semantics and calling conventions of the popular API functions which add listeners, initialize threads, or activate an activity by an intent. For example, an instruction that calls `Button.setOnClickListener()` is translated into a command of Dalvik Core as `addcallback Button.onClickListener`.

C. Reflection

Reflection is a feature of Java that allows dynamic code generation. It is possible to instantiate new objects and invoke methods from the name of the classes or methods.

SCANDAL handles simple cases of reflection by analyzing string values. We run a string analysis using prefix domain, which keeps a prefix of a string as the abstracted value. By this string analysis, we can narrow down the names of objects that should be instantiated. For example, SCANDAL handles method `Class.forName()` by creating `java.lang.Class` objects according to the abstract string value of the argument.

V. Experiments

In this section, we show experiment results of SCANDAL. We tested our analyzer with applications from the official market and third-party markets.

A. Official Android Market Applications

We analyzed 90 free applications from Android Market [1]. 10 popular free applications from 9 random categories are chosen, as in July 2011. We extracted packaged applications as .apk files using adb (Android Debug Bridge), which is included in the Android SDK [2].

SCANDAL detected privacy leaks in 11 applications. Table 1 summarizes the analysis result. We manually inspected the result and confirmed every application included actual leaks. We also identified the sinks.

1) Location to advertisement server: 6 applications sent location information to advertisement servers. 3 applications included `AdMob` modules and the other 3 included `AdSenseSpec` modules. Location information was either sent directly to their servers, or used as the part of the HTML code which `WebView` module loads to render the advertisement.

2) Location to file or analytics tool server: 5 applications outputted location information as a file, and some of the applications are believed to send location information to an application analytics tool server (`http://data.flurry.com/aar.do`). All 5 applications included `FlurryAgent` module in their bytecode.

3) Phone identifiers to remote server: 1 application sent out IMEI to a remote server.

4) Eavesdropping: We found no applications that have video or audio eavesdropping behaviors.

Application	*size*	*time*	*mem*	Detected leak
Kids Preschool Puzzle	87	1	56	Location*
Job Search	167	1	108	Location**
Kids Shapes	225	1	155	Location*
Kids ABC Phonics	134	3	119	Location*
Backgrounds HD Wallpapers	109	4	141	IMEI
Bible Quotes	138	8	263	Location**
ES Task Manager	158	19	423	Location**
Multi Touch Paint	198	47	727	Location* **
Adao File Manager	255	62	1149	Location**
(D-Day) The Day Before	293	224	2657	Location**
Kids Numbers and Math	101	538	185	Location*

Table I
PRIVACY LEAKS DETECTED IN ANDROID MARKET APPLICATIONS

size is the size of the dex (Dalvik Executable) file (KB). *time* is the CPU time spent (sec). *mem* is the peak memory consumption (MB). Flurry* and advertisement** servers are identified.

B. Black Market Applications

We analyzed 8 known malicious applications from third-party ("black") markets. These applications are originally free and can be downloaded via Android Market. However, infected applications, which seems to be the same as original ones, can be found in third-party markets. We downloaded .apk files of these applications from the Internet.

Table 2 summarizes the analysis result. All of the 8 applications sent out the phone number, IMEI, IMSI, ICC-ID, and the location information. We manually inspected the result and confirmed every application included actual leaks. When the infected applications are executed, it immediately sends several private information to a remote host, which is believed to be a malicious server.

Application	*size*	*time*	*mem*
Shot Gun	95	36	164
Baseball Superstars 2010	165	61	285
CacheMate for Root Users	174	67	245
Monkey Jump 2	169	74	442
Protector	107	75	209
Gold Miner	191	81	481
Mini Army	480	174	1292
Xing Metro	253	23049	1784

Table II
PRIVACY LEAKS DETECTED IN BLACK MARKET APPLICATIONS

C. False Positives

Although every application that SCANDAL detected included actual leaks, some of the paths were identified as false positives. After computing a sound approximation of every machine state of the Dalvik Core program, SCANDAL reports a list of paths, each of which is represented as a pair of an information source and a sink, which are potential privacy leaks. We manually identified every path detected from 11 Android Market applications. Table 3 summarizes the result.

Application	*#path*	*#true*	*#false*	*#unknown*
Kids Preschool Puzzle	59	2	16	41
Job Search	7	7	0	0
Kids Shapes	140	2	20	118
Kids ABC Phonics	59	2	16	41
Backgrounds HD Wallpapers	10	1	4	5
Bible Quotes	3	3	0	0
ES Task Manager	3	3	0	0
Multi Touch Paint	80	2	6	72
Adao File Manager	14	2	2	10
(D-Day) The Day Before	14	2	2	10
Kids Numbers and Math	59	2	16	41

Table III
PATHS IDENTIFIED IN ANDROID MARKET APPLICATIONS

#path is the total number of reported paths. *#true* is the number of actual leaks which are identified. *#false* is the number of false positives which which are identified. *#unknown* is the unidentified paths.

False positives mainly occur because the actual leaks are exaggerated when analyzing library calls. SCANDAL computes a sound approximation of privacy leak information of an input application. When library functions are called during a path of a privacy leak, SCANDAL soundly approximates that private information can be exchanged between its arguments in every possible way, which leads to an exaggeration. Unless a library function call doesn't flow any information between its arguments and also discontinues itself, which is rare, there exist at least one actual leak among the exaggerated set of paths.

We failed to identify some paths for several reasons. Some of third-party Android libraries included in the applications were heavily obfuscated and made it hard to manually follow the bytecode. Some of the applications heavily used java data structures (array, list, map, ...) or JSON and made it hard to accurately identify the data flow within their structure.

VI. LIMITATIONS AND FUTURE WORK

A. Performance

The time and memory consumption during the analysis can be improved further. We have presented sparse [13] and localization [12], [14] techniques. These are strong and general optimization techniques for C static analyzer. One of our main future work is to adopt such techniques to SCANDAL and improve the performance. These performance improvements will offer different scenarios and possibilities for the use of the analyzer.

B. Java Native Interface

SCANDAL does not support the Java Native Interface (JNI) methods. JNI defines a way to interact with native code. Application developers may use JNI to incorporate the C/C++ libraries into the application. Since the target language of our analyzer is Dalvik VM bytecode, we are unable to run our analysis on JNI libraries. 16 out of 90 Android Market applications we collected for the experiments included JNI methods. Since it is possible to obtain private information using JNI, it may cause new security problems. Another one of our main future work is to extend the target language of our analyzer to support JNI.

C. Reflection

SCANDAL does not fully support reflection-related APIs. All 8 black market applications we collected for the experiments used reflection while leaking private information. We manually coded the semantics of reflection-related API functions used in such cases. However, there are other ways to use reflection, and we do not currently support them.

Also, SCANDAL can only handle simple use of reflection. By using reflection, it is possible to instantiate new objects and invoke methods from their names. In all 8 applications, the names of the classes and methods were given as string constants. We believe that malicious developers can heavily obfuscate their behaviors using reflection. In such cases, our analyzer might not be able to precisely detect privacy leaks without adopting more complicated string analysis techniques.

D. Extending Analysis

Our static analysis framework can be extended to analyze other properties of Android applications. SCANDAL is an abstract interpreter that computes a sound approximation of privacy leak information of an input application. By modifying domains and their abstractions, we can fairly easily achieve another abstract interpreter that verifies other semantical and interprocedural properties, such as array bound checking or dead code checking.

VII. Related Work

Several tools and techniques have been presented to detect privacy leaks in smartphones using static analysis, but to our best knowledge, this is the first static analyzer based on the abstract interpretation framework for Android which targets Dalvik VM bytecode. PiOS [6] presented a static analysis for Objective-C code and detected privacy leaks in iPhone applications. DroidRanger [15] and Enck et al. [8] extensively studied Android market applications and gave better understanding of the current ecosystem. TaintDroid [7] and SCanDroid [11] are, respectively, dynamic and static analyzer detecting privacy leaks in Android applications.

TaintDroid [7] monitors Android applications at runtime, sacrificing runtime performance. TaintDroid monitors Android smartphone and tracks how applications leak private information. SCANDAL is fully automated, while TaintDroid needs to execute an application to initiate privacy leaks. Also, TaintDroid causes about 14% CPU overhead for tracking, which is not an issue for SCANDAL. We believe that by combining both approaches, both work can be complementary to each other. For example, if SCANDAL guarantees an application to be safe, we can turn off the realtime monitoring while using the application.

SCanDroid [11] is limited because it cannot analyze packaged Android applications. SCanDroid is an automated static analyzer that reasons about data flows in Android applications. It checks whether data flows through an application are consistent with its permissions. However, it is not tested on real-world Android market applications. They used WALA, a collection of open-source libraries to analyze Java programs. So it cannot be immediately applied to packaged applications without the original Java code.

Enck et al. [8] used decompiling techniques that fail in some cases. They studied 1,100 popular free Android applications by using automated tools and manual inspections. They showed that Android applications often misuse users' private data. They designed and implemented a Dalvik decompiler to recover Java source of an application before analyzing. However, they failed to recover the source code of about 5% of the total classes in the applications. SCANDAL does not use reverse engineering techniques and deals directly with the bytecode. So we do not have to suffer from decompiling issues.

VIII. Conclusion

We provide a formal, sound, and automatic static analysis for detecting privacy leaks in Android applications. We tested our analyzer, SCANDAL, with real-world applications, both from the official Android market and black markets. It detected privacy leaks in 11 applications out of 90 popular applications from Android market. It also detected various privacy leaks in 8 known malicious applications.

Acknowledgment

This work was supported by Samsung Electronics DMC R&D Center, the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2012-0000468), and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University.

References

[1] Android Market. `http://market.android.com`

[2] Android SDK. `http://developer.android.com/sdk`

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short Paper: A Look at SmartPhone Permission Models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM 2011)*, 63–68, 2011

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unied lattice model for static analysis of programs by construction or approximation of xpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, 238-252, 1977.

[5] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. In *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[6] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2011)*, Feb. 2011.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*, 1–6, Oct. 2010.

[8] William Enck, Damien Octeau, Patrick Mcdaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[9] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009)*, 235–245, 2009.

[10] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS 2011)*, 627–638, 2011.

[11] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certication of Android Applications. University of Maryland Department of Computer Science Technical Report CS-TR-4991, November 2009.

[12] Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. Access Analysis-Based Tight Localization of Abstract Memories. In *Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, volume 6538 of *Lecture Notes in Computer Science*, 356–370, Jan. 2011, Springer-Verlang.

[13] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and Implementation of Sparse Global Analyses for C-like Languages. To Appear in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012)*.

[14] Hakjoo Oh and Kwangkeun Yi. Access-Based Localization with Bypassing. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS 2011)*, volume 7078 of *Lecture Notes in Computer Science*, 50–65, Dec. 2011. Springer-Verlang.

[15] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, Feb 2012.

[16] ScanDal Website. http://ropas.snu.ac.kr/scandal