Journal of Information Processing Vol.24 No.4 700–710 (July 2016)

[DOI: 10.2197/ipsjjip.24.700]

Regular Paper

LogChamber: Inferring Source Code Locations Corresponding to Mobile Applications Run-time Logs

Yuki Ono[1,a) Kouhei Sakurai[1,b) Satoshi Yamane[1,c)

Received: October 5, 2015, Accepted: January 25, 2016

Abstract: We present a development support tool, called *LogChamber*, which infers source-code locations by analyzing run-time logs of mobile applications. During development, developers insert log functions into applications calls in order to confirm that the applications correctly run as expected. After that, they need to have a process for estimating a program's runtime behavior in order to identify the locations of unintended behavior. Such processes rely on the abilities of the developers and are not easy in many cases. Most runtime environments of mobile applications provide only limited resources, and as a result, cannot save sufficiently many runtime logs. The situation is made even worse by careless insertions of log-function calls. The method presented in this paper analyzes static source code and runtime logs. After that, it supports developers by quickly inferring candidates of log function calls. For fast inference of candidates, it extracts log strings from the source code and constructs an index of their locations in advance. We implemented our method as a plugin tool on Android Studio, one of the major integrated development environments for Android applications. We report our experiments with the implementation on real open-source applications.

Keywords: logs, static analysis, debug, mobile application, Android

1. Introduction

Mobile devices and applications for mobile operating systems ("mobile apps" for short) have grown in recent years. The major mobile OSs include Android OS by Google, iOS by Apple, and Windows Phone by Microsoft. To create a mobile app, a developer first writes source code in a development environment. After that, he or she checks the actual running of the application in a runtime environment with real devices like smartphones and tablets. To guarantee that the app runs correctly, the developer needs to 1) write different versions of the source code for different runtime environments and 2) check and debug them in these environments.

In many cases, developers insert log-function calls into the source code of the application, which generate *runtime logs* at runtime, in oder to check whether the mobile app runs an expected. In many cases, the runtime logs are text lines generated by log-function calls inserted into the source code that contain constant values of strings and runtime values of variables which will serve as hints for the developer. If the application contains an unintended behavior, the runtime logs of the abnormal behavior will reflect it.

To identify the locations of unintended behaviors, developers need to precisely estimate the runtime behavior of the program from the generated logs. Most developers of mobile apps rely on only the runtime logs. Because the runtime environments for apps tend to have limited resources, and it is difficult to apply debuggers recording execution histories [8]. Moreover, it is not easy to reproduce unintended behaviors occurred on various runtime environments and identify corresponding locations from runtime logs, and thus such tasks increase debugging time. For example, there was a case in which the Android OS had an issue that a camera application behaved differently on a specific device, and the developer needed to find the cause of the unintended behavior on that device (The details of this case are illustrated in Section 2).

In this study, we propose a developer support tool called *LogChamber*, which makes correspondences between runtime logs and source-code analyses and infers candidate locations where the runtime logs might have been generated. LogChamber analyzes the program of the target application beforehand, and it identifies the locations of the log-function calls generating the runtime logs. Furthermore, it constructs an index by analyzing the arguments of the function calls in detail so that it can quickly make correspondences with the runtime logs. By using LogChamber, the developer can estimate the locations of the runtime logs that are inserted on an ad hoc basis and easily analyze the behavior of the application. In addition, since the inference and display of locations in LogChamber is fast, the developer can see the corresponding location in the source code as soon as the application on the running device generates a runtime log.

We implemented LogChamber as a plug-in of an integrated development environment for Android OS (called Android Studio) and carried out experiments using it on real Android mobile apps. In this paper, we present the procedure of identifying actual unintended behavior in a mobile app by using LogChamber.

Section 2 describes the problems of debugging mobile apps

1 Graduate School of Natural Science & Technology, Kanazawa University, Kanazawa, Ishikawa 920–1192, Japan
a) ono@csl.ec.t.kanazawa-u.ac.jp
b) sakurai@ec.t.kanazawa-u.ac.jp
c) syamane@is.t.kanazawa-u.ac.jp

through an illustrative example. Section 3 describes LogChamber in detail. In Section 4, we report our experiments applying LogChamber to actual open-source applications. Section 5 discusses related work, and Section 6 concludes the paper.

2. Debugging of Mobile Applications

In this section, we explain the problem of checking and debugging mobile apps with the help of an illustrative example.

A mobile app, which is different from a desktop application or web application, runs on low resource environments and on many types of devices, and it tends to produce unintended behaviors corresponding to its behavior on a specific device with its own hardware configuration. Below, we give an example illustrating debugging using the runtime logs of mobile apps. In Section 2.2, we explain the problems of debugging mobile apps with popular breakpoint debuggers. In Section 2.3, we summarize the problems of using the runtime logs.

2.1 Example: Obtaining the Orientation in a Camera Application

Figure 1 shows a portion of code that implements taking a picture by a camera in an Android application. It contains an unintended behavior that appears when it is used on a specific device and environment, and the unintended behavior is that it outputs a picture rotated at an unnatural angle. To check for this unintended behavior and fix it, the code contains lines of code for generating runtime logs for debugging (lines 4, 7, 13, 15 and 18).

When a user takes a picture with the camera, the `onPictureTaken` method is called as a callback, and the JPEG data of the taken picture is given as the argument `data` (line 2). The method saves the picture in a file (lines 4–9), obtains the orientation of the device of the shot from the Exif data (meta-data), and assigns it to `r` (line 10).

Regarding the unintended behavior, when the device takes a picture and displays its JPEG data, the method needs to obtain the orientation of this data and determines the correct orientation of the picture. For example, if the user takes a picture while turning the device 90 degrees, the generated JPEG data by the device should also be rotated 90 degrees. Thus, the display of the picture at this angle is an unnatural one for the user. To resolve this issue, one needs to obtain and check the meta-data of the JPEG data and manually rotate the picture to the correct angle.

Lines 13 and 15 are log-function calls for debugging purposes, and they generate a value `r` with a message.

For instance, the case of "`JPEG rotation = 1`" means displaying the JPEG data as it is; in this case, the application does not need to rotate the picture data. The variable `r` takes values from 1 to 8 [*1], and it needs to run a rotation process corresponding to each value. Furthermore, there are devices that take `r = 0` (UNDEFINED); in such cases, it needs to run a special rotation process considering the aspect ratio of the picture [*2].

2.2 Problem with the Usage of the Breakpoint Debugger

Breakpoint debuggers are popular tools for checking the behavior of and debugging applications. With the breakpoint debugger, the developer first attaches breakpoints at arbitrary locations in the target program and can temporarily stop the running of the program while inspecting its states including the actual values of variables. The debugger then executes the program step by step, i.e., from one breakpoint to the next, and the developer checks whether the program behaves as expected in each step. For example, for code of Fig. 1, we can confirm that an actual value of the variable `r` by execution with setting a breakpoint at the line 11.

However, the breakpoint debuggers have difficulty managing many breakpoints and steps; these problems result in heavier workloads for the developer.

In addition, for debugging, developers need to reproduce unintended behaviors. It is not practical for them to attempt to reproduce an unintended behavior on the specific device, like in the example shown in Section 2.1.

Instead, they should try to write log-functions in the application and checks the generated runtime logs against the actual source code in order to estimate the actual behavior of the program.

```
 1  // a callback after generation of a JPEG image
 2  public void onPictureTaken(byte[] data,
 3                              Camera camera) {
 4    Log.d(TAG, "Start Save JPEG");
 5    boolean failed = saveImageToFile(data,FILE_NAME);
 6    if (failed) {
 7      Log.d(TAG, "Save JPEG Failed");
 8      return;
 9    }
10    int r = getExitInterface(FILE_NAME);
11    if (r != 0) { //The JPEG rotation value is
12                  //different from the device
13      Log.d(TAG, "JPEG rotation = " + r);
14    } else {
15      Log.d(TAG, "Illegal rotation = " + r);
16      return;
17    }
18    Log.d(TAG, "Saved JPEG Size " + data.length);
19    camera.startPreview();//continues previewing
20  }
```

Fig. 1 Example of taking a picture with different results by the type of the device.

```
1  10-04 04:31:31.076 820-5820/camera_sample
     W/Resources Converting to string:
     TypedValue{t=0x5/d=0xa01 a=1 r=0x10500d8 }
2  10-04 04:34:34.887 820-5820/camera_sample
     D/screen2camera Start Save JPEG
3  10-04 04:34:35.023 820-5820/camera_sample
     D/screen2camera Illegal rotation = 0
```

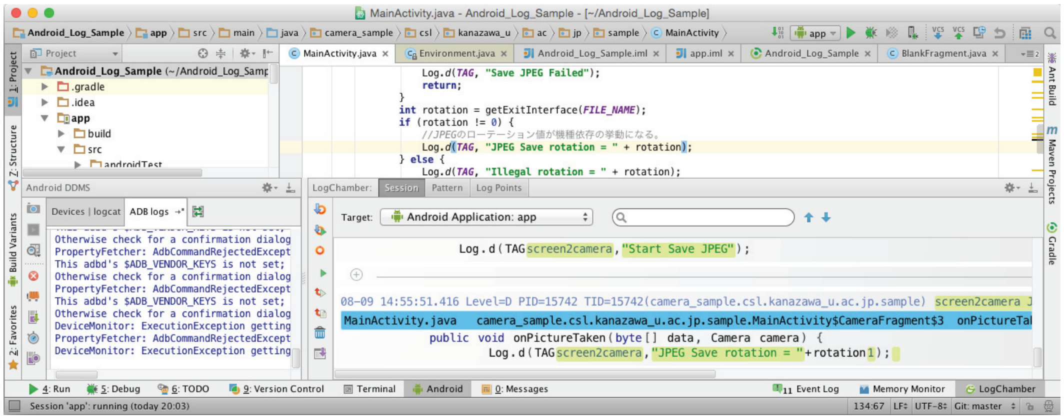
Fig. 2 Example of runtime logs.

2.3 Problems of Uniqueness and Identification of Runtime Logs

Runtime logs contain lines generated by the OS and applications other than the debugging target, so the developer needs to identify the logs generated by the target application. Once the developer has identified the application, he or she needs to identify the corresponding locations in the source code of the application.

Figure 2 shows actual runtime logs generated by the application including the code shown in Fig. 1. Each log contains a timestamp, a process ID, and a tag string as hints for the identifi-

*1 http://www.sno.phy.queensu.ca/~phil/exiftool/TagNames/EXIF.html
*2 http://hackmylife.net/archives/7400448.html



```
MainActivity.java - Android_Log_Sample - [~/Android_Log_Sample]

            Log.d(TAG, "Save JPEG Failed");
            return;
        }
        int rotation = getExitInterface(FILE_NAME);
        if (rotation != 0) {
            //JPEGのローテーション値が機種依存の挙動になる。
            Log.d(TAG, "JPEG Save rotation = " + rotation);
        } else {
            Log.d(TAG, "Illegal rotation = " + rotation);
```

```
Otherwise check for a confirmation dialog
PropertyFetcher: AdbCommandRejectedExcept
This adbd's $ADB_VENDOR_KEYS is not set;
Otherwise check for a confirmation dialog
PropertyFetcher: AdbCommandRejectedExcept
This adbd's $ADB_VENDOR_KEYS is not set;
Otherwise check for a confirmation dialog
DeviceMonitor: ExecutionException getting
PropertyFetcher: AdbCommandRejectedExcept
DeviceMonitor: ExecutionException getting
```

```
Target: Android Application: app

        Log.d(TAG screen2camera, "Start Save JPEG");

08-09 14:55:51.416 Level=D PID=15742 TID=15742(camera_sample.csl.kanazawa_u.ac.jp.sample) screen2camera J
MainActivity.java   camera_sample.csl.kanazawa_u.ac.jp.sample.MainActivity$CameraFragment$3   onPictureTal
      public void onPictureTaken(byte[] data, Camera camera) {
            Log.d(TAG screen2camera, "JPEG Save rotation = "+rotation1);

Session 'app': running (today 20:03)                         134:67  LF  UTF-8  Git: master
```

Fig. 3 Screenshot of LogChamber running on the integrated development environment Android Studio.

cation. The tag string is used as a string constant value that is supplied to the log-function calls, and in Fig. 1, it is `TAG`, the first argument of line 13. The developer can make use of the identification of generated logs by assigning the string `"screen2camera"` to `TAG` of line 13 in advance. In Fig. 2, line 1 does not contain `"screen2camera"`, so it comes from the API. On the other hand, lines 2 and 3 contain this tag, so they were generated by line 13 in Fig. 1.

Log-function calls are not always systematically written in order to be identifiable; thus, the uniqueness of the generated logs is not guaranteed. For example, if there are usages of the API of the camera shot, the developer might write a log-function call similar to line 13 for each location in which it is used. Lines 4, 15 and 18 in Fig. 1, use the same `TAG` string.

Log-function calls are prone to be repeatedly inserted and deleted in an ad hoc manner during debugging. As a result of the frequent rewriting of calls, even a hint for identification like `TAG` will have a certain amount of granularity; i.e., at the level of a Java package or a class module. Such hints are often limited to the level a module of the target applications (instead of the exact lines of source code).

Furthermore, because the runtime logs are generated from the expressions in the source code, more complex expressions make identification of log locations more difficult. For example, although `+ r` at line 13 of Fig. 1 concatenates the string `r`, this string is actually converted from an integer, and the identification becomes harder if the developer does not know that. The expression in the example is simple, yet there are log-function calls that contains multiple variables. As a result, the identification of the source-code locations from the generated logs becomes time consuming. We investigated the logs generated by several applications and found that they are generally comprehensible as English words. Hence, our basic idea was to infer log-function calls in the source code by dividing the log lines into words.

3. LogChamber: A Tool for Inferring Source-code Locations from Runtime Logs

We developed a tool that infers the locations of source code from the runtime logs of a mobile app. In particular, we focused on Android applications and developed a plugin called *LogChamber* for the standard integrated development environment, Android Studio (IntelliJ IDEA). This section explains how LogChamber is run on a target Android application.

3.1 Overview

LogChamber analyzes the runtime logs and source code of the application and infers the locations in the source code responsible for generating those logs.

Generally, as shown in Section 2.3, the runtime logs contain mixed outputs from multiple applications and the OS. There is also a limited amount of storage available for log outputs. It is difficult to uniquely identify the exact location which generated a runtime log from the log itself. Therefore, our method tries to infer locations that might have generated the runtime logs.

Figure 3 shows a screenshot of an example of LogChamber running on Android Studio. LogChamber reads the runtime logs and displays them and the fragments of source code inferred from them in the lower part of the window.

Figure 4 illustrates the entire LogChamber system and indicates the individual steps using numbers. The locations generating logs are inferred as follows.

(1) Extract the pre-defined log-function calls from bytecodes contained in the application packages.

(2) Statically analyze the arguments of each log-function call, and divide constant values of the strings within the arguments into words. The system also constructs a word list of the log-function calls and associates them with information on the occurrence locations in the source code.

(3) Analyze the runtime logs and divide each line into words by using particular delimiters like white space. The system calculates a coincidence degree for the word list of each log-function call from the word list of each log line.

(4) Display the location of the source code that has highest coincidence as the candidate of the inferred log-function call for the runtime log. The system displays the associated argument values of the log-function call and sub-strings.

The above identification procedure has the following advantages.

- The developer does not need to manage tag strings, etc., thanks to the inferences made by the system, so he or she can easily handle even code including changes in log-function calls in an ad hoc manner.

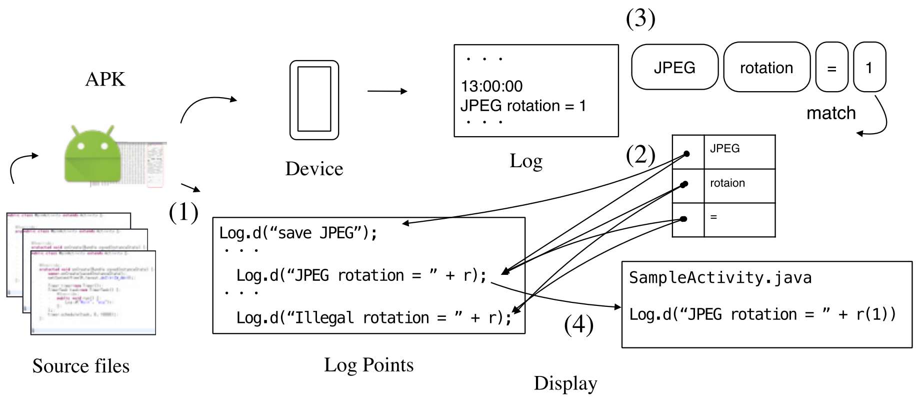


Fig. 4 Structure of LogChamber.

Table 1 Examples of log functions of Android.

`Log`	Debugging message class (Android standard)
`Logger`	Debugging message class (Java standard)
`PrintStream#print`	Output function (Java standard)
`PrintWriter#print`	Output function (Java standard)
`Console`	Console class (Java standard)

- Thanks to the feature of supplying saved runtime logs, the system can artificially reproduce the behavior of the log outputs.
- The system can accept onsite runtime logs from a running application as inputs and lets the developer cooperate with the breakpoint debugger.

The following sub-sections explain the above steps.

3.2 Extracting Expressions of Log-function Calls

In order to extract the expressions of log-function calls that could be candidate locations, LogChamber analyzes the bytecode of the Dex file contained in the application package (APK) file of the Android application. When the developer starts LogChamber, the system automatically builds the project in Android Studio and generates an APK file containing information for debugging. Then, it runs the analysis by using Soot [11] [*3], which is a Java bytecode analysis framework.

Bytecodes generated by Java compilers save instructions corresponding to the source code. Therefore, traversals for extracting methods and the data-flow analysis explained in Section 3.3 can produce the same results as analyzing the source code.

In the analysis, the system searches for the method calls listed in **Table 1** in the instructions of each method body. These methods are log functions used in Android. Developers can specify other methods by using regular expression patterns.

The system regards expressions of the searched log-function calls as targets of the analysis explained in Section 3.3, in order to infer locations of runtime logs.

3.3 Analyzing Expressions of Log-function Calls

To identify the target log functions from the runtime logs, the system analyzes the strings used in the arguments of the functions as keywords and constructs a list of words with the associated locations in the program. The system uses the getLocAndWords

$$\begin{aligned}
&\mathrm{getLocAndWords}(m(e_1,\ldots,e_x)_{loc}) = \\
&\qquad (loc, \mathrm{words}(e_1) \oplus \mathrm{words}(e_2) \oplus \cdots \oplus \mathrm{words}(e_x))
\end{aligned}$$

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathrm{words}(e_i) = \mathrm{match}\ e_i\ \mathrm{with}} \\
\mathit{Const} \rightarrow & \mathrm{splitWords}(\mathit{Const}) \\
\mathit{Var} \rightarrow & \mathrm{words}(\mathrm{def}(\mathit{Var}).\mathrm{rhs}) \\
e_l + e_r \rightarrow & \mathrm{words}(e_l) \oplus \mathrm{words}(e_r) \\
\texttt{String.format}(\texttt{"}c_f\texttt{\%}p_f\texttt{"}, e_f) \rightarrow & \mathrm{words}(\texttt{"}c_f\texttt{"}) \oplus (\texttt{\%}f) \\
\mathrm{otherwise} \rightarrow & (\texttt{\%}i)
\end{array}$$

Fig. 5 Algorithm for obtaining locations of log-function calls and a word list.

function shown in **Fig. 5** to analyze the locations in the program and the argument strings.

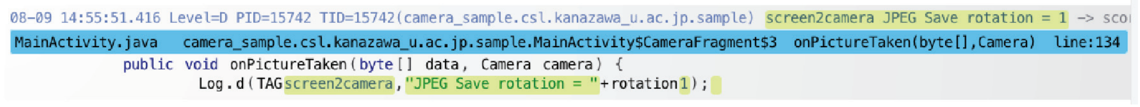
getLocAndWords takes the program expression in the syntax-tree $m(e_1,\ldots,e_x)_{loc}$ for log functions which take x arguments and returns the source-code locations of the log functions *loc* and a word list (w_i) of the strings composing the logs. Let $\oplus$ be the operation for concatenating two word lists and *words* be a function for getting a word list from an expression.

words(e_i) returns a result depending on the type of the ith expression e_i in the program.

- For a string constant value *Const*, it splits the string into a word list. Note that words here include special symbols. This is because a string of a generated log is usually short and can be characterized by symbols. For example, the string `"JPEG rotation ="` yields the word list (`JPEG`, `rotation`, =). In addition, a static field like `TAG` in Fig. 1 is handled as a string constant value.
- For a local variable reference *Var*, it recursively obtains a word list from the expression of the right hand side .rhs of a uniquely determined assignment expression def(*Var*). Although multiple assignment expressions for a local variable may exist as a result of a join of branches and so on, we do not consider such cases for the sake of simplicity [*4]. Recursive applications of words via such the local variable reference *Var* and its assignment expression def(*Var*) refer to expressions in other statements of the same method, and this means traversing the method. To handle such expansions of local variables, the system analyzes the data flows within the method.
- For a string concatenating expression $e_l + e_r$, it concatenates

[*3] https://github.com/Sable/soot: We use the version supporting Dex file analysis (commit ID: `36f9765`).

[*4] The system handles an individual log-function call for each combination of statements of local variable assignment in the entire expression.



```
08-09 14:55:51.416 Level=D PID=15742 TID=15742(camera_sample.csl.kanazawa_u.ac.jp.sample) screen2camera JPEG Save rotation = 1 -> sco
MainActivity.java   camera_sample.csl.kanazawa_u.ac.jp.sample.MainActivity$CameraFragment$3   onPictureTaken(byte[],Camera)   line:134
            public void onPictureTaken(byte[] data, Camera camera) {
                  Log.d(TAGscreen2camera,"JPEG Save rotation = "+rotation1);
```

Fig. 6 A log-function call and the surrounding source code.

word lists obtained from the left and right expressions. The analysis of the bytecode and string concatenating operations in Java are implemented using `StringBuilder`, so such an expression actually becomes an `append` method of the `StringBuilder` class.

- Runtime logs are frequently generated from formatting function calls (`String.format`); thus, the system specifically analyzes such calls. A format function takes arguments ("c_f%p_f", e_f). These arguments consist of a sequence of conversions and the fth of them corresponds to the string c_f concatenated while conversion, the conversion specifier %p_f, and the expression e_f generates the value of the conversion. For such a format function call, the system returns the line that concatenates the special word %f which indicates a word generated by the expression e_f of the conversion.
- For other expressions, the system handles a special word %i that is dynamically generated from an expression e_i. Note that these special words are used later for displaying candidates, but are not used in the calculation of coincident degrees.

For example, from line 14 of the source code in Fig. 1, we obtain the word list
(`screen2camera`, `JPEG`, `rotation`, =, %i).

3.4 Analysis of Runtime Logs and Calculation of Coincidence

To associate the log-function calls with the lines of the logs, LogChamber calculates their coincidence degrees. We define the calcuation of the coincidence degree as the ratio of the number of all lines including words in word lists obtained from each log-function in the source-code. Regarding the definition of coincidence, we consider that word lists are usually short and that those of the runtime logs may contain words that are not included in the word list of the source code but are generated from a variable.

For log line r consisting of word list $(n_{i_1}, \ldots, n_{i_{|r|}})$, the coincidence of each log function calls consisting of word list $(n_{j_1}, \ldots, n_{j_{|l|}})$ is provided from the following expressions.

$$s(r, l) = \sum_{k=1}^{|r|} f_1(l, i_k) + f_2(l, i_k, i_{k+1})$$

$$f_1(l, x) = \begin{cases} \frac{1}{|\{l' : n_x \in l'\}|} & \text{if } n_x \in l \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(l, x, y) = \begin{cases} \frac{1}{|\{l' : (n_x, n_y) \lhd l'\}|} & \text{if } (n_x, n_y) \lhd l \\ 0 & \text{otherwise} \end{cases}$$

where the operator $\lhd$ represents that (n_x, n_y) is sequentially included in a word list l for each log-function call. This means that a word list contains words like $(\ldots, n_x, n_y, \cdots)$.

f_1 and f_2 calculate the frequency of word occurrences. $|\{l' : n_x \in l'\}|$ represents the number of log-function calls that contain a word n_x. Similarly, $|\{l' : (n_x, n_y) \lhd l'\}|$ represents the number of log-function calls that contain a sequence of two words n_x, n_y with the same order.

The system selects the l whose coincidence degree $s(r, l)$ is the highest of all log-function calls as the location of the source code generating the log line.

As an example, let us consider the five log-function calls (4, 7, 13, 15, 18 lines) in Fig. 1 and their correspondence to line 2 of the log. For simplicity, we suppose there are no other log functions.

Suppose that L_x represents the location of line x of each log-function call in Fig. 2 ; then for the word list (`screen2camera`, `Start`, `Save`, `JPEG`) of the log line, a log-function call including each word is calculated as follows:

$$(\{L_4, L_7, L_{13}, L_{15}, L_{18}\}, \{L_4\}, \{L_4, L_7\}, \{L_4, L_7, L_{13}, L_{18}\})$$

From the words and occurrence frequencies, the totals of f_1 and f_2 and the values of s are calculated as follows:

L_x	$\sum f_1$	$\sum f_2$	s
L_4	$0.2 + 1 + 0.5 + 0.25$	$1 + 1 + 0.5$	4.45
L_7	$0.2 + 0 + 0.5 + 0.25$	$0 + 0 + 0.5$	1.45
L_{13}	$0.2 + 0 + 0 + 0.25$	$0 + 0 + 0$	0.45
L_{15}	$0.2 + 0 + 0 + 0$	$0 + 0 + 0$	0.25
L_{18}	$0.2 + 0 + 0 + 0.25$	$0 + 0 + 0$	0.45

For L_4, because all words occur, f_1 and f_2 add up to a value larger than 0, and this sum has the largest s value of all locations. As a result, L_4 is the inference candidate with the highest s value, and it is associated with the log line.

3.5 Displaying Inference Candidates of Log Functions

LogChamber displays a log-function call that has a word list of the highest coincident degree for a runtime log. To help developers intuitively understand the list, the display includes the source code around lines of the log-function call, as well as sub-strings of the log-line probably generated from expressions in the source code.

Figure 6 shows an example of the display. The first line is the generated log line, and the second line represents the corresponding location in the source file and its line number. The third line is the method signature part of the corresponding location, and fourth line shows the composition of the log-function call with sub-strings of the generated log.

Let us consider a word list
$(\ldots, n_i, n_{i+1}, \ldots, n_{j-1}, n_j \ldots)$ of a runtime log and %x specifying a word generated from an expression e_x, and suppose that the word list $(\ldots, n_i, \%x, n_j, \ldots)$ of the log-function call that has a highest coincidence degree is obtained. A partial word list of the runtime log $(n_{i+1}, \ldots, n_{j-1})$ corresponds to %x, and it is a string generated

Table 2 Performance of the analysis of applications

App	Time(sec)	Classes	Methods	LPs	SR	Words	$\|LP\|_{avr}$
Chapel Hill Transit	67.22	4,445	30,481	683	0.61	2,907	6.96
Tacere	84.45	4,433	35,562	454	0.80	2,727	8.79
WiGLE Wifi Wardriving	65.89	3,993	28,230	732	0.62	2,726	6.77
Aizōban	46.53	3,032	21,988	449	0.75	2,264	8.03
Anthology for Gives Me Hope	26.07	2,635	19,329	283	0.81	1,765	9.80
ownCloud News Reader	28.48	2,261	20,257	500	0.83	2,809	8.44
Password Store	55.84	4,363	35,707	358	0.82	2,156	9.59
HackWinds	18.78	1,900	14,187	309	0.81	1,626	8.57
weiciyuan	23.99	1,728	11,902	184	0.93	1,268	10.57
Lightning Browser	23.42	969	6,357	234	0.80	1,356	9.85
Mean	44.07	2,976	22,400	419	0.78	2,160	8.74

from e_x. When displaying the lines of the source code, the system inserts the word list $(n_{i+1}, \ldots, n_{j-1})$ just after e_x.

`%x` on a word corresponding to e_x is obtained using words described in Section 3.3. The words do traversal by using data-flow of local variables, thus expressions in the method including the log-function calls. The traversal is actually carried out on the bytecode; thus, the locations for inserting sub-strings of the runtime logs are determined by referring to the location information of the source code of the debugging information corresponding to e_x.

4. Evaluation

To assess the effectiveness of *LogChamber*, we carried out experiments on open-source Android applications.

The environment consisted of Mac OS X (10.10.5), Intel Core i5 1.4 GHz CPU, 16 GB RAM Java 1.8.0_20, Android Studio 1.3.2 (-Xmx 8g), and Google Nexus 5 (the device running the target applications).

Section 4.1 describes the performance of LogChamber's program analysis, and Section 4.2 examines its effectiveness at fixing unintended behaviors of an actual application.

4.1 Performance of LogChamber

LogChamber requires computation time and memory ((1) and (2) in Section 3.1) of the development environment in order to analyze the target application. Hence, we opened a development project in Android Studio and measured the performance of the application process of the LogChamber analysis.

Section 4.1.3 analyzes the log-function calls in the source code and compares LogChamber with existing text-search methods, Section 4.1.4 examines the limitations of LogChamber, and Section 4.1.5 tackles the question of whether the log functions affect the success rates of inferring log-function calls.

4.1.1 Measurement Targets and Methods

As measurement targets, we selected apps registered in F-Droid [*5], which is a distribution site for open-source Android apps. We automatically built the apps from the registered repositories on F-Droid, analyzed the generated APK files, and selected nine apps that had larger numbers of classes and the Lightning Browser app (described in Section 4.2).

As performance criteria of LogChamber, we can consider not only the time for the analysis but also the accuracy of inferring candidate log-function calls (the success rate). However, it is difficult to define and measure accuracy because it is affected by the dynamic log generations of apps. Consequently, we regarded word lists that are statically extracted from the log-function calls by the algorithm explained in Section 3.3 as the generated logs and determined whether those function calls become candidates of the inferred log-function calls as they are.

4.1.2 Measurement Results

Table 2 summarizes the results. The columns of the table list the following values:

App:	The application name.
Time:	The elapsed analysis time (in seconds).
Classes:	The number of classes included in the APK file. Note that the APK file contains classes of libraries (except for Android SDK) that the application depends on.
Methods:	The number of method definitions included in the APK file.
LPs:	The number of log-function calls.
SR:	The success rate of inferring log-function calls. We regard a statically generated word list from each log-function call as a generated log and define a success as a case when the original log function becomes a candidate inferred by analysis. In actual cases, however, the output values are generated from dynamic values of variables; thus, the actual success rate may be lower than this value.
Words:	The size of the set of all words included in the application.
$\|LP\|_{avr}$:	The mean number of words of the log-function calls.

The results show that the analysis finished within 100 seconds even if the application had over 4,000 classes. Although LogChamber traversed the entire source code of the program, it was a so-called intra-procedural analysis; i.e., it did not handle inter-methods flows or global data flows. For that reason, it was sufficiently fast to be practical.

The success rates (SR) of inferring the log-function were almost 60–80%. As mentioned above, in actuality, the generated logs would contain output values generated from dynamic values; thus, the actual success rate would be lower. Meanwhile, from the entire values, we expect that the success rate does not closely correlate with the size of the application but is affected by the style in which the log-function calls are written. In summary, it is expected that when a developer writes log-function calls, LogChamber practically works if he or she is aware of a variety of words used in the string constant values.

[*5] https://f-droid.org

```
public static final int SCHEMA_VERSION = 4;
Log.i("greenDAO", "Creating tables for
                 schema version " + SCHEMA_VERSION);
```

Fig. 7 A code fragment whose log-function call contains a reference to a constant value.

4.1.3 Comparison with Text Searching Techniques

This section compares the performance of LogChamber with that of text search tools using pattern matching (like `grep`) by using examples from the target applications.

When string constant values are written in log-function calls, we can use text search tools to infer the candidates of the source-code locations. For example, the developer can use `grep` to search the entire source code while at the same time replacing arbitrary words in a log line with wildcards (`.*`).

However, if an expression of the corresponding log-function calls constructs an output string through dynamic computations, the pattern matching of a naive text search will fail.

For instance, the `DaoMaster` class in the ownCloud News Readers app has a log-function call like that shown in **Fig. 7**. If the searched for pattern contains the value 4 of `SCHEMA_VERSION` included in the generated logs, a naive text search cannot find the site. Accordingly, there are cases of computations constructing arguments of log-function calls.

The developer needs to estimate patterns for using text searching, but there would likely be cases in which the boundaries between sub-strings generated from variables and constant string values are unclear. In the example shown in Fig. 7, a tag string like `greenDao` and the words `tables`, `schema` and `version` in the string constant value are also used in other log-function calls; thus, we cannot make an identification by using only those words as a pattern.

Our inference based on the source-code analysis works better than a naive text search when some of the words occurring in the generated logs are in string constant values composed of arguments of a log-function call. The structures of the source code and the strings of the logs are such that locations of the definition of a constant value and the log-function call may be far away from each other or that many locations use the same constant value; thus, with a naive text search, we need to manually search log-function calls repeatedly.

In addition, our method automatically divides entire lines of the generated logs into words. As a result, the developer does not need to estimate a string pattern for searching.

4.1.4 Limitations of Our Algorithm

We analyze cases in which inaccurate candidates were inferred on the measurement targets of Section 4.1.

The locations of many of the failed inferences are log-function calls in classes contained in the sub-packages of `android.support` and `com.google.android`, in the additional libraries included in the Android standard SDK [*6]. Developers rarely refer to those locations, and in LogChamber, we can easily exclude them by using package names. For example, the Chapel Hill Transit app has 683 locations of log-function calls; 267 cases (SR = 0.61) failed, but there were only 18 locations (about 2%) other than those of the additional libraries of the standard SDK. The other applications showed similar tendencies.

[*6] These are selectively used besides the basic classes in the standard SDK, thus, they are included in the APK file.

```
public static PublicKey generatePublicKey
    Log.e(TAG, "Invalid key specification.");
    Log.e(TAG, "Invalid key specification.");
```

Fig. 8 Example of identical log-function calls.

```
} catch (FileNotFoundException e) {
  Log.e(TAG, e.getMessage());
} catch (IOException e) {
  Log.e(TAG, e.getMessage());
```

Fig. 9 A log-function call that does not use any string constant values as its message body.

```
void logDebug(String msg) {
 if (mDebugLog) Log.d(mDebugTag, msg);
}
```

Fig. 10 A log-function definition that does not refer to any constant values.

The causes of these failures are that the algorithm shown in Section 3.3 produces identical word lists for multiple candidates (We did not observe cases in which the word lists were different but their coincidence degrees were identical). For example in **Fig. 8**, the Tacere app contains two different locations whose log-function call generates identical content.

There were other cases in which the word lists are identical for multiple candidates. They were locations that use no constant values and from which no words (or a word of the tag) could be extracted. **Figure 9** shows the case of the `AboutLicenseActivity` class in the Tacere app; the entire log-message is generated as a dynamic value. In these cases too, multiple log-function calls have identical word lists, and the inference fails.

There were also cases in which words failed to be extracted. The application defined the original method calling the log function, like the `logDebug` method included in the `IabHelper` class in the Tacere app (**Fig. 10**). For these cases, as explained in Section 3.2, we can accurately infer candidates by including the `logDebug` method as a log output function and making it a target of extraction [*7].

Like the above cases, when there are multiple log-function calls that have identical word lists, it is difficult to identify a candidate from the log outputs without modifying the log-function calls. We will attempt to remedy this situation in the future. If the source code can be modified, we can append identifiable information (e.g., a line number embedded in the class file for each log output) to each log-function call. We could also indicate that there are multiple candidates on the user interface.

4.1.5 Mutual Influence of Log-function Calls

We illustrate, for a log-function call and logs generated from the call, how the success rate of inferred candidates is affected by descriptions of other log-function calls.

We investigated how the coincidence degree explained in Section 3.4 changes when the number of log-function calls changes. We selected the Lightning Browser app as the target, gradually changed the number of expressions extracted as candidates of log-function calls, and measured the success rate (SR). We varied the

[*7] In the experiment described in Section 4.1, for comprehensive analysis, we did not specify any additional log output functions for each application.

Table 3 Success rates for different numbers of candidates

LPs	SR	Words
1	1.00	5
10	1.00	138
25	0.84	139
26	0.80	138
37	0.84	209
55	0.83	317

```
225  Log.d(Constants.TAG, "
       "resolved url is empty.  Url is:  " + url);
379  Log.e(Constants.TAG, responseCode +
       " url:" + urlAsString + " resolved:" + newUrl);
```

Fig. 11 Log-function calls sharing the same words.

number of candidate LPs (1, 26, 37, and 55; all candidates except libraries in the app).

Table 3 shows the results. The "LPs" column lists the number of candidates, the "Words" column lists the total number of words in the candidates, and the "SR" column lists the success rates of the candidates.

From the results, we can see that the success rates for multiple candidates are about 80% and change within 3–5%. Therefore, we can expect that the change in the number of candidates would not largely affect to the results of the inference.

Figure 11 shows several lines of code in the `HtmlFetcher.java`. The log-function call in line 379 was successfully inferred in the cases of 1 candidate and 26 candidates. However, it failed in the cases of 37 and 55 candidates. The failures happened when line 225 was added as a candidate; the reason for the failure is that the algorithm in Section 3.4 computes the same score for the word lists `"url,resolved"` in those candidates. This is the same problem as the limitation described in Section 4.1.4.

4.2 Fixing an Unintended Behavior

To confirm the effectiveness of LogChamber, we carried out a debugging task on a real application. As the target of the study, we chose a web browser application named Lightning Browser. The application is a typical Android app [*8] having a moderate size and open source code [*9].

We chose to deal with an unresolved issue (named Issue#127 [*10]) that was reported by a user of the application as an unintended behavior to the developer. This unintended behavior was expressed as "cannot search words with a leading "about:"" and the developer of the application labeled it a "bug". Accordingly, we supposed that it was an unresolved and unintended behavior of the source code.

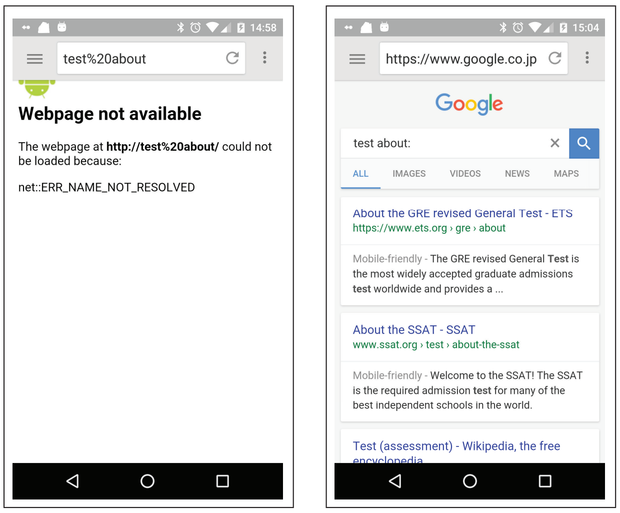
Lightning Browser, just like other general Web browsers, displays the address bar at the top of the screen, and when the user inputs a URL string, it goes to the associated page. Also, when the user inputs a search string into the address bar, it goes to the results page provided by the search site.

The target unintended behavior was that when a user inputs a string containing "`about:`", the application fails to search and

[*8] https://play.google.com/store/apps/details?id=acr.browser.barebones&hl=ja

[*9] https://github.com/anthonycr/Lightning-Browser

[*10] https://github.com/anthonycr/Lightning-Browser/issues/127



test%20about

Webpage not available

The webpage at **http://test%20about/** could not be loaded because:

net::ERR_NAME_NOT_RESOLVED

https://www.google.co.jp

Google

test about:

About the GRE revised General Test - ETS
https://www.ets.org › gre › about

Mobile-friendly - The GRE revised General **Test** is the most widely accepted graduate admissions **test** worldwide and provides a ...

About the SSAT - SSAT
www.ssat.org › test › about-the-ssat

Mobile-friendly - Welcome to the SSAT! The SSAT is the required admission **test** for many of the best independent schools in the world.

Test (assessment) - Wikipedia, the free encyclopedia

Fig. 12 Screens of the application showing an error (left) and regular behavior (right).

goes to a blank page (the left side of **Fig. 12**). The expected regular behavior is on the right side of Fig. 12, which displays the results page of the search when the user inputs "`test about:`". The task of the study was to correct the unintended behavior when the user inputs "`test about:`" in the address bar.

The authors used a version [*11] of the source code of the target application consisting of about 9000 lines in Java. The number of log-function calls except for libraries was 55.

We applied LogChamber and tried to identify the location of the unintended behavior in the source code without any preliminary knowledge about the target source code. After identifying the location, we analyzed how LogChamber worked in the task. We also reported the correcting code to the developer of the application and confirmed whether the correction was accepted or not.

4.2.1 Addition of Log-function Calls

First, the authors tried to identify the target unintended behavior from 55 log-function calls originally inserted in the application. The generated logs were 92 lines at the launching time of the application and 26 lines at the time of inputting the string "`test about:`". But the logs included lines generated from the system, and only two lines were generated by Lightning Browser (**Fig. 13**, the lines beginning with `I/Lightning`. Note that the figure omits timestamps and process ID information for readability.).

Second, the authors inputted "test about:" into the address bar just before the line 5 of the log output of 13. Consequently, the logs from line 5 in the figure were generated by the input. LogChamber inferred that lines 4–5 `onResume` and `onPause` were generated from `BrowserActivity.java` and further browsing of the source code successfully identified the log-function calls associated with the logs. The logs indicated the lifecycle state of the application and were not descriptions that could be used as hints of the unintended behavior; thus, the authors concluded that it was difficult to identify locations of the behavior from the logs.

The log-function calls did not include related variables or calls for the unintended behavior of this study; thus, they lacked key

[*11] commit ID: `a2f2fbc`

```
1558 | Log.d(Constants.TAG, "searchTheWeb query =" +query);
     | ...
1567 | boolean isIPAddress =(TextUtils.isDigitsOnly(query.replace(".", ""))
     | && (query.replace(".", "").length() >= 4) &&   query.contains("."));
1569 |  boolean aboutScheme = query.contains("about:");
1570 |  boolean validURL =(query.startsWith("ftp://") ||  query.startsWith(Constants.HTTP)
     | ...
1573 |  boolean isSearch =  ((query.contains(" ") || !containsPeriod) && !aboutScheme);
     | ...
1579 | Log.d(Constants.TAG,"isSearch = " + isSearch);
1580 |  if (isSearch)
```

Fig. 15 Source-code lines relating to the unintended behavior.

```
 1 | W/Resources Converting to string:  TypedValue
   |    {t=0x5/d=0x601 a=2 r=0x7f090011}
 2 | W/Resources Converting to string:  TypedValue
   |    {t=0x5/d=0xa01 a=2 r=0x7f09000e}
 3 | W/cr.BindingManager
   |    Cannot call determinedVisibility() -
   |    never saw a connection for the pid:  5488
 4 | D/Lightning onPause
 5 | I/Lightning onResume
 6 | E/ProxyChangeListener Using no proxy
   |    configuration due to
   |    exception:java.lang.NullPointerException:
   |    Attempt to invoke virtual method'java.lang.Object
   |    android.os.Bundle.get(java.lang.String)'
   |    on a null object reference
 7 | W/art?  Suspending all threads took:  5.370ms
 8 | W/art?  Suspending all threads took:  67.753ms
 9 | W/BindingManager Cannot call determinedVisibility()
   |    - never saw a connection for the pid:  6405
10 | I/Lightning onPause
```

Fig. 13 Example runtime logs of Lightning Browser before inserting the supplementary log-function calls.

Table 4 Code for supplementary log-function calls.

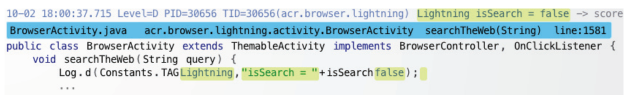
File name (Target method) Line number	Inserted code
`BrowserActivity.java` (`searchTheWeb`) Line 1558	`Log.d(Constants.TAG, "searchTheWeb query =" +query);`
`BrowserActivity.java` (`searchTheWeb`) Line 1579	`Log.d(Constants.TAG, "isSearch = "+ isSearch);`
`BrowserActivity.java` (`searchTheWeb`) Line 981	`Log.d(Constants.TAG, "findInPage query =" +query);`
`SearchAdapter.java` (`doInBackground`) Line 296	`Log.d(Constants.TAG, "query ="+ query);`
`SearchAdapter.java` (`doInBackground`) Line 358	`Log.d(Constants.TAG, "downloadSuggestionsForQuery query=" +query);`

information for identifying the target unintended behavior.

Third, the authors inserted the five log-function calls shown in **Table 4** and tried to identify the unintended behavior by repeatedly inputting the same string and applying LogChamber. The authors hierarchically browsed files in the source code and supposed that the unintended behavior was related to URL querying and searching. After quickly browsing the files, the locations related to URL searching and querying were found in `BrowserActivity.java` and `SearchAdapter.java`. The authors decided that the values of `query` and `isSearch` (listed in Table 4) that were used in the files would be generated as supplementary log outputs. **Figure 14** shows a portion of the generated logs.

```
1 | I/Lightning onResume
2 | E/ProxyChangeListener Using no proxy
  |    configuration due to
  |    exception:java.lang.NullPointerException:
  |    Attempt to invoke virtual method'java.lang.Object
  |    android.os.Bundle.get(java.lang.String)'
  |    on a null object reference
3 | D/Lightning downloadSuggestionsForQuery query
  |     = test about:
4 | D/Lightning searchTheWeb query = test about:
5 | D/Lightning isSearch = false
6 | W/BindingManager Cannot call determinedVisibility()
  |    - never saw a connection for the pid:  6405
7 | I/Lightning onPause
```

Fig. 14 Example of generated logs of Lightning Browser after inserting supplementary log-function calls.



```
10-02 18:00:37.715 Level=D PID=30656 TID=30656(acr.browser.lightning) Lightning isSearch = false -> score
BrowserActivity.java   acr.browser.lightning.activity.BrowserActivity   searchTheWeb(String)   line:1581
public class BrowserActivity extends ThemableActivity implements BrowserController, OnClickListener {
    void searchTheWeb(String query) {
        Log.d(Constants.TAGLightning,"isSearch = "+isSearchfalse);
        ...
```

Fig. 16 Display of the inference result for the variable `isSearch`.

4.2.2 Steps for Identifying Unintended Behavior

Line 5 in Fig. 14 was generated from a supplementary log-function call; it is

`isSearch = false`.

This indicated the input string was handled as not a search query but as a URL. The application tried to access the URL `http://test%20about`. Consequently, the authors concluded that the line was abnormal and should be `isSearch = true`.

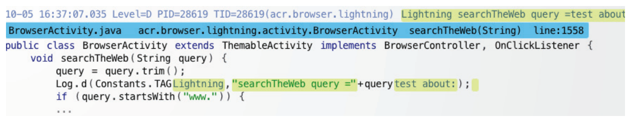
LogChamber inferred the location generating the log line was line 1579 in `BrowserActivity.java` and displayed the result as in **Fig. 16**. **Figure 15** shows the location in the source code and the lines around it relating to the unintended behavior. Line 1579 supplied a string generated from `isSearch` to the arguments of the `Log.d` method, which is one of the log functions. The variable `isSearch` had an assignment statement at line 1573, and it was determined from three variables, i.e., `query`, `containsPeriod` and `aboutScheme`.

From the names of these variables, the authors estimated that `query` was the input string, `containsPeriod` indicates whether the input has a period (.), and `aboutScheme` indicates whether the input starts with "`about:`".

LogChamber inferred that the log of line 4 in Fig. 14,

`searchTheWeb query = test about:`

corresponded to the log-function call of the supplementary inserted line 1558 (**Fig. 17**). Form this result, it could be seen that `query` was obviously the input string, and the value of `query` itself was correct.



```
10-05 16:37:07.035 Level=D PID=28619 TID=28619(acr.browser.lightning) Lightning searchTheWeb query =test about
BrowserActivity.java   acr.browser.lightning.activity.BrowserActivity  searchTheWeb(String)  line:1558
public class BrowserActivity extends ThemableActivity implements BrowserController, OnClickListener {
    void searchTheWeb(String query) {
        query = query.trim();
        Log.d(Constants.TAGLightning,"searchTheWeb query ="+querytest about:);
        if (query.startsWith("www.")) {
        ...
```

Fig. 17 Display of the inference result for the variable `query`.

The next candidate relating to the unintended behavior was estimated to be `abountScheme` from its name, and its value was determined at line 1569.

4.2.3 Correction of the Unintended Behavior

The authors deleted the uses of `aboutScheme` from line 1573 and changed the line to

```
boolean isSearch = (query.contains(" ") ||
!containsPeriod);
```

After that, the authors ran the application again and it worked, showing the expected normal behavior as on the right side of Fig. 12. `aboutScheme` was identified as the abnormal value.

The value of `aboutScheme` was generated from line 1569 in Fig. 15,

```
query.contains("about:")
```

, so the authors changed the expression to

```
query.startsWith("about:")
```

and created and submitted a patch to the developer of the application. The patch was accepted by the developer as a bug fix [*12].

4.2.4 Effectiveness of LogChamber

In this study, we presented a case in which LogChamber aided the correction of an unintended behavior.

At the beginning of the study, the information obtained from the generated logs was not sufficient. From the logs, LogChamber identified the locations of the existing log-function calls; then it helped the authors to understand that those locations were not related to the unintended behavior (Section 4.2.1).

In addition, from the generated logs with the supplementary inserted log-function calls, LogChamber successfully inferred the locations generating the logs. The authors checked the values of the variables presented in the results display of the locations and identified the unintended behavior (Section 4.2.2).

Without LogChamber, the entire task would have been the responsibility of the developers (in this case, the authors in the study) and rather time consuming.

5. Related Work

SherLog proposed by Yuan et al. [14] is a tool that infers runtime behaviors from generated runtime logs by using static program analysis, and its idea is similar to that of LogChamber. SherLog analyzes the runtime logs generated by a C program and infers both the control flow of function calls (path inference) and the data flow of variables (value inference) at runtime.

On the other hand, LogChamber does not employ such inferences. This is because 1) Android apps are event-driven GUI applications, and there are many isolated executions by event handler methods; thus, inferences on control or data flows are insufficient, and 2) the whole program analysis consumes too much time, thus the quick displaying on GUI is infeasible. SherLog focuses not only on a log output and its corresponding log-function calls, but also on more global inferences including flows of function calls. As a result, it cannot work with event handlers of callbacks repeatedly executed within an event loop as in the case of mobile apps. LogChamber makes inferences from each line of generated logs and visually displays the result on the IDE screen as soon as the log line is generated.

Similarly, the log-analysis techniques by Mariani et al. [9], and by Wei et al. [12], [13], LogEnahcnement [15], AutoLog [16] and lprof [17] do not focus on fast (immediate) log inferences for mobile apps.

Symbolic execution [6] is a technique to avoid runtime workloads, and it can be used to identify unintended behaviors and make verifications [2], [4]. There are studies that apply symbolic execution to mobile application debugging [5], [10]. However, mobile apps may have different behaviors depending on the device on which they run, and it would be hard to reproduce those differences among devices through static execution; hence, these approaches are not suitable for examining the unintended behaviors on specific devices that our work focuses on.

As for work on analyzing mobile apps, FlowDroid [1], Scandroid [3], and the technique of Klieber et al. [7] are static analysis methods for more accurate control flow, and they can be used to find security vulnerabilities. FlowDroid is an extension of Soot [11], an analysis framework for Java applications, that is specialized for Android applications, and it can determine the difference in effect that two methods have on a specified variable. In order to implement LogChamber, we used the function for loading APK files provided by FlowDroid; we think that this framework can also be used for searching the call graph.

6. Conclusion

We proposed LogChamber, a tool for analyzing runtime logs and identifying locations of outputs, in order to address the problems of debugging mobile apps. Despite that ad-hoc and poorly reproducible log-function calls are often found in mobile apps, LogChamber can automatically infer the corresponding log-function calls for the generated logs without requiring manual management of the calls. We implemented LogChamber as a plug-in on a real IDE, and it can connect to running applications for inferring log outputs on site. The tool can arrange strings in log lines corresponding to the arguments of the inferred log-function calls on a display of the source code. Thanks to these features, LogChamber can shorten the time needed to debug mobile apps.

We applied LogChamber to real mobile apps and confirmed that it is feasible for large-scale programs and is fast enough. We also tried to identify and correct actual unresolved unintended behaviors of the real application and illustrated a case in which LogChamber worked effectively.

Although the techniques presented in this paper can practically infer the locations of log-function calls in many cases, there are other cases in which they infer incorrect locations. These cases can be avoided by the developer who carefully selects words used in the log-function calls, but in the future, we may develop features for scoring the uniqueness of such log-function calls or suggesting word candidates.

[*12] commit ID: `33eb739`

Also, our current technique does not analyze control flows between multiple log lines and so on. As a future extension, we would like to develop techniques to efficiently identify global data flows and control flows and to reproduce program execution histories.

Acknowledgments We gratefully acknowledge the work of past and present members of our laboratory. This work was supported by a JSPS KAKENHI Grant, Number 26730034.

References

[1] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. and McDaniel, P.: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, *SIGPLAN Not.*, Vol.49, No.6, pp.259–269 (2014).

[2] 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, *Proc. 9th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'10*, pp.1–6, USENIX Association (2010).

[3] Fuchs, A.P., Chaudhuri, A. and Foster, J.S.: Scandroid: Automated security certification of android applications, Technical Report, University of Maryland (2009).

[4] Jensen, C.S., Prasad, M.R. and Møller, A.: Automated Testing with Targeted Event Sequence Generation, *Proc. 2013 International Symposium on Software Testing and Analysis*, *ISSTA 2013*, pp.67–77, ACM (2013).

[5] Jeon, J., Micinski, K.K. and Foster, J.S.: SymDroid: Symbolic execution for Dalvik bytecode, Technical report, Department of Computer Science, University of Maryland, College Park (2012).

[6] King, J.C.: Symbolic Execution and Program Testing, *Commun. ACM*, Vol.19, No.7, pp.385–394 (1976).

[7] Klieber, W., Flynn, L., Bhosale, A., Jia, L. and Bauer, L.: Android Taint Flow Analysis for App Sets, *Proc. 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, *SOAP '14*, pp.1–6, Edinburgh, United Kingdom (June 2014).

[8] Lewis, B.: Debugging Backwards in Time, *Proc. 5th International Workshop on Automated Debugging*, *AADEBUG'03* (2003).

[9] Mariani, L. and Pastore, F.: Automated identification of failure causes in system logs, *Software Reliability Engineering, 2008, 19th International Symposium on ISSRE 2008*, pp.117–126, IEEE (2008).

[10] Mirzaei, N., Malek, S., Păsăreanu, C.S., Esfahani, N. and Mahmood, R.: Testing Android Apps Through Symbolic Execution, *SIGSOFT Softw. Eng. Notes*, Vol.37, No.6, pp.1–5 (2012).

[11] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot—a Java Bytecode Optimization Framework, *Proc. 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, *CASCON '99*, p.13 (ref 1–13), IBM Press (1999).

[12] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.: Experience Mining Google's Production Console Logs, *Proc. 2010 Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, *SLAML'10*, p.5, USENIX Association (2010).

[13] Xu, W., Huang, L., Fox, A., Patterson, D. and Jordan, M.I.: Detecting Large-scale System Problems by Mining Console Logs, *Proc. ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, *SOSP '09*, pp.117–132, ACM (2009).

[14] Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y. and Pasupathy, S.: SherLog: Error Diagnosis by Connecting Clues from Run-time Logs, *SIGARCH Comput. Archit. News*, Vol.38, No.1, pp.143–154 (2010).

[15] Yuan, D., Zheng, J., Park, S., Zhou, Y. and Savage, S.: Improving Software Diagnosability via Log Enhancement, *SIGARCH Comput. Archit. News*, Vol.39, No.1, pp.3–14 (2011).

[16] Zhang, C., Guo, Z., Wu, M., Lu, L., Fan, Y., Zhao, J. and Zhang, Z.: AutoLog: Facing Log Redundancy and Insufficiency, *Proc. 2nd Asia-Pacific Workshop on Systems*, *APSys '11*, pp.10:1–10:5, ACM (2011).

[17] Zhao, X., Zhang, Y., Lion, D., Ullah, M.F., Luo, Y., Yuan, D. and Stumm, M.: Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems, *Proc. 11th USENIX Conference on Operating Systems Design and Implementation*, *OSDI'14*, pp.629–644, USENIX Association (2014).

Yuki Ono has a master's degree from the Graduate School of Natural Science Technology, University of Kanazawa in 2013. He expects to receive a doctoral degree from the same institution in 2016.

Kohei Sakurai received his Ph.D. degree from the University of Tokyo, Japan in 2009. From 2009 to 2011, he was a postdoctoral researcher at the Shibaura Institute of Technology. Since 2011, he has been an assistant professor in the faculty of electrical and computer engineering at the Institute of Science and Engineering of Kanazawa University, Japan. His research interests include programming, software testing, and debugging.

Satoshi Yamane received his B.S. (1982) and M.S. (1984) degrees from Kyoto University, Japan. He is a professor in Kanazawa University. He is interested in formal verification and distributed computing.