DIGITAL FORENSIC RESEARCH CONFERENCE

LogExtractor: Extracting Digital Evidence from Android Log Messages via String and Taint Analysis

By:
Chris Chao-Chun Cheng (Iowa State University), Chen Shi (Iowa State University),
Neil Zhenqiang Gong (Duke University), and Yong Guan (Iowa State University)

From the proceedings of

The Digital Forensic Research Conference

DFRWS USA 2021

July 12-15, 2021

Forensic Science International: Digital Investigation 37 (2021) 301193

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2021 USA - Proceedings of the Twenty First Annual DFRWS USA

LogExtractor: Extracting digital evidence from android log messages via string and taint analysis

Chris Chao-Chun Cheng [a, b, 1], Chen Shi [a, b, 1], Neil Zhenqiang Gong [c], Yong Guan [a, b, *]

[a] NIST Center of Excellence in Forensic Sciences - CSAFE, USA
[b] Department of Electrical and Computer Engineering, Iowa State University, USA
[c] Department of Electrical and Computer Engineering, Duke University, USA

ARTICLE INFO

Article history:

Keywords:
Android log messages
Mobile app forensics
String analysis

ABSTRACT

Mobile devices are increasingly involved in crimes. Therefore, digital evidence on mobile devices plays a more and more important role in crime investigations. Existing studies have designed tools to identify and/or extract digital evidence in the main memory or the file system of a mobile device. However, identifying and extracting digital evidence from the logging system of a mobile device is largely unexplored.

In this work, we aim to bridge this gap.Specifically, we design, prototype, and evaluate *LogExtractor*, the first tool to automatically identify and extract digital evidence from log messages on an Android device. Given a log message, LogExtractor first determines whether the log message contains a given type of evidentiary data (e.g., GPS coordinates) and then further extracts the value of the evidentiary data if the log message contains it.

Specifically, LogExtractor takes an *offline-online* approach. In the offline phase, LogExtractor builds an *App Log Evidence Database (ALED)* for a large number of apps via combining string and taint analysis to analyze the apps' code. Specifically, each record in the ALED contains 1) the string pattern of a log message that an app may write to the logging system, 2) the types of evidentiary data that the log message includes, and 3) the segment(s) of the string pattern that contains the value of a certain type of evidentiary data, where we represent a string pattern using a *deterministic finite-state automaton*. In the online phase, given a log message from a suspect's Android device, we match the log message against the string patterns in the ALED and extract evidentiary data from it if the matching succeeds. We evaluate LogExtractor on 65 benchmark apps from DroidBench and 12.1 K real-world apps. Our results show that a large number of apps write a diverse set of data to the logging system and LogExtractor can accurately extract them.

1. Introduction

Mobile devices are increasingly involved in physical and cybercrimes. As a result, digital evidence on mobile devices plays an important role in investigating crimes. For instance, the GPS coordinates and timestamps extracted from a suspect's mobile device can aid a forensic investigator to determine whether the suspect was at the crime scene or not when the crime happened. Indeed, digital evidence on mobile devices has been used to aid crime investigation in real-world cases (Raphael, 2016; Leone, 2017). Roughly speaking, digital evidence could be stored in the *main memory* and *permanent storage* of a mobile device. Several studies (Bhatia et al., 2018; Saltaformaggio et al., 2015a, 2015b, 2016) have explored extracting digital evidence from the main memory that is allocated to a particular app, e.g., reconstructing the app's context. Moreover, Cheng et al. (2018) proposed EviHunter to identify which files on the permanent storage of an Android device contain a given type of evidentiary data. However, identifying and extracting digital evidence in the *logging system* of a mobile device is largely unexplored.

In this work, we aim to bridge this gap. Android logging system aims to facilitate app development by enabling developers to easily

* Corresponding author. NIST Center of Excellence in Forensic Sciences - CSAFE, Iowa State University, Ames, Iowa, 50011, USA.
E-mail addresses: cccheng@iastate.edu (C.C.-C. Cheng), cshi@iastate.edu (C. Shi), neil.gong@duke.edu (N.Z. Gong), guan@iastate.edu (Y. Guan).
[1] Student.

https://doi.org/10.1016/j.fsidi.2021.301193

log and analyze the app runtime information. The Android logging system consists of multiple buffers in the main memory. Compared to existing studies on memory forensics (Bhatia et al., 2018; Saltaformaggio et al., 2015a, 2015b, 2016), which aim to recover an app's internal data structure, identifying and extracting digital evidence from the logging system faces new opportunities and challenges, e.g., all log messages are strings. Given a log message from the logging system on a suspect's Android device, we aim to answer two questions: does the log message include a given type of evidentiary data (e.g., GPS coordinates)? What is the value of the evidentiary data?

A naive approach to solve the two problems is that a forensic investigator manually inspects the log message and extracts evidentiary data using some clues such as keywords. For instance, to extract GPS coordinates, the forensic investigator could search for keywords such as GPS, latitude, and longitude. However, such a manual analysis or keyword search approach faces two key challenges. First, they are not applicable if the log message does not have clear clues. Second, they are not reliable when a log message is *formatted confusingly*. For instance, suppose the log message is "-30.45678,20.1234". It is challenging to determine if this log message corresponds to GPS coordinates. Even if this log message does correspond to GPS coordinates, a forensic investigator cannot reliably determine whether the format is "longitude, latitude" or "latitude, longitude". In our experiments, we will show case studies where both formats are observed in real-world apps.

Our work: In this work, we aim to address these challenges. In particular, we develop *LogExtractor*, the first tool to automatically identify and extract evidentiary data in a logging system on an Android device. Our key observation is that the log messages are written by apps and thus apps' code includes rich information about what data they may write to the logging system and in what format. Based on this observation, LogExtractor takes an offline-online approach. In the offline phase, LogExtractor builds an *App Log Evidence Database (ALED)* for a large number of apps, which includes the types of evidentiary data that an app may write to the logging system and the string format of the evidentiary data. In the online phase, LogExtractor matches each log message in the logging system on a suspect's Android device against the ALED to identify and extract evidentiary data.

Building the ALED is a key component of LogExtractor. In our ALED, each record contains three important fields: 1) the string pattern of a log message that an app may write to the logging system, 2) the types of evidentiary data that the log message includes, and 3) the segment(s) of the string pattern that contains the value of a certain evidentiary data. We represent the string pattern using *deterministic finite automaton (DFA)*. Existing static string analysis and taint analysis tools are insufficient to build our ALED. Specifically, existing string analysis tool (e.g., Java String Analyzer (Christensen et al., 2003)) can only produce the first field, i.e., a DFA to represent a log message; and existing static taint analysis tools (e.g. (Lu et al., 2012; Yang and Yang, 2012; Arzt et al., 2014; Wei et al., 2014; Li et al., 2015; Gordon et al., 2015; Backes et al., 2016; Calzavara et al., 2016),) cannot produce any of the three fields. Specifically, existing static taint analysis tools essentially track data flow from a source to a sink (e.g., logging system), and thus they can only determine whether an app writes data to the logging system without being able to provide fine-grained information on each log message.

To address the challenges, LogExtractor combines string and taint analysis to build the ALED. Specifically, we design LogExtractor by extending the Java String Analyzer (JSA) (Christensen et al., 2003). Given a java program and a *hotspot* (i.e., an argument of a sink method), JSA outputs the string pattern for the hotspot in the java program, where the string pattern is represented as a DFA. In particular, JSA associates a DFA with a variable and propagates/merges the DFA via analyzing the statements in the java program. We extend JSA in multiple aspects. First, since JSA was designed for the general java program, we extend it to support unique Android features such as the app's lifecycle and inter-component communications. In particular, we leverage FlowDroid (Arzt et al., 2014) to model an Android app's lifecycle and inter-component communications. Second, we propose *tainted DFA* and associate a tainted DFA with each variable. Specifically, a tainted DFA includes a DFA and a *taint table*, where the DFA describes a variable's string pattern, while the taint table contains the states in the DFA that include evidentiary data and the types of evidentiary data they include. The taint table is initialized at a source method that returns a certain type of evidentiary data of interest to forensic investigation, e.g., the Android API that returns latitude. Third, we extend JSA's propagation rules to propagate the tainted DFA in an Android app's code. For instance, when we concatenate two string variables, we merge their DFAs and taint tables.

In the online phase, given a log message from an app, we match it against each DFA in our ALED that belongs to the app. If the log message does not match against any DFA, then LogExtractor predicts that the log message does not include evidentiary data. Otherwise, LogExtractor further extracts the value of the evidentiary data from the log message using a new algorithm that we design. Roughly speaking, the segments of the log message that match the states in the taint table correspond to the value of the evidentiary data.

We first evaluate our LogExtractor on 65 benchmark apps from DroidBench (2019), which write data to the logging system. We use LogExtractor to build an ALED for these apps. We install each benchmark app and use it on an Android device. Then, we retrieve the log messages generated by these apps and match each of them against the ALED to identify and extract data. The benchmark evaluation results show that LogExtractor achieves 97.7% precision and 79.2% recall at identifying whether a log message contains evidentiary data. Moreover, LogExtractor correctly extracts the data value in all of the log messages that LogExtractor correctly identifies as containing evidentiary data. We also evaluate LogExtractor on 12.1 K real-world apps. We found that a large number of apps write a diverse set of data to the logging system. We also performed an end-to-end manual verification of 91 real-world apps. Specifically, we install these apps on an Android device and use them. These apps generated 90 K log messages, 266 of them contain evidentiary data (we consider device ID, latitude, longitude, and text input to be evidentiary data). LogExtractor correctly identifies that 230 log messages contain evidentiary data, indicating a 86.5% precision and 91.3% recall at identifying whether a log message contains evidentiary data or not. Moreover, LogExtractor correctly extracts the data values from all of the 230 log messages. Finally, we adapt case studies to show how LogExtractor can help to mitigate the challenges brought by the confusingly-formated log messages.

Our contributions are summarized as follows:

- We design and implement LogExtractor, the first tool to automatically identify and extract digital evidence from Android log messages.
- We extend JSA to support Android apps as well as track string patterns and taint tags simultaneously.
- We evaluate LogExtractor using both benchmark apps and real-world apps.

2. Problem formulation

2.1. Android logging system

Android logging system aids app development and tracks information such as system events and error reports from running apps. The logs are stored in four memory buffers, “main”, “radio”, “events”, and, “system”. Apps can only write log messages to the “main” buffer, and the remaining buffers are reserved for system logs. Android defines 14 APIs in the class `android.util.Log` for apps to write log messages with different log level. When a buffer is full, the oldest messages will be overwritten. Given an Android device, we can use the Android Debug Bridge (ADB) interface (Android Debug Bridge, 2019) to retrieve the log messages without altering the data.

Message structure: Fig. 1 shows two example log messages and their structure. Each log message consists of the following fields: timestamp, process identifier (PID), thread identifier (TID), package name, log level, tag and body. While the first three fields are assigned by the system and the package name only appears in the debug mode, log level, tag, and body are defined in the app's program.

2.2. Threat model

Consider the following scenario where a user's private information (e.g. GPS coordinates or the timestamp of sending out an SMS message (Alonso Parrizas, 2015)) was archived in the logging system by the app `A` and then leaked by the app B, which can access other apps' log entries without the root permission (2020, 2020). A security/forensic analyst aims to determine the evidence of app A's behavior of archiving the private data on the logging system. When the analyst investigates log entries extracted from the victim's device, he/she encounters two basic questions. First, is there any evidentiary data (e.g., GPS coordinates) in the log messages? Second, what is the value of the evidentiary data, e.g., what are the latitude and longitude if GPS coordinates are in the log messages?

Listing 1: Example of writing location to logging system.

```
1 void foo(Location loc){
2   String prefix = "Lat-";
3   StringBuffer buff = new StringBuffer(prefix);
4   double lat = loc.getLatitude();
5   buff.append(Double.toString(lat));
6   String toStr = buff.toString();
7   Log.d("Msg1", toStr);
8   Log.d("Msg2", "Lat-29.302");
9 }
```

One natural solution to the two problems is that the forensic analyst can manually inspect the log message to identify and extract evidentiary data. However, the log message could mislead the parsing result, making it challenging for manual analysis to reliably identify and extract digital evidence. For instance, suppose the two log messages shown in Fig. 1 are written by the function in Listing 1. The keyword “Lat” may or may not indicate latitude data. In particular, the first log message includes latitude data, while the second log message does not. It is challenging for a human to distinguish the two cases by simply inspecting the log messages. Moreover, even if we know the first log message contains latitude data, it is still challenging to reliably extract the latitude value. In particular, is the latitude value −30.03 or 30.03? Another intuitive case could be a log message “Send SMS message: 1234567890”. The string “1234567890” could denote the UNIX timestamp of sending out the message. How could forensic analysts know that the string should be parsed as a timestamp instead of the content of the associated message?

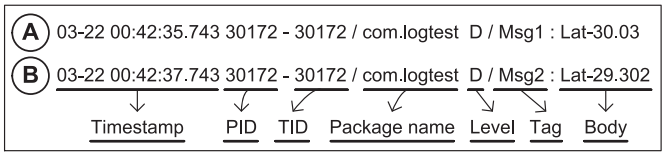


Fig. 1. Two Android log messages and their structure.

The existing string analysis tool (e.g., JSA (Christensen et al., 2003)) can produce the pattern of a log message via analyzing the app's code, where the pattern is represented as an automaton. However, they cannot answer whether a log message contains evidentiary data that is of interest to a forensic investigator or what the value of the evidentiary data is. Existing taint analysis tools (e.g., FlowDroid (Arzt et al., 2014)) can determine whether an app writes a certain type of evidentiary data to the logging system. However, they cannot determine which specific log message contains the evidentiary data and they cannot extract the value of the evidentiary data from a log message.

2.3. Problem definition

We formalize the two aforementioned research challenges into the following problems, which we aim to solve in this work:

Definition 1. (Evidence Identification Problem) Given a log message, the evidence identification problem is to determine whether the log message contains a certain type of evidentiary data, e.g., GPS coordinates, URL, or text input.

Definition 2. (Evidence Extraction Problem) Given a log message that contains a certain type of evidentiary data, the evidence extraction problem is to extract the value of the evidentiary data from the log message.

We develop LogExtractor to solve the two problems. LogExtractor combines string and taint analysis. For the two log messages in Fig. 1 that are written by the function in Listing 1, our LogExtractor can determine that the first log message contains latitude and the value of the latitude is 30.03.

3. Preliminaries

Our LogExtractor relies on finite-state automaton (FSA), Java String Analyzer (JSA), and taint analysis. Therefore, we briefly review these concepts before introducing the technical details of LogExtractor in the next section.

3.1. Finite-state automaton (FSA)

Using FSA to represent a string pattern: Finite-State Automaton (FSA) is a graph-based model composed of three parts: state, transition, and input. Equivalent to the regular expression, FSA can be used to denote a string pattern via regulating the sequence of acceptable characters. For example, Fig. 2 is the deterministic finite-state automaton (DFA) of variable `toStr` at line 7 in Listing 1. It denotes a string pattern starting with a constant string “Lat-” followed by a floating number such as “30.03”.

Matching a given string against a DFA: Given a DFA as the one in Fig. 2 and a log message “Lat-30.03”, we start from the initial state of DFA, S1, and the first character of the log message, ‘L’. Starting from state S1, we take the input symbol ‘L’ in the string that will transit from S1 to S2. Similar procedures are repeated until

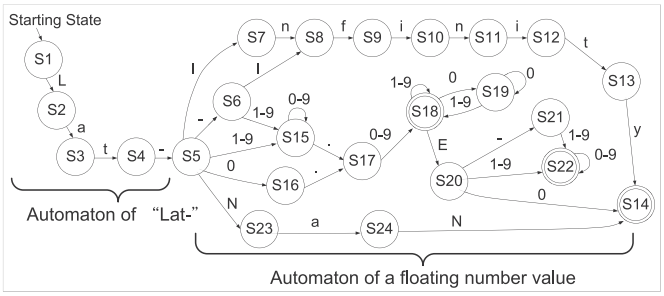


Fig. 2. DFA of `toStr` variable at line 7 in Listing 1.[21]

reaching state S18, which is an acceptor. At the time when reaching an acceptor, we can conclude string "Lat-30.03" is matched by this DFA.

3.2. Java String Analyzer

Java String Analyzer (JSA) (Christensen et al., 2003) is an automata-based string analysis tool developed for approximating string expressions in Java program. The implementation of JSA consists of multiple phases, each of them takes input from the previous one and transforms it into a new form. The tool takes Java program (`.class` file) and the target variables (called hotspots), specified by their argument indices and sink methods, as inputs, and utilize Soot (Vallée-Rai et al., 1999) to transform the input binary program to Jimple code (a three-address intermediate representation). Later, for each variable matching the input and found in the program, JSA outputs a DFA summarizing its runtime string pattern. The technical details of JSA can be found in its technical report (Feldthaus and Møller, 2009).

Limitations: JSA can approximate string expression in Java program through defining and computing the automatons. However, JSA has several limitations when being used in our research problems formulated in this paper as follows:

1. JSA does not consider the unique features of Android apps such as lifecycle and inter-component communications (ICC). This is because JSA was designed for general java program.
2. JSA does not track the flow of evidentiary data. As a result, JSA cannot determine whether a log message contains a given type of evidentiary data nor the value of the evidentiary data.
3. JSA does not have complete propagation rules. Specifically, JSA does not consider statements of binary operations such as subtraction and division when propagating DFA. Moreover, JSA does not precisely model string patterns of primitive variables, e.g., JSA uses a DFA with a single character to model a primitive variable.

Our LogExtractor aims at addressing these limitations. In particular, we leverage FlowDroid (Arzt et al., 2014) to model an app's lifecycle and inter-component communications; we design a new data structure called *tainted DFA* to track the data flow and string pattern simultaneously; and we extend the propagation rules to be more complete and propagate the tainted DFAs.

4. Our LogExtractor

4.1. Overview

Fig. 3 overviews LogExtractor. LogExtractor takes an *offline-online* approach. In the offline phase, LogExtractor builds an App Log Evidence Database (ALED) for a large number of apps via combining string and taint analysis to analyze each app's code, extracted from the app's application package (APK) file. Specifically, in the ALED, each record includes 4 components, i.e., Package Name, Log Level, Tainted DFA for message tag, and Tainted DFA for the message body. The package name and log level are used to reduce the number of tainted DFAs that we need to match against a given log message in the online phase. The tainted DFA is a new data structure that we design to propagate both the string patterns and evidentiary data in an app's code. Specifically, a tainted DFA includes a usual DFA, which models a variable's string pattern, and a taint table, which includes the states in the DFA that include evidentiary data and the types of evidentiary data. Note that a taint table could be empty.

In the online phase, given a log message from a suspect's Android device, LogExtractor matches the log message against the tainted DFAs in the ALED using a Matcher. The Matcher essentially matches a given string to a DFA as we described in Section 3.1. If the log message matches against a DFA whose taint table includes evidentiary data, then LogExtractor predicts that the log message includes the evidentiary data. Then, LogExtractor further uses the Extractor to extract the value of the evidentiary data from the log message. Specifically, we design a new algorithm to do so. Roughly speaking, our algorithm extracts the segment of the log message that corresponds to the states in the taint table as the value of the evidentiary data.

Next, we introduce how we build the ALED and design the Matcher and Extractor.

4.2. Building ALED

To solve our research problems in Section 2.3, ALED provides sufficient information to identify the evidentiary log messages and the positions of string values carrying digital evidence. The *Evidence Identification Problem*, whose goal is to determine which log message contains evidence, requires ALED to provide a complete description of the patterns of evidentiary log messages. For each pattern of evidentiary log message in ALED, we describe it using four parts: package name, log level, message tag, and message body. As introduced in Section 2.1, because the other parts (PID, TID, timestamp) of log messages are either assigned at the system level or depend on runtime environment, it is hard to infer their patterns.

Comparing to the message tag and body, package name and log level are naive string values, which can be represented in the format of constant string. In contrast, the string patterns of message tag and body are complicated because the digital evidence (e.g., GPS coordinates) is usually parsed as variables. As aforementioned, DFA can only approximate the string expression of a log message but is insufficient to solve either the *Evidence Identification Problem* or the *Evidence Extraction Problem*. We, therefore, propose *tainted DFA* to represent the patterns of log message tag as well as the body. Moreover, we propose new propagation rules to propagate tainted DFAs in an app's code.

4.2.1. Tainted DFA

A tainted DFA consists of a DFA and a *taint table*. A taint table maps a state identifier to a set of evidence types corresponding to the state. By DFA and taint table, tainted DFA takes advantage of both string and taint analysis that not only represents the patterns of a log message but also the positions of evidentiary data inside a DFA. When matching a log message against a tainted DFA, for each movement between two states, a character of the log message that satisfies the input of transition is consumed. If a character is consumed by the movement between two states having the same evidence types, then the character is determined as a part of digital

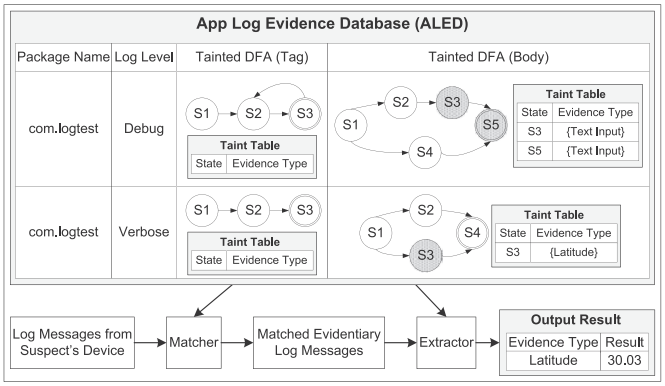


Fig. 3. Overview of LogExtractor.

evidence whose types are identified by the two states. For instance, the tainted DFA in Fig. 4 represents the pattern of the log message body `toStr` in Listing 1. Suppose we are given a log message "Lat-30.03". When matching the character '-', we move from an untainted state S4 to a tainted state S5, so we do not treat it as a part of any evidentiary data. On the contrary, when matching the next character '3', we treat it as a part of GPS latitude since both S5 and S6 are tainted by the same evidence type, i.e., latitude. Following the extraction rule state by state, we can obtain the string values of digital evidence from the given log message.

Recall that each evidentiary log message pattern in ALED has four data fields: package name, log level, message tag, and message body. While the first two fields are saved in string constant format, we represent the message tag and message body using tainted DFA. A log message is identified evidentiary only if it can be matched to these four data fields of some record in ALED.

We propose to construct a tainted DFA by implementing taint analysis in JSA. Specifically, we label an evidence type as a taint and associate taints with a DFA. Moreover, we extend JSA to propagate DFA and taint table at the same time. Since the compilation and propagation of DFA are performed by JSA, our task is to implement the propagation rules of taint tables, whose details we elaborate in Section 4.2.2. Moreover, because JSA has multiple limitations when applied to analyze real-world Android apps. We address these limitations. For example, to model the data-flow propagation in the Android app framework, such as callback methods, lifecycle, and inter-component communication (ICC), we employ FlowDroid (Arzt et al., 2014) and IccTA (Li et al., 2015) to construct a more precise call graph. Such extensions reduce the false negatives that are caused by the missing data flow of evidence type.

4.2.2. Propagation rules of tainted DFA

We implement the following rules in the analysis phase of JSA to propagate and compute runtime evidence types. We extend the determinization and minimization algorithms of DFA to process the computation of each variable's taint table and associated string patterns.

Rules of sources and sinks: Given an app, we are interested in the evidentiary log messages that will be written by this app during runtime. In addition to the data-flow propagation implemented by JSA and our extensions, the completeness of analysis result depends on the coverage of source and sink methods, where a source method defines the evidence types that should be assigned to a tainted DFA and a sink method writes messages to the logging system.

In our proposed approach, since we only focus on the evidence archived in Android logging system, we configure the sink methods to be 14 Android logging APIs introduced in Section 2 and the hotspots are arguments denoting a message's tag and body. To the best of our knowledge, these APIs cover all public methods provided to write log messages in Android app development.

Following the open-sourced list of source methods deployed by existing works (Gordon et al., 2015; Arzt et al., 2014; Calzavara et al., 2016; Cheng et al., 2018; Rasthofer et al., 2014)*, we combined the collected method signatures and configured them as the evidentiary source methods in LogExtractor. Currently, the evidence types defined by these source methods are categorized according to an existing machine-learning approach called SuSi* (Rasthofer et al., 2014)*. Note that LogExtractor has the flexibility of expanding the definition of evidence type as well as source methods for future extensions if needed.*

Rules of system native methods: Due to the fact that Android native methods potentially have data-flow effects on input arguments and return values, we use a similar approach as that in (Cheng et al., 2018) to apply the over-approximate summary for native methods. Specifically, we first obtain a list of Android native methods from DroidSafe (Gordon et al., 2015). For each native method, we compute the union of its input arguments' evidence types. Then, we update the evidence types of the arguments' and return value's (if any) tainted DFA as the union. However, since the approximation of DFA requires a reliable model of a native method, we do not modify the DFA part of a tainted DFA.

Rules of primitive data: The original implementation of JSA models primitive data as a single character with any value, which causes failure to match a log message having a floating number such as GPS latitude. Our improvement classifies these primitive types into four categories: char, integer, floating number, and boolean. We assign a DFA to each of them, which summarizes its string expression.

Moreover, JSA lacks the computation of binary operation expression between primitive data. In particular, JSA only models the string concatenation if both operands of an addition expression are of string type. However, JSA does not define the string expression computed in other binary operation expressions. As a result, a final output DFA is easily corrupted and destroyed. For example, in our evaluation of app *fishnoodle.koipond_free* (Koi Free Live Wallpaper, 2019), we observed that a tainted DFA is corrupted because it involves string expressions of primitive binary operations. Therefore, for other binary operation expressions involving primitive data, we model each operand's DFA based on their primitive type. Moreover, we compute the union of operands' evidence types and assign the result to the return value's taint table. For example, given a statement `p1 = p2 - p3`, where all operands are floating numbers, their DFAs are assigned as a DFA denoting any floating number while all states' evidence type in `p1`'s taint table is updated to the union of evidence types of all states in `p2` and `p3`.

$$v_1.method(v_2, v_3, \cdots); \tag{1}$$

$$v_0 = v_1.method(v_2, v_3, \cdots); \tag{2}$$

Data-flow summary of Android features: The original design of JSA implements the string data-flow summaries for Java system libraries such as "java.util.Collection" and "java.util.Iterator". However, JSA does not implement the data-flow summaries for Android APIs such as those under "android.os.Bundle" and "android.content.Intent", which are massively utilized for ICC. Since JSA corrupts all variables' DFAs within a method-invoking statement when the method is not recognized and summarized

[2] The DFA is equivalent to the regular expression (Lat-<int>.(0|[0–9]* [1-9])(E<int>)?|NaN|Infinity|-Infinity) where <int> is (0|-?[1-9][0–9]*).

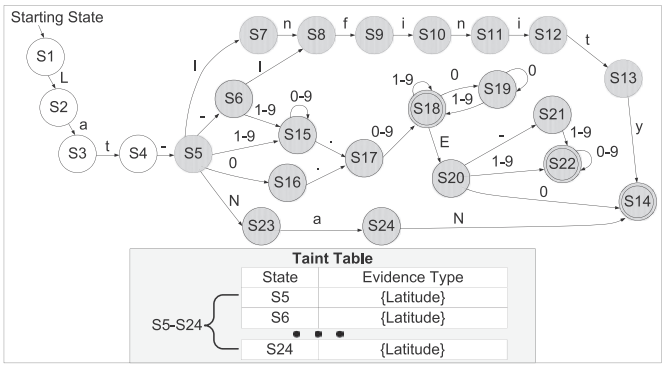


Fig. 4. Tainted DFA of the log message body assigned at Line 7 in Listing 1.

appropriately, we manually model the data-flow effect by JSA intermediate representation for such methods. In total, we model the data-flow summaries for 30 frequently used Android APIs. We collected these APIs from manual verification results by testing LogExtractor on both benchmark apps in DroidBench (2019) and real-world apps. For precisely identifying the relationship between different variables in an API, we abstract a method-invoking statement in two cases: Equations (1) and (2). Note that the data-flow summary abstracts the relationship between variables' DFAs only. We implement data-flow summary via JSA intermediate statements. For example, the summary $v_2 \leftarrow v_2 \cup v_4$ of *arraycopy*(…) is implemented to *ArrayAddAll*(v_2, v_4), where both v_2 and v_4 are `Array` type variables.

4.3. Identifying and extracting evidence

In this section, we introduce how to leverage tainted DFAs to identify and extract evidence from a given log message on a suspect's device. Specifically, we match the package name (if any), log level, message tag, and message body of a log message against the records in ALED. A log message is determined as evidentiary if it can be matched by some pattern of log message in ALED. Algorithm 1 shows our algorithm to extract evidence from a log message when it has evidentiary data. Basically, we use two pointers in the corresponding tainted DFA, one points to the previous state and the other one points to the current state. And we adapt two buffers *prevTaints currTaints* to record the taint information when moving pointers. After consuming each input character in the log message string, two pointers move ahead and apply the following rules: However, to solve *Evidence Extraction Problem*, we present the Algorithm 1 that takes the input of given log messages and tainted DFAs.

The last two inputs *auto* and *taintTable* are the two components of a tainted DFA, which are the original DFA and taint table, respectively. Each time a single character *c* is read, we determine if the next state is reachable from the current state. We leverage the method `nextState(char input)` at Line 7 to find the next state. As one of the characteristics defined in the automata theory, for any state in DFA, there is one destination state (next state) for a given unique input.

After moving to the next state, *prevTaints* stores the set of taint methods belonging to the previous state while *currTaints* stores the current one. The reasonings of extracting string belonging to a certain evidence type are explained as follows:

1. If an evidence type is involved in *currTaints* only, an empty string buffer is created and bound to this type.
2. If an evidence type is involved in *prevTaints* only, we output the string value in the corresponding string buffer as one of the evidence extraction results and remove the buffer.
3. If an evidence type is involved in both *prevTaints* and *currTaints*, the matched character is appended to the corresponding string buffer.

Line 13–20 in Algorithm 1 corresponds to the implementation of the above-mentioned procedure. With the tainted states, we obtain the potential evidence that a character belongs to and decide to perform appropriate string operation. The termination is treated in the same way as that in the matching procedure. As illustrated in Line 21, Algorithm 1 terminates when it reaches an acceptor of DFA and the end of the given log message string.

We use the log message "Lat-30.03" and tainted DFA in Fig. 4 as an example to show how our algorithm works. With the tainted DFA, we find no evidence type in the first three characters "Lat". After taking '-' as input and moving to State S5, evidence type `Latitude` is found in currTaint while prevTaint is empty. An empty string buffer is therefore created and bound to evidence type `Latitude`. The string buffer continues to append the rest of the log message until reaching state S18, which is an acceptor. Since we also reach the end of the log message, string "30.03" will be reported as output (evidence data) and bound with evidence type `Latitude`, as the table "Output Result" in Fig. 3 shows.

5. Evaluation

We evaluate LogExtractor with respect to answering our two research problems, namely, *Evidence Identification Problem* and *Evidence Extraction Problem*. To be specific, we first construct an ALED by using LogExtractor to analyze a large number of apps. Then, we run an app on a real device, Google Pixel 2 with Android 8.1 installed, and we extract the log messages from the device via ADB (Android Debug Bridge, 2019). After parsing the logs into messages by Android log parser Spell (Du and Li, 2016; Zhu et al., 2019; He et al., 2016), we identify evidentiary log messages through matching log messages' package name (if any), log level, tag, and body against the records in the ALED database. If a log message is matched, we apply LogExtractor to extract the evidence and compare it with the ground truths to evaluate the performance. First, we evaluate our tool's performance using publicly available benchmark apps from DroidBench (2019). Second, we evaluate our tool on publicly available real-world apps downloaded from AndroZoo (Allix et al., 2016) and PlayDrone (Viennot et al., 2014). Moreover, we show our manual verification results.

Our implementation of LogExtractor is based on JSA. Moreover, LogExtractor incorporates FlowDroid (Arzt et al., 2014) *to construct an Android call graph and IC3* (Dering et al., 2015) *to analyze and generate inter-component communications models. For large-scale real-world apps experiment, we deploy 16 cloud instances on Google Cloud Platform* (Google Cloud Platform, 2019) *and Amazon Web Service* (Amazon Web Services, 2019)*, each of which is equipped with 2 CPUs,* 13 GB *memory, and* 60 GB *SSD space.*

5.1. Results on benchmark apps

DroidBench includes a collection of 120 benchmark apps developed by the authors of FlowDroid (Arzt et al., 2014), DroidSafe (Gordon et al., 2015), and ICCTA (Li et al., 2015). These benchmark apps are frequently adopted to evaluate Android static taint analysis tools. Therefore, we also use them to evaluate our tool. Among these benchmark apps, 65 of them write log messages into the logging system. Therefore, we evaluate LogExtractor using these 65 benchmark apps. Specifically, we first use LogExtractor to build an

ALED for these 65 apps. Then, we install and use each app on an Android device. We retrieve the log messages generated by these apps from the Android device's logging system using ADB. In total, these benchmark apps generated 104 log messages and 53 of them contain certain data. We match each log message against the ALED to identify and extract data.

Table B2 in Appendix shows the results for each log message from each benchmark app. A log message is a *false positive* if LogExtractor predicts that the log message contains data but it actually does not contain data. A log message is a *false negative* if LogExtractor predicts that the log message does not contain data but it actually contains data. Overall, LogExtractor identified that 43 of log messages contain data, 1 of which is a false positive. Therefore, LogExtractor achieves 97.7% precision and 79.2% recall for the *Evidence Identification Problem*. Among the 42 log messages that LogExtractor correctly identifies as containing data, LogExtractor correctly extracts the data value from all of them, indicating a 100% accuracy for the *Evidence Extraction Problem*. We analyze the details of the 11 false negatives for the Evidence Identification Problem as follows:

5.2. Results on real-world apps

We use LogExtractor to analyze 12.1 K Android apps from AndroZoo and PlayDrone. Specifically, we run LogExtractor in parallel on our 16 cloud instances.

Results: Like many existing static analysis tools for Android (Yang et al., 2015; Cao et al., 2015; Octeau et al., 2015; Cheng et al., 2018; Calzavara et al., 2016), LogExtractor takes a long time to finish analyzing many real-world apps and has to set a timeout for the analysis of large-scale apps. Due to the limited resource, we set a 3-h timeout for each app. In particular, if our LogExtractor does not finish analyzing an app within 3 h, we force the analysis to stop. In total, LogExtractor finishes analyzing 70.0% of our apps with a 3-h timeout for each app. We note that, with a 2-h timeout, LogExtractor only finishes analyzing 17.4% of our apps. We evaluate the performance of the analysis time in detail at Section Appendix A to demonstrate why we need to set a fix timeout value for analyzing real-world apps.

Table 1 shows some statistical results about the ALED generated for the 12.1 K apps. We obtained the source methods for evidentiary data from SuSi (Rasthofer et al., 2014). We found that a large number of apps write a diverse set of data to the logging system. In particular, 4,464 of the 12.1 K apps do not write log messages to the logging system, 2,337 apps write log messages but they do not contain evidentiary data, while the remaining 5,382 apps write log messages that contain evidentiary data. On average, these 5,382 apps may write log messages with 8 different tainted DFAs that contain evidentiary data.

Manual evaluation: We also performed a best-effort end-to-end manual evaluation of LogExtractor on real-world apps. In this manual evaluation experiment, we focus on four types of evidentiary data, i.e., "Device ID", "Latitude", "Longitude", and "Text Input". We obtained the corresponding source methods for the four types of evidentiary data from SuSi (Rasthofer et al., 2014). Specifically, we randomly sampled 91 apps from the 12.1 K apps. We installed and used each of them on an Android device by giving sufficient permissions. Then, we retrieved the log messages generated by them from the Android device's logging system. In total, we collected over 90 K log entries.

We manually inspected the log messages to check whether they contain any of the four types of evidentiary data. For instance, we know the ground truth latitude and longitude coordinates of our own location. If a log message contains similar numbers to our latitude and/or longitude coordinates, then we label the log message as containing latitude and/or longitude. Similarly, we manually checked whether a log message contains Device ID or Text Input. Overall, we found that 266 of the 90 K log messages contain one of the four types of evidentiary data. We match each of the 90 K log messages against the ALED we built for the 12.1 K apps.

We found 22 false negatives and 36 false positives, which yield 86.5% precision and 91.3% recall in evidence identification. Moreover, for the correctly identified evidentiary messages, LogExtractor precisely extracts the corresponding evidence value without errors. Besides the limitations exposed by benchmark apps evaluation, we find the false negatives are primarily caused by the string representation of JSON object, which is an un-ordered set of key—value pairs and has its own stringify encoding rules such as automatically adding backslash and changing the order. Since parsing these behaviors requires runtime information and the used third-party libraries, LogExtractor cannot parse the string patterns. Furthermore, we find the false positives are from the app *com.ei.designateddriver1*(version: 1.0). Specifically, the app saves GPS coordinates into two consecutive log messages. For example, to save the GPS coordinates (20.1234, −30.45678), the app respectively writes the latitude and longitude into log messages "onLocationChange: 20.1234" and "onLocationChange: −30.45678", where "onLocationChange" is the tag of the log messages. In this case, LogExtractor over-approximately reports both "20.1234" and "-30.45678" correspond to the latitude and longitude types.

5.3. Case study

GPS coordinates: We use GPS coordinates as an example to illustrate how confusing log messages could mislead the investigation result and the challenges of manual inspection and keyword-based methods. The first three examples of Fig. 5 show that apps encode GPS coordinates in log messages. The first example has keywords "lat" and "lon" that a forensic investigator can leverage to determine the GPS coordinates. When assuming the latitude-longitude format, a forensic investigator may still be able to manually parse the latitude and longitude coordinates in the second example. However, such manual parsing fails for the third example, which follows the longitude—latitude format. In all three examples, LogExtractor correctly extracts the latitude and longitude coordinates.

UNIX timestamp: Fig. 5(d) demonstrates another real-world case where the exact UNIX timestamp value is attached in the log message and extracted by LogExtractor. Because existing log parsers (e.g. LogCluster (Vaarandi and Pihelgas, 2015)) can only determine the variable part "1601529204" is a number based on the regex format, they will miss the time evidence during the forensic investigation.

6. Discussion and limitations

LogExtractor shares the same limitations with existing Android static program analysis tools (Fuchs et al., 2009; Kim et al., 2012; Gibler et al., 2012; Lu et al., 2012; Yang and Yang, 2012; Arzt et al., 2014; Li et al., 2015; Wei et al., 2014; Gordon et al., 2015; Backes et al., 2016; Cheng et al., 2018; Lin et al., 2018). Such limitations include reflection, data flow and string operations in the native code, and implicit flow. For instance, one false negative missed by LogExtractor for the benchmark apps is caused by implicit flow, which is a common challenge for static program analysis. LogExtractor cannot track data that are first written to a file and then read out and written to the logging system. This is because we did not include reading from a file as a source of evidentiary data as it incurs many false positives. Since LogExtractor leverages FlowDroid and IC3 to model ICCs, LogExtractor inherits their limitations, e.g.,

they cannot parse all ICCs. For instance, 7 false negatives of LogExtractor for the benchmark apps are caused by the failure of ICC modeling.

LogExtractor also inherits the limitations of JSA as LogExtractor is based on JSA. For instance, JSA and LogExtractor do not model the string patterns of two-dimensional objects such as `HashMap` and `JSON`. In LogExtractor, we over-approximate a two-dimensional object by flattening it to a one-dimensional array, which prevents the object from being corrupted in JSA intermediate representations. However, such over-approximation sacrifices precision. Another limitation is that the model accepts any string value, such as the user text input. Since the any-string model combines all adjacent models into one, it deteriorates the performance of LogExtractor. Nevertheless, in our proposed tainted DFA, the taint table is independent of the DFA. Therefore, the propagation of tainted states can be easily extended when new string pattern models are proposed. We leave modeling the patterns of two-dimensional and any-string objects as future work.

As we observed in evaluation, when the evidence (e.g. GPS latitude and longitude) are written into two consecutive log messages with the same patterns, LogExtractor does not have timing information to differentiate the two log messages and thus loses precision. The mitigation of this issue requires modeling of runtime execution sequence and we leave it as the future work. Another limitation of logging-system forensics is that the logging system consists of small memory buffers, which are overwritten by the latest log messages. Therefore, any logging-system forensics tool can only identify and extract digital evidence that corresponds to the recent use (e.g., one week) of the Android device. However, the goal of digital forensics, in general, is to extract as many evidentiary data as possible. Therefore, logging-system forensics and LogExtractor are still valuable.

A limitation comes from our best-effort manual verification by using real-world data is that we can only verify the results with ground truths. For example, given a set of GPS coordinates, we can verify its existence in the log entries if they are written in plaintext format, however, our false-negative rate of the detection and false positive rate of the extraction may be overly optimistically counted if they are encrypted, hashed, or formatted by patterns unknown to the public. Additionally, our manual verification may under-approximate the false-negative rate due to our best-effort approach because we cannot cover 100% program coverage of Android apps during runtime, which is an open challenge (Li et al., 2019) in Android software testing.

Table 1
Statistical results of the ALED that LogExtractor builds for the 12.1 K real-world apps. We obtained the source methods for the evidence type from SuSi (Rasthofer et al., 2014).

Evidence Type	Tainted DFA		Apps
	Tag	Body	
Device ID	0	153	358
Latitude	55	6116	1688
Longitude	55	6116	1688
Altitude	0	30	30
Mailing address	2	7	6
Text input	2	539	407
Database information	4	86	25
Network information	0	269	1470
Calendar information	0	2835	1247
Others	1913	96783	4549

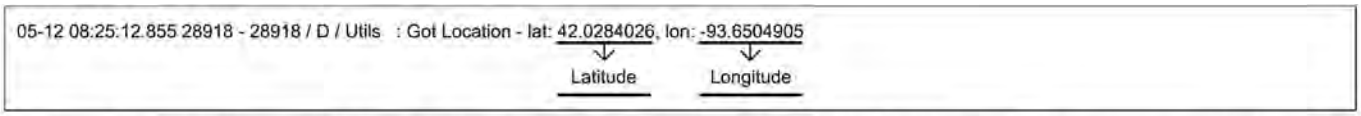
[a] We present a detail list of "Others" type evidence in Appendix Appendix C.

1. We find that 7 false negatives are caused by the failures of building ICC models. Specifically, IC3 cannot parse the destination components in the corresponding benchmark apps.
2. We find that 1 false negative is caused by the cases where the data is first written to a file and then read out later; another 2 false negatives are due to the lack of string models for the Java libraries called `Formatter` and `Matcher`; and the other 1 false negative is caused by the implicit flow case.
3. We find that the only 1 false positive is triggered in the "Button2" app. Since our approach relies on FlowDroid to generate the callback models, we inherit its limitation on over-approximating this test case as well.

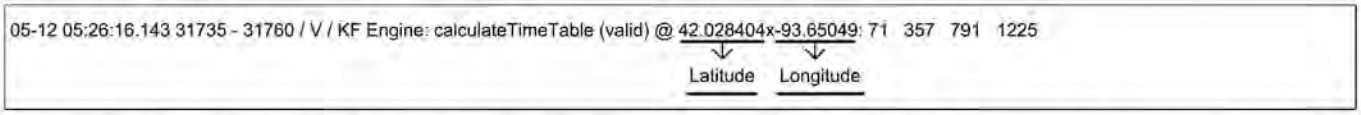
7. Related work

Forensic analysis on logging system: Although there are many studies on forensic analysis of Android apps, only a few of them mentioned the analysis of logging system. Even the NIST guideline of mobile device forensic analysis (Ayers et al., 2014) misses logging system. Satrya et al. (2016) performed both live and offline forensic analysis on three popular social messenger apps. They concluded that the evidence found in live analysis has no difference with the one found in offline analysis. Baggili et al. (2015) proposed to collect log messages of system events in Android smart watch since they might contain useful evidence. These studies took a manual analysis approach to analyze log messages, which suffers from the limitations we discussed in Introduction. A few studies (Htun and Thwin, 2017; Khandelwal et al., 2014; Htun et al., 2018) mentioned that the mobile forensic analysis should consider Android logging system as an evidentiary data sink. However, they are just limited to discussion without performing any analysis or designing any tools. While not focusing on the Android logging system, two existing approaches, OmegaLog (Ul Hassan et al., 2020) and UIScope (Yang et al., 2020), aim to identify the root causes of events by analyzing and correlating the log messages. UIScope identifies root causes by monitoring and correlating UI events to system events. OmegaLog took an offline-online approach. In particular, OmegaLog uses symbolic execution on C/C++ binary programs to analyze the log message patterns offline and identifies the root causes online using the log message patterns. Unlike OmegaLog, LogExtractor analyzes complicated string-expression constructions, e.g., StringBuffer, Arrays.toString(), and replace(). Moreover, LogExtractor combines fine-grained taint analysis and automaton-based string analysis to analyze log message patterns.

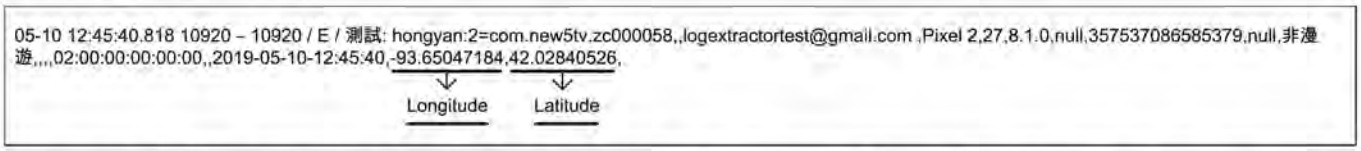
Log parsing: The goal of log parsing is to automatically categorize log messages into different specific patterns. For example, given a message "Received network packet of size 56 KB from 10.251.91.84.", a log parser aims to convert it to "Received network packet of size <*> from <*>.", where <*> is a position parameter denoting target variables. Most of existing studies (Vaarandi, 2003; Vaarandi and Pihelgas, 2015; Fu et al., 2009; Du and Li, 2016; Makanju et al., 2009; Nagappan and Vouk, 2010; Messaoudi et al., 2018; He et al., 2017; Hamooni et al., 2016) summarize log message patterns by deploying data mining techniques on massive amount of log messages, e.g., frequent pattern mining (SLCT (Vaarandi, 2003), LogCluster (Vaarandi and Pihelgas, 2015)), longest common subsequence computation (Spell (Du and Li, 2016)), hierarchical clustering (LKE (Fu et al., 2009)), etc. Zhu et al. (2019) and He et al. (2016) implemented most of these approaches and made the tools publicly available. However, these log parsers are insufficient for our forensics problem. This is because they can only provide the information on which string segment is variable but cannot determine which type of evidence the string segment corresponds to.



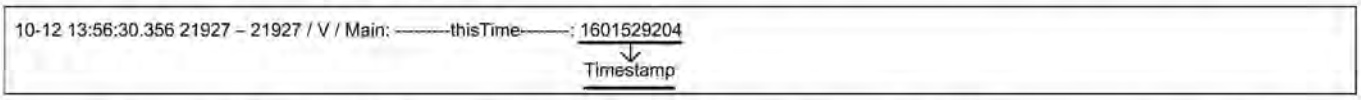
(a) Example 1: GPS coordinates follow the latitude-longitude format and have identifiable keywords. Log message was captured from *com.denite.watchface.blackmetal* (version: 2.6.4).



(b) Example 2: GPS coordinates follow the latitude-longitude format without keywords. Log message was captured from *fishnoodle.koipond_free* (version: 1.9).



(c) Example 3: GPS coordinates follow the longitude-latitude format without keywords. Log message was captured from *com.new5tv.zc000058* (version: PRM_2.1).



(d) Example 4: Event's UNIX timestamp parsed from the log message generated by *jsg.capturebrowser* (version: 1.5).

Fig. 5. Four examples of real-world log messages. LogExtractor correctly extracts the latitude, longitude coordinates and the UNIX timestamp.

Taint analysis: Program analysis, in particular, taint analysis, is generally applied for security and privacy analysis in Android apps. While there is a dynamic approach (Xu et al., 2018) implemented for Android app forensics, log messages are primarily covered by static program analysis. By treating logging system APIs as sink methods and determining if a tainted variable (by source API, e.g. `getDeviceId()`) reaches the sink methods, taint analysis can give a coarse-scale answer on whether certain kinds of sensitive information is written into logging system. Prior Android taint analysis works that cover sensitive data leakage through logging system include CHEX (Lu et al., 2012), LeakMiner (Yang and Yang, 2012), FlowDroid (Arzt et al., 2014), AmanDroid (Wei et al., 2014), IccTA (Li et al., 2015), DroidSafe (Gordon et al., 2015), R-Droid (Backes et al., 2016), and HornDroid (Calzavara et al., 2016). AmanDroid observed user password was leaked by a music player app to logging system. LeakMiner is the first work introducing logging system APIs as sink methods in taint analysis and reported hundreds of apps leak sensitive information through logging system from 1,750 real-world apps. Similarly, FlowDroid, IccTA, and R-Droid reported a number of sensitive data leakages by real-world apps to the logging system. Additionally, due to the large number of data leakages through logging system, R-Droid performed a case study on apps that write user credentials into the logging system. These tools are insufficient for our logging-system forensics problems, as they did not model the string patterns for each log message.

String analysis: As a significant branch of static program analysis, string analysis focuses on discovering all possible string patterns of given variables. Unlike taint analysis and/or constant analysis, due to the complicate string operations in a program, such as concatenation, insertion, and replacement, string analysis often consumes a large memory and requires a long time to compute the results for a large program. Java String Analyzer (Christensen et al., 2003) (JSA), which builds on top of Soot (Vallée-Rai et al., 1999), is a popular tool for string analysis of java programs. JSA was also extended to other languages. For instance, Christodorescu et al. (2005) extended JSA to recover C-style string in x86 executables. Symbolic Pathfinder (Corina et al., 2013) incorporates JSA to support symbolic string resolution. Kausler et al. (Scott and Sherman, 2014) compared the modeling cost and accuracy of multiple string analysis tools. JSA ranked the second but the first one is developed for PHP language. Both DroidSafe (Gordon et al., 2015) and Bagheri et al. (2016) used JSA to analyze the intent API argument's value to resolve the intent target. To analyze the GUI label text on user interface of an Android app, GUILeak (WangXue et al., 2018) leveraged JSA to analyze the string values associated with all API methods that will set text to GUI views. Violist (Ding et al., 2015) leveraged the summary based technique to reduce the memory requirement of string analysis. The major limitation of JSA and its extensions is that they did not track flow of evidentiary data. As a result, they are insufficient for our forensics problems. To address the limitation, our LogExtractor combines string and taint analysis.

8. Summary and future work

In this paper, we propose LogExtractor, the first tool to automatically identify and extract digital evidence from Android log messages. LogExtractor takes an offline-online approach. In the offline phase, LogExtractor builds an App Log Evidence Database (ALED) for a large number of apps via combining string and taint analysis. Specifically, to build ALED, LogExtractor leverages a new data structure called tainted DFA, associates a tainted DFA with each variable in an Android app, and propagates the tainted DFAs among an Android app's code. In the online phase, LogExtractor

matches each log message on a suspect's Android device against the ALED to identify and extract evidentiary data. We evaluate LogExtractor on both benchmark apps and real-world apps. Our results show that a large number of apps write a diverse set of evidentiary data into log messages, and our LogExtractor can accurately identify and extract them. An interesting future work is to extend LogExtractor to analyze the string patterns of content in the Android file system (e.g. content provider), which aids the extraction of evidentiary data. Analyzing string patterns for the file system is more challenging because the content in a file may have more complicated formats than log messages.

Acknowledgments

This work was partially supported by NIST CSAFE under Cooperative Agreement No. 70NANB20H019, NSF under grants No. CNS-1527579, CNS-1619201, CNS-1730275, DEB-1924178, ECCS-2030249, and Boeing Company. We appreciate anonymous reviewers for their valuable suggestions and comments.

Appendix

Appendix A. Performance Evaluation

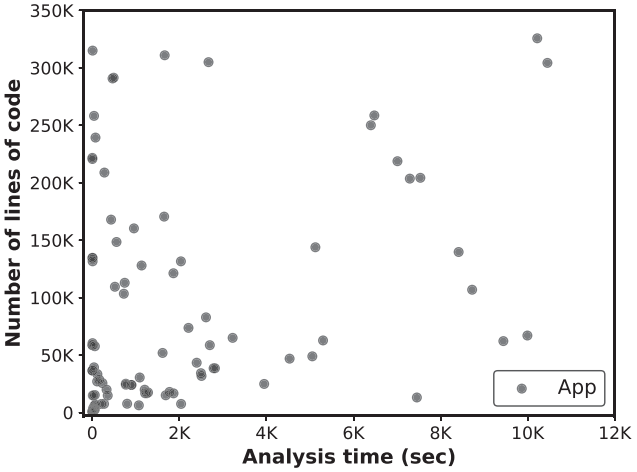


Fig. A.6. Number of lines of code vs. analysis time.

We did performance analysis on randomly sampled 129 apps whose analysis finished within 3 h and 129 apps whose analysis did not. Our tool is based on FlowDroid, IC3, and JSA. In total, the tool roughly has 13 stages, e.g., call graph generation, liveness, aliasing analysis, and reaching definition analysis. Among the 129 apps whose analysis finished within 3 h, reaching definition analysis and aliasing analysis are the top-2 most time consuming stages, which took 1,000 s and 571 s on average, respectively. We categorize the 129 apps whose analysis did not finish within 3 h as follows:

- 72 (55.8%) of them timeout because of aliasing analysis. They spent 10,297 s (on average) for aliasing analysis before timeout. Our observation is consistent with previous work (Leone, 2017) that also reported the heavy cost of aliasing analysis in JSA.
- 30 (23.2%) of them spent 6,917 s (on average) for reaching definition analysis before timeout.
- 16 (12.4%) of them spent 9,727 s (on average) for liveness analysis before timeout.
- 10 (7.8%) of them cannot have their call graphs generated within 3 h by FlowDroid and IC3.
- 1 (0.8%) of them timeouts in the last stage. Aliasing analysis and reaching definition analysis took 3,869 s and 6,691 s for the app.

Figure A.6 shows the app size (lines of code) vs. analysis time for 100 randomly sampled apps whose analysis finished within 3 h. Each dot in the figure corresponds to an app. The Pearson Correlation Coefficient between app size and analysis time is 0.35, which means that there is moderate positive correlation between app size and analysis time.

Algorithm 1. Extracting evidentiary data from a log message.

Input: Log message string $message$, DFA $auto$, Map<state's id, set of evidence types> $taintTable$
Output: Map<evidence type, set of matched strings> $output$
Ensure: $auto$ accepts $message$
Initialization:
$workMap \leftarrow$ empty map <evidence type, StringBuffer>
$workState \leftarrow$ starting state of $auto$

if $taintTable$.containsKey($workState$) **then**
  $prevTaints \leftarrow taintTable$.get($workState$)
  for each evidence type $m \in prevTaint$ **do**
    $workMap$.put(m, empty buffer)
else $prevTaints \leftarrow \varnothing$
for each character $c \in str$ **do**
  $nextState \leftarrow workState$.nextState($c$)
  if $nextState =$ `null` **then return** $output$
  if $taintTable$.containsKey($nextState$) **then**
    $currTaints \leftarrow taintTable$.get($nextState$)
  else $currTaints \leftarrow \varnothing$
  for each evidence type $m \in prevTaints \cup currTaints$ **do**
    if $m \in prevTaints \cap currTaints$ **then**
      Append c to the buffer mapped by m in $workMap$
    else if $m \in prevTaints$ **then**
      $workMap$.put(m, empty buffer)
    else
      $extractStr \leftarrow workMap$.get($m$).toString()
      Update $output$ by m and $extractStr$
      $workMap$.remove(m)
  if $nextState$ is an acceptor $\wedge$ c is the last character **then**
    for each evidence type $m \in workMap$.keySet() **do**
      $extractStr \leftarrow workMap$.get($m$).toString()
      Update $output$ by m and $extractStr$
  $workState \leftarrow nextState$
  $prevTaints \leftarrow currTaints$
return $output$

Appendix B. DroidBench Evaluation Result

Table B.2
DroidBench evaluation results of LogExtractor. Each symbol represents a log message for an app, e.g., if an app has 3 symbols in the column "Match" or "Extract", then the app generates 3 log messages.

Benchmark App	Match[a]	Extract[b]	Benchmark App	Match	Extract
ArrayCopy1	⊛	✔	EventOrdering1	⊛	✔
ArrayToString1	⊛	✔	SharedPreferenceChanged1	◯	✘
MultidimensionalArray1	⊛	✔	Clone1	⊛	✔
AnonymousClass1	⊛	✔	FactoryMethods1	⊛⊛	✔✔
Button2	⊛⊛∗	✔✔✘	Serialization1	◯	✘
Button5	⊛	✔	StaticInitialization3	⊛	✔
LocationLeak1	⊛⊛	✔✔	StringFormatter1	◯	✘
LocationLeak2	⊛⊛	✔✔	StringPatternMatching1	◯	✘
LocationLeak3	⊛	✔	StringToCharArray1	⊛	✔
MethodOverride1	⊛	✔	StringToOutputStream1	⊛	✔
MultiHandlers1			UnreachableCode		
RegisterGlobal1			VirtualDispatch1	⊛	✔
Echoer			VirtualDispatch2		
SendSMS	⊛	✔	ApplicationModeling1	⊛	✔
StartActivityForResult1	⊛◯	✔✘	InactiveActivity		
ActivityCommunication2	◯	✘	LogNoLeak		
ActivityCommunication3	◯	✘	PrivateDataLeak1		
ActivityCommunication4	⊛	✔	PrivateDataLeak2	⊛	✔
ActivityCommunication5	⊛	✔	PublicAPIField1	⊛	✔
ActivityCommunication6	◯	✘	PublicAPIField2	⊛	✔
ActivityCommunication7	⊛	✔	ImplicitFlow1	◯	✘
ActivityCommunication8	◯	✘	ImplicitFlow2		
BroadcastTaintAndLeak1	⊛	✔	ImplicitFlow3		
ComponentNotInManifest1			ImplicitFlow4		
Ordering1			AsyncTask1		
IntentSource1			JavaThread1	⊛	✔
ServiceCommunication1	⊛	✔	JavaThread2	⊛	✔
SharedPreferences1	◯	✘	Executor1	⊛	✔
Singletons1	⊛	✔	Looper1	⊛	✔
UnresolvableIntent1	⊛	✔	Emulator: ContentProvider1	⊛	✔
ActivitySavedState1			Emulator: IMEI1		
AsynchronousEventOrdering1	⊛	✔	Emulator: PlayStore1	⊛	✔
BroadcastReceiverLifecycle2	⊛	✔			

[a] ⊛ = true positive, ∗ = false positive, ◯ = false negative.
[b] ✔= correctly extract the data value, ✘= incorrectly extract the data value.

Appendix C. Distribution of Methods Classified as "Others" Evidence

In SuSi (Rasthofer et al., 2014), if a system method cannot be classified into any of their predefined categories, it will be determined as "Others". LogExtractor reuses its categorization result and lists the distribution of "Others" methods in the real-world app evaluation.

Table C.3
Top 30 methods whose return value is embedded as "Others" types of evidence in log messages generated by real-world apps.

Class Name	Method Name	Percentages in Evaluation
java.lang.Class	getSimpleName	6.82%
android.view.MotionEvent	getAction	3.61%
android.graphics.Bitmaps	getHeight	3.56%
android.view.MotionEvent	getX	3.21%
android.graphics.Bitmap	getWidth	3.15%
android.view.MotionEvent	getY	3.10%
java.lang.StackTraceElement	getMethodName	2.29%
android.view.Display	getWidth	2.18%
android.view.Display	getHeight	2.01%
android.content.res.Resources	getIdentifier	1.66%
android.view.View	getId	1.61%
java.io.File	getAbsolutePath	1.49%
android.view.View	getHeight	1.43%
java.lang.Thread	getName	1.43%
android.view.View	getWidth	1.15%
java.lang.StackTraceElement	getClassName	1.15%
java.lang.StackTraceElement	getLineNumber	1.15%
android.view.View	getVisibility	1.15%
android.graphics.drawable.Drawable	getIntrinsicWidth	1.03%
android.view.View	getLeft	1.03%
android.view.MotionEvent	getRawX	1.03%
java.lang.StackTraceElement	getFileName	0.97%
android.graphics.drawable.Drawable	getIntrinsicHeight	0.97%
android.view.ViewConfiguration	getScaledTouchSlop	0.92%
android.view.View	getRight	0.92%
android.view.VelocityTracker	getXVelocity	0.92%
android.view.ViewConfiguration	getScaledMaximumFlingVelocity	0.86%
android.view.VelocityTracker	getYVelocity	0.86%
android.view.Display	getOrientation	0.86%
android.view.Display	getRotation	0.86%

References

2020, 2020. Logcat reader. https://play.google.com/store/apps/details?id=com.dp.logcatapp.

Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L., 2016. AndroZoo: collecting millions of android apps for the research community. In: MSR.

Alonso Parrizas, A., 2015. Forensic Analysis on Android: A Practical Case. SANS Institute InfoSec Reading Room, 2015.

2019 Amazon Web Services, 2019. https://aws.amazon.com/.

2019 Android Debug Bridge, 2019. https://developer.android.com/studio/command-line/adb.

Arzt, Steven, Rasthofer, Siegfried, Fritz, Christian, Bodden, Eric, Bartel, Alexandre, Klein, Jacques, Le Traon, Yves, Octeau, Damien, McDaniel, Patrick, 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI.

Ayers, Rick, Brothers, Sam, Jansen, Wayne, Ayers, Rick, Brothers, S., 2014. Guidelines on Mobile Device Forensics. NIST Special Publication, 2014.

Backes, Michael, Bugiel, Sven, Derr, Erik, Gerling, Sebastian, Hammer, Christian, 2016. R-droid: leveraging android app analysis with static slice optimization. In: AsiaCCS.

Baggili, I., Oduro, J., Anthony, K., Breitinger, F., McGee, G., 2015. Watch what you wear: preliminary forensic analysis of smart watches. In: ARES.

Bagheri, H., Sadeghi, A., Jabbarvand, R., Malek, S., 2016. Practical, formal synthesis and automatic enforcement of security policies for android. In: DSN.

Bhatia, Rohit, Saltaformaggio, Brendan, Yang, Seung Jei, Aisha, I., Ali-Gombe, Zhang, Xiangyu, Xu, Dongyan, Richard, Golden G., 2018. Tipped off by your memory allocator: device-wide user activity sequencing from android memory images. In: NDSS.

Calzavara, Stefano, Grishchenko, Ilya, Maffei, Matteo, 2016. HornDroid: practical and sound static analysis of android applications by SMT solving. In: EuroS&P.

Cao, Yinzhi, Fratantonio, Yanick, Egele, Manuel, Bianchi, Antonio, Kruegel, Christopher, Vigna, Giovanni, Chen, Yan, 2015. EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In: NDSS.

Cheng, Chris Chao-Chun, Shi, Chen, Gong, Neil Zhenqiang, Guan, Yong, 2018. EviHunter: identifying digital evidence in the permanent storage of android devices via static analysis. In: CCS.

Christensen, Aske Simon, Møller, Anders, Michael, I., Schwartzbach, 2003. Precise analysis of string expressions. In: SAS.

Christodorescu, Mihai, Kidd, Nicholas, Goh, Wen-Han, 2005. String analysis for x86 binaries. In: PASTE.

Corina, S Păsăreanu, Visser, Willem, Bushnell, David, Geldenhuys, Jaco, Mehlitz, Peter, Rungta, Neha, 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. In: ASE, vol. 20, p. 3, 2013.

Dering, Matthew, Jha, Somesh, McDaniel, Patrick, Octeau, Damien, Luchaup, Daniel, 2015. Composite constant propagation: application to android inter-component communication analysis. In: ICSE.

Ding, Li, Lyu, Yingjun, Wan, Mian, William, G., Halfond, J., 2015. String analysis for java and android applications. In: ESEC/FSE.

2019 DroidBench, 2019. https://github.com/secure-software-engineering/DroidBench.

Du, M., Li, F., 2016. Spell: streaming parsing of system event logs. In: ICDM.

Feldthaus, Asger, Møller, Anders, 2009. The Big Manual for the Java String Analyzer. Department of Computer Science, Aarhus University, 2009.

Fu, Q., Lou, J., Wang, Y., Li, J., 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In: ICDM.

Fuchs, Adam P., Chaudhuri, Avik, Foster, Jeffrey S., 2009. Scandroid: Automated Security Certification of Android. Technical Report.

Gibler, Clint, Crussell, Jonathan, Erickson, Jeremy, Chen, Hao, 2012. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: TRUST.

2019 Google Cloud Platform, 2019. https://cloud.google.com/.

Gordon, Michael I., Kim, Deokhwan, Perkins, Jeff H., Gilham, Limei, Nguyen, Nguyen, Rinard, Martin C., 2015. Information flow analysis of android applications in DroidSafe. In: NDSS.

Hamooni, Hossein, Debnath, Biplob, Xu, Jianwu, Zhang, Hui, Jiang, Guofei, Abdullah, Mueen, 2016. LogMine: fast pattern recognition for log analytics. In: CIKM.

He, P., Zhu, J., He, S., Li, J., Lyu, M.R., 2016. An evaluation study on log parsing and its use in log mining. In: DSN.

He, P., Zhu, J., Zheng, Z., Lyu, M.R., 2017. Drain: an online log parsing approach with fixed depth tree. In: ICWS.

Htun, Naing Linn, Thwin, Mie Mie Su, 2017. Proposed workable process flow with analysis framework for android forensics in cyber-crime investigation. IJES 6, 1, 2017.

Htun, N.L., Thwin, M.M.S., San, C.C., 2018. Evidence data collection with ANDROSICS tool for android forensics. In: ICITEE.

Khandelwal, Pooja, Sahu, Divya Rishi, Tomar, D.S., 2014. Scrutinize evidences for android phones. In: IJCSIT, vol. 5, p. 2, 2014.

Kim, Jinyung, Yoon, Yongho, Yi, Kwangkeun, Shin, Junbum, SWRD Center, 2012. ScanDal: static analyzer for detecting privacy leaks in android applications. In: MoST, vol. 12, 2012.

2019 Koi Free Live Wallpaper, 2019. https://play.google.com/store/apps/details?id=fishnoodle.koipond_free.

Leone, Hannah, 2017. GPS Evidence Can Be Used in Aurora KFC Shooting Case, 2017. https://tinyurl.com/y5ayk63s.

Li, Li, Bartel, Alexandre, Bissyandé, Tegawendé F., Klein, Jacques, Le Traon, Yves, Arzt, Steven, Rasthofer, Siegfried, Bodden, Eric, Octeau, Damien, McDaniel, Patrick, 2015. IccTA: detecting inter-component privacy leaks in android apps. In: ICSE.

Li, Yuanchun, Yang, Ziyue, Guo, Yao, Chen, Xiangqun, 2019. Humanoid: a deep learning-based approach to automated black-box android app testing. In: ASE.

Lin, Xiaodong, Chen, Ting, Zhu, Tong, Yang, Kun, Wei, Fengguo, 2018. Automated forensic analysis of mobile applications on Android devices. Digit. Invest. 26, 2018.

Lu, Long, Li, Zhichun, Wu, Zhenyu, Lee, Wenke, Jiang, Guofei, 2012. CHEX: statically vetting android apps for component Hijacking vulnerabilities. In: CCS.

Makanju, Adetokunbo A.O., Zincir-Heywood, A. Nur, Milios, Evangelos E., 2009. Clustering event logs using iterative partitioning. In: KDD.

Messaoudi, Salma, Panichella, Annibale, Bianculli, Domenico, Briand, Lionel, Sasnauskas, Raimondas, 2018. A search-based approach for accurate identification of log message formats. In: ICPC.

Nagappan, M., Vouk, M.A., 2010. Abstracting log lines to log event types for mining software system logs. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 114–117.

Octeau, Damien, Luchaup, Daniel, Dering, Matthew, Jha, Somesh, McDaniel, Patrick, 2015. Composite constant propagation: application to android inter-component communication analysis. In: ICSE.

Raphael, Meghan, 2016. Android's Location Data Can Now Help Police Track Suspected Criminals to the Scene of the Crime, 2016. https://tinyurl.com/y46g6cuv.

Rasthofer, Siegfried, Arzt, Steven, Bodden, Eric, 2014. A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS.

Saltaformaggio, Brendan, Bhatia, Rohit, Gu, Zhongshu, Zhang, Xiangyu, Xu, Dongyan, 2015a. GUITAR: piecing together android app GUIs from memory images. In: CCS.

Saltaformaggio, Brendan, Bhatia, Rohit, Gu, Zhongshu, Zhang, Xiangyu, Xu, Dongyan, 2015b. VCR: app-agnostic recovery of photographic evidence from android device memory images. In: CCS.
Saltaformaggio, Brendan, Bhatia, Rohit, Zhang, Xiangyu, Xu, Dongyan, Richard III, Golden G., 2016. Screen after previous screens: spatial-temporal recreation of android app displays from memory images. In: Usenix Security Symposium.
Satrya, G.B., Daely, P.T., Shin, S.Y., 2016. Android forensics analysis: private chat on social messenger. In: ICUFN.
Scott, Kausler, Sherman, Elena, 2014. Evaluation of string constraint solvers in the context of symbolic execution. In: ASE.
Ul Hassan, Wajih, Noureddine, Mohammad A., Datta, Pubali, Adam, Bates, 2020. OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis. In: NDSS.
Vaarandi, R., 2003. A data clustering algorithm for mining patterns from event logs. In: IPOM.
Vaarandi, R., Pihelgas, M., 2015. LogCluster - a data clustering and pattern mining algorithm for event logs. In: CNSM.
Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, Sundaresan, Vijay, 1999. Soot - a java bytecode optimization framework. In: CASCON.
Viennot, Nicolas, Garcia, Edward, Nieh, Jason, 2014. A measurement study of Google play. In: SIGMETRICS.
Wang, Xiaoyin, Xue, Qin, Mitra, Bokaei Hosseini, Slavin, Rocky, Breaux, Travis D., Niu, Jianwei, 2018. GUILeak: tracing privacy policy claims on user input data for android applications. In: ICSE.
Wei, Fengguo, Roy, Sankardas, Ou, Xinming, Robby, 2014. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: CCS.
Xu, Zhen, Shi, Chen, Cheng, Chris Chao-Chun, Gong, Neil Zhengqiang, Guan, Yong, 2018. A dynamic taint analysis tool for android app forensics. In: SADFE.
Yang, Z., Yang, M., 2012. LeakMiner: detect information leakage on android with static taint analysis. In: WCSE.
Yang, Wei, Xiao, Xusheng, Benjamin, Andow, Li, Sihan, Xie, Tao, Enck, William, 2015. AppContext: differentiating malicious and benign mobile app behaviors using context. In: ICSE.
Yang, Runqing, Ma, Shiqing, Xu, Haitao, Zhang, Xiangyu, Chen, Yan, 2020. UISCOPE: accurate, instrumentation-free, and visible attack investigation for GUI applications. In: NDSS.
Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., Lyu, M.R., 2019. Tools and benchmarks for automated log parsing. In: ICSE-SEIP.