DroidForensics: Accurate Reconstruction of Android Attacks via Multi-layer Forensic Logging

Xingzi Yuan[†] Omid Setayeshfar[†] Hongfei Yan[†] Pranav Panage[†] Xuetao Wei[‡] Kyu Hyung Lee[†]
[†]University of Georgia, Athens, GA, USA
[‡]University of Cincinnati, Cincinnati, OH, USA
{x.yuan, omid.s, hfyan, pranav.panag, kyuhlee}@uga.edu, weix2@ucmail.uc.edu

ABSTRACT

The goal of cyber attack investigation is to fully reconstruct the details of an attack, so we can trace back to its origin, and recover the system from the damage caused by the attack. However, it is often difficult and requires tremendous manual efforts because attack events occurred days or even weeks before the investigation and detailed information we need is not available anymore. Consequently, forensic logging is significantly important for cyber attack investigation.

In this paper, we present DroidForensics, a multi-layer forensic logging technique for Android. Our goal is to provide the user with detailed information about attack behaviors that can enable accurate post-mortem investigation of Android attacks. DroidForensics consists of three logging modules. API logger captures Android API calls that contain high-level semantics of an application. Binder logger records interactions between applications to identify causal relations between processes, and system call logger efficiently monitors low-level system events. We also provide the user interface that the user can compose SQL-like queries to inspect an attack. Our experiments show that DroidForensics has low runtime overhead (2.9% on average) and low space overhead (105 $\sim$ 169 MByte during 24 hours) on real Android devices. It is effective in the reconstruction of real-world Android attacks we have studied.

1. INTRODUCTION

Android devices are becoming increasingly popular but at the same time, also constantly attract cyber criminals. For example, a recent "stagefright" attack [28] exploits a vulnerability in Android core component, which potentially infects 950 million Android devices.

Consequently, there is an increasing need of detecting and investigating Android attacks. Forensic logs are critical to the cyber attack investigation. For example, when a user detects a symptom of an attack, we can analyze forensic logs to reconstruct the attack path from the symptom to the "entry point" of the attack. It is also important to understand the damage have been conducted by the attack. For instance, understanding what system objects are compromised or what data was exfiltrated is important to resolve the damage.

Forensic logging captures behaviors of the system execution and their relations. For instance, audit logging techniques [11] are widely used for cyber attack forensics. They records system properties such as users, processes, files or network sockets and their relations such as a process receives data from network socket, a user log-in to the system or a system file is replaced by a process. It can be used for backward and forward tracking [43, 45] to locate the origin of an attack and to identify the damage to the system. Recent work [45, 49, 29] show that forensic logging is an effective technique for cyber attack forensics in desktop or server environments.

However, unlike traditional desktop or server applications, most Android applications run within the virtual machine called Android Run-time (ART) to provide isolated execution environment for each process. There are two major hindrances to use traditional system call logging in Android. First, system calls might be too low-level to capture the rich semantics of application behaviors. Second, Android has unique inter-process communication (IPC) protocol, called `binder` and it is difficult to accurately capture IPC from the system calls. For example, if the Android application steals a contact list from the device and sends it to SMS message, system calls cannot capture the critical behaviors such as *reading contact information* and *sending SMS message to the attacker's number*. Because the Android application cannot directly access contact or SMS, but it uses `binder` call to interact with Android service providers such as `ContentProvider` or `SMSManager` to access contact or SMS.

Recently, CopperDroid [63] and DroidScope [66] have proposed techniques to analyze the behaviors of Android malware. CopperDroid developed system-call based analysis techniques for Android attack reconstruction. DroidScope [66] is the Android malware analysis engine that provides unified view of hardware, kernel and Dalvik virtual machine information. However both are built on top of emulated environments (e.g., QEMU [22]) and it generally incurs nontrivial runtime and space overhead for resource-constrained mobile devices.

In this paper, we develop a multi-layer forensic logging technique for Android, called DroidForensics. DroidForensics captures important Android events from three layers; Android API, Binder and system calls. Our API logger can capture high-level semantics of application, Binder logger ac-

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3052984

curately captures interactions between applications, and system call logger records low-level events such as system calls. In addition, DroidForensics provides easy-to-use, SQL-like user interface that the user can compose queries to inspect an attack. DroidForensics generates a causal graph to answer the query and the user can iteratively refine queries based on the previous graph. We do not require an emulated environment and DroidForensics is designed for real devices. In summary, this paper makes the following contributions:

- We design and implement a multi-layer forensic logging system for Android. Our system consists of three modules to capture different levels of information from high-level application semantics to low-level system events. We also accurately capture inter process communication via Android's binder protocol.
- We develop a light-weight system call logging technique for Android. Existing Android audit system [4] causes up to 46% overhead in Nexus 6 that would be too expensive to be active during normal execution. Our runtime overhead on Nexus 6 is only less than 4.05%. We can also reduce the space consumption substantially.
- We develop an easy-to-use user interface to aid the attack investigation. The attack reconstruction is carried out by writing SQL-like queries. Our pre-process automatically converts the user query to SQL-queries, and the post-processor generates causal graphs.
- We evaluate the efficiency, effectiveness and compatibility of DroidForensics. The results conducted on widely used Android benchmarks show that our runtime overhead is only about 2.9% on average and 6.16% in the worst case. We present that 31 android malwares are effectively resolved by querying various levels of information. The compatibility results produced by Android Compatibility Suite (CTS) show that DroidForensics maintains the same compatibility-level comparing with original Android.

The rest of this paper is organized as follows. Section 2 introduces the overview of DroidForensics and motivating example using Android malware called AVPass. Section 3 discusses our design and implementation details. In Section 4, we evaluate DroidForensics for efficiency, effectiveness and compatibility. We discuss limitations of DroidForensics and our future plans in Section 5. Section 6 presents related works and we conclude the paper in Section 7.

2. SYSTEM OVERVIEW AND MOTIVATING EXAMPLE

In this section, we present an overview of DroidForensics and we use a real-world Android malware, `AVPass` [14], to motivate our work.

A high-level overview of DroidForensics is depicted in Figure 1. It has three logging modules, namely API logger, Binder logger and System call logger. API logger captures important Android APIs such as accessing database, controlling sensitive devices (e.g.,a camera, GPS or microphone). Binder logger monitors interactions between processes via IPC or RPC, record their information such as process id for the caller (or the client) and the callee (or the server), and a message shared between them. Finally system call logger records forensic-related system calls such as calls that

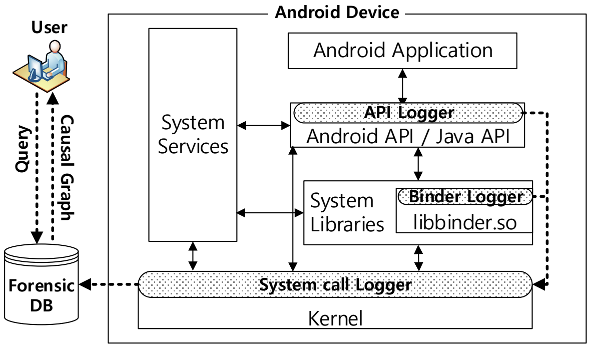


Figure 1: High level overview of DroidForensics

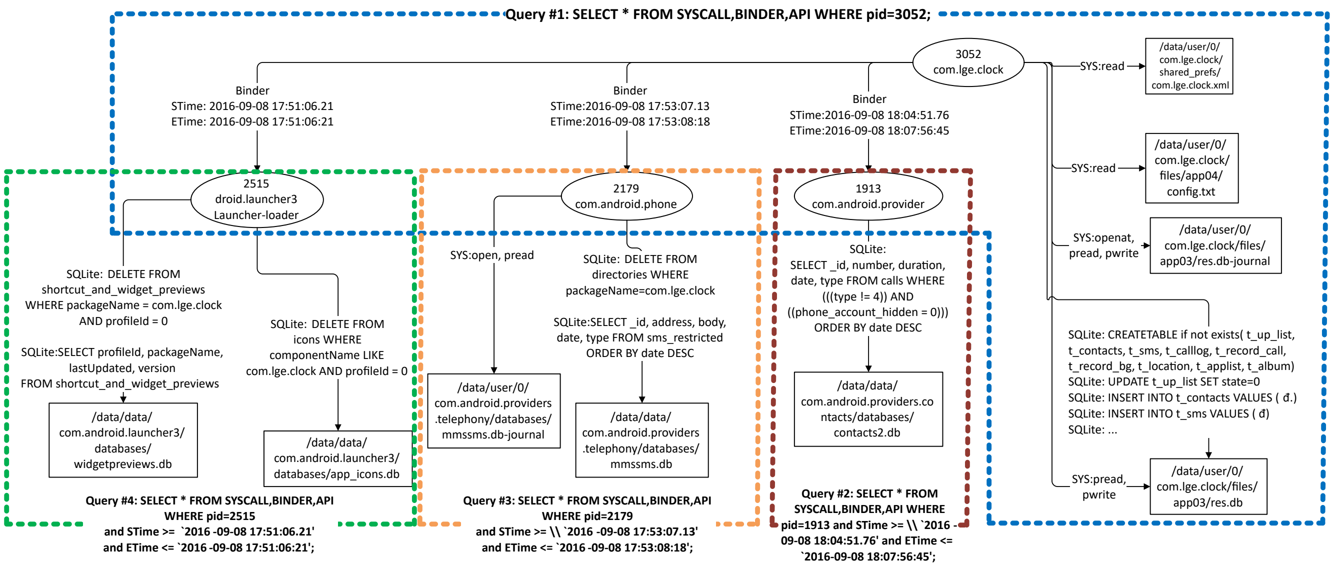
affect other processes (e.g., fork, kill) or other system object (e.g., read, write, recv, send). To record a global order of these events from different layers, API and Binder loggers forward their events to system call logger and system call logger stores them with global timestamps. The dotted line in Figure 1 shows the flow of collected forensic logs. DroidForensics periodically transfers those forensic data to an external server through wifi and three layers of logs are encoded uniformly into a relational database. Finally the user can compose SQL-like queries to investigate an attack. DroidForensics converts the user query to SQL-queries and also generates a causal graph using the output from forensicDB. The user can observe malicious behaviors from different layers in a unified causal graph, and refine queries for the further investigation.

Attack Description: Suppose John carries an Android smart phone, and falls victim to a social engineering malware download attack by clicking on a link in an advertisement page. The malware, `AVPass` [14], silently installed in John's device. The malware deletes an icon and a widget preview to hide from the user, then steals sensitive information such as contacts, SMS messages from the victim device. Finally, the malware stores sensitive data into the local SQLite database for exfiltration.

Forensic Analysis: John accidentally detects that a suspicious process, `com.lge.clock` with pid 3052, is running in the background. He wants to identify what this process has done in his device. However, the malware's activities happened a while ago, and the inspection of the malware process or the device states does not provide a clear evidence of the attack. He then tries to reconstruct the process behavior using our technique. DroidForensics successfully captured the behaviors of the malware in three layers, Android API, binder, and system calls. John composes the first query to find out the events invoked by the process 3052:

```
$ SELECT * FROM SYSCALL,BINDER,API WHERE pid=3052;
```

DroidForensics's pre-processor converts the user-query into SQL queries. John's first query is converted to SQL union query to retrieve the output from multiple sources. Our post-processor generates a causal graph from the output of the query. The blue dotted box in Figure 2 shows an (simplified) output from John's first query. The graph shows that the suspicious process read four files through system calls, namely `com.lge.clock.xml, config.txt, res.db-journal` and `res.db`. From the read events to `res.db-journal` and `res.db`, John understands that process 3052 accesses a local



Query #1: SELECT * FROM SYSCALL,BINDER,API WHERE pid=3052;
3052
com.lge.clock
Binder
STime: 2016-09-08 17:51:06.21
ETime: 2016-09-08 17:51:06:21
Binder
STime:2016-09-08 17:53:07.13
ETime:2016-09-08 17:53:08:18
Binder
STime:2016-09-08 18:04:51.76
ETime:2016-09-08 18:07:56:45
SYS:read
/data/user/0/com.lge.clock/shared_prefs/com.lge.clock.xml
SYS:read
/data/user/0/com.lge.clock/files/app04/config.txt
SYS:openat, pread, pwrite
/data/user/0/com.lge.clock/files/app03/res.db-journal
SQLite: CREATETABLE if not exists(t_up_list, t_contacts, t_sms, t_calllog, t_record_call, t_record_bg, t_location, t_applist, t_album)
SQLite: UPDATE t_up_list SET state=0
SQLite: INSERT INTO t_contacts VALUES (đ.)
SQLite: INSERT INTO t_sms VALUES (đ)
SQLite: ...
SYS:pread, pwrite
/data/user/0/com.lge.clock/files/app03/res.db
2515
droid.launcher3
Launcher-loader
SQLite: DELETE FROM shortcut_and_widget_previews WHERE packageName = com.lge.clock AND profileId = 0
SQLite:SELECT profileId, packageName, lastUpdated, version FROM shortcut_and_widget_previews
/data/data/com.android.launcher3/databases/widgetpreviews.db
SQLite: DELETE FROM icons WHERE componentName LIKE com.lge.clock AND profileId = 0
/data/data/com.android.launcher3/databases/app_icons.db
Query #4: SELECT * FROM SYSCALL,BINDER,API WHERE pid=2515 and STime >= `2016 -09-08 17:51:06.21' and ETime <= `2016 -09-08 17:51:06:21';
2179
com.android.phone
SYS:open, pread
/data/user/0/com.android.providers.telephony/databases/mmssms.db-journal
SQLite: DELETE FROM directories WHERE packageName=com.lge.clock
SQLite:SELECT _id, address, body, date, type FROM sms_restricted ORDER BY date DESC
/data/data/com.android.providers.telephony/databases/mmssms.db
Query #3: SELECT * FROM SYSCALL,BINDER,API WHERE pid=2179 and STime >= \\ `2016 -09-08 17:53:07.13' and ETime <= `2016 -09-08 17:53:08:18';
1913
com.android.provider
SQLite: SELECT _id, number, duration, date, type FROM calls WHERE (((type != 4)) AND ((phone_account_hidden = 0))) ORDER BY date DESC
/data/data/com.android.providers.contacts/databases/contacts2.db
Query #2: SELECT * FROM SYSCALL,BINDER,API WHERE pid=1913 and STime >= \\ `2016 -09-08 18:04:51.76' and ETime <= `2016-09-08 18:07:56:45';

Figure 2: Generated causal graph from the users queries.

database, but he wants more specific information such as an exact query that 3052 used. Fortunately, DroidForensics's API logger captures SQLite API invoked by process 3052 and shows the query as well as the target database in the graph. There were multiple SQLite queries from the malicious process (bottom right corner in Figure 2). It creates table, updates table state, and inserts data into various tables.

The graph also shows multiple binder transactions to three different servers, pid 2515, 2179 and 1913. Now John wants to further understand what have happened in those servers. Each binder event has a start time (STime) and an end time (ETime) of each binder transaction. They show the time stamps when the binder sends a request to the server process and receives a reply. For example, process 3052 invoked the binder call to the server 2151, and the binder driver sends this request to process 2151 at 2016-09-08 17:51:06.21, and the binder driver received the replay from 2151 at 2016-09-08 17:51:06:21. We assume that all server behaviors between two time stamps are causally related to the client. We discuss details of binder events in the section 3.2. John composes following three queries to further investigate behaviors from the server processes: `$ SELECT * FROM SYSCALL,BINDER,API`

```
WHERE pid=1913 and STime >=
'2016-09-08 18:04:51.76' and ETime <= '2016-09-08 18:07:56:45';
$ SELECT * FROM SYSCALL,BINDER,API WHERE pid=2179 and STime >=
'2016-09-08 17:53:07.13' and ETime <= '2016-09-08 17:53:08:18';
$ SELECT * FROM SYSCALL,BINDER,API WHERE pid=2515 and STime >=
'2016-09-08 17:51:06.21' and ETime <= '2016-09-08 17:51:06:21';
```

Figure 2 shows a generated graph from the queries. The output from each query is merged into the previous graph so that John can see the full graph. It shows that process 1913 retrieves contact lists, 2179 reads SMS messages from the phone, and 2515 deletes the malware icon and widget preview to hide itself from the user. Now John fully understands the attack flow. The malware steals private information and stores it into a local database *(com.lge.clock \files\app93\resdb)*, and also the malware deletes the icon and preview.

Existing Techniques: Traditional system call logging [11, 12] does not show interactions between the malware process and service providers through binder protocols. For example, John only knows that the malware accesses the local database file, but system call logs do not show binder transactions or SQLite queries. Therefore, he will miss most attack behaviors.

CopperDroid [63] developed a technique to capture binder IPC from `ioctl` system calls, but it may hurt the runtime performance due to heavy logging (it pulls out the user memory used by `ioctl`) and analysis requirements. More importantly, it cannot capture the high-level API behaviors such as SQLite queries. Accessing SQLite does not always invoke system calls. Because it is common for SQLite to cache a whole table into memory, and accessing memory-loaded table does not need a system call. For example, process 1913 and 2515 in Figure 2 do not show any system call because target tables are already loaded in the memory. System call logs will completely miss those behaviors.

DroidScope [66] is a malware analysis engine that provides a unified view of malware behaviors. Similar to our approach DroidScope monitors multiple aspects of malware executions. However, DroidScope was designed for the Dalvik virtual machine. Dalvik environment is not available anymore on Android-6.0 and newer versions, so it makes DroidScope's analysis engine for Dalvik bytecode infeasible. Furthermore, both DroidScope and CopperDroid require emulated environments (e.g., QEMU) to monitor and analyze the process execution that generally incurs high overhead.

Taint tracking techniques [31, 25, 65, 62] might be able to detect what information got stolen, but cannot show how the attack unfolded. Additionally, taint tracking generally requires instruction-level monitoring to propagate taint tags that causes high overhead.

3. SYSTEM DETAILS

Before we introduce the details of our implementation, we will briefly discuss an overview of Android framework. Android is a Linux-based Operating System for mobile devices and tablet computers. Even though Linux kernel is the core part of Android, there are important differences in application execution models.

First, Android does not allow to use traditional System V based IPC or RPC protocols, but Android applications need to use binder, a custom implementation of IPC/RPC protocol. Binder communication is an important source to capture causal relations between processes.

Second, Android provides various Android-specific APIs such as framework APIs, system APIs, and resource APIs. The API usage often represents high-level program semantics.

Most Android applications are written in Java, but Android also allows developers to write code in native components (C or C++) to enhance the performance. Native components can invoke methods in Java libraries and also directly call lower-level instructions such as system calls.

3.1 Android API Logging

API logger captures application's activities at the API level to reason about how it interacts with the Android runtime framework. It can also capture an interaction between native components and Android APIs. We directly instrument Android source code to capture important APIs with their arguments and return values. In particular, we identify the set of Android APIs that potentially induce causal relations with system objects such as the device resources or private data. We instrument 21 Android APIs and they mainly fall into three categories:

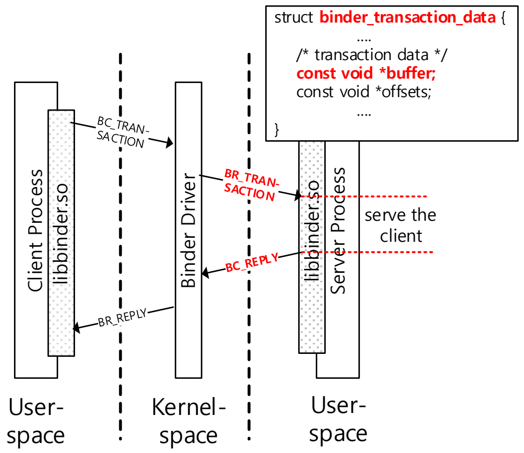
- Framwork APIs: We capture APIs that can handle Android framework resources. For instance, SMSManager APIs can send and receive SMS messages, TelephonyManager APIs can call or receive calls and also get device's IMEI number. PackageManager APIs allow to install or uninstall APK packages and also scan installed application lists.
- System APIs: We monitor APIs that can access camera, GPS, and microphone defined in Camera, Location and MediaRecorder classes, respectively.
- Resource APIs: We log APIs that can access device contents such as a local storage or a database. For instance, we capture SQLite database APIs and content provider APIs.

Our instrumented code snippets collect API information and send it to system call logger where we assign each event a global timestamps to show the happens-before relations between different logs. We store API information into heap memory, and we use `openat()` system call to deliver the data to system call layer. `openat` has three arguments, `(int fd, char* path, int oflag)` and we use `fd` as an indicator of log type. For example we use -255 for API and -256 for Binder. `path` points to the process heap memory where we store API information. Note that we can send an arbitrary length of data through the memory. `openat` with a negative `fd` simply returns an error from the kernel and does not cause any side-effect only except Linux errno [9]. When a system call fails, the kernel sets an errno to indicate a reason of the error. We save the current errno before `openat` and restore it after the system call to avoid a side effect from errno.

Alternative approach: We studied an alternative approach that can reduce the manual instrumentation. The idea started from an observation that most API calls from Android applications invoke `DoCall()`, `Invoke()` or `Execute()` methods defined in Android runtime class. Then they search the destination API address and jump to the target API. If the call arrives at `DoCall()`, `Invoke()` or `Execute()`, the target API name and their arguments can be retrieved from `ShadowFrame` data structure. Our idea is to instrument only those three methods to capture API calls and their arguments. We can monitor all API calls go through those methods and we can detect APIs we are interested (e.g., SQLite query) by simple string comparison.

However, it has a limitation. We cannot guarantee that all APIs are going through runtime methods. For instance, if Android compiler (`dex2oat)` optimization applies method inlining or direct offset calling, the application can directly jump to the target API without passing through aforementioned runtime methods. Native components are another problem. It can directly call Android APIs and runtime methods cannot observe them either. This approach has advantage as the user can easily configure API list to monitor and minimize Android modification, but we decide not to use it due to above limitations.

Another approach we considered is APK rewriting. It can directly instrument arbitrary codes in APK file, but this approach has limitations. APK rewriting or static instrumentation is known to be vulnerable to the code obfuscation, and it cannot instrument native code. Furthermore, Android applications can load an additional code at runtime, called dynamic loading code. Android attacks techniques often use dynamic loading code [54, 68, 32, 57] to avoid offline analysis systems, but APK rewriting technique cannot handle dynamically loaded codes.



```
struct binder_transaction_data {
    ....
    /* transaction data */
    const void *buffer;
    const void *offsets;
    ....
}
```

Figure 3: An overview of `binder` protocol.

3.2 Binder Logging

Another challenge to build an effective forensic analysis system for Android is its unique Inter-Process Communication (IPC) mechanism. Android applications are not allowed to use traditional System V based IPC or RPC protocols, but required to use Android `binder`, a custom implementation of the OpenBinder protocol [5]. Android applications use binder protocols to invoke methods of remote objects (e.g., services or activities) to interact with other applications. For instance, in order to send SMS message, Android applications need to invoke remote procedure, `sendTextMessage` provided by `com.android.sms` process (i.e., SMSManager). Similarly, Android applications use `binder` to access photos, contacts, map or other data stored in Android's main storage. In fact, all 31 Android malwares we have inspected use binder calls to steal information or to send unauthorized text message. The user application also can be a service provider. For instance, *Facebook* and *Twitter* provide sign-in services that enable people to log into the app with *Facebook* or *Twitter* accounts.

Consequently, IPCs or RPCs are important sources for forensic analysis but existing Linux-based logging techniques cannot effectively capture them. We need to understand semantics of binder protocol (e.g., client's and server's process ids, invoked remote method and a data object that is transferred between the client and the server) and capture them.

Figure 3 shows a simplified data flow in the binder protocol. To provide a service to other processes, the server first registers a service into `ServiceManager`, a special binder object that is used as a registry and lookup service for other binder objects. Once the service is registered, client processes can find and interact with it through binder protocol. The client (or caller) process X first interacts with `ServiceManager` to find the remote method name and invokes the remote method. The binder protocol sends `BC_TRANSACTION` message to Binder driver. It is delivered to Binder driver with multiple `ioctl` system calls. Then the Binder driver lookups a server process Y who can provide the service to the client X, and sends `BR_TRANSACTION` message to process Y. When the process Y finishes, it sends `BC_REPLY` message to the Binder driver and the driver forwards it to process X via `BR_REPLY`.

We log `BR_TRANSACTION` and `BC_REPLY` messages along with the information of the client (process X) and the server (process Y). We assume that all server's behaviors between `BR_TRANSACTION` and `BC_REPLY` are causally related to the client process. If the server concurrently receives requests from multiple clients, our conservative assumption may introduce false positives, but we will not miss any information. In practice, we do not observe any false dependences in our experiments.

In some cases, `BR_TRANSACTION` or `BC_REPLY` contains a message shared between the client and server that can be informative for the forensic analysis (e.g., SMS message, a recipient's number, IMEI). Figure 3 also presents a data structure that `BR_TRANSACTION` and `BC_REPLY` use. "void *buffer" contains a shared memory address that can be accessed by both the client and server. We log the first 128 bytes of the buffer if it can be a useful information for forensic analysis. We log the buffer that goes to SMSManager or sent from TelephonyManager because they possibly contain outgoing SMS message or devices's IMEI number. Note that logging the first 128 bytes is enough in our evaluation scenarios and the length of data to be logged can be easily tunned to meet one's demand of security level.

Alternative Approaches: In API-layer, it is possible to capture `intent` calls which initiate binder protocols. However it has following limitations. `Intent` declares a recipient by an action string or a component name. At the run-time, we can specify the recipient process, but that information might not be available in post-mortem forensic analysis. Note that arbitrary applications can register the service, and service name alone is not enough information to understand the behaviors. Furthermore, native component can use binder protocol through binder library without using `intent` API.

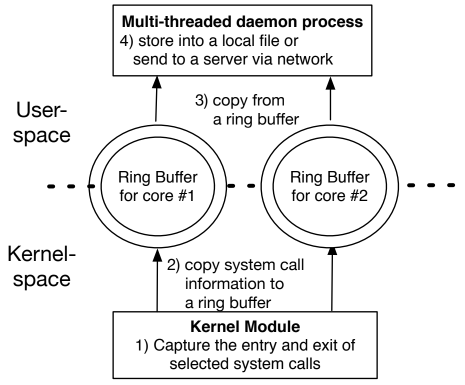

Multi-threaded daemon process
4) store into a local file or send to a server via network
3) copy from a ring buffer
User-space
Ring Buffer for core #1
Ring Buffer for core #2
Kernel-space
2) copy system call information to a ring buffer
Kernel Module
1) Capture the entry and exit of selected system calls

Figure 4: System call logging overview.

CopperDroid [63] proposed a technique that analyzes the semantics of binder via `ioctl` system call. However, CopperDroid is build on top of QEMU and it requires the out-of-the-box analysis to understand a payload of each `ioctl` calls. Unfortunately, their technique could be too heavy to be implemented in a real device.

Accordingly, we decide to monitor IPC/RPC in the binder library (`libbinder.so`) where we can collect all information.

3.3 System call Logging

In the previous sections, we discussed Android API and `binder` logging techniques to monitor high-level application behaviors and interactions between applications or services. However, they are not enough to fully capture application behaviors. For instance, Android applications can contain native components written in C/C++. Native components can directly invoke lower-level instruction such as system calls that API logger can not observe. Malicious apps frequently hide their activities in native code [61, 17, 56, 52] to evade the Java-code analysis techniques [21, 37, 47, 50, 64, 69]. Recent study [17] shows that 37% of Android applications (446k out of 1.2 million Android apps) potentially use native components. Therefore we develop system call logger that can capture system calls from native components.

System call logging is a popular technique in traditional desktop or server forensics. For instance, Linux Audit [11] is a default package in most Linux distributions, and DTrace [8] is shipped with FreeBSD operating system. Linux Audit is also available in Android [4], however, it causes too much run-time overhead (up to 38%) and space consumption to use in practice (e.g., always-on forensic logging) in resource-constrained mobile devices. To address this problem, we have developed a light-weight system call logging module for Android. We borrow an idea from state-of-the-art Linux system call logging techniques, ProTracer [49] and Sysdig [12]. Thankfully, Sysdig is an open-source project under GNU General Public Licenses (free to share and change the code), and we reuse part of their code to build our logger for Android. Our system call logger causes only 1.99% $\sim$ 4.56% run-time overhead in Nexus 6 smartphone.

Similar to ProTracer [49] and Sysdig [12], our system consists of two parts, a kernel module and a user-space daemon process. Figure 4 shows the architecture of our system. The kernel module leverages `tracepoints` [15] to capture the entry and exit of each system call (e.g,. `sysenter` and

	Benchmark	Runtime Overhead			Space Consumption		
		Linux Audit [4]	DroidForensics without comp.	DroidForensics with comp.	Linux Audit [4]	DroidForensics without comp.	DroidForensics with comp.
Android-6.0.1_r42	PCMark-work	15.31%	0.26%	1.99%	166MB	110MB	16MB
	TabletMark-web/email	22.61%	1.44%	3.57%	590MB	402MB	61MB
	TabletMark-photo/video	37.38%	1.41%	2.12%	612MB	509MB	64MB
	3DMark	18.98%	3.12%	3.75%	56MB	44MB	7.3MB
Android-5.1.0_r3	PCMark-work	18.34%	1.84%	2.32%	150MB	101MB	14MB
	TabletMark-web/email	24.19%	3.77%	4.14%	612MB	421MB	67MB
	TabletMark-photo/video	38.73%	2.21%	3.41%	661MB	469MB	69MB
	3DMark	19.15%	3.19%	4.05%	59MB	46MB	8.1MB
Average		24.34%	2.16%	3.17%	363MB	263MB	38.3MB

Table 1: Overhead of system call logging: Linux Audit and DroidForensics' system call logger.

`sysexit`). At runtime, the kernel module collects system call information and stores it into a kernel ring buffer. To avoid race conditions, we generate a separate ring buffer for each CPU cores. The user-space daemon reads from ring buffers, compresses them and sends to the local file or an outer server through the network. We only capture forensic-related system calls such as calls that affect other processes (e.g., fork, kill) or other system objects (e.g., read, write, recv, send). We record 52 out of 328 system calls. Our selection of system calls are similar to previous Linux-based forensic logging techniques [49, 45]. API and binder logs delivered via `openat()` system call are handled here. They are further processed to retrieve API and binder information in the server when we inject logs into a relational database.

If the kernel module generates data faster than the user-level daemon can consume, we might lose data. To prevent the loss of information in the ring buffer, we make sure that the ring buffer has enough space to store the current system call. When we intercept the entry of a system call, we check the remaining space of the ring buffer. If the ring buffer is full, the kernel module suspends the execution of the system call and allows the user-space daemon to consume the ring buffer. We allocate each ring buffer with 16 MBytes, for example, Nexus 6 needs four ring buffers and Nexus 9 has two, one for each CPU-core. In our experiments with I/O intensive workloads and Android benchmark applications, ring buffers hardly become full. The user-space daemon is multi-threaded process that efficiently consumes ring buffers. It compresses the data using `zlib` before writing to the storage, then periodically sends the compressed log to a predefined server via *ssh* protocol. In our current setup, we send the log every 10 minutes if wifi is available.

Table 1 shows a runtime and space overhead of Linux audit and DroidForensics's system call logger. In this experiments, we use popular Android benchmark applications, namely PCMark [10], TabletMark [13], and 3DMark [1]. These benchmarks have been used by IT magazines and developers to compare performances between different devices and Android versions.

PCMark-work benchmark simulates basic office work tasks such as web-browsing, video editing, writing, photo editing and parsing data. PCMark-storage accesses various types of files located in internal storage, external storage and local database. TabletMark simulates web-browsing, email accessing, and watching and editing photos and videos. 3DMark uses OpenGL ES benchmarks to measure CPU and GPU performance. Each benchmark execution takes 20 to 60 min.

The third column shows runtime overhead of Linux audit system. It goes up to 38%. The forth and fifth columns show overhead from our technique while we turn-off the compression, and turn-on the compression, respectively. DroidForensics with the compression is a little slower than without the compression, but it still shows much lower overhead (3.17% on average and 4.14% in the worst case) than Linux audit. The last three columns show space consumption of system call logs generated by Audit, DroidForensics without the compression, and DroidForensics with the compression, respectively. DroidForensics with the compression shows much smaller log size than other two cases (9.5 times smaller than Audit and 6.9 times smaller than without the compression). We can observe similar results from Nexus 9 tablet (details are elided). We argue that 3% to 4% of runtime overhead from DroidForensics with the compression is acceptable in practice. All other experiments in this paper are conducted with the compression turned on.

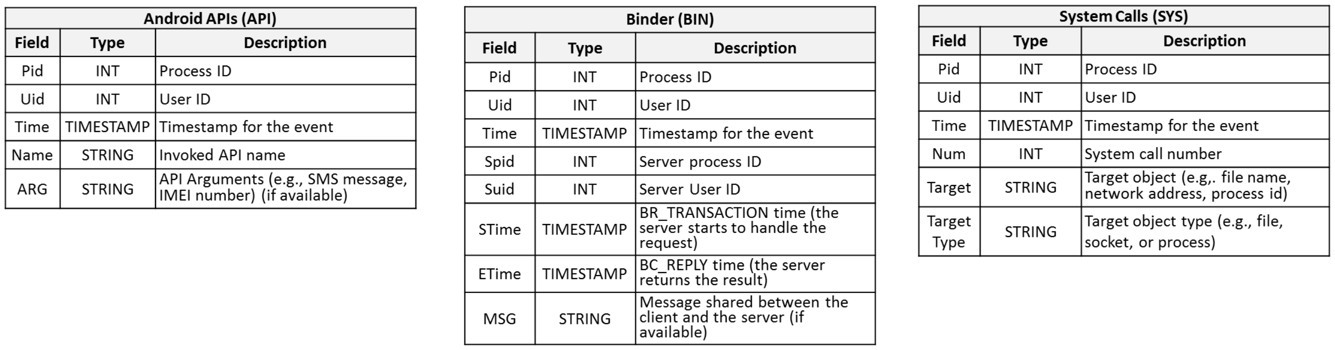
3.4 User Interface

DroidForensics uses MySQL database as a back-end storage and we provide a declarative interface based on SQL-like language to the user. Various levels of logs are encoded into SQL database and the user can compose one or more queries on relations to reconstruct Android attacks. We defined a set of relations that describe aspects of Android API, binder and system calls. Figure 5 presents the detailed schemas. *Pid* and *Uid* shows the process id and the user id of the process, and *Time* means the timestamp of the event. Binder event shows a causal relation between the client and the server processes. It has *Spid* and *Suid* fields for the server's process id and user id. *Stime* and *Etime* are timestamps for `BR_TRANSACTION` and `BC_REPLY` event respectively. We assume that the server's events happens between *Stime* and *Etime* are causally related to the client process (details are discussed in the section 3.2). If additional information is available (e.g., SMS body or a recipient's address), we store them in *MSG* field.

API event presents the interaction between Android app and underlying Android Framework. We use *Name* and *ARG* fields to represent the API name and their important arguments (e.g., SQLite query).

System call event shows the interaction between the process and Android kernel. We use *Num* for system call number and *Target* for the target system object. We additionally define *TargetType* field to show the type of the object such as file, socket, or process. It is useful to understand low-level process behaviors such as process X reads file A, process Y send a packet to IP 1.2.3.4 or process X kills process Y.

Figure 6 shows how DroidForensics' user interface works. DroidForensics accepts SQL-like query from the user and our pre-processor converts it into SQL queries. Our post-processor generates a causal graph from the query results



Android APIs (API)		
Field	**Type**	**Description**
Pid	INT	Process ID
Uid	INT	User ID
Time	TIMESTAMP	Timestamp for the event
Name	STRING	Invoked API name
ARG	STRING	API Arguments (e.g., SMS message, IMEI number) (if available)

Binder (BIN)		
Field	**Type**	**Description**
Pid	INT	Process ID
Uid	INT	User ID
Time	TIMESTAMP	Timestamp for the event
Spid	INT	Server process ID
Suid	INT	Server User ID
STime	TIMESTAMP	BR_TRANSACTION time (the server starts to handle the request)
ETime	TIMESTAMP	BC_REPLY time (the server returns the result)
MSG	STRING	Message shared between the client and the server (if available)

System Calls (SYS)		
Field	**Type**	**Description**
Pid	INT	Process ID
Uid	INT	User ID
Time	TIMESTAMP	Timestamp for the event
Num	INT	System call number
Target	STRING	Target object (e.g,. file name, network address, process id)
Target Type	STRING	Target object type (e.g., file, socket, or process)

Figure 5: Database schema for log properties

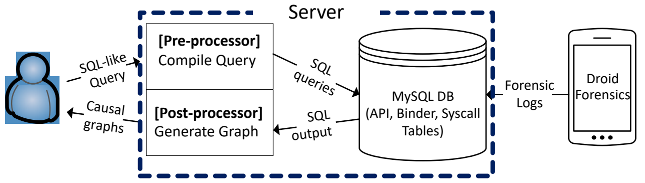


Figure 6: The user inferface of DroidForensics.

and the user can iteratively compose queries based on generated causal graphs. Our post-processor can merge the new results into the previous graph so that the user can inspect the attack with a unified graph. We demonstrated how the user can utilize DroidForensics to reconstruct the Android attack in section 2.

4. EVALUATION

To establish the practicality of DroidForensics, we measure the runtime and space overhead it incurs for forensic logging. We also evaluate DroidForensics on 30 real-world and one crafted Android malwares, and we can easily reconstruct their behaviors. Finally, we use Android Compatibility Test Suite (CTS) on our modified Android framework and the results show that our modification does not affect the compatibility. The experiments were performed on two devices; Nexus 6 with Snapdragon 805 CPU (Quad-core 2.7 GHz, 32-bit) and 3GByte RAM, and Nexus 9 with Tegra K1 CPU (Dual-core 2300 MHz, 64-bit) and 2GByte RAM. We use Android-6.0.1_r42 for Nexus 6 and 6.0.1_r46 for Nexus 9.

4.1 Logging Overhead

Runtime Overhead: In this experiment, we examine the runtime overhead incurred by DroidForensics. Overhead was measured by widely used Android benchmark programs including PCMark for Android [10], 3DMark [1], Antutu [3], DiscoMark [7], and TabletMark [13].

PCMark simulates basic office work tasks such as web-browsing, video editing, writing, photo editing and parsing data. 3DMark uses OpenGL ES benchmarks to measure CPU and GPU performance. Antutu measures performance of the device in multiple aspects. For instance, 3D-test evaluates the performance of 3D rendering, UX examines the performance of multi-tasking and application executions, CPU and RAM tests use CPU-intensive and memory workloads to measure the device performance. DiscoMark developed at ETH and it opens and closes installed applications multiple times to measure the application's launch time. TabletMark simulates web-browsing, email accessing, and watching and editing photos and videos. Each benchmark execution takes 10 to 60 minutes and we run each benchmark 5 times and report the average.

Figure 7 shows the runtime overhead. The graph shows separate results from each test and overall bar represents a final score reported from each benchmark suite. Web-browsing benchmark in PCMark shows the highest overhead, which is 6.16% slower than original Android without DroidForensics. Overall results show that DroidForensics only causes negligible runtime overhead (2.58% on average).

Space Overhead: Figure 8 presents the forensic log size changes over 24 hours. In this experiment, we install DroidForensics on Nexus 6 and Nexus 9 devices and ask graduate students to use them for 24 hours. Both Nexus 6 and Nexus 9 cannot use SMS, phone or LTE because Nexus 9 is a wifi-only model and we removed a sim card from Nexus 6 for this experiment. The users stayed wifi-available places such as home and office over 90% of time during this experiments. We installed Chrome web-borwser, Gmail app, and a few other utility and game applications before the experiment. We also implemented a simple script app that records the current size of our forensic log when the user clicks a button. We ask each user to click the button every hour until he goes to bed at midnight. Next morning at 8am, the user clicked the button again to get the final log size. Because we do not have log size data between 12am to 8am, we present an average rate of log increase during that period. The results show that Nexus 6 log grows at 4.75MB/hour in a daytime and 3.45MB/hour at night and the log in Nexus 9 grows at an rate of 8.9MB/hour in a daytime and 3.2MB/hour at night. Note that DroidForensics can transfer the log to the server (if wifi is available), but we did not transfer any log in this experiment. If the user can use wifi every one hour, average space consumption of DroidForensics will be 5.6MB on average and 16.21MB in the worst case. If the user can send the log every 10 minutes, the device requires only less than 3MB additional storage for the forensic log. The majority of the logs (97.9%) are system call logs, 1.6% of the logs are generated from Binder, and 0.5% of are from API.

4.2 Effectiveness

We collect 30 real-world Android malware samples and evaluate DroidForensics on them. We install each malware package to Android-6.0.1_r42 on Nexus 6 device. Then we execute a malware while DroidForensics collects forensic logs. After the execution, we use SQL-like queries to reconstruct

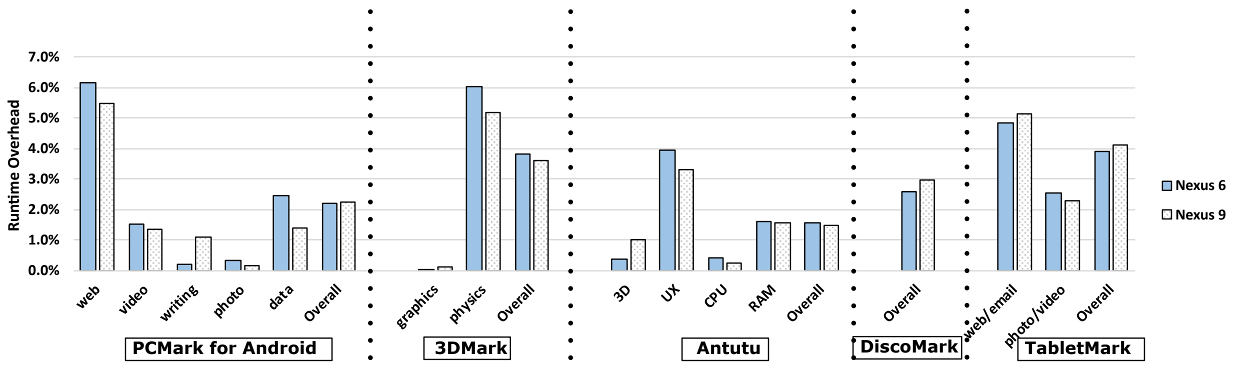


Figure 7: The runtime overhead of DroidForensics

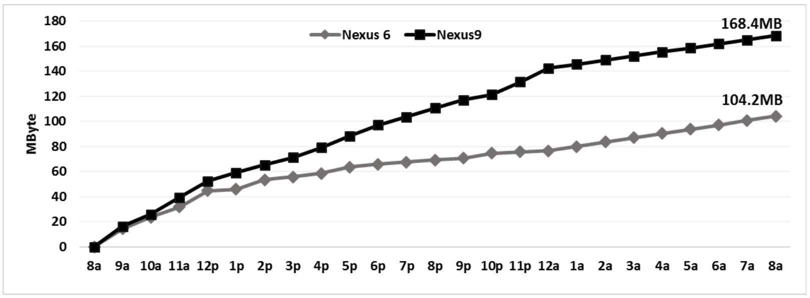


Figure 8: Accumulated log size from the one-day execution.

behaviors of the malware. We start from a query, `SELECT * from API,BIN,SYS where pid=malware_pid;`, then we compose additional queries based on the output of the previous query until we find all relations from the malware process. To evaluate the effectiveness of DroidForensics, another graduate student studied each malware with manual approaches such as understanding various analysis reports on the web and inspecting the malwares using APK analyzer, ADB and log-cat. We compare the analysis outputs' from DroidForensics and the manual inspection.

Table 2 shows the result. The first and second columns present malware family and package names. The last column presents the comparison between DroidForensics and the manual inspection. "Full" represents DroidForensics can discover all attack behaviors. "Partial" means DroidForensics misses part of attack behaviors. It happens from two ransomware samples, namely LockScreen and FBI.Locker. Both show similar behaviors. They first lock the device, then send a SMS message in the background to indicate a succesful infection, and finally display a ransom message (html page) via Android webview. DroidForensics successfully captures SMS sending and a ransom message events, but we failed to capture the behaviors for the device locking. To lock the device, they manipulate event handlers to hinder the user from doing any activity on the device. For instance, a malware overrides event handlers for all actions to home button, back button, and power button to completely ignore user actions. Some of touch actions are ignored as well. Our current implementation does not capture function overriding events. However, we believe this is not a fundamental limitation of our approach and we plan to support them in near future. Specifically, if the application overrides any handler, we will capture the overriding event and records the name of old and new event handlers.

The columns from third to fifth represent the type of logs we needed to understand the attack. We mark (✓) if a log is needed in attack reconstruction. For example, `com.android.mms20` malware deletes its launcher icon, then steals IMEI, IMSI and GPS location. It also collects a list of installed applications. After that, the malware attempts to send SMS message to a hard-coded number. We can reconstruct all those behaviors from API and binder logs and we did not need system call logs to understand the attack. Apparently, the generated graph from our queries (e.g., `SELECT * from API,SYS,BIN where pid=malware;`) contains system call edges, but we can fully reconstruct the attack without them. It can happen mainly because of the following reasons. First, system calls may not be involved in the attack behaviors (e.g., SQLite query to memory-loaded tables). Second, in some cases, we can detect system call events that contribute the attack, but it can be more clearly explained by the higher-level logs. For example, `mms20` deletes its launcher icon to hide from the user. We captured two different events that show deleting application icon event: 1) an API log shows a SQLite query, `DELETE FROM icons WHERE componentName LIKE com.android.mms20;`, 2) a system call log, `pwrite app_icons.db` shows a low-level action that modified app_icon.db file. Obviously, the API log is much clearer and easier to understand. So we do not mark the system call column for `mms20`.

The last row (`com.nativeCode`) is a crafted malware that uses native components to access files, establish a connection to C&C server, and send information to the server. As expected, API and Binder logs are not useful, but we can reconstruct all behaviors from system calls. The results from this experiment show that all three loggers are essential to reconstruct attacks. All the samples (except the crafted one) we have used in this study are publicly available on Contagio Mobile [6].

Malware family	Package Name	Is this log needed?			Attack Reconstruction?
		API	Binder	Syscall	
Worm.Gazon	app.rewards.amazon.com.amazonrewards	✓	✓	✓	Full
Android. Smsstealer	com.dsifakf.aoakmnq	✓	✓	✓	Full
Android.Windseeker	com.example.windseeker	✓	✓	✓	Full
Android.Tetus	com.droidmojo.celebstalker	✓	✓	✓	Full
AVPass	com.lge.clock	✓	✓	✓	Full
unknown	com.android.mms20	✓	✓		Full
BadNews B	ru.blogspot.playsib.savageknife	✓	✓	✓	Full
CutTheRope	com.atools.cuttherope	✓	✓	✓	Full
unknown	com.android.systemsecurity	✓	✓	✓	Full
LockScreen	qqkj.qqmagic	✓	✓		Partial
AngryBird	com.elite	✓	✓	✓	Full
HongTouTou	com.bytedroid.liveprints	✓	✓	✓	Full
AndroidArmour	com.armorforandroid.security	✓	✓		Full
unknown	com.andnottech.morningandnight	✓	✓		Full
Android.FakeToken	token.generator	✓	✓	✓	Full
FantaSDK	com.fanta.services	✓	✓		Full
Pincer.A	com.security.cert	✓	✓	✓	Full
unknown	com.example.android.service	✓	✓	✓	Full
Android.Titan	com.Titanium.Gloves	✓	✓	✓	Full
Qicsomos A	org.projectvoodoo.simplecarrieriqdetector	✓	✓		Full
Android.Exprespam	frhfsd.siksdk.ujdsfjkfsd	✓	✓		Full
FakeInstragram	com.software.application	✓	✓	✓	Full
unknown	il.co.egv	✓	✓	✓	Full
HGSpy	com.exp.tele	✓	✓		Full
FBI.Locker	com.android.locker	✓	✓		Partial
Android.Fakeplay	com.googleprojects.mmsp	✓	✓	✓	Full
Android.Fakenotify B	wap.syst	✓	✓		Full
Android.Fakeinstaller	imauyfxuhxd.qhlsrdb	✓	✓		Full
Android.Fakedaum	com.tmvlove	✓	✓	✓	Full
Android.Fakebank B	com.example.adt	✓	✓	✓	Full
crafted	com.nativeCode			✓	Full

Table 2: The reconstruction of Android attacks with DroidForensics. Mark (✓) means that the log from that layer is needed to reconstruct an attack. "Full" means DroidForensics discover full attack behaviors, and "Partial" means DroidForensics misses part of malicious behaviors.

Test pakages	Nexus 6			Nexus 9		
	# of fails		# of tests	# of fails		# of tests
	Ori.	Our		Ori.	Our	
Access	7	7	316	7	7	316
Device	4	4	53	4	4	53
Core	0	0	2,917	0	0	2,914
Graphic	0	0	1,393	0	0	1,390
Native	0	0	1,060	0	0	1,060
Media	0	0	1,776	0	0	1,776
Contents	0	0	619	0	0	619
Other	0	0	1,127	0	0	1,127
Total	11	11	9,261	11	11	9,254

Table 3: Compatibility Tests. "Ori." shows a number of failed tests with original Android and "Our" shows a number of failed tests with DroidForensics. Both failed the same set of test cases.

4.3 Compatibility

One may think that DroidForensics can cause compatibility issues because it requires a modification of Android framework and an additional kernel module. To identify this concern, we evaluate the compatibility of DroidForensics using Android Compatibility Test Suite (CTS) [2]. We use the *CTS-public-small* plan which contains around 9,200 test cases. The summarized results are in Table 3. In all tests, DroidForensics and original Android failed on the same set of test cases. We believe the failed cases are caused by device environments, for instance both Nexus 6 and 9 do not have external SD card and tests that try to access the external storage failed. The results show that DroidForensics maintains the same compatibility-level to compare with original Android.

5. DISCUSSION

In this section, we discuss limitations and future work of DroidForensics. First, a kernel-level attacks could disable DroidForensics. Although we periodically (e.g., every 10 minutes) transfer the log to the outer server, the attacker can tamper with logs remained in the device. We believe this is an on-going research area [51, 24] that is orthogonal to the main focus of DroidForensics.

Second, our prototype uses `openat()` system calls to transfer API and binder logs (see the section 3.1). If a malicious application invokes `openat()` to trick DroidForensics, it can introduce bogus causal relations (e.g., bogus binder edges in the output graph), and it will make the investigation difficult. However, it only introduces false positives but cannot hide true positives (i.e., malicious behaviors).

We also plan to mitigate this problem as following. We will add a user-defined system call with three arguments and use the first argument as a secret session keys between higher-level loggers and system call logger. We plan to build a simple module that randomly assigns the key at boot-up time to mitigate the vulnerability.

Third, our binder logger intercepts IPC/RPC in the native binder library, *libbinder.so.* Both Java and native codes use this library to invoke binder calls. However, native components can directly invoke `ioctl` system calls to send binder message to the binder driver in the kernel. We never observe that in practice, but it is theoretically possible. Our binder logger cannot capture them. To address this limitaion, we can port the binder logger to the binder driver in kernel-space, then we can capture all binder communications.

Finally, DroidForensics requires manual instrumentation to Android API functions. In the future, we plan to develop more automated techniques to determine instrumentation points including important call-back functions and handlers. For example, we can leverage DroidAPIMiner [16] to automatically identify instrumentation locations from important Android APIs, call-backs, and event handlers.

6. RELATED WORK

Forensic Logging: Tracking system-level dependence is a popular technique for attack analysis in desktop and server environments [43, 41, 18, 35, 44, 42, 53, 27]. They record system events (e.g., system calls) during the execution and interpret them to analyze causal dependences between system subjects (e.g., process) and system objects (e.g., network socket or file) to reconstruct an attack. Recently, BEEP [45], ProTracer [49] and WinLog [48] propose techniques that pro-actively analyze and instrument application binaries to improve an accuracy of attack reconstruction. They focus on logging system-level event in desktop or server systems, however, it is not effective in Android framework due to it's unique execution environment, Android Runtime (ART), and binder IPC protocol. DroidForensics enables logging in multi-layer to capture accurate information from different layers, and we provide easy-to-use user interface to query them.

LogGC [46] proposes a garbage collection techniques for forensic logs. It removes redundant or unnecessary events from the log (e.g., accessing temporary files). In the future, we plan to develop a similar technique for Android to fundamentally reduce the size of log.

Recently, Android attack reconstruction techniques have been proposed. CopperDroid [63] proposes system-call logging and analysis technique for Android attack reconstruction. CopperDroid is a VMI-based approach and it is built on top of QEMU [22]. It provide a smart way to analyze `ioctl` system call to understand semantics of binder protocol, but it requires buffer contents of each `ioctl` and it might causes too much runtime and space overhead for real devices. Furthermore, it could miss important events that can be observed only in higher-layer (e.g., API).

DroidScope [66] is a QEMU-based malware analysis engine that provides the unified view of hardware, kernel and Android virtual machine (Dalvik). Unfortunately, DroidScope's analysis engine for Dalvik bytecode is infeasible for recent Android ART environments. Furthermore, both DroidScope and CopperDroid were build on top of QEMU [22] emulator and it generally incurs high overhead. DroidForensics supports ART environments, and does not rely emulated environments but directly works on real devices.

Quire [30] monitors Android binder calls to detect confused deputy problem. It track privileges across inter-process boundaries. Grover et al. [38] propose an application-level technique to monitor user activities such as application install and removal, web browser history, calendar, call log or contact lists.

Android Taint Tracking: Dynamic taint tracking and information flow analysis techniques for Android [31, 25, 65, 62] have been proposed to detect information leak or privilege escalation attacks. Their approaches first assign tags to provenance sources (e.g., private data objects) and propagates the tags at each instruction through dependencies captured during the system execution. They can detect provenance tags that reaches a sink node (e.g., outgoing network socket, SMS message send) that indicate the leakage of private information. Taint tracking techniques usually require instruction-level monitoring that causes high run-time overhead and often requires emulator-based instrumentation platform such as QEMU [22]. Taint tracking only shows the flow of the data (what-provenance), but forensic analysis including DroidForensics captures both what- and how-provenance. Our system is designed for forensic logging, and comparing with taint tracking techniques, our solution directly works on real Android devices and has less runtime overhead.

Other Android Analysis Techniques: Static Analysis techniques [47, 50, 34, 25, 20] can be used to understand the behaviors of Android applications. They use APK or Java code analyzer to detect potentially malicious behaviors from Android source code. These static techniques are complementary to DroidForensics. For example, we can use static analysis as a hint and enhance runtime forensic logging for potentially malicious code regions.

Android memory forensics techniques [60, 59, 19] reconstruct the application or device states from a smartphone's memory image. Their goal is to recover the current (when the memory was dumped) state of the device to allow the user to acquire important evidences such as photo, application UIs, or authentication credentials. DroidForensics complements these techniques by logging runtime behaviors of application to reconstruct the execution. Zhang et al. [67] proposed a machine learning approach that analyzes network traffic on Android devices to detect stealthy Android malware activities.

Recording-and-replay based attack forensics are very useful because the user can replay the malicious execution as many time as he wants. Recently, record-and-replay techniques for Android applications have been studied in the software engineering community to aid in application debugging [36, 39, 40, 55]. RERAN [36] and Mosaic [39] use Android SDK's *getevent* tool to capture low-level event streams including graphical user interface (GUI) gestures (e.g., swipe, zoom, pinch, multi-touch) and sensor events. However, they are not able to replay inputs from other devices such as GPS, microphone or network. Furthermore, they are not able to record-and-replay sophisticated activities [33, 58, 23, 26], as they only record stream inputs. VALERA [40] statically instruments APK files to capture Android API calls. It leverages a bytecode rewriting tool to record and replay API calls. However, VALERA does not support native code execution [61, 17, 56, 52] and dynamic code loading [54, 68, 32, 57]. Mobiplay [55] is a client-server based recording and replay system. Android applications run on a server that emulates the exact same environment as the mobile phone, and the server transfers a GUI display to the mobile device that the user interacts with.

7. CONCLUSION

In this paper, we have presented DroidForensics, a multi-layer forensic logging technique for Android. DroidForensics captures important Android events from Android API, Binder and system calls layers. API logger collects information about Android API calls that contain high-level semantics of an application. Binder logger captures inter-process communications that represent causal relations between processes, and system call logger efficiently monitors low-level system events. We also develop an easy-to-use interface for Android attack investigation. The user can compose SQL-like queries to inspect an attack and DroidForensics provides causal graphs to the user. The user can iteratively refine queries based on previous results.

Our experiments have shown that DroidForensics has low runtime overhead (2.9% on avg.) and low space overhead (up to 168 MByte during 24 hours) on Nexus 6 and Nexus 9 devices. We evaluate DroidForensics with 30 real-world Android malwares and the results show that DroidForensics is effective in reconstruction of Android attacks. Our compatibility tests present that DroidForensics maintains the same level of compatibility as original Android.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. This research was, in part, supported by the United States Air Force and Defense Advanced Research Agency (DARPA) under Contract No. FA8650-15-C-7562 and funds from the University of Cincinnati CECH. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

8. REFERENCES

[1] 3dmark. https://www.futuremark.com/benchmarks/3dmark/android/.

[2] Android compatibility test suite (cts). https://source.android.com/compatibility/cts/.

[3] Antutu. http://www.antutu.com/en/index.shtml.

[4] Auditdandroid. https://github.com/nwhusted/AuditdAndroid.

[5] Binder ipc mechanism. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html.

[6] Contagio mobile. http://contagiominidump.blogspot.com.es/.

[7] Discomakr. https://play.google.com/store/apps/details?id=ch.ethz.disco.gino.androidbenchmarkaccessibilityrecorder&hl=en/.

[8] Dtrace. http://dtrace.org/blogs/.

[9] errno - number of last error. http://man7.org/linux/man-pages/man3/errno.3.html.

[10] Pcmark for android. https://www.futuremark.com/benchmarks/pcmark-android/.

[11] Redhat linux audit. https://people.redhat.com/sgrubb/audit/.

[12] Sysdig. http://www.sysdig.org/.

[13] Tabletmark. https://bapco.com/products/tabletmark/.

[14] Trojan:android/avpass.c. https://www.f-secure.com/v-descs/trojan_android_avpass_c.shtml.

[15] Using the linux kernel tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt/.

[16] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In SecureComm '13. 2013.

[17] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. d. Geus, C. Kruegel, and G. Vign. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In NDSS '16.

[18] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Trans. on Knowl. and Data Eng.*, 2002.

[19] D. Apostolopoulos, G. Marinakis, C. Ntantogian, and C. Xenakis. Discovering authentication credentials in volatile memory of android mobile devices. In *In Collaborative, Trusted and Privacy-Aware e/m-Services*, 2015.

[20] D. Arp, M. Spreitzenbarth, H. Malte, H. Gascon, and K. Rieck. Drebin : Effective and Explainable Detection of Android Malware in Your Pocket. In NDSS '14.

[21] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In PLDI '14.

[22] F. Bellard. Qemu, a fast and portable dynamic translator. In USENIX ATEC '05.

[23] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In S&P '15.

[24] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. *PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging*. In RAID '14.

[25] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In NDSS '15.

[26] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In Usenix Security '14.

[27] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In SSYM'04.

[28] CVE-2015-3864. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3864.

[29] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P.M. Chen. Eidetic Systems. In OSDI '14

[30] M. Dietz, A. Shu, and D. S. Wallach. Quire : Lightweight Provenance for Smart Phone Operating Systems. In Usenix Security '11.

[31] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. . G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In OSDI'10.

[32] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi. Grab 'n run: Secure and practical dynamic code loading for android applications. In ACSAC '15.

[33] E. Fernandes, Q. Chen, J. Paupore, G. J. Essl, A. Halderman, Z. M. Mao, and A. Prakash. Android ui deception revisited: Attacks and defenses. In FC '16.

[34] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In S&P '16.

[35] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In SOSP '05.

[36] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In ICSE '13.

[37] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In NDSS '12.

[38] J. Grover. Android forensics: Automated data collection and reporting from a mobile device. *Digit. Investig.*, 2013.

[39] M. Halpern, Y. Zhu, R. Peri, and V. J. Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In ISPASS '15.

[40] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In OOPSLA '15.

[41] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. Buchholz, and Y.-M. Wang. Provenance-aware tracing ofworm break-in and contaminations: A process coloring approach. In ICDCS '06.

[42] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek. Intrusion recovery using selective re-execution. In OSDI'10.

[43] S. T. King and P. M. Chen. Backtracking intrusions. In SOSP '03.

[44] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In NDSS '05.

[45] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In NDSS '13.

[46] K. H. Lee, X. Zhang, and D. Xu. Loggc: garbage collecting audit log. In CCS '13.

[47] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In CCS '12.

[48] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In ACSAC '15.

[49] S. Ma, X. Zhang, and D. Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In NDSS '16.

[50] C. Mann and A. Starostin. A framework for static detection of privacy leaks in android applications. In SAC '12.

[51] G. A. Marson and B. Poettering. Practical secure logging: Seekable sequential key generators. In ESORICS '13.

[52] C. Mulliner, W. Robertson, and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In AsiaCCS '14.

[53] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In NDSS '05.

[54] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In NDSS '14.

[55] Z. Qin, Y. Tang, E. Novak, and Q. Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In ICSE '16.

[56] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Trans. Info. For. Sec.*, 2014.

[57] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In NDSS '16.

[58] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In Usenix Security '15.

[59] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu. Guitar: Piecing together android app guis from memory images. In CCS '15.

[60] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. R. III. Screen after previous screens: Spatial-temporal recreation of android app displays from memory images. In Usenix Security '16.

[61] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In WiSec '14.

[62] M. Sun, T. Wei, and J. C.S.Lui. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In CCS '16.

[63] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In NDSS '15.

[64] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In CCS '14.

[65] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In S&P '15.

[66] L. K. Yan and H. Yin. DroidScope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In Usenix Security '12.

[67] H. Zhang, D. D. Yao, and N. Ramakrishnan. Causality-based sensemaking of network traffic for android application security. In AISec '16.

[68] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In CODASPY '15.

[69] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In NDSS '12.