Effective Security and Forensic Analysis for Mobile and IoT Environments

by

Xingzi Yuan

(Under the Direction of Kyu Hyung Lee)

Abstract

Security and forensic analysis is a well-established technique to reconstruct an attack. It provides insights from the symptom to the origin of the attack. It also helps to understand damage from attacks, recover the system and prevent similar attacks in the future. However, it is typically infeasible to directly apply existing security and forensic analysis tools to mobile or IoT environments. Mobile devices have different structure of execution environments from desktops or servers, and IoT devices comprise diverse hardware and software platforms and have restricted computing resources. Fast growing malicious content on mobile and IoT environments urges the need for effective security and forensic analysis on these platforms.

In this dissertation, we propose security and forensics analysis techniques for mobile and IoT environments. We make four contributions as follows. Our first contribution is DroidForensics, a multi-layer forensic logging technique for Android. It captures different levels of information from high-level application semantics to low-level system events, and inter-process communication via Android's binder protocol. We show that DroidForensics effectively and efficiently collects essential logs to reconstruct attacks from real-world malwares. Second, we design an Android helper application for PushAdMiner. It enables PushAdMiner to harvest Web Push Notifications (WPN) from Android Chromium, and to monitor their corresponding landing pages. We use PushAdMiner to automatically collect and analyze 21,541 WPN messages across thousands of different websites. Among these, PushAdMiner identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads, of which 51% are malicious. Third, we propose MQTTprov, a data provenance and forensic log collection technique for MQTT protocol. MQTTprov fills the gap of lacking in-depth study and solution for forensic logging in IoT environment. We show the MQTTprov's effectiveness of reconstructing real-world attacks. Fourth, we propose an attack model named ChatterHub, a novel approach accurately identifies smart-home devices' activities with only encrypted traffic in the home network. Using ChatterHub, an adversary can identify smart-home devices' specific activities without prior knowledge of the target home (e.g., list of deployed devices). We further demonstrate that ChatterHub successfully recognizes privacy-sensitive activities, including open/close of a smart door lock and turn on/off of smart LED.

INDEX WORDS: Cyber Forensic Analysis, Web Push Notification, Smart-home Security, Android, Internet of Things, Data Provenance

Effective Security and Forensic Analysis for Mobile and IoT Environments

by

Xingzi Yuan

B.S., Donghua University, China, 2014

A Dissertation Submitted to the Graduate Faculty of the
University of Georgia in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2021

Effective Security and Forensic Analysis for Mobile and IoT Environments

by

Xingzi Yuan

Major Professor: Kyu Hyung Lee

Committee: Roberto Perdisci
In Kee Kim

Electronic Version Approved:

Ron Walcott
Vice Provost for Graduate Education and Dean of the Graduate School
The University of Georgia
August 2021

Acknowledgments

Completion of this doctoral dissertation is only possible with the help and support from many people. I would like to express my sincere gratitude to all of them.

First, I am grateful for my PhD advisor, Dr. Kyu Hyung Lee for his patience and guidance throughout my whole PhD program. He is the best professor I have ever known. I want to express appreciation to my committee members, Dr. Roberto Perdisci and Dr. In Kee Kim. They have provided valuable advice and insights in the projects. I want to thank my parents, Junping Yuan and Zhongxian Sun. They are always supportive in my entire life. I'm not expressive in person but I want them to know that I love them internally forever. I also want to thank all my friends. Last but not least, a big thank you to my beloved wife Dr. Yifei Wu. She has been encouraging, caring and assistant (as well as sharing and loving and having and receiving) from the moment we were together.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the rapid growth of mobile and Internet of Things (IoT) devices, sensitive data is increasingly generated and stored in mobile and IoT devices to bring convenience to people's lives. However, at the same time, such devices and private data become attractive targets for adversaries. Recent study [1] reported that more than 3 million Android malicious applications have been released in 2020. The malicious applications steal personal information from end users, send text messages unknowingly to trigger deductions, and exploit root privileges to control the system, causing massive harm to the users [2]. A smart home filled with IoT devices could be exposed to more than 12,000 attacks in a single week [3], leading to not only privacy breach, but also financial loss or being used as a botnet to launch Distributed Denial of Service (DDoS) attacks [3].

Effective security and forensic analysis methods to these attacks in mobile and IoT environments have become increasingly crucial to allow researchers can detect, understand, and prevent such an attack in the future. For example, when a symptom of an attack is detected in a mobile device, forensic analysis allows the user to analyze forensic logs to reconstruct the attack to find out its origin. Understanding the damage from the attack is also essential because it helps to recover the system when acknowledging what data has been breached or what system objects have been compromised. Furthermore, understanding the entrance and attack vector is crucial to detect and prevent future attacks. Forensic logging is widely used technique for traditional desktop and server environments to capture behaviors of the system execution and their relations. For instance, audit logging techniques [4] are widely used for cyber attack forensics. They record 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 [5, 6] to locate the origin of an attack and to identify the damage to the system. Recent studies [6, 7, 8] show that forensic logging is an effective technique for cyber attack forensics in desktop or server environments.

However, existing security and forensic analysis tools cannot be directly applied to mobile and IoT devices. Mobile operating system is structurally different compared to desktop or server operating system. For example, Android applications run in the virtual machine named Android Runtime (ART), to provide a layer of isolation between each application. Traditional system call logging is too low-level to capture rich

semantics of application behaviors. Additionally, IoT devices bring unique challenges for security and forensic analysis approaches. First, IoT environments comprise diverse hardware and software platforms, thus there is no one solution to fit all IoT devices. Second, IoT devices typically have restricted resources such as computing power or memory. Therefore traditional security or forensic analysis techniques are infeasible in IoT environments.

In this dissertation, to mitigate the challenges which hinder the security and forensic analysis for mobile and IoT environments, we focus on developing selective logging techniques to identify and automatically collect important events from the mobile and IoT environments during runtime, and performing security and forensic analysis. Our techniques point out that collecting selective of events is sufficient enough to accurately reconstruct the attack. Real-world examples alongside with crafted never-seen-before attacks are used to illustrate and to evaluate our contributions. We also present an attack that could be applied to breach user's privacy in a smart-home environment.

1.1 Research Challenges

In this section, we discuss challenges uniquely for mobile and IoT environments.

1.1.1 Semantic-Rich Forensic Logs for Android

There are two major hindrances to use traditional system call logging in Android. First, because of the extra layer of ART, 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.

CopperDroid [9] and DroidScope [10] have proposed techniques to analyze the behaviors of Android malware. CopperDroid developed system-call based analysis techniques for Android attack reconstruction. DroidScope [10] 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 [11]) and it generally incurs nontrivial runtime and space overhead for resource-constrained mobile devices.

1.1.2 Automated Collection for WPNs in Android

Along with the rapid growth of online advertising, ad networks and ad publishers are constantly trying to pursue new strategies to maximize their revenues. They start to leverage the web push technology enabled by modern web browsers, especially for mobile web browsers. In 2020, roughly 80% of web push notification subscriptions are in mobile [12]. Meanwhile, malvertising (i.e., to deliver malicious ads)

has been abused when it comes to mobile as well. Those malvertising can disguise themselves as fake *missed call* notifications, fake *amber alerts*, "spoofed" *Gmail* or *WhatsApp* notifications, etc. Therefore, a tool is needed to automatically harvest and analyze malicious web push notifications (WPNs) on mobile environment and its landing page.

However, we identify three major challenges of applying PushAdMiner onto Android. First, the mechanism of how WPNs are displayed on a mobile OS is fundamentally different than desktop environment. Unlike on desktop devices, in which browser displays WPN messages, on Android device it is the Android OS that displays a WPN as a system notification. Generating clicks from instrumented Chromium would not work, as the Chromium runs in an isolated Android Runtime, which does not have the access to interact with system notifications. Second, assistant is needed to send seed website URLs to Chromium to start collecting WPN messages. Third, other than PushAdMiner collecting events happen inside Chromium, visual recording is also needed for accurate cluster of ad campaigns and further analysis.

1.1.3 Data Provenance and Forensics for MQTT

We identify three fundamental challenges for enabling data provenance and forensics for MQTT. First, to generate provenance data for diverse devices that use various hardware (e.g., CPUs) and software (e.g., OS, firmware), the approached approach must be hardware and software agnostic. Second, modification on end devices is difficult to achieve in practice. For instance, some devices may not be physically accessible and some devices may use proprietary software which does not allow any modification. The proposed approach should be able to derive data provenance from such devices. Third, end devices are often less powerful hence it is difficult to expect they have sufficient processing power to generate provenance data and storage space to store. Previous studies [13, 14] tries to improve MQTT data provenance using MQTT-Plan, EP-Plan and PROV-O. Its goal is to trace whether a message was forwarded to an unauthorised client, according to predefined "plan". However, their approach only captures a provenance event when the broker re-publishes (a.k.a., forwards) a message. Therefore any unauthorized publishing event without any subscriber is missed (e.g., DoS attack using CONNECT packets).

1.2 Thesis Statement

While most traditional security or forensic analysis techniques are infeasible for mobile and IoT devices, this dissertation proposes forensic logging techniques targeting mobile and IoT environments to enable effective security and forensic analysis. This dissertation shows that the proposed techniques can effectively help users reconstruct cyber-attacks and prevent similar future attacks.

1.3 Contributions

This dissertation addresses the challenges discussed in the previous section. The main contributions of this dissertation can be summarized as follow.

1. We propose DroidForensics for investigating malicious Android applications in mobile environment. Our goal is to provide the user with detailed information about attack behaviors that can enable accurate post-mortem investigation of attacks from Android malicious applications. 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.

2. We propose PushAdMiner and an Android helper application to achieve the automated information collection for Android Chromium web browser. An Android helper application is designed to empower PushAdMiner to forensic log collection by automatically clicking on the notifications, in addition to screenshot and screen recording. By performing analysis on collected information, we are able to collect a total of 21,541 push notification messages, in which 9,100 notifications are for the mobile environment. Among those WPN, we collect 12,262 valid landing pages and 2,692 of them are on mobile. Combined with the WPNs collected from the desktop, PushAdMiner identified 572 ad campaigns and a total of 5,143 WPN ads related to these campaigns. Moreover, PushAdMiner identified 51% of all WPN ads as malicious. Specifically, PushAdMiner found 318 (out of 572) campaigned to be malicious; in aggregate, these malicious campaigns included 2,615 WPN ads.

3. We present MQTTprov to fill the gap of lacking in-depth study and solutions for forensic analysis in IoT environment. Most of the IoT messaging protocols are in publish/subscribe model [15, 16], meaning that there is a central server taking charge of dispatching messages to corresponding clients, among which, MQTT is the most popular massaging protocols [15]. MQTTprov is designed to enable data provenance on MQTT network. We evaluate our tool against real world attack scenarios and prove that our tool is capable of reconstructing all those attacks.

4. We design an attack for the smart-home using limited information obtainable. This novel approach accurately identifies smart-home devices' activities with minimal monitoring of encrypted traffic in the home network. ChatterHub targets devices that can only connect to the Internet through a centralized smart-home hub (e.g., Samsung SmartThings) using *Zigbee* or *Z-wave*. Specifically, ChatterHub passively eavesdrops on encrypted network traffic from the hub and leverages machine learning techniques to classify events and states of smart-home devices. Using ChatterHub, an adversary can identify smart-home devices' specific activities without prior knowledge of the target smart home (e.g., list of deployed devices, types of communication protocols). We evaluate the accuracy and efficiency of ChatterHub in three real-world smart-home environments, and the evaluation results show that an attacker can successfully disclose smart-home devices' behaviors with over 88% F_1 score. We further demonstrate that ChatterHub successfully recognizes privacy-

sensitive activities, including open and close of a smart door lock and turn on and off of smart LED.

1.4 Outlines

This dissertation presents security and forensic analysis techniques for mobile and IoT environments. The rest of this dissertation is organized as follows.

- Chapter 2 presents a multi-layered forensic log collection system for Android operating system, called DroidForensics, and its ability to provide semantically rich attack traces. We presents details of our design and evaluation results of DroidForensics in this chapter. We also designed an Android helper application for PushAdMiner to enable automated log collection in Android, and the measurements from PushAdMiner.
- Chapter 3 demonstrates that even in an environment like IoT with very limited places to monitor, it is still possible to reconstruct most of the attacks in real life. We use real-world attack scenarios to evaluate MQTTprov. We also propose an attack to demonstrate privacy invasion in a smart-home environment called ChatterHub.
- Chapter 4 discusses the limitation of our proposed techniques.
- Chapter 5 reviews and discusses related literature.
- Chapter 6 concludes this dissertation.

Chapter 2

Effective Forensic Logging for Mobile Environment

Being the most popular mobile operating system, Android dominated the market with a close to 73% share in June 2021 [17]. The large mass of Android application economy brings a great profit for developers, but at the same time, it draws the the interest from malicious contents. Malicious applications find their way to bypass the censorship from app stores, meanwhile malicious websites leverage the relatively new web push notification from mobile browser to deliver social engineered notifications like fake *missed calls* luring the user to click on them.

In this chapter, we present DroidForensics in Section 2.1 to demonstrate its capability to reconstruct the attacks from malicious application; we present PushAdMiner for mobile environment in Section 2.2, to empower automatic forensic log collection in Android Chromium web browser.

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

In this section, 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 accurately 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 section 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 [18] 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 section is organized as follows. Section 2.1.1 introduces the overview of DroidForensics and motivating example using Android malware called AVPass. Section 2.1.2 discusses our design and implementation details. In Section 3.2.3.3, we evaluate DroidForensics for efficiency, effectiveness and compatibility.

2.1.1 System Overview and Motivating Example

In this section, we present an overview of DroidForensics and we use a real-world Android malware, `AVPass` [19], to motivate our work.

A high-level overview of DroidForensics is depicted in Figure 2.1. 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 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 2.1 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 an unified causal graph, and refine queries for the further investigation.

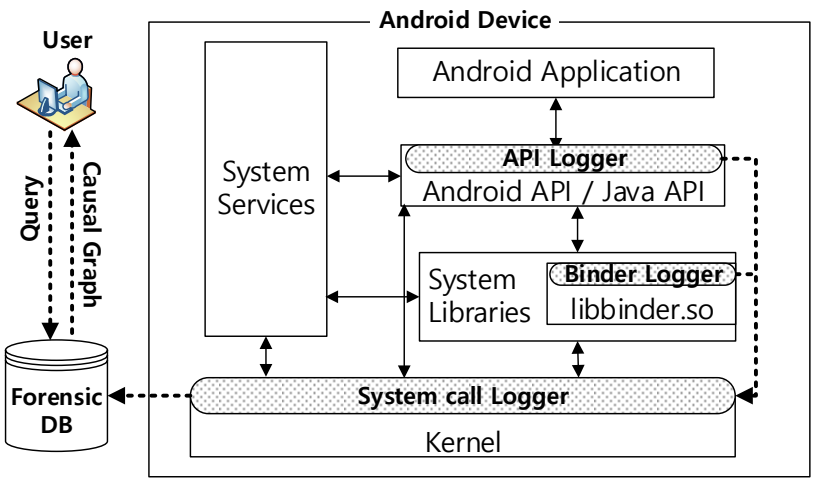


Figure 2.1: High level overview of DroidForensics

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` [19], silently installed in John's device. The malware deletes an icon and an 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 2.2 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 database, but he

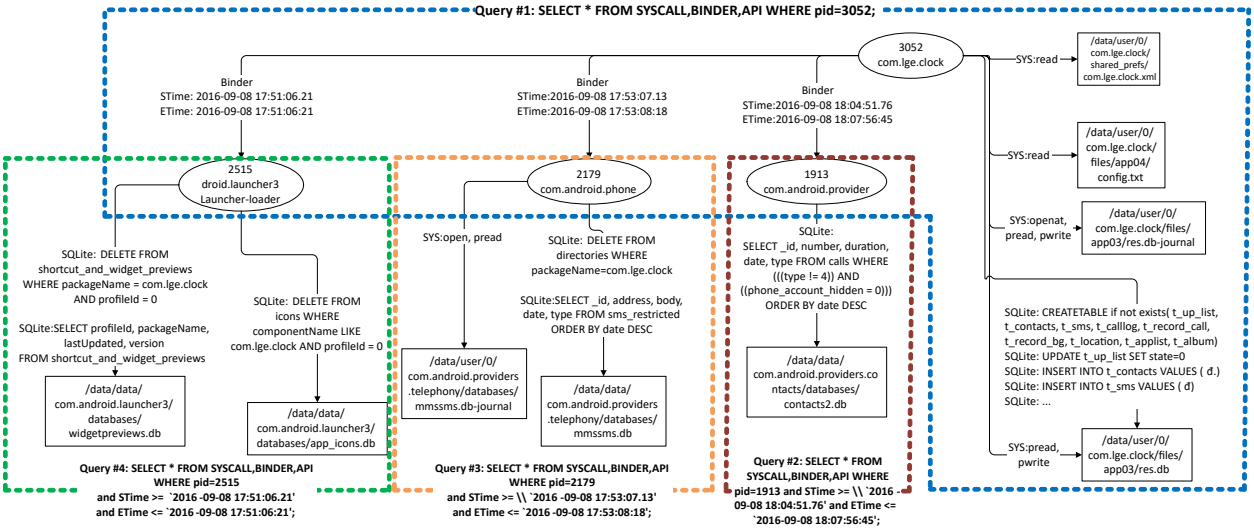


Figure 2.2: Generated causal graph from the user queries.

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 2.2). 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 2.1.2.2. 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 2.2 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 [4, 20] 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 [9] developed a technique to capture binder IPC from `ioctl` system calls, but it may hurt the runtime performance due to heavy logging (it needs to pull out the user memory used by `ioctl`) and analysis requirements. Furthermore, it cannot capture the high-level API behaviors such as SQLite queries. More importantly, 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 2.2 do not show any system call because target tables are already loaded in the memory. System call logs will completely miss those behaviors.

DroidScope [10] is a malware analysis engine that provides an unified view of malware behaviors. Similiar 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 [21, 22, 23, 24] might be able to detect what information got stolen, but cannot show how the attack unfolded. Furthermore, taint tracking generally requires instruction-level monitoring to propagate taint tags that causes high overhead.

2.1.2 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.

2.1.2.1 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 [25]. 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.

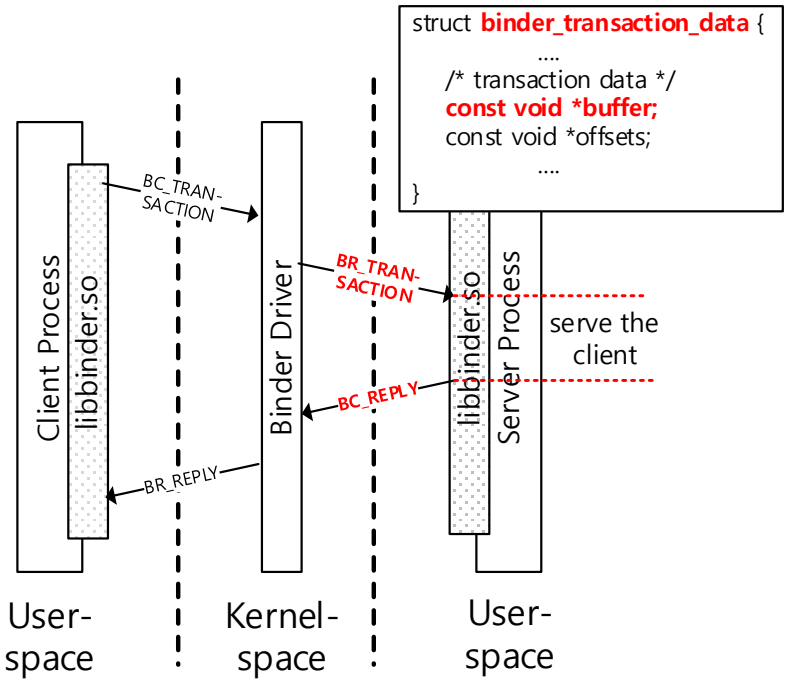


Figure 2.3: An overview of `binder` protocol.

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 [26, 27, 28, 29] to avoid offline analysis systems, but APK rewriting technique cannot handle dynamically loaded codes.

2.1.2.2 Binder Logging

Another challenge to build an effective forensic analysis system for Android is it's 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 [30]. 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 2.3 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 2.3 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.

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.

CopperDroid [9] 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.

2.1.2.3 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 [31, 32, 33, 34] to evade the Java-code analysis techniques [35, 36, 37, 38, 39, 40]. Recent study [32] 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 [4] is a default package in most Linux distributions, and DTrace [41] is shipped with FreeBSD operating system. Linux Audit is also available in Android [18], 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 [7] and Sysdig [20]. 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.

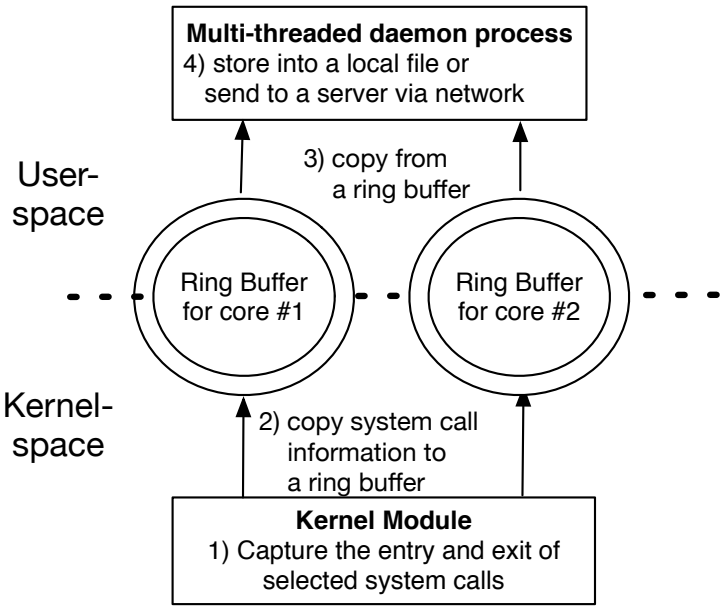


Figure 2.4: System call logging overview.

Table 2.1: Overhead of system call logging: Linux Audit and DroidForensics' system call logger.

	Benchmark	Runtime Overhead			Space Consumption		
		Linux Audit [18]	DroidForensics without comp.	DroidForensics with comp.	Linux Audit [18]	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

Similar to ProTracer [7] and Sysdig [20], our system consists of two parts, a kernel module and a user-space daemon process. Figure 2.4 shows the architecture of our system. The kernel module leverages `tracepoints` [42] to capture the entry and exit of each system call (e.g,. `sysenter` and `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 [7, 6]. 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 2.1 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 [43], TabletMark [44], and 3DMark [45]. 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

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 2.5: Database schema for log properties

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 section are conducted with the compression turned on.

2.1.2.4 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 2.5 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 2.1.2.2). 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 2.6 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 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

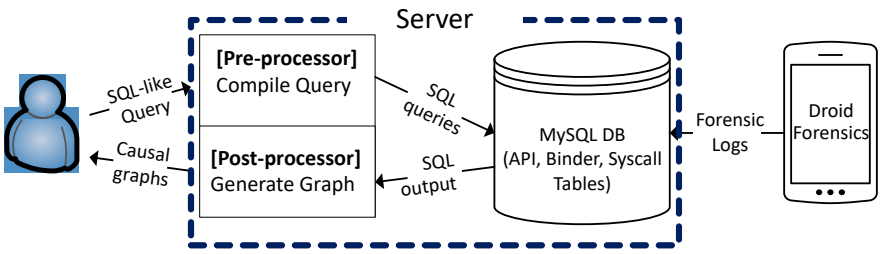


Figure 2.6: The user inferface of DroidForensics.

attack with an unified graph. We demonstrated how the user can utilize DroidForensics to reconstruct the Android attack in section 2.1.1.

2.1.3 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.

2.1.3.1 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 [43], 3DMark [45], Antutu [46], DiscoMark [47], and TabletMark [44].

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 2.7 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

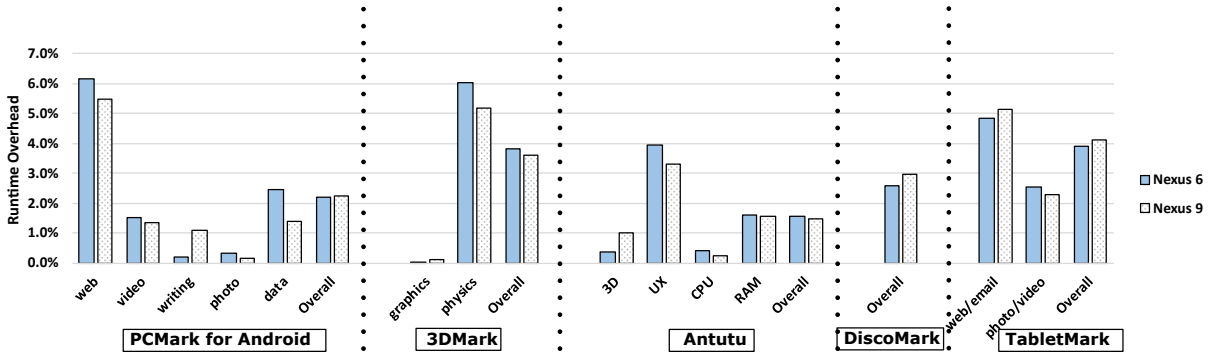


Figure 2.7: The runtime overhead of DroidForensics

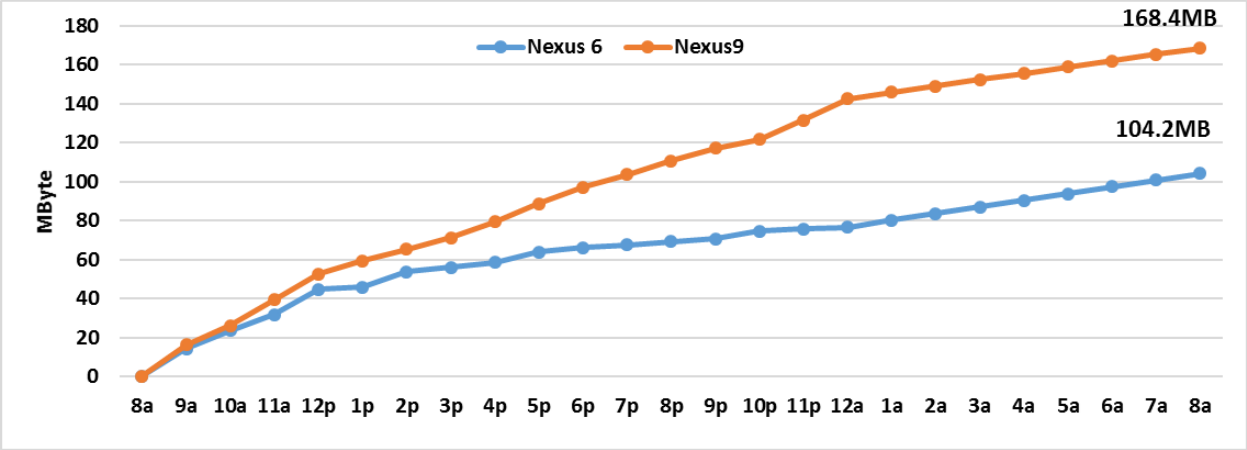


Figure 2.8: Accumulated log size from the one-day execution.

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 2.8 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.

2.1.3.2 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 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 2.2 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 it's 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

Table 2.2: 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.

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

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 it's 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 [48].

Table 2.3: 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.

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

2.1.3.3 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) [**cts**]. We use the *CTS-public-small* plan which contains around 9,200 test cases. The summarized results are in Table 2.3. 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.

2.2 Analyzing Android Web Browser Behavior

In the past few years, the rapid growth of online advertising has fueled the growth of ad-blocking software, such as new ad-blocking and privacy-oriented browsers (e.g., Brave [49]) or browser extensions (e.g., AdBlockPlus [50]). In response, both ad publishers and ad networks are constantly trying to pursue new strategies to keep up their revenues. To this end, ad networks have started to leverage the *Web Push* technology enabled by modern web browsers [51]. Until relatively recently, push notifications were mostly limited to native apps on mobile platforms, and web-based applications were unable to connect to their users out of active browsing sessions. However, now Web Push allows for web applications to send out Web Push Notifications (WPN) at any time to re-engage their users, even when the browser tab in which the web application was running is closed (the browser itself needs to be running, but does not need to be in the foreground for a WPN to be delivered to the user). Furthermore, unlike push notifications from native mobile apps, WPNs allow for notifications to be displayed on both desktop and mobile devices. Thus, they serve as a single tool with support to reach users on multiple platforms.

Although WPNs were initially designed for websites to deliver simple messages (e.g., news, weather alerts, etc.), they have become an effective way to also serve online ads, and can therefore be abused to also deliver malicious ads. In particular, the use of WPNs for ad delivery has some unique advantages. First, unlike traditional online ads (banner ads, pop-up ads or pop-under ads), advertisers do not have to wait for users to reach the web page that publishes the ad. Instead, advertisers can send out notifications that can allure users to their targeted content. Secondly, thanks to years of experience with native mobile app notifications, users have been trained to compulsively interact with push notification messages (at least on mobile devices). WPN-based ads may also be less prone to *ad blindness* [52], compared to traditional web ad delivery mechanisms such as page banners. Furthermore, ad-blocking software are not currently effective at blocking WPN-based ads (see Section 2.2.4.4), in part because browser extensions are not allowed to interfere with the *Service Workers* code through which WPNs are delivered [53]. For these reasons, some ad networks are focusing their business specifically around WPN ads (e.g., RichPush [54]).

As WPNs are relatively new, their role in ad delivery has not yet been studied in depth. Furthermore, it is unclear to what extent WPN ads are being abused for *malvertising* (i.e., to deliver malicious ads). In this section, we aim to fill this gap. Specifically, we propose a system called PushAdMiner that is dedicated to (1) *automatically* registering for and collecting a large number of web-based push notifications from publisher websites, (2) finding WPN-based ads among these notifications, and (3) discovering malicious WPN-based ad campaigns. To build PushAdMiner, we significantly extend the Chromium browser instrumentations developed by [55] and [56], which have been open-sourced by the respective authors. Specifically, neither [55] nor [56] are able to track the activities of Service Workers in detail. Therefore, we implement our own set of browser instrumentations that allows us to track WPNs in all their aspects, from registration to notification delivery, on both desktop and mobile devices. We then build a custom WPN crawler around our instrumented browser to automatically receive, track, and interact with generic WPNs, including collecting malicious WPN ads and their respective malicious landing pages. Finally,

we develop a data mining pipeline to analyze the collected WPNs and discover malicious WPN-based campaigns.

To the best of our knowledge, ours is the first systematic study that focuses on automatically collecting and analyzing WPN-based ads and on discovering malicious ad campaigns delivered via WPNs. In contrast, previous work focused on other security-related aspects of Service Workers and Push Notifications, such as building stealthy botnets [53], or social engineering attacks that attempt to force users into subscribing to push notifications [56], but without studying the resulting push messages. Lee at al. [57] study Progressive Web Apps. They collect Service Worker scripts from top-ranked website homepages and analyze their push notifications. Their work studies potential security vulnerabilities related to Service Workers, App Cache, and discusses how push notifications may be abused to launch phishing attacks, without measuring how prevalent these attacks are in the wild. Our work is different, in that we aim to *automatically* collect and analyze WPN-based ads, to discover WPN ad campaigns, and to measure the prevalence of malicious WPN-based ad campaigns in the wild.

In summary, we make the following contributions:

- We present PushAdMiner, a system that enables the automated collection and analysis of online ads delivered via web push notifications (WPNs) on both desktop and mobile devices.
- To track WPNs, we extend a Chromium-based instrumented browser developed in [55, 56] to allow for a detailed analysis of Service Workers, which are at the basis of WPN deliveries. Furthermore, we build a custom WPN crawler around our instrumented browser to collect and automatically interact with WPNs.
- Using PushAdMiner, we collected and analyzed 21,541 WPN messages by visiting thousands of different websites. Among these, our system identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads that were pushed by a variety of ad networks. Furthermore, we found that 51% of all WPN ads we collected are malicious, and that traditional ad-blockers and URL filters were mostly unable to block them, thus leaving a significant abuse vector unchecked.

2.2.1 Motivating Example and Background

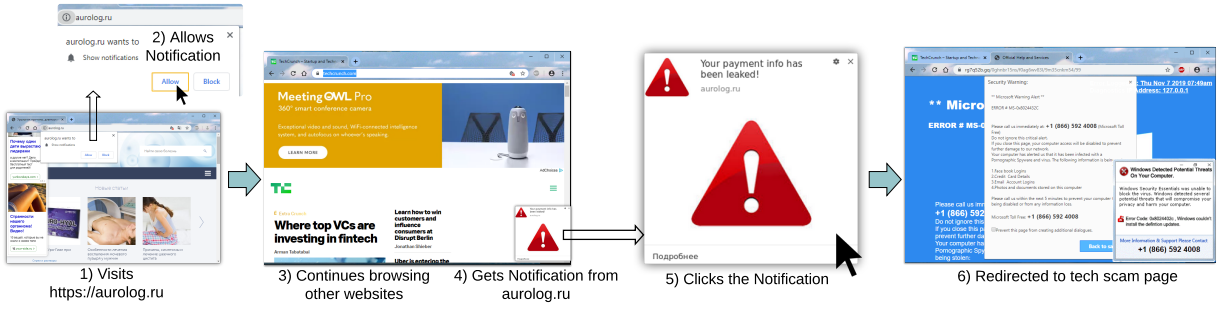


Figure 2.9: Example of malicious advertisement served through web push notifications

In this section, we provide an example of WPN-based malicious ad, and then briefly explain the concepts and technologies behind web notification services.

2.2.1.1 Motivating Example

Figure 2.9 provides an example of malicious WPN-based ad. During the preliminary stages of our research, we stumbled upon a website on `aurolog[.]ru`. When visiting the main page, the site requested permission to send us notifications. We granted permission by pressing the *Allow* button on the browser dialog box, and subsequently received a WPN ad with the following alert message: "Your payment info has been leaked" (see Figure 2.9). After clicking on the notification, we were redirected to a *tech support scam* [58]. To our surprise, the landing URL was neither blocklisted by Google Safe Browsing[59] nor detected as malicious by any of the web page scanners on Virus Total[60]. This example confirmed our suspicion that WPNs may be abused for malvertising, and sparked our investigation to determine whether such cases of malicious WPN-based ads could be automatically collected and analyzed.

2.2.1.2 Technical Background

Recent changes in HTML5 have introduced new web features, such as *Service Workers*[61], *Push Notifications*[62] and *AppCache*[63]. Websites that adopt these technologies are called Progressive Web Apps (PWAs). Throughout this section, we refer to push notifications sent by PWAs using a browser as Web Push Notifications (WPN), to distinguish them from push notifications sent by native apps on mobile devices, and refer to Service Workers as SWs for brevity.

Service Workers and Push Notifications A *Service Worker* (SW) is an event-driven script executed by the browser in the background, separately from the main browser thread and independently of the web application from which it was initially registered and that it controls. In practice, a SW comes in the form of a JavaScript file that is registered against the origin and path of the web page to which it is associated (only HTTPS origins are allowed to register a SW). In effect, SW can be viewed as "a programmable network proxy that lets you control how network requests from your page are handled"[64].

Service Worker can use the *Push API*[65] to receive messages from a server, even while the associated web application is not running. It is worth noting that a single web app is allowed to register multiple SWs. Service Workers can also use the *Notifications API*[62] to display system notifications to the user. A prerequisite is that the web application must first request permission to display notifications to the user (only allowed for HTTPS origins). If the user accepts (i.e., clicks on "Allow" instead of "Block" on the notification request popup) to receive notifications from the web application's origin, this permission persists across browser restarts, and until the user explicitly revokes the permission via browser settings/preferences (notice that non-expert users may find it difficult to understand, find, and disable notifications in the browser's settings).

Web notification messages have a number of customizable parameters, such as title, body, target URL, icon image, display image and action buttons. The user can interact with a notification by either clicking

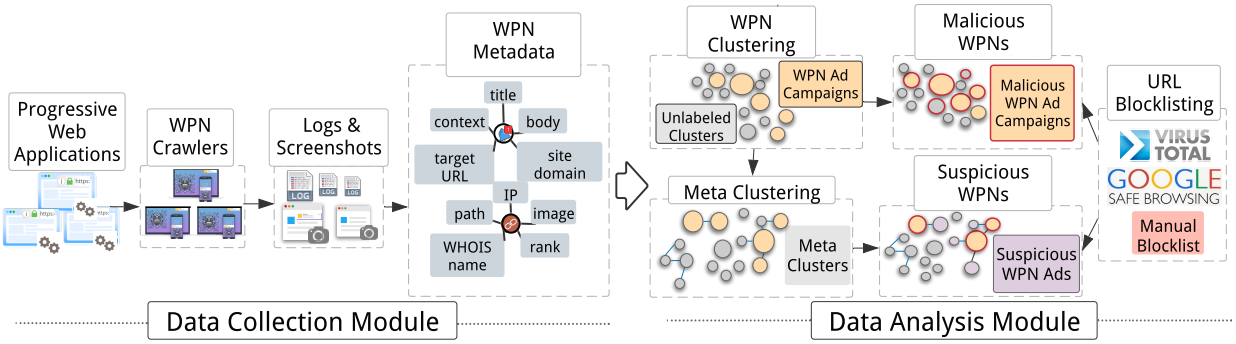


Figure 2.10: PushAdMiner System Overview

on it, closing it or performing any custom actions displayed in the notification message. SW can listen to such user events and take action according to the input. This includes loading target URL on a separate tab, following a user's click on the notification box.

Firebase Cloud Messaging (FCM) FCM is a cross-platform messaging solution for Push Notifications. It can serve as a central authority that mediates the communication between the ad server and the Service Worker. Upon initial registration, FCM creates a unique registration ID per user and per Service Worker, which is sent along with an endpoint URL [66] to the ad server. For further details, refer to FCM's online documentation [67].

2.2.2 System Overview

In this section, we provide an overview of how PushAdMiner works, leaving a detailed description of the main system's components to Sections 2.2.3. A high-level representation of the system is provided in Figure 2.10.

PushAdMiner consists of three main components: (i) an instrumented browser to collect fine-grained information about SWs and WPNs; (ii) a custom crawler that automatically visits sites and interacts with the browser, including granting notification permissions and interacting with WPNs (Section 2.2.3); and (iii) a data analysis component aimed at identifying WPN-based ad campaigns and labeling likely malicious ones.

While a number of browser automation and crawling systems have been proposed, including Selenium [68], Puppeteer [69], and others [55, 56, 70], currently they do not fully support the automatic user interactions with WPNs and collection of all details about SWs needed for our study. We therefore built an instrumented browser based on Google Chromium, by significantly extending existing open-source Chromium instrumentations [55, 56]. In addition, we leveraged Puppeteer [69] for browser automation and event logging, and wrote custom scripts to record SW registrations and network requests. Figure 2.10

presents an overview of how PushAdMiner collects information about WPNs, and how the browser logs are analyzed to identify ad campaigns in general and discover malicious ones among them. First, we discover a set of URLs that may send push notifications with the help of an ad network and filter those that actually request for notification permission (see Section 2.2.4.1 for details). For the web pages that ask for notification permissions, we log details about the responsible SW code, automatically grant permission (via browser code instrumentation), and then collect the notifications that are later pushed to our instrumented browser. When a notification is displayed by the browser, we record fine grained details about the notification message itself (including message text and icons), automatically simulate a user click on the notification box (via browser code instrumentation), and track all events resulting from the click. If the click results in a new page being open, we record detailed information about the related network requests, including all browser redirections, as well as detailed logs and a screenshot of each new page the browser visits, including the landing page (i.e., the final web page reached due to the click).

Finally, we extract relevant information from the detailed logs of our instrumented browser, and apply a clustering strategy to find notifications that are similar to each other, which allows us to identify WPN-based ad campaigns. We then leverage URL blocklists to find WPN ad campaigns that are likely malicious (e.g., because one or more landing pages are known to be malicious).

Note that in this section we do not focus on building a malicious WPN ad campaign detector, such as using statistical features or machine learning classifiers. Rather, our focus is on discovering, collecting, and analyzing WPN ad campaigns in general, and on measuring the prevalence of both benign and malicious campaigns. As we will show in Section 2.2.4, URL blocklists tend to miss a significant number of malicious URLs that we determine to be related to malicious ad campaigns. The analysis we present in this section could therefore be used as a starting point for developing an automated malicious WPN ad campaign detector. We leave this latter task to future work. Our code to collect and analyze WPNs is publicly available in a Github repo[1] and a Docker container with the instrumented Chromium is in Docker Hub [2].

Ethical Considerations To track WPN-based ads and label malicious ones, it is necessary to collect information about the landing page that an ad eventually redirects to. For instance, for most malicious ads the attack is effectively realized only once the user reaches the landing page, especially in case of social engineering and phishing attacks. As we do not know in advance what landing page will be reached by clicking on a WPN message, and whether a WPN ad is malicious or not, our system will likely click on both legitimate and malicious ads. In turn, this may cause legitimate advertisers to incur a small cost for our clicks, as they will likely have to pay a third-party publishing web page and ad network for their services (notice that we obviously receive no monetary gain whatsoever during this process). This is common to other similar studies, such as [56, 71], and we therefore address the ethical considerations for our study following previous work.

[1] `https://github.com/karthikaS03/PushAdMiner`

[2] `https://hub.docker.com/repository/docker/dockerammu/docker_puppeteer_chromium_xvfb`

To make sure we do not have a significant negative impact on legitimate third-parties, we estimated the cost incurred by these advertisers due to ad clicks performed by our system, and found that our system has negligible impact on advertisers. Specifically, among the WPN ads we identified, we consider legitimate ads to be those whose landing pages are not labeled as malicious by Virus Total's URL classification services. Then, we estimate the *cost per landing domain* based on the number of ads we clicked on that lead to a specific domain, using the *Cost Per Mille (CPM)* [72] for push notification ads according to iZooto [73]. The maximum cost per landing domain throughout our entire study was USD 1.12 (due to landing on the same domain 444 times), which we calculated using the standard CPM of USD 2.54. On average, we visited each landing domain 18 times, which corresponds to an average cost of USD 0.04 per landing domain (i.e., per advertiser). Considering these low values, we believe the impact of our system on advertisers is not significant, and is on par with previous work [56, 71].

2.2.3 Data Collection Module in Mobile Environment

In this section, we describe in detail how PushAdMiner's data collection module is implemented, particularly in Android environment. The workflow of the data collection module in desktop environment is shown in Figure 2.11.

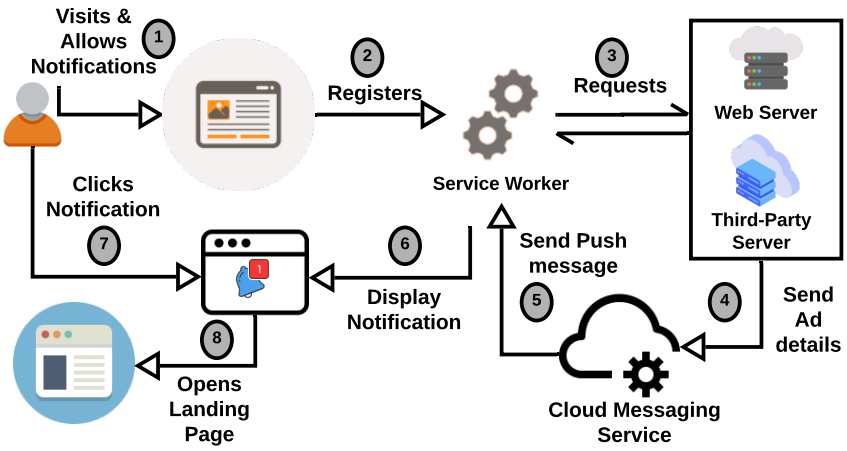


Figure 2.11: Steps involved in Serving Ads via WPNs

Due to some technical differences between how WPNs are displayed on a mobile OS, compared to desktop environments, we had to adapt some of the system components to run specifically on Android.

Logging Internal Browser Events We compile our instrumented Chromium browser for Android, so that we can collect intimate details about internal browser events related to WPNs, including recording information about the related Service Workers and the rendering of the landing page resulting from clicking on a WPN. Browser logs are sent via the `logcat` ADB command to a remote logging machine.

Interacting with Notifications Unlike on desktop devices, in which WPN messages are displayed by the browser, on Android device it is the Android OS that displays a WPN as a system notification. Also, unlike on desktop environments, the browser does not need to be activated for a WPN message to be received, though the browser may be activated after tapping on a notification (e.g., to navigate to the URL pointed to by the notification). We therefore had to implement a different mechanism to simulate user interactions with WPNs on Android. Specifically, we developed an Android application that leverages Android's *Accessibility Service*. The Accessibility Service is aimed to help people with disabilities in using the device and apps. It is a long-running privileged system service that helps users process information from the screen and lets them interact with the content meaningfully in an easy way. Android developers can leverage the Accessibility Service API and develop apps that are made aware of certain events, such as `TYPE_VIEW_FOCUSED` and `TYPE_NOTIFICATION_STATE_CHANGED`. Furthermore, the accessibility service API can also be used to initiate user actions such as click, touch and swipe.

We install our app with Accessibility Service permission on an Android physical device, and use it to interact with every notification event fired. Whenever a new notification pops up, our application will automatically swipe down the notification bar and click on the notification to complete the action, while our instrumented Android browser produces detailed logs about the consequences of such interactions (e.g., loading a new web page).

As an alternative, `Android Debug Bridge` (ADB) could be leveraged to implement the same browser automation that we implemented using Accessibility Service. However, in practice, we found that it could create large traffic overhead through the USB cable connected with the device. As we already use ADB to retrieve fine-grained browsing logs, we decided to avoid further USB overhead. Therefore, we found the use of the accessibility features to be better in practice for our specific application.

2.2.4 Measuring WPN Ads in the Wild

In this section, we report measurements on the usage of WPNs as an ad delivery platform, and provide insights into the malicious use of WPN ads.

2.2.4.1 Data Collection Setup

We first describe PushAdMiner's setup for harvesting in-the-wild WPN messages for both desktop and mobile environments. Because our internal browser instrumentations are implemented by extending the browser code provided by [56], our data collection process leverages Chroumium's code base version 64.0.3282.204, which we built for both Linux and Android environments.

Collecting WPNs in Mobile Environment During our study, we found that WPN messages sent to mobile devices tended to be somewhat different that the ones collected by desktop browsers, in that they were more tailored to mobile users. In particular, malicious mobile WPN messages included fake *missed call* notifications, fake *amber alerts*, "spoofed" *Gmail* or *WhatsApp* notifications, fake *FedEx* notifications, etc. In addition, we found that these malicious messages were much more likely to appear on real Android

devices, rather than emulated environments (likely due to some form of emulator detection). Therefore, to automatically collect mobile WPN messages we instrumented a real mobile device. Specifically, we used a Google Nexus 5 device with 2 GB of RAM and a 1080×1920 pixels display. The Android version we used was `aosp_shamu-userdebug 7.1.1 N6F26Y`.

We used a real device for the following reasons. Android emulators are easily detected by ad libraries [74, 75, 76], which may prevent us from harvesting real WPN ads. Also, while desktop Chromium with mobile view emulation enabled can receive WPN ads, we also observed that some ad networks tend to send mobile-specific WPN ads only to real mobile devices.

As we attempted to scale our PushAdMiner's mobile WPN crawlers on a real device, we identified two challenges. First, Docker or other container techniques do not support Android, and therefore we cannot easily visit multiple URLs in parallel with isolated browsing sessions. Second, we considered to use app cloning techniques [77] to open multiple browser instances separately in isolated execution environments. However, the limited computing power of mobile device restricted us to scale up and visit a large number of URLs simultaneously. Therefore, we decided to open multiple URLs in one chromium app but in separate tabs.

Table 2.4: Measurement Results of Data analysis Module

	WPNs with Landing Pages	**WPN Ad Campaigns**	**WPN Ads**	**Malicious WPN Ad Campaigns**	**Malicious WPN Ads**
Desktop	9,570	572	5143	318	2615
Mobile	2,692				
Total	12,262				

2.2.4.2 WPN Messages Dataset

We start with the 5,849 initial URLs that we collected , over 5,697 distinct second-level domain names. By clicking on WPN messages issued by these initial URLs, we collect an additional 10,898 URLs across 2,269 distinct second-level domains. When visited, many of these additional URLs presented our browser with a notification request, which our crawler automatically granted. This brought us to a total of 7,951 URLs that registered a SW with Push permission and were therefore able to push notifications to our instrumented browser instances over time.

During the course of about two months (September and October 2019), we were able to collect a total of 21,541 push notification messages, including 12,441 notifications for the desktop environment and 9,100 for the mobile environment. PushAdMiner interacted with each of these WPN messages. However, not all automated clicks on notification boxes led to a separate landing page. In addition, some landing pages appeared to cause a crash in the browser's tab (but not the browser) in which they rendered, preventing us from collecting detailed information on those pages (this was likely due to the fact that our instrumented

Chromium browser is not based on the most recent stable code base). We filtered out these notifications, leaving us with 12,262 WPN messages (9,570 on desktop and 2,692 on mobile) that when clicked on lead to a valid landing page. We then used this final set of WPN messages for clustering process .

2.2.4.3 Data Analysis Results

Summary of findings Table 2.4 summarizes the overall results of our analysis process. From the 12,262 WPN messages mentioned above, PushAdMiner identified 572 WPN ad campaigns and a total of 5,143 WPN ads related to these campaigns. Moreover, PushAdMiner identified 51% of all WPN ads as malicious. Specifically, PushAdMiner found 318 (out of 572) campaigns to be malicious; in aggregate, these malicious campaigns included 2,615 WPN ads.

This is quite a staggering result, in that it appears that ad networks that provide WPN ad services are heavily abused to distribute malicious content. Later, in Section 2.2.4.4, we also show that ad blockers are ineffective at blocking such ads, which is an additional cause of concern. In the following sections, we discuss the clustering and labeling results in more detail.

WPN Clusters and Ad Campaigns We cluster the collected WPN messages based on their message content and landing page information. After clustering 12,262 WPN messages that led to a valid landing page, we obtained 8,780 WPN clusters, of which 7,731 were singleton clusters containing only one element (i.e., only one WPN message). Of the remaining non-singleton clusters, 572 were labeled as WPN ad campaigns . In aggregate, these WPN ad campaigns pushed 3,213 WPN ad messages to our browsers, during a period of about two months.

Figure 2.12 provides some concrete examples of WPN clusters. In both `WPN-C1` and `WPN-C2`, the respective WPNs were pushed from multiple sources (i.e., multiple second-level domain names), as also shown in Figure 2.12. `WPN-C3` included 4 identical WPN messages pushed by a single source website, a bank, alerting users on their loan offers. These messages appear to be legitimate, and led back to the site that pushed them. `WPN-C4` is an example of WPN message isolated into a singleton cluster. We label `WPN-C1` and `WPN-C2` as WPN ad campaigns, because the WPNs in each of the clusters deliver very similar (or the same) message promoting very similar products from multiple sources. However, `WPN-C3` and `WPN-C4` do not meet the definition and are thus not labeled as WPN ad campaign.

Malicious WPN Ad Campaigns We submit landing page URLs related to all WPN messages to GSB [59] and VT [60]. On our initial scan, less than 1% of the URLs were detected as malicious by GSB or VT, in aggregate. For instance, initially VT flagged 108 landing page URLs as malicious, of which 88 were related to WPN messages labeled by our system as belonging to ad campaigns. Notice that for VT we consider a URL as malicious if at least one of the URL detection engine reports it as malicious, and later manually review all results to filter out possible false positives. After one month, we submitted the same set of URLs once again, and we found that 1,388 URLs (11.31%) were detected by VT, though GSB still only flagged 1% of them.

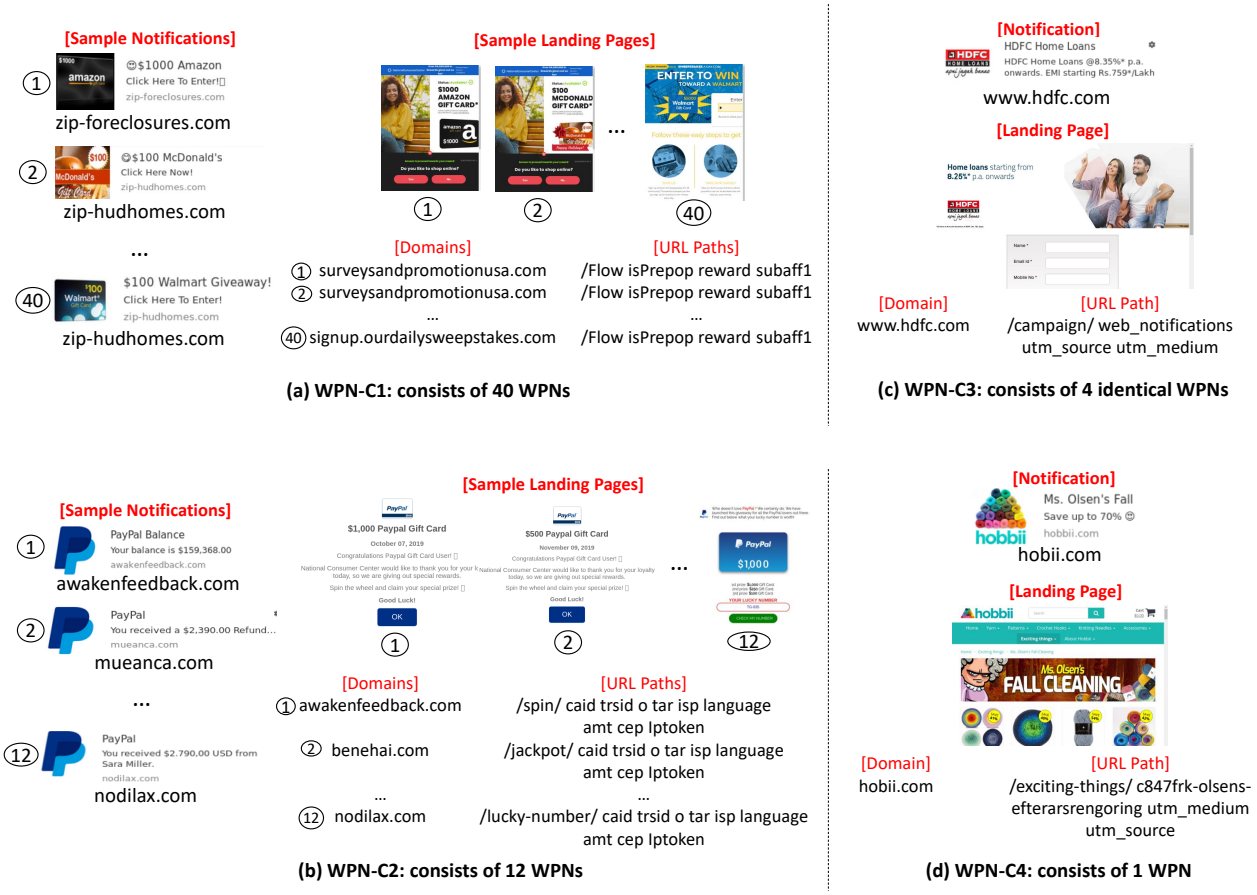


Figure 2.12: Examples of WPN clusters

PushAdMiner relies on label propagation to label WPN messages and clusters as malicious, based on results from VT and GSB . To limit the chances of amplifying possible false positives from VT and GSB, we manually verified all 1,388 URLs to check whether they actually led to malicious content. We were able to confirm that 96.8% of them indeed appeared to be malicious. Of the remaining 44 URLs that we could not confirm as malicious, 13 were found to belong to popular benign domains such as `bing.com`, `kbb.com`, `tophatter.com`, etc.; 24 URLs were related to unpopular blog/news sites; 3 led to adult websites; and 4 led to websites hosting non-English content that we could not verify. Given that these sites may be benign, since we do not have all the information VT and GSB had to label them as malicious we take a conservative stance and remove the malicious label from them. Accordingly, we label 1,344 WPNs as *known malicious*. Among them, 758 WPNs were part of 572 WPN clusters that we previously classified as ad campaigns (see Section 2.2.4.3). The remaining 586 WPN messages that led to malicious landing pages were not immediately found to belong to WPN ad clusters, as they formed separate small clusters. We will determine whether they are related to WPN campaigns later, in Section 2.2.4.3) after the meta-clustering step .

Table 2.5: Measurement Results at Stages of Clustering

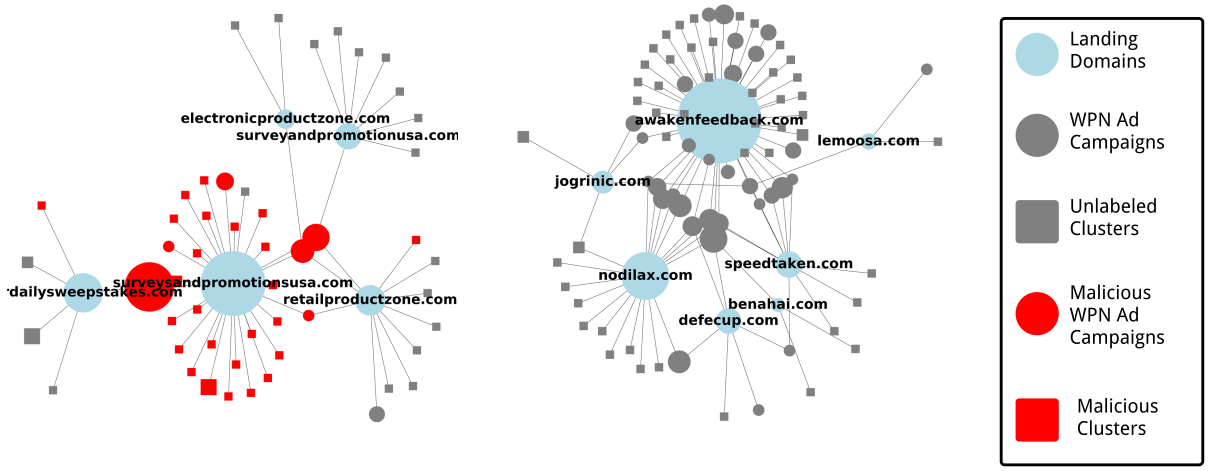
	# clusters	# ad-related clusters	# WPN ads	# known malicious ads	# additional malicious ads
After WPN Clustering	8780	572	3213	758	367
After Meta Clustering	2046	224	1930	210	1280
		Total:	5143	968	1647

By using a "guilty by association" label propagation policy , we label WPN ad campaigns as malicious if they include at least one *known malicious* WPN (remember that this policy is justified by the close similarity in content and landing page URL path between messages in the same cluster). This yielded 152 (out of 572) *malicious* WPN ad campaigns, which overall included 376 WPN (or more precisely their landing pages) that GSB or VT missed to detect as malicious. After manually inspecting these 376 WPN ads, we were able to confirm that 367 of them are indeed malicious ads that lead to survey scams, phishing pages, scareware, fake alerts, social media scams, etc. We were not able to confirm the maliciousness of the remaining 9 ads (i.e., 2.4%) that led to different pages that welcome/thank the user for subscribing to the notification all hosted on the same IP address. The take away from the above discussion is that, using our WPN clustering approach, we were able to increase the number of confirmed malicious WPN ads from 758 to 1,125 (i.e., 758 plus 367), which represents an increase of about 50% as summarized in Table 2.5, first row.

Referring to the examples provided in Figure 2.12, in cluster `WPN-C1`, 35 out of the 40 WPNs were labeled as *known malicious* WPNs, according to VT. However, PushAdMiner labeled this entire cluster as *malicious*. After manually inspecting all 40 messages, we confirmed that the remaining 5 messages in the cluster were indeed related to the 35 malicious sweepstakes/survey scam ads.

Finding Suspicious Ads So far, we have leveraged the labels provided by VT and GSB to identify malicious WPN ads, and label WPN ad campaigns. Unfortunately, both URL blocklists suffer from significant false negatives, when it comes to detecting malicious landing pages reached from WPN ads. As an example, consider cluster `WPN-C2`, which PushAdMiner identifies as an ad campaign. This cluster contains 12 WPNs; none of which were labeled as *known malicious* according to VT. However, PushAdMiner flags this cluster as suspicious since it contains *duplicate ads* and via manual inspection we found that the WPN messages in this cluster display fake PayPal alerts that lead users to survey scam pages; therefore, we manually label the entire `WPN-C2` cluster as malicious. This example demonstrates the gaps left by current URL blocklisting services, and how ineffective they could be if they were used to detect and block malicious ad notifications. Below we discuss how we use the meta-clustering approach to automatically identify and label more of such cases.

We apply a meta-clustering method to group WPN clusters that may relate to each other, as they share common landing page domains. To this end, we create a bipartite graph $\mathcal{G} = (W, D, E)$, here W is the set of all 8,780 WPN clusters we previously obtained, and D is the set of all 2,177 distinct landing page domains pointed to by WPN ads that we were able to record. By identifying and separating the connected components in this bipartite graph, we identify 2,046 WPN meta clusters. Of these, 224 contain a mix of WPN clusters that we previously labeled as *ad campaign* and other non-campaign WPN clusters. We then label all WPN messages contained in these 224 ad-related meta-cluster as *WPN ads*, thus increasing the number of WPN ads identified so far from 3,213 to 5,143.



(a) Meta Cluster of WPN-C1 WPNAC (b) Meta Cluster of WPN-C2 WPNAC

Figure 2.13: Graphical Representation Examples of Meta Clusters

Figure 2.13 provides two examples of meta clusters. Figure 2.13a contains as a node cluster `WPN-C1` from Figure 2.12, as well as other 6 related WPN ad campaigns that are likely orchestrated by the same operators. This meta-cluster contains many *known malicious* WPN ad campaigns and WPN clusters, and therefore, we label all the WPN clusters in the meta cluster as suspicious. By manual inspection, we verified that all domains involved in this meta cluster host visually similar malicious pages (e.g., online survey scam pages).

Figure 2.13b shows another example of meta-cluster, which includes cluster `WPN-C2` from Figure 2.12 as a node, along with *30* other related WPN ad clusters. In this meta-cluster, none of the WPN clusters (i.e., the nodes) were initially labeled as malicious by either VT or GSB. However, we manually inspected all landing pages pointed to by WPN messages including in the meta cluster, and we were able to confirm that these are indeed malicious, in that they display fake PayPal messages and alerts that lead users to survey scams and likely phishing-related pages.

Next, we consider all yet to be labeled WPN messages in a WPN meta cluster as suspicious if the meta cluster contains at least one *malicious* WPN cluster or if it contains *duplicate ad domains*. Out of the 572 WPN ad campaigns identified earlier, we found that 255 of them contained *duplicate ad domains*. Accordingly, we were able to label a total of 287 out of 2,046 WPN meta clusters as *suspicious*. Further, we

identified 166 (out of 572) additional WPN ad campaigns, which were not previously labeled malicious in the previous step, as *suspicious*. Overall, this translates into 1,479 suspicious WPN ads, as shown in Table 2.5. Following our manual verification process, we confirmed 1,280 (86.5%) of these ads as malicious. The remaining 199 WPN ads were flagged by PushAdMiner because they were related to *duplicate ad domains*. Of these, 166 were alerts related to job postings and led to similar pages on multiple domains listing the same job; 23 led to multiple sites that hosted content related to the horoscope; 4 led to adult websites; and 6 were subscription welcome/thank you notifications. Notice that while these 199 WPN messages may be benign, PushAdMiner helped us identify and characterize a large number of additional WPN ads that are in fact malicious and were not identified by URL blocklists such as VT or GSB. However, our current system is not designed to be an automatic malicious WPN ad detection system. In our future work, we plan to leverage the lessons learned from the measurement results obtained in this section to investigate how malicious WPN messages can be accurately detected and blocked in real time.

Singleton Clusters: Our tight first-stage clustering yielded 7,731 singleton clusters. Of these, 6,876 were found to share landing domains with WPNs in non-singleton clusters. After meta clustering, we were then left with 855 singleton clusters. By manually inspecting a sample of 200 singleton clusters, we found them to be a mix of simple alerts and spurious suspicious ads. Table 2.6 shows a few example of the text and domains related to the analyzed singleton clusters.

Table 2.6: Examples of Singleton Clusters

Title	**Body**	**Domains (S)-Source (L)-Landing**
TechNewsGadget	189 Fortnite Wants To Add Bot Players	technewsgadget.net (S)(L)
Congrats Walmart User!!	(1) Reward Waiting!	healthydreamstoday.com (S) besthealthlife.com (L)
The Mattest Blackest Liner EVER !!	Our new obsession!	hudabeauty.com (S)(L)
Coca Cola is looking for YOU ??	No experience required Training Provided!	eblog.network (S)(L)
Hire Local Service Professionals For All Your Needs	vconnect.com is your one stop destination for local services.	m.vconnect.com (S)(L)
FOX NEWS	Lose 45lbs In 4 Weeks! No Exercise!	nodilax.com (S) womenlifestyledaily.com (L)

Additional recent measurements: To measure the prevalence of WPN ads at a later point in time, compared to our initial measurements, we collected an additional and more recent batch of data for 5 days

between April 4th, 2020 and April 9th, 2020. We revisited 300 websites randomly chosen from our previous datasets, 35 of which sent us 305 notifications over 5 days. Of these 305 notifications, PushAdMiner labeled 198 WPN ads and flagged 48 of them as malicious, which we also verified via manual analysis. After checking the corresponding landing page URLs on VirusTotal, only 15 of them were flagged as malicious, confirming again that WPN-based threats often remain undetected by current defenses.

2.2.4.4 Push Ad Networks and Blocking

Figure 2.14 shows the distribution of WPN ads, including malicious ones, per ad network. As it can be seen, many of the ad networks we considered in our measurements are abused to distribute malicious WPN ads.

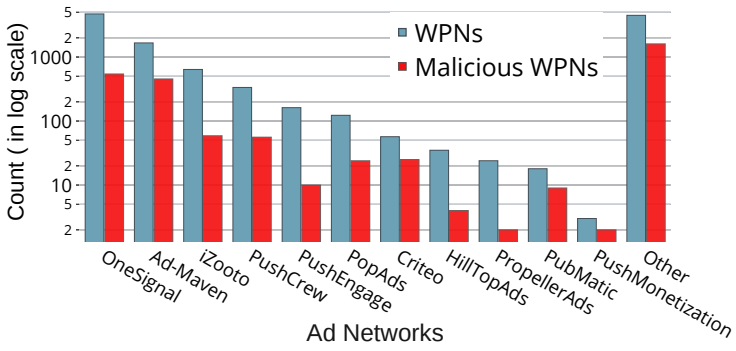


Figure 2.14: Distribution of WPNs w.r.t. Ad Networks

We also investigated whether ad blocker extensions used by desktop browsers may be effective against WPN ads. For instance, we checked the URLs of SW scripts against the Easylist filter rules[78] used by most popular ad blockers. Furthermore, we installed two highly popular ad blocker extensions in our Chromium browser and checked its blocking capability. As shown in Table 2.7, both ad blocking mechanisms couldn't block the requests issued by the installed SW that were related to WPN ads, even though Easylist was able to filter a small number (less than 2%) of such network requests. This shows that existing methods were not sufficient to mitigate WPN-based ads, including malicious ones.

While working on this study, a new feature was introduced in Chrome [79] in February, 2020 to prevent the abuse of WPNs. This feature focuses on blocking notification permission prompts from websites that have a low notifications opt-in rate. To test the effect of this feature on our dataset, we used the latest Chrome version (Chrome 80) to revisit 300 randomly chosen websites that had previously requested notifications. Since our Chromium code instrumentations could not be easily ported to this latest version of the browser, we performed detailed manual analysis. We found that all of the websites we visited were still able to request notification permissions without getting blocked. It is possible that this new Chrome feature will be able to block abusive WPNs in the future, after more training data is

collected. However, it is unclear whether and to what extent it will be effective in blocking WPN-based malicious ads.

Table 2.7: Results on Existing Ad Blockers

	No. of Blocked URLs	
	Service Worker Scripts	**Service Worker Requests**
Easylist Blocklist	0 out of 1187	132 out of 8031
AdBlockPlus	0 out of 884	0 out of 7276
AdGuard	0 out of 895	0 out of 7520

Chapter 3

Security and Forensic Analysis for Internet of Things (IoT)

With more than 10 billion active IoT devices in 2021 [80], personal information is increasingly used to provide convenience. However, a large number of privacy related attacks are crafted [80] as well. To address these scary invasions of privacy, we present MQTTprov for MQTT network in Section 3.1, providing the ability to collect provenance and forensic logs from such IoT network. We also propose an attack called ChatterHub in Section 3.2, proving that smart-home environment is easy to breach privacy even with encrypted network traffic.

3.1 MQTT Data Provenance

3.1.1 Background and Motivating Example

In this section, we briefly explain the concepts behind MQTT protocol and then present a realistic motivating example.

3.1.1.1 Technical Background

Broker A broker, also referred as MQTT server, is a program or device that acts as an intermediary between clients. A broker handles things like accepting network connections from clients, accepting messages from clients and forwarding those messages to clients with subscriptions, managing subscribing and unsubscribing requests from clients, etc. The diagram of a MQTT network is present in Figure 3.1. MQTT natively decouples the publishers and subscribers. As a result, the subscribers can not figure out where the messages are coming from.

Client A client is a program or device that uses MQTT protocol to connect to a MQTT broker. A client can both subscribe to and publish to one or more topics. To make it easier to demonstrate in this work, we call a client that publishes messages a *publisher*, and a client that receives messages a *subscriber*.

Topics and QoS MQTT messages are published into a hierarchy of topics that are addressed as UTF-8 strings, and when a client is connected to the broker, it can choose to subscribe to certain topics. Levels in the topic hierarchy are divided by forward slash characters ("/"). A wildcard in topic enables the client to subscribe to multiple topics simultaneously. A wildcard can only be used to subscribe to topics, not to push a message. There are single-level wildcard (+) and multi-level wildcard (#). If there's no restriction on the broker's rule, a client can subscribe to whatever topics it desires. With the help of wildcard "#", a client can listen to all the topics without knowing the exact topic strings. Subscribing to "#" equals to listening to all the topics that are passed through the broker. Each connection between clients and broker can specify a Quality of Service (QoS): at most once (QoS 0), at least once (QoS 1), and exactly once (QoS 2).

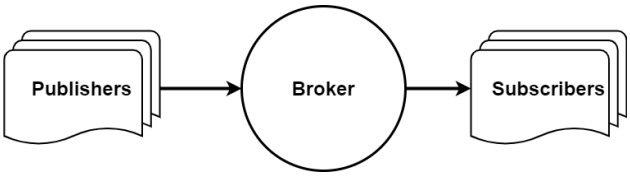


Figure 3.1: A diagram of MQTT protocol model

Dummy Subscriber listens to "#" The dummy subscriber is a naive way to monitor MQTT traffic. The administrator could use a dummy subscriber only subscribe to topic "#" to fetch all the messages going through the broker. In this monitoring method, the dummy subscriber is only able to act like a normal user, and is only able to fetch topics and messages from the publisher. It is impossible for the dummy subscriber to know who-else receives the topics and messages. At the same time, because of the decoupling nature of MQTT, it is impossible for the dummy subscriber to know the authentication of the message origin.

Verbose Log from Vanilla Broker If verbose output of a vanilla MQTT broker (for example, Mosquitto [1]) is enabled, following information could be gathered: client ID, IP address, and Topic. The format and content of verbose log could be different based on various of implementation of the broker. In this work, we use the verbose log from Mosquitto, which is the larges open-source broker.

3.1.1.2 Motivating Example

As shown in Figure 3.2, Alice has a home full of devices broadcasting status using MQTT protocol, for example, a door lock broadcasting lock/unlock events to the topic "`/Alice/home/frontdoor/doorlock/status`". This broadcasting service provides convenience to Alice, because Alice could use a light controller set to listen on "`/Alice/home/frontdoor/doorlock/status`" and to perform automatons like "*If the door is unlocked, turn on the light of the porch*". Additionally, Alice could use a central controller

[1] `https://mosquitto.org/`

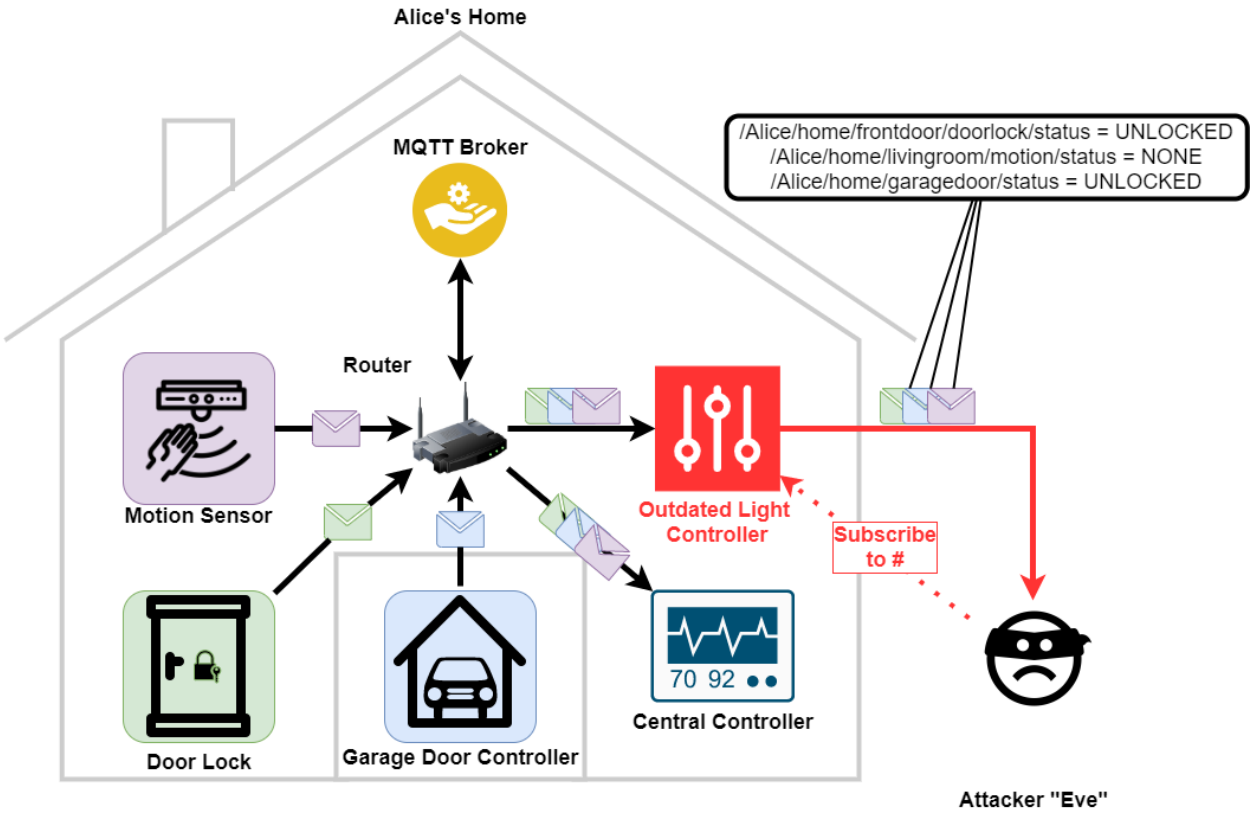


Figure 3.2: Attack scenario of attacker getting access to confidential information by subscribing to wildcard topic "#"

who listens to all the broadcast topics and conducts more complicated routines like "*If the motion sensor detects something is moving and Alice's phone's GPS location is not in the house, send a notification to Alice's phone*".

An eavesdropper Eve hacked into Alice's WiFi network and got access into an outdated light controller. Eve added a new subscription to the outdated light controller using the controller's client ID, and let it subscribe to topic "#". Eve was able to get updates like door lock status, as shown in Figure 3.2. Combined with other possible device information (lights or motion sensors), Eve drew the conclusions like "*Alice is not at home and door is unlocked*" or "*Alice is sleeping and garage door is unlock*". This attack scenario happens in real life [81, 82].

Eve noticed that Alice has a topic named "`/Alice/home/frontdoor/doorlock/command`" and the contents are "unlock" or "lock". Eve was able to apply a spoofing attack when Alice was not home. Eve used the outdated light controller to publish to that topic with the content "unlock" and successfully unlocked the front door, even the command is not from legitimate door lock controller.

Eve later entered Alice's house. To prevent Alice from **aggressively** checking the door lock status, Eve launched a tempering data attack. Now that Eve is physically inside the house, accessing the broker would be much easier in this case. Eve modified the cache inside the broker so that any content related to topic "`/Alice/home/frontdoor/doorlock/command`" would return "locked".

In order to prevent Alice's phone from **passively** receiving updates from the central controller, Eve used a device connecting to the broker with the same device ID as the central controller's, to perform a client takeover. Because the TCP connection between it and the broker was closed due to Eve's takeover, periodically the original central controller would try to reconnect. To prevent the original central controller from receiving any broadcast messages, Eve repeatedly perform takeover in a fast rate. This attack happens in real life as well [82]. After leaving the Alice's house, Eve unsubscribed to the topic "#" on the light controller and stop clients takeover on central controller.

After figuring out these had been a theft, Alice was trying to figure out four major questions:

① How did the thief know Alice was not home?

② How did the thief get in?

③ Why the door lock showed "locked" when Alice checked the door lock status?

④ Why hadn't the phone received any notification from the central controller?

The information that can be gathered by different mechanisms are shown in Table 3.1.

For Alice's question ①, from the vanilla verbose log, Alice could gather limited information. She noticed that the light controller client ID subscribed to topic "#" and received a lot of messages, including lights or motion sensors. However, because vanilla verbose log cannot provide the content, only knowing the topic is not sufficient to let Alice know the reason why the attacker knew she was not home. For example, the motion sensor always publishes to topic "`/Alice/home/livingroom/motion/status` with or without any detected movement, but the contents are different [83, 84]. This insufficient of evidence also applies to MQTT-Plan as well, as MQTT-Plan does not record message content. From the dummy subscriber mentioned above, Alice knows all the topics and contents that were exposed to the

attacker. Alice may figure out that Eve was using the motion sensor alongside with lights status to make the decision to rob the house at that time. However, Alice cannot figure out which subscriber gets those messages and therefore couldn't pinpoint which device was defected. With MQTTprov, Alice could know the subscriber ID (i.e., light controller), publisher IDs (i.e., all sensors), topics and contents. At this time, Alice could finally make the conclusion that the light controller received both statuses including motion sensor and lights, and this is how the attacker knew Alice was not at home.

For Alice's question ②, from the dummy subscriber, Alice could figure out there was a command "unlock" sent to topic "`/Alice/home/frontdoor/doorlock/command`". However, she is not able to know whether this command is accidentally sent from her phone or the attacker, because dummy subscriber does not provide publisher ID. With vanilla verbose log and MQTT-Plan, Alice knows one light controller sent some messages to "`/Alice/home/frontdoor/doorlock/command`". From both techniques, they are capable of fetching [topic, publisher ID] and [topic, subscriber ID] pairs. However, because of the message decoupling from MQTT nature, they both are not effective to provide the evidence of [publisher ID, topic, subscriber ID] for each message. In other words, it is needed to "couple" the decoupled publishers' and subscribers' messages. Moreover, without actual message content, this is not a persuasive evidence either. All the aforementioned criteria would be exposed with MQTTprov. See Section 3.1.2.2 for detailed discussion about re-coupling.

For Alice's question ③, the following requirements need to be met: 1) Alice needs both the contents sent to the broker and sent from the broker. 2) She needs to find a way mapping the contents so that every subscriber could find the source of the content. Dummy subscriber, vanilla verbose log and MQTT-Plan could not provide these features at all. MQTTprov is capable of doing such and comparing the mapped messages to notify any data tempering.

For Alice's question ④, with vanilla verbose log and MQTT-Plan, Alice is only able to know that the broker has forwarded the messages to her phone's client ID, and there are two devices (the attacker and Alice's actual phone) using her phone's client ID at the same time trying to connect to the broker. But because of the client takeover, they were both disconnecting and reconnecting while the other one was connected to the broker. From this, Alice could draw the conclusion that the other device was probably used by the attacker and this might be the reason the messages weren't received by her phone. However, she is not able to figure out what contents are sent to the attacker. With dummy subscriber or MQTTprov, Alice could extract all the messages that were meant to be sent to her phone's client ID. And then she could subtract the actual messages received by phone, from the messages that were meant to receive, to get the messages forwarded to the attacker. Noted that the contents are recorded in bytes in both mechanisms, a parser is needed to acquire the actual payload. This takeover attack will be discussed in detail in Section 3.1.3.1.

Table 3.1: The evidences needed to reconstruct each attack scenario, and the capability of each tool acquiring them

Alice's Questions	①			②				③			④	
	Topic	**Content**	**Sub ID**	**Topic**	**Content**	**Pub ID**	**Sub ID**	**Content into Broker**	**Content out from Broker**	**Compare Both Content**	**Topic**	**Content**
Dummy Sub listens to "#"	✓	✓		✓	✓						✓	✓
Vanilla verbose log	✓		✓	✓		✓	✓				✓	
MQTT-Plan	✓		✓	✓		✓	✓				✓	
MQTTprov	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

①: How did the thief know Alice was not home? ②: How did the thief get in? ③: Why the door lock showed "locked" when Alice checked the door lock status? ④: Why hadn't the phone received any notification from the central controller?

3.1.2 System Details

MQTTprov uses two layers to gather different information, making up the flaws of existing provenance techniques. A high-level overview of MQTTprov is depicted in Figure 3.3. It has two modules: network logger and library logger. Network logger captures the raw network traffic passing through MQTT broker, no matter it is sent or received by the broker, alongside with necessary information like IP address, port, client ID, etc. Library logger provides a higher level semantics for the network packages. Through library logger, it is able to retrieve the data provenance from messages. In other words, library logger is able to make causality connections between publishers and subscribers, which can be used later for detecting data tempering.

3.1.2.1 Implementing Broker or Client

We identify three fundamental drawbacks in providing data provenance in MQTT clients. First, to generate provenance data for diverse devices that use various hardware (e.g., CPUs) and software (e.g., OS, firmware), the proposed approach must be hardware and software agnostic. Second, modification on end devices is difficult to achieve in practice. For instance, some devices may not be physically accessible and some devices may use proprietary software which does not allow any modification. The proposed approach should be able to derive data provenance from such devices. Third, end devices are often less powerful hence it is difficult to expect they having sufficient processing power to generate provenance data and properly store it. Therefore implementing MQTT clients is very difficult and impractical. Fortunately, because MQTT is a centralized protocol, every network traffic is routed through MQTT broker. Therefore, implementing a MQTT broker would be the best place to monitor the traffic.

For the MQTT broker, there are many open-sourced and close-sourced MQTT implementations, with different programming languages used[2]. We choose Mosquitto because Mosquitto is the most popular MQTT broker [85]. It is an open-sourced broker maintained by Eclipse.

3.1.2.2 Mosquitto Implementation

Broker is the central server through which all the network traffic passes. As shown in Figure 3.3, network layer receives the network packages from system call `read` and forwards them to library layer, in which they get parsed with context. When handling broadcasting messages, library layer checks the context and forge message payload if previously received topic has any subscriber. Library layer then forwards the package to network layer and use the system call `write`.

Network We hook `read` and `write` system calls in `lib/netmosq.c` to monitor raw network traffic. This implementation method should be equivalent to monitoring network traffic in any place between the broker and clients. And these places are the actual system calls that are made by receiving/sending network data. By hooking these system calls, we could monitor the whole network traffic with minimum intrusion in the source code.

[2] `https://github.com/hobbyquaker/awesome-mqtt`

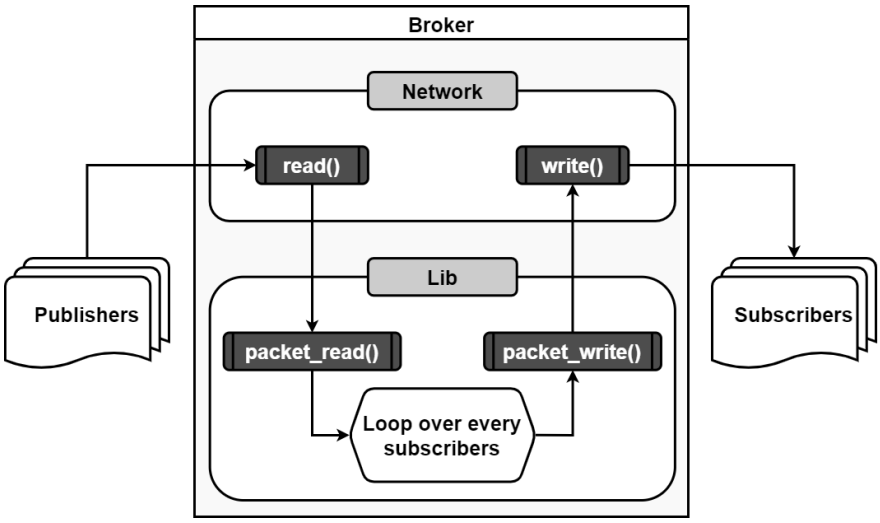


Figure 3.3: High level overview of MQTTprov

Mosquitto Lib Monitoring network traffic only is enough to get all the information defined in MQTT protocol. However, this information alone cannot fill the needs of mapping the messages to its source. To achieve this, we hooked `packet__write` and `packet__read` in `lib/packet_mosq.c`. In these methods, we could use the self-incrementing index to map source and destination.

Mapping source and destination The most important aspect of pub/sub model is the decoupling of the publisher of the message from the recipient (subscriber) [86]. That is, the subscriber does not know who the publisher is, and the publisher does not know whom the message is sent to. However in terms of data provenance, we need to re-couple the payloads to make a connection between publisher and subscriber. According to the implementation choice of Mosquitto, it uses a self-incrementing index to keep track of the messages that pass through library layer. Every message sent out by the publisher is assigned a unique index in its session lifetime.

3.1.2.3 MQTTcuties

In order to test if MQTTprov is capable of reconstructing the attack scenarios described in the motivating example in Section 3.1.1.2, and in other real life attack scenarios in Section 3.1.3, we need to create specific behavior patterns of clients to mimic the attack. For example, for evaluating the client takeover, one client must be connected to the broker first as "ClientID", and the simulated attacker client will then connect to the broker using "ClientID" as well. Current MQTT simulators [87, 88, 89] are only trying to create massive MQTT networks and stress test the broker. However, the customization of complex real life device behavior patterns is not available. Therefore we created our own MQTT client simulator, called MQTTcuties.

MQTTcuties is able to generate massive simulated clients to do defined behaviors. MQTTcuties is used in generate attack scenarios in Section 3.1.3. Individual clients can be defined the topics they interact with, their publishing behaviors, their subscription behaviors, and their generic timing. Clients can interact with the broker either through a low-level byte-level interface or a standard plain text format; this gives the clients more flexibility to launch attacks against the broker. A client's behavior is set through a minimal configuration file we call a recipe. Recipes contain the remote broker's information as well as the topics client will subscribe to and the publishing behaviors of the client. To create uniqueness and flexibility across clients, randomization, and adaptive activity timings have been designed into them.

These recipes are then read by the simulator where they will be given life. These clients will run on a resource sharing parallel model, where every set of clients will share one I/O stream, and there are many I/O stream available for scalability. The shared stream enables the maximization of resource utilization, while parallel streams will allow minimal wait times for clients. Each client can have any number of subscribers and publishers. publishers can be selected from one the following options :

3.1.3 Evaluation

In this section, we will discuss each attack that can be applied to MQTT, as shown in Table 3.2.

3.1.3.1 Denial of Service (DoS) Attacks

Persistence Session There's a subtle way to overload the broker by asking for persistent session every time using unique client ID [90]. Broker will first try to keep all information about all messages, including subscriptions, currently in-flight messages and retained messages in memory. If the persistent condition is met (for example, save on every N seconds or N events), broker will save these messages to a local database. Note that a client can subscribe to ANY topic even if this topic is not used by any publisher. Attacker could subscribe to a huge number of nonexistent topics with persistent sessions using a huge number of unique client ID. Because of the definition of persistent session, the broker stores the session indefinitely until next purging. That leaves a loophole for the attacker to fill the memory or disk and shutdown the broker, even with a slow attacking rate.

From network and library logs, we could extract connection/disconnection information with all the subscriptions. The pattern we are looking for is the pattern of large number of subscriptions linked to a large number of unique client ID. All other works cannot detect this attack because they are not logging this information.

Table 3.2: Different attacks that can accrue to the MQTT

Attack	Vulnerabilities	Dummy Sub	Verbose Log	MQTT-Plan[13, 14]	MQTTprov	Description
DoS	Persistence Session [90]				✓	Creating large amount of garbage persistent sessions
	Client Takeover [82]				✓	Client takeover bombing to prevent legitimate client from connecting
	Invalid Unicode[91, 92]				✓	Broadcasting invalidly encoded topic strings to all other clients
	CONNECT Packets [93]		✓	✓	✓	Sending multiple CONNECT packets to create half-opened sessions
	Payload [94]				✓	Spamming large size of payload
	QoS [93, 90]		✓		✓	Leveraging QoS$\geq$1
Identity Spoofing	Publisher Identifier		type 2)	type 2)	✓	Publishing MQTT messages using 1) non-existing or 2) existing client identifier
	Subscriber Identifier		✓	✓	✓	Subscribing to unauthorized topics
Information disclosure	Unsecured Broker	✓	partial		✓	Accessing the broker server and reveals confidential information
Elevation of Privilege	Wildcard Topics [81, 82]		partial	partial	✓	Subscribing to "#" topic will reveal confidential data
Tempering Data	Unencrypted Data	✓			✓	Altering the data cached in broker

Client Takeover Bombing Similar to Section 3.1.3.2, the attacker can also pretend to be other subscribers. If the attacker managed to connect to the broker using the same victim client ID repeatedly, it leads to Denial of Service (DoS) attack for the victim client. This could happen because client takeover is a essential feature supported by MQTT protocol [95] . There are two main reasons for this. The first reason is the presence of half-open connection. Once the client gets disconnected because of any reason (for instance, network issue), the connection between client and broker are now half-opened. In this case, client needs to reconnect to broker using the same client ID, and broker could close previous connection and re-connect with this client. The second reason is client upgrade. If a client needs a hard upgrade, the new client needs to connection to broker with same client ID as the old client, and broker should close the connection with old client and establish a new one with new client. However, client takeover can be used to launch DoS attacks on clients, as well as buffer overflow attacks across networks and devices [96].

This scenario can be detected by MQTTprov. If there is a client takeover, the log would show that broker closes the old client and establish a new connection with new client using the same client ID but different IP addresses (sockets).

Invalid Unicode Maggi et al. [91] pointed out a way to perform a DoS attack with invalid Unicode (U+0001..U+001F control characters; U+007F..U+009F control characters) (CVE-2017-7653) [92]. If the broker doesn't checks for disallowed UTF-8 code points in topic strings while other clients do, a malicious client would exploit this discrepancy and disconnect other clients by sending invalidly encoded strings in topic. This attack could be extracted from the network log since the log has all the binary data passing through the broker. Therefore any invalid Unicode is captured. Verbose log from vanilla broker and MQTT-Plan only records after the strings are decoded. Therefore they cannot present the actual invalidly encoded strings.

CONNECT packets If a client tries to connect to the broker and terminate the connection ungracefully (meaning the client ends the connection directly on its side without sending a DISCONNECT packet), the broker will maintain the half-open connection for a certain amount of time specifying by the broker's configuration. After this amount of time, broker will try to send a PING packet to the client, and terminate the connection on broker's side if PING is not answered. However, during the time of existing half-open connection, attacker could send multiple CONNECT packets using unique client IDs to a point the broker can't accept more connection requests [93]. CONNECT packets are recorded by verbose log, MQTT-Plan and MQTTprov.

Payload Attacker can exhaust the resource of the broker by dispatching multiple message accompanied by a large size of payload. CVE-2017-7651 [94] is an example of these attack. This attack is omitted by verbose log and MQTT-Plan as they don't record payload information. MQTTprov, however, logs the payload in binary to accommodate the situation that the payload is not a string.

Quality of Service (QoS) Because high QoS (QoS≥1) messages required additional steps to ensure message delivery, it requires more resources than a message with QoS=0. Attacker can send a large volume of messages with high QoS to exhaust the resources of the broker [93, 90]. This attack is not detectable by dummy subscriber and MQTT-Plan because QoS information is not recorded. Noted that MQTT-Plan omits control packets like PUBACK, PUBREC (Publish received - QoS 2 delivery part 1), PUBREL (Publish release - QoS 2 delivery part 2), and PUBCOMP (Publish complete - QoS 2 delivery part 3), therefore MQTT-Plan is not able to identify the QoS of the message. Verbose log, however, prints out those control packets. Similarly, MQTTprov logs all the packets, hence being able to reconstruct high QoS attack.

3.1.3.2 Identity Spoofing

Publisher Identifier There are two types of publisher identity spoofing: Type 1) attacker publishes to any topic using arbitrary client ID (even it doesn't exist in the MQTT network) as if the attacker is part of the MQTT network; Type 2) attacker pretends to be other legitimate publishers using their client ID.

- **Type 1) Attacker Publishing to Any Topic:** Because the publisher is anonymous to the subscribers as described in Section 3.1.2.2, anyone with the permission (or in other words, not restricted by the broker's rule) can publish to any topic.

  As we demonstrated the attack scenario in Section 3.1.1.2, the attacker Eve opened the door with arbitrary client ID and published "open" to topic "`/Alice/home/frontdoor/doorlock/command`". Because of the MQTT nature of anonymity, the door lock does not know who is the sender of the command.

  From MQTTprov, we could see there's a known door lock controller client ID publishing to "`/Alice/home/frontdoor/doorlock/command`" and other client IDs publishing to the same topic as well. The other client IDs could be the attacker's client IDs. Dummy subscriber, however, is at the same level with the door lock. Therefore it does not know the publisher of the command. This type of attack could be detected by verbose log and MQTT-Plan. Because they have the records of topic and publisher, both works is capable of telling unauthorized client ID publishing to any topic.

- **Type 2) Attacker Pretending to Be Other Publisher:** The only field that are used to identify a client is through either 1) client ID, 2) username-password combination, or 3) X509 client certificates [97, 98]. The third way is the most secured. But in reality, it is very hard to manage client certificates on a broker with large scale of devices. Especially, a report from Shodan [99] states that there are more than 49,000 MQTT misconfigured servers visible on the internet, including over 32,000 servers with no password protection. Notably, the transaction of username-password authentication is in plain texted, meaning any node in-between is able to intercept the packet and exposes username-password combination [100].

Once the attacker knows the client ID or username-password combination of a victim client, the attacker can claim itself to be that client using the same client ID or username-password combination. Even with X506 client certificates authentication method, as long as the attacker is able to acquire the certificates, it can pretend to be that client as well.

In Alice's case scenario, her door lock controller is using client ID "frontdoor_doorlock_controller". The attacker Eve can establish a new connection to the broker with the same client ID "frontdoor_doorlock_controller", and send out commands like "unlock". Note that this attack fundamentally different with "Attacker Publishing to Any Topic" mentioned above. "Attacker Publishing to Any Topic" attack does not need to know any existing client identity, while this attack does. In the case of broker only connecting to a predefined list of client IDs, the attacker can still perform this attack while the other attack won't succeed.

This problem is very hard to detect on the broker side. The reason is that with the same authentication identity, the only noticeable difference is the IP addresses or port numbers are different between the attacker and victim. However, as described in Section 3.1.3.1, client takeover it permitted and necessary. The attack procedure will look like this from MQTTprov logs:

1. Victim maintains a connection with broker using IP address 1.1.1.1 and broker is using port 7777;
2. Attacker establish a TCP connection to the broker using IP address 2.2.2.2 and the broker is using port 8888;
3. Attacker initializes the MQTT protocol with CONNECT control packet using same authentication identity of the victim;
4. Broker disconnects older connection (victim) from port 7777;
5. If the victim figures out it has been disconnected (thorough PING packet) and it tries to reconnect, it has to re-initate the TCP connection first. In this case, victim first connects to broker using IP address 1.1.1.1 and broker is using socket 9999 (possibly still be 7777 because it may be vacant).

Verbose log, MQTT-Plan and MQTTprov are all managed to provide this behavior pattern.

Subscriber Identifier Because subscriber client ID is captured in verbose log, MQTT-Plan and MQTTprov, any subscription to unauthorized topics is exposed using these techniques.

3.1.3.3 Information disclosure

Unsecured Broker A broker server may have security vulnerabilities that enable the attacker to access the broker server. In this case, assurance of the records for every messages that are sent to the broker is crucial, even if that message has no subscriber. This is crucial because in this attack, the attacker is interacting with the server itself. Memory, network traffic and kernel are presumably exposed to the

attacker. Therefore, we have to ensure that not only the traffic sent out by the broker, but also the traffic sent to the broker are all recorded. MQTT-Plan only captures the traffic when there is a re-publish, hence it misses the case that a message has no subscribers. Verbose log is able to record incoming messages, but it only keeps the topic in the log. Because dummy subscriber listens to all the topics, it equals monitoring all the incoming messages for the broker. MQTTprov is capable of reconstructing this attack through network layer logger.

3.1.3.4 Elevation of privilege

Wildcard Topics This type of attack is described in motivating example in Section 3.1.1.2.

3.1.3.5 Tempering data

Unencrypted Data This type of attack is described in motivating example in Section 3.1.1.2.

3.2 ChatterHub: Privacy Invasion via Smart Home Hub

The blooming of the Internet of Things (IoT) promotes massive smart-home devices to become connected to the Internet, with an estimate of 10 smart devices per home on average in 2020 [101]. We expect the number of installed smart-home devices to reach 75 billion by 2025. Smart-home devices promise to make the user's daily life more convenient. According to a recent study [102], the main reason for smart device purchase is convenience, as users can easily control and monitor smart-home devices over the Internet. Most smart-home devices can be accessed via smart apps on smartphones or smart-home platforms. e.g., *"Front door unlocked at 13:52 by code A", "Motion detected in living room at 17:03 ".*

However, this convenience comes at a cost. For example, an adversary with access to smart-home devices' state information (such as what is triggered or used and when), could acquire sensitive information about the users and their activities. These device states often contain the users' activities in their living space, and the adversary can exploit it to commit further offenses, such as burglary and aggravated robbery. Indeed, cybercriminals are increasingly targeting smart-home devices [103]. Recent studies [104, 105] demonstrated privacy invasion problems present in smart-home devices. For example, Peek-a-Boo [104] showed that attackers could identify smart-home devices' states and actions by passively listening to the wireless around a smart-home. Apthorpe et al. [105] showed an Internet Service Provider (ISP) could learn privacy-sensitive information from smart-home devices by analyzing traffic.

This work presents a novel method to attack smart-homes, called ChatterHub, enabling an adversary to infer smart home events and user activities by sniffing *encrypted* network traffic to/from a target home, even though devices are hidden behind a smart-hub (e.g., Samsung SmartThings [106]) and do not directly connect to the Internet. ChatterHub requires neither physical proximity to the target home nor prior knowledge of its setup (e.g., list or topology of smart-home devices), making attacks on smart-homes more feasible.

The intuition behind designing ChatterHub is that users' activity routine in a smart home can trigger smart devices, manifesting as distinct patterns in the network traffic, albeit encrypted, and hence the users' activities and smart devices' events are discoverable and learnable. To infer smart-home devices' events, ChatterHub employs a classification model trained with traffic patterns of popular smart-home devices and hubs. The adversary can further train ChatterHub with their own devices by providing network packet traces and event logs to the training platform. ChatterHub automatically partitions the network trace with our novel segmentation algorithms and feeds the segmented traces (with event labels parsed from the event logs) into machine learning models to detect smart home devices' events. This way, the attacker can infer the occupancy pattern of the home by analyzing the event timing and patterns.

We have evaluated the accuracy and effectiveness of ChatterHub on real-world testbed environments with Samsung SmartThings hub and 14 smart-home devices. The results show ChatterHub can successfully discover the capabilities and events of the devices, e.g., *lock*, *switch*, or *motion* based on their encrypted traffic, and reveal users' daily routines by tracking devices' activity, including changes in lock's state, smart LED's state (i.e., on→off, off→on), and multi-purpose sensor's states (i.e., detecting motion on doors or windows).

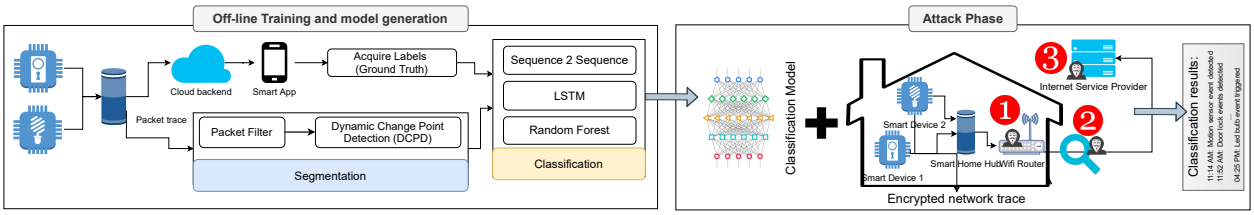


Figure 3.4: A high level overview of ChatterHub

In summary, this section makes the following contributions:

1. We explore a new adversarial approach against smart-home devices hidden behind a smart-hub, which could leak critical user's privacy, including households' daily routine.

2. We design a classification model that can accurately identify the events and usage patterns of various smart-home devices from encrypted network traffic.

3. We evaluate ChatterHub in three real smart-home environments. The evaluation results show that ChatterHub can successfully recognize smart-home devices' events with 88% F_1 score on average.

4. We show that a combination of packet padding and random sequence injection techniques can mitigate threats from ChatterHub at an average cost of 9.2MB traffic per day.

5. All the data sets, source code, and classification models used in this work are publicly available to the community[3].

3.2.1 Adversary Model, Assumption, and Goal

We assume that an attacker only passively sniffs encrypted network packets from/to the target home. In this work, we consider three potential points at which the attacker can eavesdrop on network traffic. First, the attacker can gain access to the traffic from a compromised router. Second, the attacker can eavesdrop on network traffic from the home router's uplink traffic. Third, the attacker can be the one who can monitor the network traffic of the target home, Considering these scenarios, encryption remains the only form of protection for users' data. Nonetheless, our adversary model is a passive attacker who collects encrypted network traffic (e.g., TLS/SSL). The attacker can only observe the size of each incoming and outgoing packet, the source and destination IPs, and timestamps. In addition, the attacker does not rely on decoding or interpreting the information inside traffic packets.

We also assume the attacker has access to a trained model or can collect his data from a hub and desired devices to train a model. However, the attacker does not require prior knowledge of a targeted smart-home topology or devices deployed.

[3] `https://github.com/karthikaS03/ChatterHub`

The Goal of the Adversary Once the network packet traces from the target home are obtained, the adversary proceeds to leverage a classification model, provided by ChatterHub or trained on the attacker's own hub and devices. By doing so, the adversary can understand the pattern of network traffic generated by the smart-home devices of interest. We consider that the attacker can achieve the following goals (but not limited to):

- **Scout Attack:** The attacker targets a range of IP addresses to find vulnerable home routers, similar to Mirai attack [107]. After gaining access to the routers, the attacker analyzes traffic either in the routers or through a virtual redirection to a sniffer installed device. Understanding smart-home devices' behaviors will allow the attacker to find vulnerable targets for a further offensive campaign, such as burglary.
- **Targeted Attack:** The attacker first gains access to network sniffing tools [108] and sniffs the outgoing traffic. After enough scouting, the attacker can understand the smart-home devices' behaviors, identify household activities patterns, and use the patterns for physical assault.
- **ISP-level Tracking:** Internet providers such as ISPs and VPNs who has complete access to users' traffic can learn the patterns of the households' daily life. Such information can be used for targeted advertising based on user behaviors or other activities, potentially violating users' privacy [73].

Target Devices In general, two types of smart-home devices are available on the market; 1) WiFi or Ethernet-enabled devices and 2) devices equipped with home automation network modules i.e., *Zigbee*, *Z-wave*, or Bluetooth Low Energy (*BLE*). The first type of device can directly connect to the access point. On the other hand, devices in the second category cannot connect to the Internet directly, so they require a smart-home hub to manage communications among devices. Additionally, since the second type of devices is hidden behind the hub, they are considered to be more secure against remote attackers [109]. A large body of work [105, 110, 111, 112] studied security and privacy of the first type of devices, while the security of home automation network devices (the second type of devices) has gained little attention. This work focuses on the second type for their high market share and diversity [113, 114].

3.2.2 System Design

Minimally intrusive monitoring is the most important goal of ChatterHub as the adversary only requires access to the network traffic from/to home. Obtaining access at this level is ascertained to be relatively simpler compared to using eavesdropping devices that have to be placed near the target devices [115, 116, 105].

Fig. 3.4 illustrates an overview and the control flow of ChatterHub. In ChatterHub's training, all the communication from the devices are transmitted through the hub. We collect these communication packets through 1) accessing the cloud backend logs and 2) monitoring the network traffic. Network traffic will be passed to a *segmentation* module, which separates network traces into sequences associated with events.

3.2.2.1 Training Data Collection

We first collect network packets to/from our smart-home setup and smart-devices' event logs, then label them for model training. In this work, we used 15 different devices with 12 unique capabilities as described in Table 3.3 (list of devices), and Table 3.4 (capabilities) shows events associated with each capability. In our dataset, an event is represented as the combination of a capability and its event (e.g., switch-on, lock-unlocked). We used the following setups for model training.

1. **Single device:** We connect a single device to the hub and observe the network traffic generated. This is to understand the unique traffic patterns generated by each device.
2. **Multiple devices:** We connect multiple devices to the hub and monitor traffic concurrently generated by all devices; we use this data to train our model with a more realistic setup. For example, we observed packets (generated by multiple device events) often overlapped each other. We connect not only smart-home devices to the hub, but also other home appliances (e.g., computers, tablets, smartphones) to the router to create more realistic traffic.
3. **Only the hub:** We also observe the network traffic from an isolated hub's (with no other devices attached) operations to understand the hub's behaviors (e.g., firmware update).

We connect `Wireshark` installed on a laptop to Samsung SmartThings hub through a bridged network to monitor the network traffic. We obtain event labels from the logs delivered through the hub. Samsung SmartThings hub stores event logs (e.g., all events and commands sent to/by smart-home devices along with timestamps). We collect the logs regularly by using "Simple Event Logger" [117] provided by the manufacturer. We have collected over 200,000 network packets from the smart-hub with over 60,000 event logs and use them for training the classification model in ChatterHub.

3.2.2.2 Trace Segmentation, Labeling, and Feature Extraction

We first design a method to filter out network packets that are not related to the SmartThings hub (i.e., packets generated by PCs or tablets), and then we perform packet segmentation.

Packet Segmentation When the hub is registering, it connects to an authentication server (*Auth-Server*) in the cloud. The hub exchanges authentication keys with *Auth-Server*, and receives an IP address of a communication server (*Comm-Server*) in the cloud. The hub is then connected to *Comm-Server* for further communications and operations. This communication channel between the hub and *Comm-Server* remains established, and hub relays the information through this channel. Suppose the devices communicate with *Comm-Server* directly via Wi-Fi. In that case, traffic is segmented based on each session, and each device's traffic can be separated based on their unique destination and source IP addresses. However, the challenge we encounter is that all communications go through the hub, and there is a lack of discerning parameters. Thus, it is not possible to partition packets based on the network flow information. Also, the communication interval in the sequence of packets between two events is relatively large compared to the interval between packets sent for a single event. Thus, we apply a segmentation method to

Table 3.3: List of devices and capabilities. Communication is shown by (Zigbee) for Zigbee and (Z-Wave) for Z-Wave. Capability references correspond to Table 3.4

Type	Device	Device Name	Cap.
Sens.	Multi Sensors	Centralite Micro Door Sensor (Zigbee)	9, B, 6
		Smartthings Multipurpose Sensor (Zigbee)	B, A, 6
		Samsung Multipurpose Sensor (Zigbee)	B, A, 6
	7 Sensors	Iris Smart 7 Sensor (Zigbee)	C, 6, 7
		Centralite 7 Sensor (Zigbee)	C, 6, 7
	3 Sensors	Centralite 3 Sensor (Zigbee)	3, 6, C
		Samsung 3 Sensor (Zigbee)	3, 6, C
Act.	Smart Lights	SYLVANIA Smart 10Y A19 TW (Zigbee)	4, 5, 8
		SYLVANIA Smart + Adjust. (Zigbee)	4, 5, 8
		Sengled Element Plus (Zigbee)	4, 5, 8
	Smart Plugs	Centralite Smart Outlet (Zigbee)	4
		Sylvania SMART+ Smart Plug (Zigbee)	4
	2s	Kwikset 10-1 Deadbolt (Z-Wave)	2, C
	Switches	OSRAM LIGHTIFY Dimming 4(Zigbee)	1
Hub	Hub	Samsung SmartThings Hub (Zigbee, Z-Wave)	ping

Table 3.4: Event types for Capabilities

Capabilities	# Events	Commands
button (1)	410	`push`, `held`
lock (2)	584	`lock`, `unlock`
motion (3)	406	`inactive`, `active`
switch (4)	1562	`on`, `off`
switchLevel (5)	3181	`change`
temperature (6)	790	`change`
water (7)	117	`dry`, `wet`
colorTemperature (8)	853	`change`
activity (9)	31	`online`, `offline`, `hub disconnection`
status (A)	572	`open`, `close`
contact (B)	708	`open`, `close`
battery (C)	16	`change`

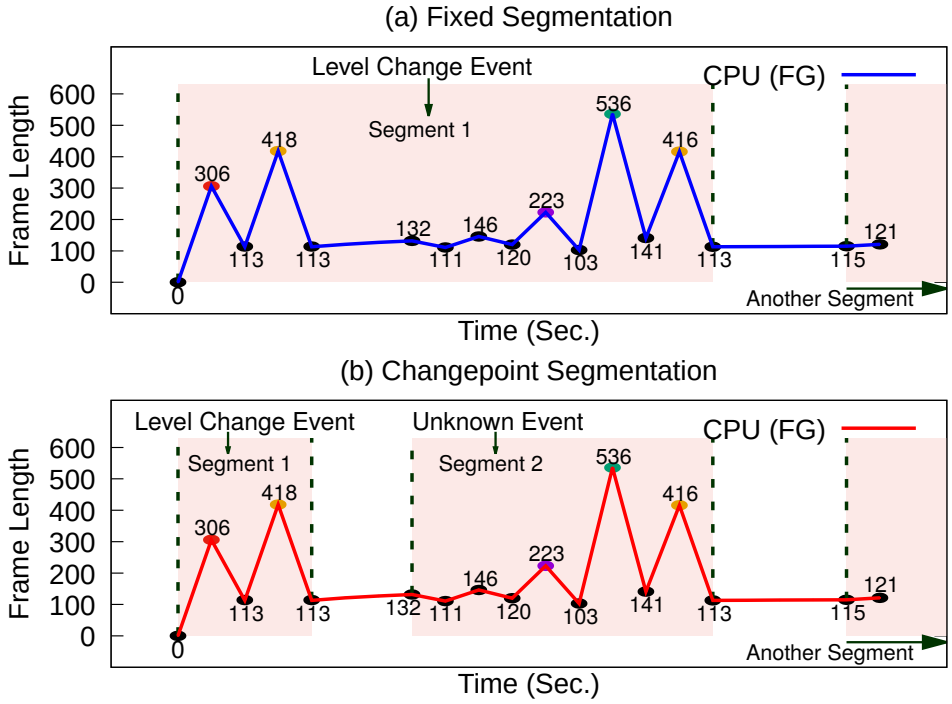


Figure 3.5: Fixed Segmentation vs. Changepoint Segmentation

divide the network traces into small bursts of packets. To segment the network flows into separate bursts, we try to leverage approaches from previous studies [118, 115] that use a fixed threshold of 4.5 seconds to segment network packets into multiple bursts. Previous works show that 4.5 seconds is enough for the communication between a client and server to complete packets exchange. However, we observed that the time gap between packet exchange for a single event could last longer than 4.5 seconds. Fig. 3.5 shows a case where a fixed-threshold approach fails to separate the *level change* event from other events. Also, events of the hub (e.g., ping, status) can occur along with other device events within an interval of shorter than 4.5 seconds. As such, the segmentation based on a fixed threshold often fails to correctly segment the device events from other packets (e.g., ping and status). Therefore, we develop a dynamic segmentation technique using *change point detection* [119] to segment these packets into bursts correctly.

Dynamic Change Point Detection (CPD) A *change point* is a temporal point when the statistical properties of its previous and subsequent time points are different. In our smart-home setup, the network packets for a single event are issued in short intervals compared to the intervals between two distinct events. Therefore, a change point will be when a sequence of packets for a single event starts or ends. Since our logs are collected over a long time, multiple change points need to be identified to segment all events. CPD is an approach to find abrupt changes in time-series [119]. CPD can also be used for estimating the temporal point when the statistical properties of a sequence change [120]. ChatterHub employs PELT

(Pruned Exact Linear Time) [120] because it is computationally efficient and outperforms other exact CPD search methods [121]. We present detailed evaluation results of Dynamic CPD and fixed threshold segmentation algorithms on our dataset in Section 3.2.3.1.

Labeling After we segment the packets from network traces into different bursts, we obtain event labels from the hub's logs. We use timestamps to align the labels and the segmented trace. However, we observe that slight time differences between *generation of event log* and *packet capture* can occur. Hence, we allow for ± 5-the second difference between the two; then, we map the event to a specific burst of packets. We also observe special cases where *a single user activity enables multiple events in a device*. For example, a "switch on" user event from the app triggers two events (switch-on and level-change). Therefore, a single burst of packets could be mapped to multiple events. We also observe that a number of segments are not associated with any labels (i.e., no logs from the hub and *Comm-Server*) so that we label them as *unknown*. To characterize the unknown packets, we further analyzed the source code of device handlers [122] and found that the handler generates the event logs, and some of the handlers do not emit any logs. We found that most of the missing events are less important for the user (e.g., device refresh, device ping).

Feature Extraction For feature extraction, we begin by forming a signature via fetching the frame length of multiple packets in each segment. We then use this signature as the feature for our classifier.

These signatures show a significant amount of collision across different classes. These collisions are the result of events happening in small intervals or events that some happen together, e.g., when a user opens a door, both contact-open and status-open events occur concurrently.

3.2.2.3 Classification Models

We train classification models using extracted network features to classify smart home network traffic into smart home devices' capabilities and events. Given the dynamic nature of the data, we consider the following machine learning models: 1) Random Forest, 2) OneVsRest classifier, and 3) SEQ2SEQ.

Random Forest (RF) Model RF constructs an ensemble of decision trees by taking a random subset of the features to decide a node split in building each tree. We only use RF as a baseline to identify better algorithms because RF largely depends on the training data's completeness.

OneVsRest Classifier A key characteristic of our data is that a single traffic segment may contain the data related to multiple capabilities that usually occur together or were subsequently activated. Therefore, we need a multi-class classifier that can identify all the classes in segmented traffic. As a result, we follow the *one-vs-rest strategy* that uses a classifier for each class fitted against all other classes. This method ensures that each classifier is independently optimized to identify features for the corresponding class. As this entails a large number of classifiers, we use XGBoost (Extreme Gradient Boosting) . XGBoost is an ensemble that applies Gradient boosting on decision trees to boost the performance of the various models [123, 124, 125]. In this project, we use the `XGBClassifier` of `XGBoost` library [126] with its default parameters.

We use `CountVectorizer` as vectorizer, with `ngram` range from 1 to 4 so that the relationship between the packets in the sequence is maintained. The output of the vectorizer is directly fed into the XGBoost model.

SEQ2SEQ Model Sequence-to-sequence (SEQ2SEQ) model solves sequential problems. The input to SEQ2SEQ is a series of data units, and the output is also a sequence of data units [127]. SEQ2SEQ model is applied to address various problems in multiple disciplines. Specifically, SEQ2SEQ caught our attention because of its application in natural language translation, for which the input is usually a sentence and the output is a sentence in a different language. In our model, we have a sequence of package lengths, and the output is a sequence of *events*. We use sequences of capabilities and events as labels. It is worth noting that SEQ2SEQ is a model to translate natural languages, so the order of the sequence will affect the result. We maintain the original order from ground truth, even if one label appears multiple times. The SEQ2SEQ framework contains two main components: an *encoder* and a *decoder*. The encoder reads the input, and the decoder translates the encoder's output to a final sequence of outputs [127].

3.2.3 Evaluation Results

We evaluate ChatterHub with real-world smart-home environments. In the smart-home setup, we deploy a set of smart-home devices and other Internet-connected devices (e.g., laptops, smartphones), and then we connect them to the hub.

3.2.3.1 Network Trace Segmentation

We perform trace segmentation on the captured traffic to partition the overall traffic flow between the hub and cloud servers (e.g., *Comm-Server*) into a set of small bursts, which map to specific commands. Therefore, ChatterHub first needs to identify the IP address of the target hub, and then it performs network trace segmentation, which will generate a set of proper packets related to a specific command/event from the devices at a time. An accurate segmentation will ensure that each packet burst contains a negligible amount of noise packets. Note that noise or noisy packets indicate *unknown* packets or packets for the hub's status report. The hub randomly sends these packets to the cloud servers.

Identifying the IP address of the Hub We monitor all network traffic from and to the target home router and identify the hub's IP using the pattern signature of "hub's ping" events. While the hub keeps changing the IP address of *Comm-server* from time to time (usually over days), we can successfully identify the IP address of *Comm-Servers*. Then, we can extract necessary traffic between the hub and *Comm-Server* (excluding the traffic from other devices in the home).

Network Trace Segmentation As we discussed in Section 3.2.2.2, we develop a PELT-based Dynamic Change Point Detection (CPD) algorithm to segment the network traffic.

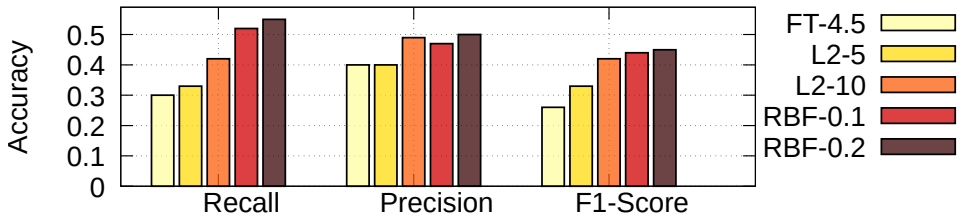


Figure 3.6: Performance Evaluation of various Segmentation Methods. (FT-4.5: Fixed Threshold with 4.5 seconds, L2-5: L2 Cost Penalty-5, L2-10: L2 Cost Penalty-10, RBF-0.1: RBF Cost Penalty - 0.1, RBF-0.2: RBF Cost Penalty - 0.2)

The PELT algorithm can be used with different cost functions, and it takes the output of the cost function as a penalty value, which affects the segmentation results. We compare the two most dominant cost functions, least squared deviation (L2) and kernalized mean change with radial basis function (RBF) kernel by running their output through our baseline model. Fig. 3.6 shows the results of classification for different parameters. We use PELT (RBF cost function and penalty value of 0.2), which achieved the best F_1 and precision.

3.2.3.2 Evaluation in Smart-home Environments

To evaluate ChatterHub in the real world, we set up three smart-home environments at three homes along with other devices and record the network traffic from their home router for a total of 10 days.

We train the classification models with data collected from the lab setting (explained in Section 3.2.2.1) plus the data obtained from one of three home configurations. We then test the model on data from two remaining smart homes not used for training.

After the model training, we conduct two experiments; 1) an attacker tries to infer the capabilities of devices, and 2) an attacker tries to detect specific events of those capabilities. For example, the attacker will be made aware of "switch" being present and used in the first experiment's target home. The attacker will then infer if a "switch on" or "switch off" has happened in the second experiment.

It is worth noting that when we test the classification models, we add more sensors and devices (e.g., water sensor), which do not exist in the training dataset, to two test smart-homes to test the scenario where the attacker does not have a list of installed devices in the target home.

Classification Accuracy We generate the ground truth for two different sets of labels (capabilities and events) so that we can train our classification models on both data sets to classify capabilities and events separately. Table 3.5 reports the classification accuracy (recall, and F_1-score) of events from each device, such as switch-on, switch-off, motion-active and motion-inactive. If some devices in the target home have not been used in the model training, ChatterHub categorizes the events and capabilities belonging

Table 3.5: Classification results for capabilities and events

Capabilities	**Random Forest**		**SEQ2SEQ**		**XGBoost**	
	R.	F_1	R.	F_1	R.	F_1
button-held	0.00	0.00	0.00	0.00	0.27	0.18
button-pushed	0.00	0.00	0.61	0.57	0.98	0.73
colorTemperature	0.23	0.37	0.17	0.27	1.00	0.96
contact-closed	0.25	0.32	0.29	0.30	0.40	0.44
contact-open	0.37	0.45	0.50	0.47	0.47	0.54
switchLevel	0.87	0.42	0.63	0.33	0.89	0.55
lock-locked	0.71	0.50	0.77	0.46	0.77	0.53
lock-unlocked	0.12	0.17	0.06	0.10	0.77	0.68
motion-active	0.08	0.12	0.10	0.13	0.48	0.17
motion-inactive	0.28	0.23	0.30	0.25	0.62	0.36
ping-ping	0.96	0.98	0.96	0.98	1.00	0.99
status-closed	0.49	0.57	0.50	0.52	0.69	0.80
status-open	0.60	0.63	0.80	0.66	0.77	0.74
switch-off	0.58	0.71	0.55	0.52	0.71	0.80
switch-on	0.13	0.17	0.29	0.18	0.32	0.38
temperature	0.63	0.25	0.07	0.10	0.77	0.37
unknown	0.59	0.73	0.94	0.92	0.95	0.90
F_1 Known Average	0.83	0.72	0.78	0.81	0.92	0.76
F_1 Average	0.69	0.73	0.88	0.88	0.93	0.82

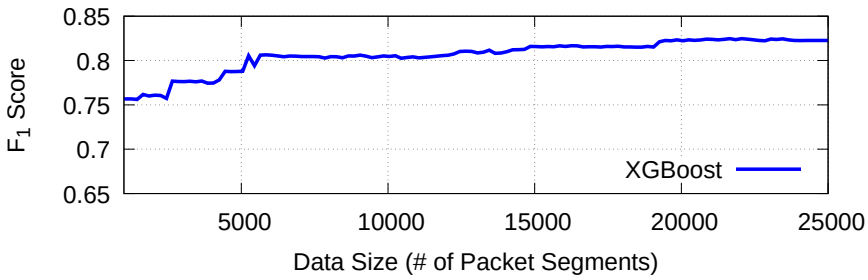


Figure 3.7: Effect of training set size to the F_1-score (XGBoost)

to this device as *unknown*. This is also observed from our results in the case of *water* capability, as shown in Table 3.6; where "0" for water sensor activities means water sensor was not used while training.

Overall, our classifiers generate multi-label outputs, indicating that a single segment of traffic packets can be classified into more than one class. Thus, our models identify multiple activities happening concurrently without an explicit time gap in the transmission of the network packets. The classification results reported in Table 3.5 are the accuracy of each class. To decide the models' overall performance, we calculate the micro average score for F_1 and recall (μAvg) [128], which takes into consideration the imbalanced class sizes. μAvg calculates a F_1-score across different classes by adding up their respective confusion quadrants. This shows how a multi-class model considers the whole class list to reduce bias towards their underlying class distribution. The average of F_1-score is calculated by $\frac{2\times\sum tp_i}{2\times\sum tp_i+\sum fp_i+\sum fn_i}$, where tp, tn, fp, fn indicate true positive, true negative, false positive, and false negative, respectively. fp_i denotes the number of false positives for the i^{th} class. We report this result as `Average` in Table 3.5. However, since the focus of our system is to detect *known* device activities, we calculate the μAvg for only the *known* classes and exclude *unknown* classes from the computation. The average results are reported as `known-average`.

Among three classification models, RF (the baseline model) shows the lowest recall, and precision 3.5. Overall, SEQ2SEQ gives us the highest F_1-score (0.81) for *known-average*, compared to the XGBoost model's F_1 result of 0.76. Although the XGBoost model has a higher individual F_1-score for some of the capabilities and events, the average F_1-score is lower because of higher false positive cases resulting in lower precision score. On the other hand, SEQ2SEQ shows higher precision results, indicating that SEQ2SEQ's accurate performance for identifying the events of devices. Based on this observation, this limitation of SEQ2SEQ in identifying some activities is in overlapped packet sequences. But XGBoost is more resilient to such noise in the data [129]. Hence, it shows higher accuracy in the presence of overlapped of packets. However, XGBoost's misclassification is a result of signature conflicts between multiple activities from a same device.

Effect of Training Data Size As this attack relies on the model to accurately classify traffic, we further analyze the impact of training data size on a fixed set of test data classification accuracy. Fig. 3.7 shows

Table 3.6: Classification results w/ and w/o a specific (Lock) device in different home setups. (R: Recall, P: Precision)

Capability	Case #1 : Home w/ Lock			Case #2 : Home w/o Lock		
	OneVsRest – XGBoost					
	R	P	F_1	R	P	F_1
Contact	0.88	0.35	0.50	0.78	0.45	0.57
Lock	**0.94**	**1.00**	**0.97**	-	-	-
Switch	0.76	0.94	0.84	0.61	0.85	0.71
Unknown	0.99	0.99	0.99	0.99	0.98	0.99
Average	0.90	0.82	0.83	0.80	0.76	0.75

how XGBoost F_1-score changes as the size of training data grows. The results show the accuracy increases with the size of training data. And, decent accuracy is possible with fewer training data.

3.2.3.3 In-Depth Analysis of Smart-Home Results

Our threat model is based on attackers' capabilities to monitor network traffic in a smart home to infer the smart home devices' activities and the user's behavior. Therefore, while our efforts are to create a model that works best in all scenarios, we demonstrate our model is useful for attackers to identify private information about the user and her home correctly. In this section, we explain such cases in detail.

Target Classification Model for Specific Devices We discuss how an attacker can use a specific model to obtain more accurate information on a targeted device from its capabilities. In this case, we train our XGBoost model only to detect three capabilities (contact, lock, and switch). To this end, we use a single trained XGBoost model and design two different evaluation test sets with intentionally deploying different devices (e.g., lock). In the first case setup (case #1), we create a smart-home testbed with all devices, including lock and water. In the second case (case #2), we create another smart-home testbed with all devices, including water except the *lock*. As shown in Table 3.6, in case #1, our model was able to detect *lock* with a precision of "1.00" indicating the model has no false-positives. Moreover, even if our model was trained on signatures of *lock*, in case #2 (the testbed without *lock* device), no lock capability was detected. Therefore, our model is highly accurate in detecting the presence of the devices.

Identifying Recurring Patterns Fig. 3.9 shows the activities of a smart lock at various times of day. We measured the device's activities for 5 consecutive days. The results show that at 11:00 and at 23:00, the lock had the events on multiple days at the same time. Based on this observation, the attacker can infer the homeowner's daily schedule. Hence, with further analysis of such patterns, the classification results could reveal information on the smart-home devices and the users.

Another example is the *switch-on/off* events reported in Table 3.5. F_1-scores of these events by XGBoost are 0.38 (*switch on*) and 0.80 (*switch-off*). Although F_1-scores are less than 0.8, ChatterHub can still identify user actions with light switches (e.g., user turning lights on/off). Fig. 3.8 shows ChatterHub correctly identifies 20 out of 25 events. ChatterHub only has three misclassifications (i.e., event *on* recognized as *off*, and vice versa) and two false detections (i.e., non-switch events recognized as switch events but part of the switch device itself). Further, the patterns of *on*/*off* events provide more confidence in the actual presence of a smart light in the home.

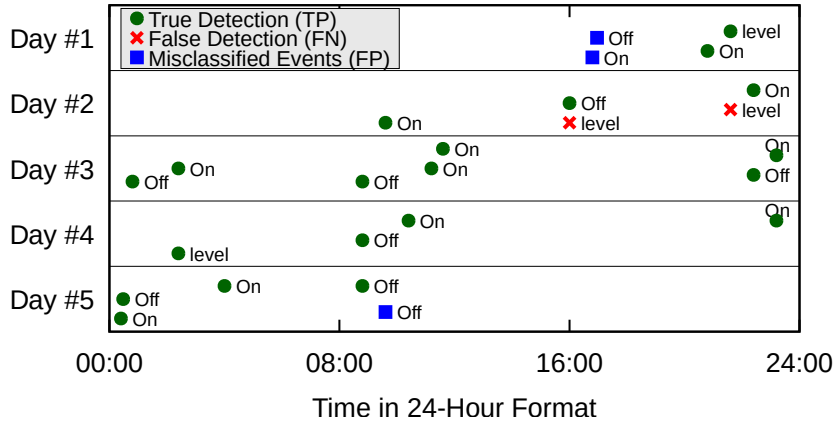


Figure 3.8: `Switch` events detected by ChatterHub.

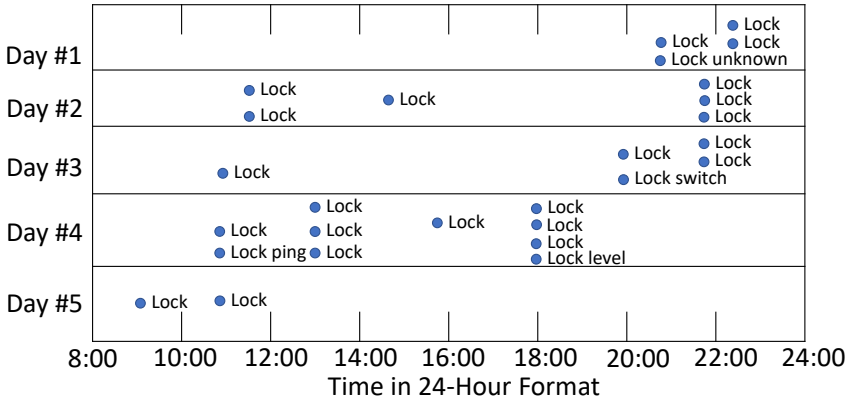


Figure 3.9: `Lock` events detected by ChatterHub.

Chapter 4

Limitations

In this chapter, we examine the limitations of proposed techniques. We first discuss the limitations of proposed forensic analysis techniques for Android, DroidForensics and PushAdMiner, in Section 4.1. In Section 4.2, we examine the limitations of proposed techniques for IoT environment.

4.1 Forensic Analysis in Mobile Environment

While DroidForensics and PushAdMiner enables semantically rich provenance trace for forensic analysis in Android, they have limitations as well.

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 [130, 131] that is orthogonal to the main focus of DroidForensics.

Second, DroidForensics uses `openat()` system calls to transfer API and binder logs (see the section 2.1.2.1). 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. `openat()` has three arguments and we use the first argument as an indicator of log type. For example, we are using -255 for API log and -256 for binder log. However, we can use the first argument as well as the third 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.

Fifthly, DroidForensics requires manual instrumentation to Android API functions. This limitation could be mitigate by developing more automated techniques to determine instrumentation points includ-

ing important call-back functions and handlers. For example, we can leverage DroidAPIMiner [132] to automatically identify instrumentation locations from important Android APIs, call-backs, and event handlers.

Finally, at the time when we started building our system, Puppeteer [69] did not appear to support Android Chromium automation. Only recently there have been online posts in which Puppeteer users describe how they have been able to "hack" their configurations to remotely control an Android browser. We therefore built our own browser automation framework that works via the Android Debug Bridge (ADB). The capabilities of our ADB-based automation framework are limited, but sufficient for enabling data collection for PushAdMiner.

4.2 IoT Security and Forensic Analysis

Because of the limited resourced obtainable in IoT environment, our approaches have their limitations.

First, a kernel-level attack could disable MQTTprov. The attacker has the capability to tamper our log, or even disable the logging if the kernel is compromised.

Second, because MQTTprov records the payload of each message, the runtime overhead and space overhead are high. However, research has the option to omit payload with the trade-off of losing the evidence to reconstruct certain attacks (e.g., wildcard topic subscription attack). The runtime overhead is caused by accessing user space memory multiple times. If we are permitted to modify the server kernel other than just MQTT broker, memory access could be faster. A ring-buffer can be introduced for enabling asynchronous writes for the logs to reduce the runtime overhead as well. Space overhead can be reduced by periodically compressing the generated log.

Finally, as we discussed in Section 3.2.2.1, the overlapping of packet sequences is one of the major challenges to accurate classification in ChatterHub. Suppose the target home has a larger number of smart-home devices than our experiment setup. In that case, there will be more chances for overlapping of packet sequences, implying that ChatterHub's classification results can be less accurate. However, our setup conservatively constitutes realistic smart-home setups as we deploy many devices (14+) that repeatedly generate network traffic, so we believe there will be minimal impact on the classification accuracy with more devices. Another potential limitation is when the target home has multiple devices of the same type, ChatterHub cannot tell which one contributes to the detected capability. For example, if the target home has two identical smart lock devices installed on two separate doors, the attacker would be able to recognize all the lock activities but cannot distinguish one lock from the other.

Chapter 5

Related Work

5.1 Forensic Logging

Tracking system-level dependence is a popular technique for attack analysis in desktop and server environments [5, 133, 134, 135, 136, 137, 138, 139]. 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 [6], ProTracer [7] and WinLog [140] 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. However, our approach 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 [141] 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 [9] 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 [11]. 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). For example, if an application invokes a SQLite query to a table loaded in the memory, the query does not cause any system calls.

DroidScope [10] 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 [11] emulator and it generally incurs high overhead. DroidForensics

supports ART environments, and is not rely on QEMU or other emulated environments but directly works on real devices.

Quire [142] monitors Android binder calls to detect confused deputy problem. It track privileges across inter-process boundaries. Grover et al. [143] propose an application-level technique to monitor user activities such as application install and removal, web browser history, calendar, call log or contact lists.

5.2 Android Taint Tracking

Dynamic taint tracking and information flow analysis techniques for Android [21, 22, 23, 24] 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 [11]. 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.

5.3 Other Android Analysis Techniques

Static Analysis techniques [37, 38, 144, 22, 145] 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 results as a hint and enhance runtime forensic logging for potentially malicious code region.

Android memory forensics techniques [146, 147, 148] 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.

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 [149, 150, 151, 152]. RERAN [149] and Mosaic [150] 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 [153, 154, 155, 156], as they only record stream inputs. VALERA [151] 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 [31, 32, 33, 34] and dynamic code loading [26, 27, 28, 29]. Mobiplay [152] is a client-server based recording and replay system. Android applications run on a Mobiplay 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.

5.4 Data Provenance for IoT

Data provenance for IoT systems has been a research topic for many years. It is also discussed in file system [157], database [158, 159] and sensor networks [160]. Markovic et al. [161] proposes EP-Plan, a vocabulary for linking the different levels of granularity of a plan with their respective provenance traces in general IoT environment. In the research of data provenance for MQTT, Markovic et al. [13, 14] tries to improve MQTT data provenance using MQTT-Plan, EP-Plan and PROV-O. Its goal is to trace whether a message was forwarded to an unauthorised client, according to predefined "plan". However, it can't handle most of the attack scenarios described in Section 3.1.3. Notably, their approach only captures a provenance event when the broker re-publishes (a.k.a., forwards) a message. Therefore any unauthorized publishing event without any subscriber is missed (e.g., DoS attack using CONNECT packets). On contrary, our approach MQTTprov is able to reconstruct these attacks.

Colombo et al. [162] tried to enforce access control on MQTT brokers. It can specify rules regulating the rights to receive/publish messages on specific topics. It may help preventing some attacks beforehand. However, in the scenario of client takeover bombing, it cannot stop the attacker from receiving the messages instead of legitimate subscriber, as they are using the same client ID. MQTTprov is able to extract the attack pattern.

Andy et al. [100], Potrino et al. [93] and Hintaw et al. [163] present attack scenarios over MQTT protocol and solutions to mitigate these attacks. However, it is generally hard to safeguard the system from future attacks. MQTTprov provides such capability to reconstruct an attack when it indeed happens.

5.5 WiFi Enabled IoT Devices

Most of the research studies that follow fingerprinting smart-home devices from network traffic focus on independent devices that directly connect to WiFi [164, 165, 166, 167, 168, 169] unlike our setup where devices are connected to a central device (hub). These studies also requires tapping to the local network for information on individual devices [170, 171] unlike our threat model where the network tapping can be acquired remotely.

Pingpong [172] proposes packet-level network traffic analysis to identify activities of smart-home devices. Similar to ChatterHub, Pingpong analyzes packet-level traffic to create unique signatures for smart home devices' activities. However, they only study WiFi-connected devices, but our focus is smart-home devices hidden behind the hub, increasing the complexity of network traffic. We observe that many security-critical devices (e.g., smart lock, motion sensor, smart switch) are hidden behind the hub to be more secure.

5.6 Insider Analysis

HoMonit [115] is a smart-home monitoring system that identifies misbehaving smart apps. By using this system, encrypted network traffic between the hub and smart-home devices has been analyzed to fingerprint each device. However, it requires tapping into the network between the hub and the devices, which is not piratical in real life scenario. Similarly, Peek-a-Boo [104] focuses on capturing the traffic between devices and a hub. Notably, ChatterHub focuses on the communication between a hub and the cloud servers (e.g., *Auth-Server* and *Comm-Server*). Due to hub devices' inter-operability, fingerprinting devices by analyzing the hub and server communications becomes complicated and difficult, compared to the encrypted traffic analysis done on HoMonit. Also, Zhou et al. [173] investigated potential security flaws in communications between the smart-home devices and the cloud servers, but ChatterHub does not focus on identifying smart-home devices' activities inferred from encrypted traffic.

5.7 Smart-home Applications

A number of studies focus on the security analysis and improvement for smart-home applications [174, 175, 176, 114, 177, 178, 179] where they discovered security vulnerabilities, i.e., private data leakage, privilege abuse, and malicious activities. Other studies focus on the analysis of information flow among smart apps, cloud backend, and IoT devices to discover vulnerabilities in the chain of information transfer [180, 110, 181, 182]. While existing solutions mainly focus on preventing the leak of sensitive data from the context of smart apps, cloud blackened, and/or the smart-home platform, ChatterHub demonstrates that an adversary can still infer activities and states of smart-home devices by eavesdropping on encrypted network traffic.

Chapter 6

Conclusion

The increasing number of mobile and smart devices offers new cyber channels for attackers, and thus, it is imperative for us to conduct effective security and forensics analysis on these smart devices for hardening the platforms and preventing future attacks. In this dissertation, we focus on developing techniques to identifying and automatically collecting logs that are essential to reconstruct the attack, from the mobile and IoT environments during runtime, and perform security and forensic analysis. We also present an attack that could be used to breach user's privacy in a smart-home environment.

We present 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. Our techniques point out that collecting selective of events is sufficient enough to accurately reconstruct the attack. We use real-world examples alongside with crafted never-seen-before attacks to illustrate and evaluate out contributions.

To help harvesting logs from Android Chromium web browser, we propose PushAdMiner and an Android helper application. We have studied how web push notifications (WPNs) are being used to deliver ads, and measured how many of these ads are used for malicious purposes. PushAdMiner, both on desktop and mobile environments, allowed us to automatically collect and analyze 21,541 WPN messages across thousands of different websites. Among these, our system identified 572 WPN ad campaigns, for a total of 5,143 WPN-based ads, of which 51% are malicious. We also found that traditional ad-blockers and

malicious URL filters are remarkably ineffective against WPN-based malicious ads, leaving a significant abuse vector unchecked.

We present MQTTprov, a multi-layered and automated provenance trace collection tool for IoT environment, specifically, MQTT network. We prove that even in an environment like IoT with very limited places to monitor, it is still possible to reconstruct most of the attacks in real life. We evaluate MQTTprov's capability of reconstructing real world attacks with real-world attack scenarios. We thoroughly compare our work with existing techniques.

We propose ChatterHub, a novel attack method that can correctly identify smart-home devices' capabilities with passive sniffing of encrypted home network traffic. With ChatterHub, an attacker does not need any prior knowledge of the target home. Our evaluation results from three realistic smart-home environments show that the attacker can successfully recognize smart-home devices' capabilities from the encrypted network traffic. This, in turn, leads the attacker to discover device behaviors, such as door being locked or motion in the room. Such information can be used to reveal a household's daily routine.

Bibliography

[1] *Malware Statistics & Trends Report | AV-TEST*. https://www.av-test.org/en/statistics/malware/. (Accessed on 07/10/2021).

[2] *McAfee Mobile Threat Report*. https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf. (Accessed on 07/11/2021).

[3] *How a smart home could be at risk from hackers – Which? News*. https://www.which.co.uk/news/2021/07/how-the-smart-home-could-be-at-risk-from-hackers/. (Accessed on 07/11/2021).

[4] *RedHat Linux Audit*. https://people.redhat.com/sgrubb/audit/.

[5] Samuel T. King and Peter M. Chen. "Backtracking Intrusions". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. 2003.

[6] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. "High Accuracy Attack Provenance via Binary-based Execution Partition". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2013.

[7] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2016.

[8] David Devecsery et al. "Eidetic systems". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 525–540.

[9] Kimberly Tam et al. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2015.

[10] Lok Kwong Yan and Heng Yin. "DroidScope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Usenix Security. 2012.

[11] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. USENIX Association, 2005.

[12] *15 Must-Know Web Push Notification Statistics*. https://gravitec.net/blog/15-must-know-web-push-notification-statistics/. (Accessed on 07/11/2021).

[13] Milan Markovic et al. “Towards transparency of iot message brokers”. In: *International Provenance and Annotation Workshop*. Springer. 2018, pp. 200–203.

[14] Milan Markovic and Peter Edwards. “Enhancing Transparency of MQTT Brokers For IoT Applications Through Provenance Streams”. In: *Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things*. 2019, pp. 17–20.

[15] *Top 5 IoT Messaging Protocols | Build5Nines*. https://build5nines.com/top-iot-messaging-protocols/. (Accessed on 07/11/2021).

[16] *Which Are the IoT Messaging Protocols? - DZone IoT*. https://dzone.com/articles/which-are-the-iot-messaging-protocols. (Accessed on 07/12/2021).

[17] • *Mobile OS market share 2021 | Statista*. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. (Accessed on 07/10/2021).

[18] *AuditdAndroid*. https://github.com/nwhusted/AuditdAndroid.

[19] *TROJAN:ANDROID/AVPASS.C*. https://www.f-secure.com/v-descs/trojan_android_avpass_c.shtml.

[20] *Sysdig*. http://www.sysdig.org/.

[21] William Enck et al. “TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. 2010.

[22] Yinzhi Cao et al. “EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework”. In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2015.

[23] Mingyuan Xia et al. “Effective Real-Time Android Application Auditing”. In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP ’15. 2015.

[24] Mingshen Sun, Tao Wei, and John C.S.Lui. “TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. CCS. 2016.

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

[26] Sebastian Poeplau et al. “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. NDSS ’14. 2014.

[27] Yury Zhauniarovich et al. “StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications”. In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. CODASPY ’15. ACM, 2015.

[28] Luca Falsina et al. “Grab ’N Run: Secure and Practical Dynamic Code Loading for Android Applications”. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC 2015. ACM, 2015.

[29] Vaibhav Rastogi et al. “Are These Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. NDSS ’16. 2016.

[30] *Binder IPC Mechanism*. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html`.

[31] Mengtao Sun and Gang Tan. “NativeGuard: Protecting Android Applications from Third-party Native Libraries”. In: *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*. WiSec ’14. ACM, 2014.

[32] Vitor Afonso et al. “Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. NDSS ’16. 2016.

[33] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. “Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks”. In: *Trans. Info. For. Sec.* (2014). URL: `http://dx.doi.org/10.1109/TIFS.2013.2290431`.

[34] Collin Mulliner, William Robertson, and Engin Kirda. “VirtualSwindle: An Automated Attack Against In-app Billing on Android”. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS ’14. ACM, 2014.

[35] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. ACM, 2014.

[36] Michael Grace et al. “Systematic Detection of Capability Leaks in Stock Android Smartphones”. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. NDSS ’12. 2012.

[37] Long Lu et al. “CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. ACM, 2012.

[38] Christopher Mann and Artem Starostin. “A Framework for Static Detection of Privacy Leaks in Android Applications”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. ACM, 2012.

[39] Fengguo Wei et al. “Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. ACM, 2014.

[40] Yajin Zhou and Xuxian Jiang. "Detecting Passive Content Leaks and Pollution in Android Applications". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. NDSS '12. 2012.

[41] *DTrace*. http://dtrace.org/blogs/.

[42] *Using the Linux Kernel Tracepoints*. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt/.

[43] *PCMark for Android*. https://www.futuremark.com/benchmarks/pcmark-android/.

[44] *TabletMark*. https://bapco.com/products/tabletmark/.

[45] *3DMark*. https://www.futuremark.com/benchmarks/3dmark/android/.

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

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

[48] *Contagio Mobile*. http://contagiominidump.blogspot.com.es/.

[49] *Brave Ad Block*. https://brave.com. (Last accessed Sep.17, 2020). 2020.

[50] *Ad Block Plus*. https://adblockplus.org. (Last accessed Sep.17, 2020). 2020.

[51] Joseph Medley. *Web Push Notifications: Timely, Relevant, and Precise*. https://developers.google.com/web/fundamentals/push-notifications. (Last accessed Nov.1, 2019). 2019.

[52] Wikipedia. *What is Banner Blindness?* https://en.wikipedia.org/wiki/Banner_blindness. (Last accessed Nov.11, 2019). 2019.

[53] Panagiotis Papadopoulos et al. "Master of Web Puppets: Abusing Web Browsers for Persistent and Stealthy Computation". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019.

[54] *Richpush Ad Network*. https://richpush.co. (Last accessed Sep.17, 2020). 2020.

[55] Bo Li et al. "JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions". In: *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. 2018.

[56] Phani Vadrevu and Roberto Perdisci. "What You See is NOT What You Get: Discovering and Tracking Social Engineering Attack Campaigns". In: *Proceedings of the Internet Measurement Conference*. ACM. 2019, pp. 308–321.

[57] Jiyeon Lee et al. "Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1731–1746.

[58] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. "Dial One for Scam: A Large-Scale Analysis of Technical Support Scams". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017.

[59] *Google Safe Browsing : Blocklisting Platform*. https://safebrowsing.google.com/. 2020.

[60] *Virus Total: Blocklisting Platform*. https://www.virustotal.com/. 2020.

[61] *Using Service Workers*. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers. (Last accessed Sep.17, 2020). 2020.

[62] *Notifications API*. https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API. (Last accessed Sep.17, 2020). 2020.

[63] *Using Application Cache*. https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache. (Last accessed Sep.17, 2020). 2020.

[64] *Introduction to Service Worker*. https://developers.google.com/web/ilt/pwa/introduction-to-service-worker. (Last accessed Sep.17, 2020). 2020.

[65] *Push API*. https://developer.mozilla.org/en-US/docs/Web/API/Push_API. (Last accessed Sep.17, 2020). 2020.

[66] *Introduction to Push Notifications*. https://developers.google.com/web/ilt/pwa/introduction-to-push-notifications. (Last accessed Sep.17, 2020). 2020.

[67] Google. *Set up a JavaScript Firebase Cloud Messaging client app*. https://firebase.google.com/docs/cloud-messaging/js/client. (Last accessed Nov.1, 2019). 2019.

[68] Selenium. *Selenium: Web Browser Automation Tool*. https://www.seleniumhq.org/. (Last accessed Nov.11, 2019). 2019.

[69] Google. *Puppeteer: Chormium Browser Automation Tool*. http://liwc.wpengine.com/compare-dictionaries/. (Last accessed Nov.11, 2019). 2019.

[70] Jordan Jueckstock and Alexandros Kapravelos. "VisibleV8: In-browser Monitoring of JavaScript in the Wild". In: *Proceedings of the Internet Measurement Conference*. IMC '19. Amsterdam, Netherlands, 2019, pp. 393–405. ISBN: 978-1-4503-6948-0.

[71] M. Zubair Rafique et al. "It's Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services". In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. 2016.

[72] *Cost per mille*. https://en.wikipedia.org/wiki/Cost_per_mille. (Last accessed Sep.17, 2020). 2020.

[73] *The State of Push Notification Advertising*. https://www.izooto.com/hubfs/TheStateofPushNotificaiZootoreport.pdf. (Last accessed Sep.17, 2020). 2020.

[74] Timothy Vidas and Nicolas Christin. "Evading Android Runtime Analysis via Sandbox Detection". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14. Kyoto, Japan: ACM, 2014, pp. 447–458. ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590325. URL: http://doi.acm.org/10.1145/2590296.2590325.

[75] Elie Bursztein et al. "Picasso: Lightweight device class fingerprinting for web clients". In: *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM. 2016, pp. 93–102.

[76] Alexei Bulazel and Bülent Yener. "A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web". In: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*. ACM. 2017, p. 2.

[77] LBE Tech. *Parallel Space - Multiple accounts and Two face*. http://parallel-app.com/. (Last accessed Nov.1, 2019). 2019.

[78] *Easylist*. https://easylist.to/. (Last accessed Sep.17, 2020). 2020.

[79] *Google Quiet UI for Notifications*. https://blog.chromium.org/2020/01/introducing-quieter-permission-ui-for.html. (Last accessed Sep.17, 2020). 2020.

[80] *45 Fascinating IoT Statistics for 2021 | The State of the Industry*. https://dataprot.net/statistics/iot-statistics/. (Accessed on 07/12/2021).

[81] *Open MQTT Servers Raise Physical Threats in Smart Homes | Threatpost*. https://threatpost.com/open-mqtt-servers-raise-physical-threats-in-smart-homes/136586/. (Accessed on 08/20/2020).

[82] *Tripwire Research: IoT Smart Lock Vulnerability Spotlights Bigger Issues*. https://www.tripwire.com/state-of-security/featured/tripwire-research-iot-smart-lock-vulnerability/. (Accessed on 04/14/2021).

[83] *MQTT Room Presence - Home Assistant*. https://www.home-assistant.io/integrations/mqtt_room/. (Accessed on 06/29/2021).

[84] *MQTT motion detection | mqtt-motion-detection*. https://thenailz.github.io/mqtt-motion-detection/. (Accessed on 06/29/2021).

[85] *MQTT Brokers Hosting Guide*. http://www.steves-internet-guide.com/mqtt-hosting-brokers-and-servers/. (Accessed on 04/12/2021).

[86] *Publish & Subscribe - MQTT Essentials: Part 2*. https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/.

[87] *MQTT Simulator*. https://www.gambitcomm.com/site/mqttsimulator.php. (Accessed on 07/04/2021).

[88] *mqtt-spy*. http://kamilfb.github.io/mqtt-spy/. (Accessed on 07/04/2021).

[89] *Softblade*. https://softblade.de/en/welcome/. (Accessed on 07/04/2021).

[90] Bogdan-Cosmin Chifor, Ion Bica, and Victor-Valeriu Patriciu. "Mitigating DoS attacks in publish-subscribe IoT networks". In: *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE. 2017, pp. 1–6.

[91] Federico Maggi, Rainer Vosseler, and Davide Quarta. "The fragility of industrial IoT's data backbone". In: *Trend Micro Inc., https://www. trendmicro. com/vinfo/us/security/news/internet-of-things/mqtt-and-coap-security-and-privacy-issues-in-iot-and-iiot-communication-protocols* (2018).

[92] *CVE - CVE-2017-7653*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7653.

[93] Giuseppe Potrino, Floriano De Rango, and Amilcare Francesco Santamaria. "Modeling and evaluation of a new IoT security system for mitigating DoS attacks to the MQTT broker". In: *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE. 2019, pp. 1–6.

[94] *CVE - CVE-2017-7651*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7651. (Accessed on 06/18/2021).

[95] *Your MQTT Applications: Are they resilient enough?* https://www.hivemq.com/blog/are-your-mqtt-applications-resilient-enough/#takeover. (Accessed on 10/01/2020).

[96] Min Chen et al. "A survey of recent developments in home M2M networks". In: *IEEE Communications Surveys & Tutorials* 16.1 (2013), pp. 98–114.

[97] *X509 Client Certificate Authentication - MQTT Security Fundamentals*. https://www.hivemq.com/blog/mqtt-security-fundamentals-x509-client-certificate-authentication/.

[98] *Generate client certificates for test.mosquitto.org*. https://test.mosquitto.org/ssl/.

[99] *Security Researchers Find Vulnerable IoT Devices and MongoDB Databases Exposing Corporate Data*. https://blog.shodan.io/security-researchers-find-vulnerable-iot-devices-and-mongodb-databases-exposing-corporate-data/. (Accessed on 07/11/2021).

[100] Syaiful Andy, Budi Rahardjo, and Bagus Hanindhito. "Attack scenarios and security analysis of MQTT communication protocol in IoT system". In: *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE. 2017, pp. 1–6.

[101] Danny Yuxing Huang and et al. "IoT Inspector: Crowdsourcing Labeled Network Traffic from Smart Home Devices at Scale". In: *IMWUT* (2020).

[102] Parks Ass. *Evolution of Smart Home and the Internet of Things*. https://bit.ly/3bzk9sr. 2020.

[103] A. Ng. *IoT Attacks are Getting Worse – and No One's Listening*. https://tinyurl.com/ydf7ma9b. 2018.

[104] A. Acar, H. Fereidooni, and et al. "Peek-a-Boo: I See Your Smart Home Activities, Even Encrypted!" In: *CoRR* abs/1808.02741 (2018).

[105] N. Apthorpe and et al. "Spying on the Smart Home: Privacy Attacks and Defenses on Encrypted IoT Traffic". In: *CoRR* (2017).

[106] Samsung Elec. *SmartThings*. `https://www.smartthings.com/`. 2020.

[107] M. Antonakakis and et al. "Understanding the Mirai Botnet". In: *USENIX Security*. 2017.

[108] N. T. Anh and R. Shorey. "Network Sniffing Tools for WLANs: Merits and Limitations". In: *IEEE ICPWC*. 2005.

[109] Zigbee Alliance. "Zigbee: Securing the Wireless IoT". In: (2017).

[110] O. Alrawi et al. "SoK: Security Evaluation of Home-Based IoT Deployments". In: *IEEE S&P*. 2019.

[111] C. Lee and et al. "Securing Smart Home: Technologies, Security Challenges, and Security Requirements". In: *IEEE CNS*. 2014.

[112] Y. Xiao and et al. "Security Services and Enhancements in the IEEE 802.15.4 Wireless Sensor Networks". In: *IEEE GLOBECOM*. 2005.

[113] M. Ammar, G. Russello, and B. Crispo. "Internet of Things: A survey on the security of IoT frameworks". In: *J. Info. Sec. and App.* 38 (2018).

[114] E. Fernandes, J. Jung, and A. Prakash. "Security Analysis of Emerging Smart Home Applications". In: *IEEE S&P*. 2016.

[115] W. Zhang, Y. Meng, and et al. "HoMonit: Monitoring Smart Home Apps from Encrypted Traffic". In: *ACM CCS*. 2018.

[116] K. Meng, Y. Xiao, and S. V. Vrbsky. "Building a Wireless Capturing Tool for WiFi". In: *Security and Communication Networks* (2009).

[117] *Simple Event Logger*. `https://bit.ly/3v3GdDg`. 2020.

[118] V. F. Taylor and et al. "AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic". In: *Euro S&P*. 2016.

[119] Y. Kawahara and M. Sugiyama. "Sequential Change-Point Detection Based on Direct Density-Ratio Estimation". In: *SDM*. 2009.

[120] G. Wambui and et al. "The Power of the Pruned Exact Linear Time (PELT) Test in Multiple Changepoint Detection". In: *AJTAS*. 2015.

[121] R. Killick and et al. "Optimal Detection of Changepoints with a Linear Computational Cost". In: *J. American Statistical Association*. 2012.

[122] Samsung Electronics. *SmartThings Public GitHub Repo*. `https://bit.ly/2S62VvN`. 2020.

[123] T. Chen and C. Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *ACM KDD*. 2016.

[124] Z. Chen and et al. "XGBoost classifier for DDoS attack detection and analysis in SDN-based cloud". In: *IEEE BigComp*. 2018.

[125] S. S. Dhaliwal, A.-A. Nahid, and R. Abbas. "Effective Intrusion Detection System Using XGBoost". In: *Information* 9.7 (2018), p. 149.

[126] *XGBoost Library*. `https://xgboost.readthedocs.io/`.

[127] D. Britz and et al. "Massive Exploration of Neural Machine Translation Architectures". In: *CoRR* abs/1703.03906 (2017).

[128] V. Van Asch. "Macro-and Micro-Averaged Evaluation Measures". In: *Belgium: CLiPS* 49 (2013).

[129] A. Gómez-Ríos and et al. "A Study on the Noise Label Influence in Boosting Algorithms: AdaBoost, GBM and XGBoost". In: *HAIS*. 2017.

[130] Giorgia Azzurra Marson and Bertram Poettering. "Practical secure logging: Seekable sequential key generators". In: *European Symposium on Research in Computer Security*. Springer. 2013, pp. 111–128.

[131] Kevin D Bowers et al. "Pillarbox: Combating next-generation malware with fast forward-secure logging". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 46–67.

[132] Yousra Aafer, Wenliang Du, and Heng Yin. "Droidapiminer: Mining api-level features for robust malware detection in android". In: *International conference on security and privacy in communication systems*. Springer. 2013, pp. 86–103.

[133] Xuxian Jiang et al. "Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach". In: *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. ICDCS '06. IEEE Computer Society, 2006.

[134] Paul Ammann, Sushil Jajodia, and Peng Liu. "Recovery from Malicious Transactions". In: *IEEE Trans. on Knowl. and Data Eng.* (2002).

[135] Ashvin Goel et al. "The Taser Intrusion Recovery System". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. 2005.

[136] Samuel T. King et al. "Enriching Intrusion Alerts Through Multi-Host Causality". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2005.

[137] Taesoo Kim et al. "Intrusion Recovery Using Selective Re-execution". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. 2010.

[138] James Newsome and Dawn Xiaodong Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2005.

[139] Jim Chow et al. "Understanding data lifetime via whole system simulation". In: *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*. SSYM'04. USENIX Association, 2004.

[140] Shiqing Ma et al. "Accurate, Low Cost and Instrumentation-Free Security Audit Logging for Windows". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC '15. 2015.

[141] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. "LogGC: garbage collecting audit log". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. CCS '13. 2013.

[142] Michael Dietz, Anhei Shu, and Dan S Wallach. "Quire : Lightweight Provenance for Smart Phone Operating Systems". In: *Proceedings of the 20st USENIX Conference on Security Symposium*. Usenix Security. 2011.

[143] Justin Grover. "Android Forensics: Automated Data Collection and Reporting from a Mobile Device". In: *Digit. Investig.* (2013).

[144] Yanick Fratantonio et al. "TriggerScope: Towards Detecting Logic Bombs in Android Applications". In: *IEEE Security and Privacy*. 2016.

[145] Daniel Arp et al. "Drebin : Effective and Explainable Detection of Android Malware in Your Pocket". In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. 2014.

[146] Brendan Saltaformaggio et al. "Screen after Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images". In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[147] Brendan Saltaformaggio et al. "GUITAR: Piecing Together Android App GUIs from Memory Images". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. ACM, 2015.

[148] Dimitris Apostolopoulos et al. "Discovering Authentication Credentials in Volatile Memory of Android Mobile Devices". In: *In Collaborative, Trusted and Privacy-Aware e/m-Services*. 2015.

[149] Lorenzo Gomez et al. "RERAN: Timing- and Touch-sensitive Record and Replay for Android". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. IEEE Press, 2013. URL: http://dl.acm.org/citation.cfm?id=2486788.2486799.

[150] M. Halpern et al. "Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. 2015.

[151] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. "Versatile Yet Lightweight Record-and-replay for Android". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. ACM, 2015. URL: http://doi.acm.org/10.1145/2814270.2814320.

[152] Zhengrui Qin et al. "MobiPlay: A Remote Execution Based Record-and-replay Tool for Mobile Applications". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. ACM, 2016.

[153] Earlence Fernandes et al. "Android UI Deception Revisited: Attacks and Defenses". In: *20th International Conference on Financial Cryptography and Data Security*. FC '16. 2016.

[154] Chuangang Ren et al. "Towards Discovering and Understanding Task Hijacking in Android". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. USENIX Association, 2015.

[155] Antonio Bianchi et al. "What the App is That? Deception and Countermeasures in the Android User Interface". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. IEEE Computer Society, 2015.

[156] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. "Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. USENIX Association, 2014.

[157] Kiran-Kumar Muniswamy-Reddy et al. "Provenance-aware storage systems." In: *Usenix annual technical conference, general track*. 2006, pp. 43–56.

[158] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. "Why and where: A characterization of data provenance". In: *International conference on database theory*. Springer. 2001, pp. 316–330.

[159] Wang Chiew Tan et al. "Provenance in databases: past, current, and future." In: *IEEE Data Eng. Bull.* 30.4 (2007), pp. 3–12.

[160] Hyo-Sang Lim, Yang-Sae Moon, and Elisa Bertino. "Provenance-based trustworthiness assessment in sensor networks". In: *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks*. 2010, pp. 2–7.

[161] Milan Markovic et al. "Semantic modelling of plans and execution traces for enhancing transparency of iot systems". In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE. 2019, pp. 110–115.

[162] Pietro Colombo and Elena Ferrari. "Access control enforcement within MQTT-based internet of things ecosystems". In: *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*. 2018, pp. 223–234.

[163] Ahmed J Hintaw et al. "MQTT Vulnerabilities, Attack Vectors and Solutions in the Internet of Things (IoT)". In: *IETE Journal of Research* (2021), pp. 1–30.

[164] M. R. Shahid and et al. "IoT Devices Recognition Through Network Traffic Analysis". In: *IEEE BigData*. 2018.

[165] N. Apthorpe, D. Reisman, and N. Feamster. "A Smart Home Is No Castle: Privacy Vulnerabilities of Encrypted IoT Traffic". In: *CoRR* (2017).

[166] D. Kumar and et al. “All Things Considered: An Analysis of IoT Devices on Home Networks”. In: *USENIX Security*. 2019.

[167] T. J. OConnor and et al. “HomeSnitch: behavior transparency and control for smart home IoT devices”. In: *ACM WiSec*. 2019.

[168] Liangdong D. and et al. “IoTSpot: Identifying the IoT Devices Using their Anonymous Network Traffic Data”. In: *MILCOM*. 2019.

[169] Vijay Srinivasan and et al. “Protecting your daily in-home activity information from a wireless snooping attack”. In: *UbiComp*. 2008, pp. 202–211.

[170] B. Bezawada and et al. “IoTSense: Behavioral Fingerprinting of IoT Devices”. In: *CoRR* (2018).

[171] M. Miettinen and et al. “IoT SENTINEL: Automated Device-Type Identification for Security Enforcement in IoT”. In: *ICDCS*. 2017.

[172] R. Trimananda and et al. “Packet-Level Signatures for Smart Home Devices”. In: *NDSS*. 2020.

[173] W. Zhou and et al. “Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms”. In: *USENIX Security*. 2019.

[174] Y. J. Jia and et al. “ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms.” In: *NDSS*. 2017.

[175] E. Fernandes and et al. “FlowFence: Practical Data Protection for Emerging IoT Application Frameworks”. In: *USENIX Security*. 2016.

[176] E. Fernandes and et al. “Security Implications of Permission Models in Smart-home Application Frameworks”. In: *IEEE Secur. & Priv.* 15.2 (2017).

[177] D. Kumar and et al. “Emerging Threats in Internet of Things Voice Services”. In: *IEEE Secur. & Priv.* 17.4 (2019).

[178] D. T. Nguyen and et al. “IotSan: Fortifying the Safety of IoT Systems”. In: *ACM CoNext*. 2018.

[179] Z. Celik and et al. “Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opport.” In: *ACM CSUR* (2019).

[180] Y. Li and et al. “Deep Content: Unveiling Video Streaming Content from Encrypted WiFi Traffic”. In: *IEEE NCA*. 2018.

[181] Y. Jia and et al. “A Novel Graph-based Mechanism for Identifying Traffic Vulnerabilities in Smart Home IoT”. In: *IEEE INFOCOM*. 2018.

[182] S. Marchal and et al. “AuDI: Toward Autonomous IoT Device-Type Identification Using Periodic Communication”. In: *IEEE JSAC* (2019).