Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps

Xueqiang Wang, ***Indiana University Bloomington;*** **Yuqiong Sun and Susanta Nanda,** ***Symantec Research Labs;*** **XiaoFeng Wang,** ***Indiana University Bloomington***

https://www.usenix.org/conference/usenixsecurity19/presentation/wang-xueqiang

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps

Xueqiang Wang
Indiana University Bloomington

Yuqiong Sun
Symantec Research Labs

Susanta Nanda
Symantec Research Labs

XiaoFeng Wang
Indiana University Bloomington

Abstract

Smart home IoT devices have increasingly become a favorite target for the cybercriminals due to their weak security designs. To identify these vulnerable devices, existing approaches rely on the analysis of either real devices or their firmware images. These approaches, unfortunately, are difficult to scale in the highly fragmented IoT market due to the unavailability of firmware images and the high cost involved in acquiring real-world devices for security analysis.

In this paper, we present a platform that accelerates vulnerable device discovery and analysis, without requiring the presence of actual devices or firmware. Our approach is based on two key observations: First, IoT devices tend to reuse and customize others' components (e.g., software, hardware, protocol, and services), so vulnerabilities found in one device are often present in others. Second, reused components can be indirectly inferred from the mobile companion apps of the devices; so a cross analysis of mobile companion apps may allow us to approximate the similarity between devices. Using a suite of program analysis techniques, our platform analyzes mobile companion apps of smart home IoT devices on market and automatically discovers potentially vulnerable ones, allowing us to perform a large-scale analysis involving over 4,700 devices. Our study brings to light the sharing of vulnerable components across the smart home IoT devices (e.g., shared vulnerable protocol, backend services, device rebranding), and leads to the discovery of 324 devices from 73 different vendors that are likely to be vulnerable to a set of security issues.

1 Introduction

Smart home IoT devices have become favored targets for attackers [41] as much for the lack of user awareness [35] as for their poor security design [46]. As the motivation for attackers grows (e.g. IoT botnets, personal data theft), security incidents for smart home devices are only expected to increase. Securing these devices is challenging on several fronts. First, a good fraction of vendors in this space are small and medium-sized businesses that lack the budget for software quality control and security best practices, resulting in numerous insecure devices in the market. Second, many of these devices are relatively inexpensive (often less than $100) and cannot afford to have support for expensive security infrastructure, such as monitoring agents, encryption and authentication hardware, etc. Consequently, when a device is found vulnerable, there is very little incentive and capability for the vendor to release a fix. Third, high vendor fragmentation makes it hard to manage and distribute software/firmware patches.

One way to address this issue is to identify vulnerable devices before they get deployed and take appropriate measures to protect the device. Examples of such measures may include upgrading the device firmware, identifying and blocking traffic that can exploit the vulnerability, or quarantining the device completely. To identify the vulnerable devices beforehand, multiple approaches have been proposed [10, 14, 16, 17, 19, 20, 24, 25, 27, 34, 42, 47, 49, 58]. One line of research [19, 27] focused on launching an Internet-scale scan to detect trivially vulnerable devices (e.g., devices with weak passwords, certificates, and keys) that are publicly accessible. However, these approaches often cannot help identify devices with more sophisticated vulnerabilities or devices hidden behind NAT. Another line of research [10, 14, 16, 17, 20, 24, 25, 34, 42, 47, 49, 58] focused on statically and/or dynamically analyzing an IoT device or its firmware to evaluate its security. Although these approaches tend to yield more comprehensive and accurate results for individual devices, they do not scale well for a large-scale analysis. First, getting physical access to all the devices on the market is not a viable option because of restricted availability of devices in certain geographies and their prohibitively high acquisition cost. Similarly, device firmware is not always available due to the highly fragmented market that involves a lot of small integration and distribution vendors [1]. Second,

[1] Integration vendors are the ones that integrate components, tools, and SDKs, provided by OEMs. Distribution vendors simply acquire the device from an OEM and re-brand with their own before selling in the market.

even with a device or its firmware, the analysis itself is often tedious, error-prone and difficult, especially considering the "device shell" that is often put in place by the device vendors (e.g., packing, obfuscation and encryption). As a result, the market would benefit from an approach where vulnerable devices can be quickly identified at scale and the scope of analysis can be narrowed down.

Approach. In this paper, we present a platform that accelerates vulnerable device discovery and analysis without requiring access to a physical device or its firmware. Our approach is based on two observations. First, smart home IoT device vendors, especially small and medium-sized ones, often rely on same components (e.g., software built from open source projects, hardware components from common suppliers) to build their devices. Consequently, the same vulnerabilities or bad security practices often transfer from one IoT device to another. We can thus propagate vulnerability information to an *unknown* device by evaluating its similarity with devices *known* to be vulnerable. Second, similarities of devices are often reflected in their mobile companion apps, which are widely accessible. Combining these two observations enables us to build a platform that identifies vulnerable devices in a scalable way without requiring the physical devices themselves or their firmware images.

In our platform, we try to expedite the process of identifying vulnerable devices by providing two functions: (1) *app analysis*: find the characteristics of a device by analyzing its companion app, and (2) *cross-app analysis*: find device families, i.e. cluster of devices, that have similarity in some of the characteristics found in app analysis by analyzing multiple apps. Clustering helps identify apps that have a similar set of vulnerabilities based on shared components [8].

Results Overview. For our experiments, we crawled Google Play Store [3] to search for potential IoT companion apps and downloaded 3,094 of them. After filtering out some noise, we were left with a dataset of 2,081 apps (see Section 2.2 for more details). These apps were then analyzed by our platform.

First, we found the device clusters, i.e., device families, containing devices that are similar in various aspects such as software or hardware components, back-end services, and network protocols. For instance, in our analysis, we found 19 device families covering 139 apps from 122 different vendors where devices in a family shared similar software components. As another example, we found 48 different families covering 460 devices that shared similar back-end services.

Second, we tried to identify devices that are impacted by a given vulnerability using the device families already identified. In one case, we were able to discover devices from four different vendors (apps of which is estimated to be installed by more than 215,000 users) that were previously not known to be vulnerable to a software vulnerability and independently confirm the existence of the vulnerability on 45 devices from four different vendors that were previously confirmed by other sources. In another case, we were able to identify 67 devices from 16 different vendors that are impacted by a hardware security issue. In total, our platform has identified 324 potentially vulnerable devices from 73 different vendors. During the process of validation, we could reach a decision (confirm or disapprove) about 179 devices from 43 vendors, among which 164 (91.6%) are confirmed to be vulnerable.

Contributions. This paper makes the following contributions:

- It demonstrates how companion mobile apps for IoT devices can provide insights into the security of the devices themselves.
- It shows the effectiveness by using this approach to assess the security posture of IoT devices when neither the physical devices nor their firmware images are available.
- It proposes a platform to perform mobile app collection, filtering, analysis, and clustering at a large scale. It demonstrates its use by analyzing more than 2000 apps and clustering them in multiple dimensions.
- It reports the discovery of 324 devices from 73 distinct vendors that are likely to be vulnerable to a set of security issues.

2 System Design

2.1 Overview

Figure 1 presents an overview of our platform. The first component of our platform is the *IoT App Database*, which stores the companion apps of smart home IoT devices crawled from the Google Play Store [3]. The database is extended constantly by fetching more apps (e.g., when new IoT devices are on market or old apps get updated).

The apps stored in IoT App Database are then analyzed by the *App Analysis Engine*. The goal of the App Analysis Engine is to estimate the profile of an IoT device (i.e., what the device is like) based on code analysis. Specifically, the App Analysis Engine computes three things: the network interfaces of a device, the unique strings (called *imprints*) that a device may include, and code signature of the companion app. The results of App Analysis Engine are stored in the *App Analysis Database*.

A *Cross-App Analysis Engine* queries the App Analysis Database and identifies correlations across different devices in order to build a *device family*. A device family groups together different devices from different vendors based on their *similarity*. The similarity can be in terms of different dimensions (e.g., similar software, similar hardware, similar protocols, and similar cloud back-end services). The device family allows propagation of vulnerability information among similar devices. Specifically, it allows evaluation of IoT device

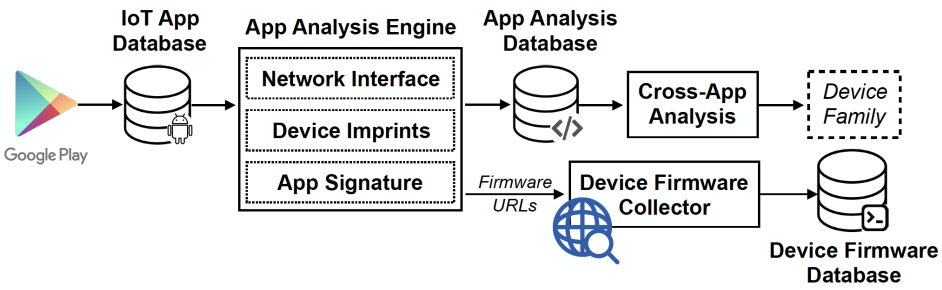

Figure 1: An overview of the platform

security from the perspective of either a device or a threat: 1) for a specific device, the similarity allows to quickly assess whether or not the device is vulnerable and if so to which vulnerabilities, and 2) for a specific vulnerability, find the set of devices on the market that might be affected by the vulnerability.

To facilitate vulnerability confirmation, our platform contains an additional component called *Device Firmware Collector*. It leverages the code analysis results output by the App Analysis Engine (e.g., Firmware URLs) as well as Internet search results to download firmware images into a *Device Firmware Database*. These firmware images later enable us to further confirm the vulnerabilities found by the Cross-App Analysis Engine. Note that the Device Firmware Collector is not an essential or required component of our platform. Rather, it is utilized as one of several means to help confirm the findings from the platform (See Section 3.2 for more details).

In the remainder of this section, we describe each component of our platform in detail.

2.2 App Collection

The first step of our platform is to gather mobile companion apps of smart home IoT devices for analysis. To achieve this goal, we crawled Google Play Store[2]. In total, we downloaded 3,094 Android apps, out of which 2,081 were included in the final dataset and analyzed by our platform.

The challenge during app collection is to identify apps that are mobile companion apps of IoT devices. To address this problem, we initialized the crawler with 281 seed apps manually selected from the online smart home products database *SmartHomeDB* [5], and used snowball sampling to collect more apps via the connections between the seed apps and other apps on Google Play (e.g., keywords, suggestions and categories). As a result, 3,094 candidate apps are initially downloaded. However, we observed that snowball sampling may sometimes introduce noise. For example, apps that manage phone camera are confused with the apps that manage home security cameras. Apps that lock phone's screen are confused with smart home locks. To eliminate such noise from the dataset, we performed filtering. The filtering is based on a clustering model (Affinity Propagation [26]) that clusters apps based on the permissions that the apps request on installation and the sensitive Android APIs that the apps may invoke at runtime. We deploy the filtering on apps that are nominated by the same seed sample and remain the largest cluster. This approach turns out to be effective: a random manual inspection of 200 apps after filtering shows that 98.5% of them are real mobile IoT companion apps. After further deduplication, 2,081 apps are left in the dataset and fed into the App Analysis Engine for analysis. Note at the first phase of the research, we worked with a relatively small dataset and focused more on validating the approach. Our platform is constantly running to collect more apps for future analysis at a larger scale.

[2]We based our analysis primarily on Android but most IoT device vendors provide mobile companion apps in both iOS and Android.

2.3 App Analysis Engine

The *App Analysis Engine* analyzes mobile companion apps collected in order to build a device profile for individual devices. Unlike previous works [12, 32, 48, 60] that focused on apps themselves, the goal here is to compute what the *device* is like, indirectly from the app. We achieved this goal by independently applying three methods: a device interface analysis that computes the network interfaces of a device, an imprint analysis that computes unique strings a device might be related to, and a fuzzy hash analysis that computes code signature of a mobile companion app. In practice, we found that the first method is more comprehensive and informative. Nevertheless, the rest two methods are still useful in filling in gaps where the first method cannot easily apply.

2.3.1 Device Interface Analysis

The device interfaces are often a good reflection of what the device is like, e.g., the protocol that the device speaks, the service that the device runs, the function that the device supports, and sometimes the hardware components in use by the device. Without directly examining a device or its firmware, we estimate the device interfaces based on analysis of its mobile

companion app, as the app and the device complement each other in their network interfaces. A peer-to-peer connection between the app and the device can benefit this estimation, as the app interfaces, in this case, are direct reflections of the device interfaces; however, this is not a necessary condition. For devices where a cloud or backend service is involved, popular backend services like Microsoft Azure IoT Hub [40] are often generic: they tend to relay the connection between the app and the device without much meddling. Such devices also work well with our approach since their app interfaces still closely reflect that of the devices. We performed a study over the online IoT device database (SmartHomeDB), and found that majority of the devices (76.3%) produced by small and medium-sized vendors support a peer-to-peer connection between the app and the device. Even large vendors like TP-Link support both cloud and peer-to-peer mode for network outage and privacy reasons. This enables us to have a good estimation of the device interfaces for many of the IoT devices, especially vulnerable ones, that are sold on the market.

We used a backward approach to compute the network interfaces of an app, starting from the network response messages that the app may receive, as these messages are information output by the device. We first identify message handling functions in the app and statically decide what the response message may look like. We then identify the request that may trigger the response. Finally, we partially instantiate and execute the app code to reconstruct the request [12, 56]. Figure 2 shows an example of the request and response extracted from the mobile companion app *com.Zengge.LEDWifiMagicColor* of Zengge Wi-Fi Bulb. With many different pairs of requests and responses (e.g., with UDP/48899, TCP/5577 of the bulb and also the cloud server **.magichue.net*), we obtain a good estimation of the device interfaces.

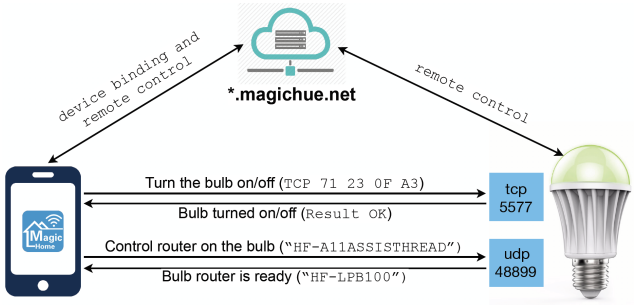

Figure 2: Interfaces of Zengge Wi-Fi Bulb

Response Extraction. We rely on symbolic execution [33] to estimate what the response messages from a device may look like, without actually running the device. We first built a Control Dependency Graph (CDG) and Data Dependency Graph (DDG) of a mobile companion app using Soot [51]. We then start from standard network receiving functions in Android (e.g., <*java.net.DatagramSocket: void receive(java.net.DatagramPacket)*>) and forward execute the mobile companion app symbolically. Whenever we encounter a branch that is dependent on the content of a response message (e.g., fields of the response are checked against a value), we capture the check as a symbolic constraint and fork the execution. After all executions terminate, the conjunction of the symbolic constraints is stored as a "description" of the response message. In order for response messages from two devices to be similar, they have to satisfy the same set of symbolic constraints.

One practical issue is to decide when to terminate a symbolic execution. In our experience, we found that a valid response from the device (i.e., the response passes checks performed by the app) often triggers state changes of the app. Such state changes could be either UI element changes (e.g., updating device status displayed to the user) or modifications to the local registry (e.g., storing device information to configuration files, shared preferences or databases). To confirm this heuristic, we randomly sampled 200 response handling procedures that exist in 179 apps from our app set and evaluated manually the impact of valid responses. Among these responses, 162 of them had an influence on UI elements, and 76 of them resulted in modifications to the local registry (with some overlapping cases where responses changed both); only eight of them would not trigger such changes, but the app stored response content (e.g., login token) in global variables. This study shows that state changes can be a good approximation for the termination of valid response handling. We thus mark such state changes as the point where we terminate the symbolic execution and produce the conjunction of the symbolic constraints. In addition, we supplement this method with the observation that invalid responses are discarded quickly by the apps (i.e., within few lines of code). We thus also set a threshold on the number of procedures to execute before we terminate the execution. Utilizing these two heuristics, we could produce a small but meaningful set of constraints that closely describe a valid response that an IoT device may produce.

Pairing Request and Responses. The next step is to identify the request sent by the app that will trigger the response from the device. In many cases, the request is straightforward to identify: it co-locates with the response message handling functions. In other cases, however, it is trickier as the request can be located in a different procedure or class, especially when the communication between app and device is asynchronous. In these cases, static code analysis can be limited in identifying the matching request.

Fortunately, we observe that a matching pair of request and response often share a large code base of their handling functions (i.e., classes and methods used to process the request and response). Such similarities are reflected in the stack at runtime. To confirm this observation, we examined the paired requests and responses for the same set of 200

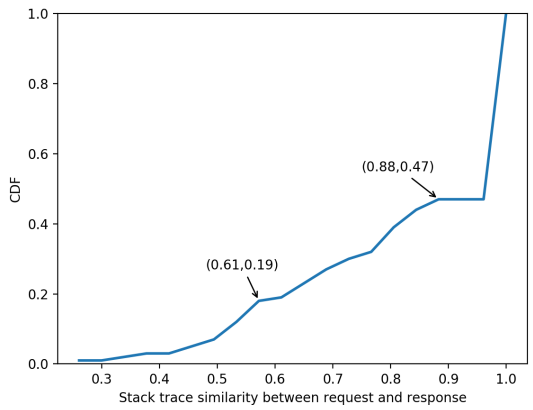

Figure 3: Cumulative Distribution Function (CDF) of request and response similarity

response handling procedures, and evaluated the similarities between stack traces of the responses and the requests. Figure 3 shows the Cumulative Distribution Function (CDF) of the Jaccard Similarity: 81% pairs of request and response share over 61% of their stack frames, and more than half (53%) of the request-response pairs have over 88% frames in common. For unpaired requests and responses, the similarity reduces to almost zero. Thus, by recording and comparing the execution stack of the app when the app is making requests (i.e., via concrete execution) and processing responses (i.e., via program dependence graph), we can pinpoint with good accuracy, among multiple request sending functions, the one that most likely will trigger the target response. As an example, Figure 4 shows the stack traces of a request and response that are used by Chuango Wi-Fi alarm system. The request and response are matched based on the common stack frames (e.g., those triggered by the same user click) despite being located in different classes that run in different threads.

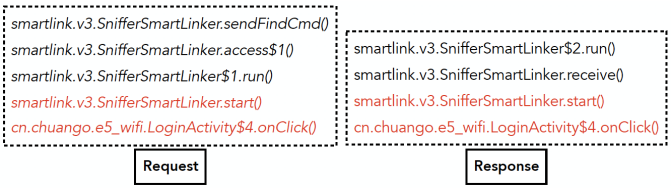

Figure 4: A matching asynchronous request and response in *cn.chuango.e5_wifi*

Request Reconstruction. After identifying a matching pair of request and response, the next step is to reconstruct the request string. Unlike responses, requests are produced by the app. Therefore, we may reconstruct a complete request string as compared to a set of symbolic constraints for responses. A number of techniques have been developed for reconstructing program values via program slicing and execution [12, 29, 48, 56]. We adopted the *Instantiated Partial Execution (IPE)* technique developed in Tiger [12] in our platform. The advantage of using IPE is that it evaluates and instantiates variables to concrete values if they are found to be irrelevant to the request string thereby dramatically reducing the number of paths need to be explored. In addition, IPE also caches outputs of code slices and reuses the results if applicable, further reducing the analysis complexity. By using IPE, we were able to reduce the time needed to reconstruct a request to under a minute.

The result produced by device interface analysis is a set of request and response pairs. The requests are fully or partially[3] reconstructed request strings and responses are sets of symbolic constraints. A device is said to have a similar interface as another device if they both accept similar requests and output similar responses.

2.3.2 Imprints Analysis

Device imprints (i.e., unique strings) found in an app can help correlate different devices. We are particularly interested in imprints that show up in the communication between the app and the device, as they are indicative of the uniqueness of the *device*. In contrast, there are also app imprints, such as app developer emails or special class names, that identify an app or library. However, they are less indicative of the device.

Table 1: Examples of device imprints

Type	Imprints	Device
device keywords	"20140930073702357" (dir. name in firmawre)	Homeboy Wi-Fi Security Camera
	"0622707c-da97-4286-cafe-"* (UUID of the device family)	SensingTEK Cameras
cert and comm. keys	"Ztwy518518puy518" (AES key)	Zhongteng Smart Home Devices
user & pwd	"P0rtal@123!" (account pwd)	Pro1 Thermostats
special URLs	"qjg7ec".internetofthings .ibmcloud.com (MQTT *orgID*)	Max Smart Home Devices

Inspired by previous work done by Costin et al. [16] that directly extracts imprints from embedded firmware images, we also focus on four types of device imprints: device (backdoor) keywords, certificates and keys, non-trivial usernames and passwords, and special URLs. The method we used to identify imprints is simple: we build a Data Dependence Graph of an app and check backward from network APIs to find constant strings in the app that affect parameters of those APIs. Note that these APIs are used to communicate with the device. In other words, we only use unique strings as imprints if they are related to the device (i.e., they are part of either requests to or responses from the device). A parser later decides which category the constant strings fall into and

[3] Certain requests require user input (e.g., login request). In these cases, we partially reconstruct the request with <NONE> string replacing the missing user input.

whether or not they are commonly seen (e.g., *admin* for both username and password is ignored). Table 1 shows an example of a few imprints we collected in our dataset. When two apps have the same imprints (and both imprints affect the communication with devices), it serves as a strong indicator of the similarity between devices. For instance, by using imprint *"OBJ-000165-PBKMW"*, we were able to correlate *VStarcam* and *OUSKI* IP Cameras (the latter is later confirmed to be a rebranding of the former).

Although imprints can serve as strong evidence of correlation, imprint analysis as a method is less applicable in general since many times imprints of a device do not manifest themselves in the app. For example, we were not able to spot the existence of any magic keyword, like the "*xmlset_roodkcableoj28840ybtide*" (i.e., edit by 04882 joel backdoor in reverse) keyword used by a number of devices for debugging purposes reported by Constin et al. [16]. This makes sense since the magic keyword is built into the firmware images for debugging purposes, and device debugging is generally not a critical functionality required for customer facing apps. However, it highlights the limitation of imprint analysis, and the reason why we need a fully fledged device interface analysis.

2.3.3 Fuzzy Hash Analysis

Another method we used is to assess code similarity via fuzzy hash. Similar mobile companion apps often indicate similar devices. We thus compute *ssdeep* of objects found in an app, including classes, libraries, and other types of resources (e.g., texts), and compare the results across apps. The benefit of using fuzzy hash as compared to traditional hashing algorithm (e.g., *SHA1*) is that we can relate objects that are similar but are not exactly the same. Through this way, we were able to identify a few similar devices. For example, the companion apps of *CHITCO* and *EDUP* smart switches are found to have 50.7% objects matched with 80/100 similarity, and these two devices are later confirmed to share similar software. Note, however, similarities between devices do not necessarily mean similarities in the apps. We observed in many cases that similar devices have different apps (e.g., apps are developed independently), and therefore cause failures to fuzzy hash analysis. Code similarity is more useful for identifying obvious correlations as well as for cases where other analysis methods have some difficulties to apply (e.g., for native libraries).

2.3.4 Modularity

A special consideration we made while building the App Analysis Engine is the modularity of the analysis. The reason we took this extra step as compared with generating analysis result per app is to accommodate the *modular similarity* that often appear across IoT devices. It is common that IoT device vendors, especially smaller ones, comprise their products from a number of existing modules on market, such as hardware components from common suppliers, software built from open source projects, binary driver code for protocols and etc. For example, the *HiFlying* Wi-Fi module is used by a number of vendors to manage Wi-Fi connectivity for their devices. Thus it is important for our analysis to be modular as well, in order to track device similarities and detect vulnerability propagation at a finer granularity of individual device components (Refer to Section 2.4 and Section 3.3 for more details of the components that we can track).

We based our design on the observation that device components are often managed by different code modules in the app (e.g., class, package). Taking the previous HiFlying Wi-Fi module as an example, devices such as *BeSMART* thermostat that uses the module often have two separate classes, *com.hiflying.smartlink.v3.SnifferSmartLinkerSendAction* and *com.besmart.thermostat.MyHttp*, for handling Wi-Fi connection and user interaction over HTTP, respectively. We thus infer such modularity from the app (e.g., based on class hierarchy and invocation stack) and apply the above analysis method on individual modules.

2.4 Cross-App Analysis Engine

The analysis results output by the App Analysis Engine are stored into the *App Analysis Database*, which is then queried by the *Cross-App Analysis Engine*. The Cross-App Analysis Engine is designed to detect modular similarities between different devices. In particular, the comparison is made to detect four types of similarities: similar software components, similar hardware components, similar protocol, and similar backend services.

Similar Software Components. Similar device interfaces, especially application interfaces, are indicative of strong connections between software components of different devices. For example, we were able to correlate 72 different smart home IoT devices from 16 distinct vendors that might have used the same version of GoAhead web server[4]. Such correlation is powerful, as in many times security weaknesses manifest them in software and security weakness found in one device can directly impact the security of others. For example, we were able to identify seven previously unreported devices that are vulnerable to a known vulnerability, as detailed in Section 3.3.

Another interesting phenomenon detected is device rebranding. In the smart home IoT industry, smaller vendors sometimes do not develop their own products. Instead, they customize IoT devices from OEMs and resell with their own branding. As reflected in the app analysis results, rebranded devices have almost identical device interfaces across multi-

[4]GoAhead is a simple web server specifically designed for embedded devices.

ple modules as the original OEM devices. Although device rebranding itself is not an issue, it complicates the security practices in firmware update and patching. In some cases, for example as shown in Section 3.3, a vulnerability is inherited by the rebranded devices from the OEM but the security patch that fixes the vulnerability is not.

Similar Hardware Component. Smart home IoT devices may be built upon similar hardware (e.g., Wi-Fi module). Such similarities in hardware components are sometimes reflected in device companion apps due to the need for the app to configure or interact with the hardware component. Due to the specialty of the hardware, such device-app interfaces can be unique, allowing a strong correlation of different devices using the same hardware. For example, we found that two Wi-Fi modules with a known security weakness of credential leakage are potentially being used by 166 devices from 35 different device vendors. The total downloads of these apps together are over 278,000 times.

Similar Protocol and Backend Service. A specific protocol often has its own request and response format. Similarly, a specific backend service often exposes standard APIs. Cross-App Analysis Engine can detect similarities in network interfaces and thus correlate devices that speak the same protocol or speak with the same backend service, even if such protocols or backend services are not documented. For example, we found that 39 different devices from 11 vendors are very likely to speak the SSDP protocol, which was known to be vulnerable as a reflector for DDoS attacks. As another example, we found that 32 devices from 10 vendors relied on the same cloud service to manage their devices, and the cloud service has a reported security weakness that allows attackers to take full control of the IoT devices by device ID and password enumeration.

Future Work. There are additional dimensions that security evaluation of an IoT device can benefit from similarity analysis. For example, previous works [16, 28] have shown that same developers or sub-contractor may follow a similar way of coding thus having the same set of bad security practices or vulnerabilities built into their devices. Similarly, the same development toolchain (e.g., compiler) may transform code in a similar way that leads to the same set of security issues [7, 52, 54]. As a future work, we plan to extend our analysis to cover more dimensions of similarities in order to obtain a more accurate and complete evaluation of smart home IoT devices.

2.5 Device Firmware Collector

Our platform features an additional component called *Device Firmware Collector* which enriches the *Device Firmware Database* through downloading firmware images of devices corresponding to the apps being analyzed. The purpose of the firmware images is to help us confirm the findings from the cross-app analysis phase. In our current platform, we collect device firmware in two ways. First, we utilize the firmware downloading links that are embedded in the mobile companion apps. As IoT devices are usually headless (i.e., no keyboard or screen for user interaction), they often deploy firmware updates via the companion apps. As a result, links are sometimes built into the app by the vendor. Such links are often special URLs that can be extracted through imprint analysis. Second, we follow the app pages on Google Play, which often direct to device vendors, to crawl potential firmware files. Specifically, we used Google Custom Search API to programmatically search through vendor websites for firmware image files.

For the files collected, we filter out non-firmware files by checking their format using *Binwalk* [2]. Binwalk is a well-known firmware unpacking tool which extracts various data from a binary blob through pattern matching. Once a file is decided to be a firmware, a special effort is made to correlate firmware version with app version. As we will discuss in Section 3.3, this helps us to decide at which version a particular vulnerability is fixed and whether or not that fix has an impact on the app.

Note that not all device firmware could be downloaded. Even for the ones that we collected, there is still a considerable amount of firmware encrypted or obfuscated that renders the analysis difficult. This is a limitation yet to overcome in vulnerability confirmation, as discussed in Section 4.2.

3 Dataset and Results

3.1 Dataset and Platform Statistics

Dataset. In total, our dataset comprises of 2,081 apps collected through the method described in Section 2.2. The average size of the apps is 13MB (Min. 23KB and Max. 142MB). These apps spread globally (271 languages) and have a total download exceeding 1.2 billion. The apps cover 1,345 different device vendors and, by our estimate, about 4,720 different device models. We note that this dataset is still incomplete: by comparing certain types of devices in our dataset (e.g., IP camera) against online lists of devices [13, 21–23] of the same type, we estimate the dataset to cover ∼5-20% of the total IoT companion apps.

Testbed and App Processing. Our app analysis platform ran on a 4 Core, 3.33GHz Ubuntu 16.04 server with 16 GB RAM and 1TB hard drive. The Android emulator is compiled from Android Open Source Project, AOSP 4.4.4.

In total, our platform needed ∼68.3 hours to process the 2,081 apps, with an average processing time of 118.2 seconds per app. In our experiment, we set the maximum processing time to 10 minutes and the majority of the apps are processed successfully within this time frame. The platform was not able to fully analyze 73 (3.5%) apps within the timeout win-

dow and therefore only partial analysis results are available for them. In addition, 43 (2.1%) apps were not analyzed by the platform because the tool we used (i.e., Soot) to build CDG and DDG failed to handle the app bytecode during interpretation. Overall, about 98% of the apps were either fully or partially analyzed.

One practical concern is the obfuscation of the app and its impact on the analysis. As reported in previous study, the majority (85.8%) of the device companion apps are produced by the standard tool in Android SDK (i.e., Proguard) [53], which mangles the apps by renaming classes, methods and fields. While Proguard introduces skews to the fuzzy hash analysis, it does not affect our main analysis method (i.e., network interface analysis) since it does not obfuscate network APIs, data-flow and control-flow. Another concern is the packing of the app—some developers use packers to encrypt their code, which would also have an impact on the network interface analysis. However, consistent with observations made by prior research [53], packers are often seen in malware, and less adopted by benign apps. In our dataset, only a handful of apps used commercial packers. Currently, we did not apply any special processing to these apps. There is an orthogonal line of research on developing better unpacking tools(e.g., DexHunter [59] and PackerGrind [57]) and our platform can be supplemented by these tools.

Table 2: IoT device families

Type	Number of Families	Covered Apps	Covered Vendors
Software	19	139	122
Rebranding	28	156	104
Hardware	14	61	51
Protocol	40	271	210
Backend	48	460	422

Device Family. Table 2 shows all the device families detected via our cross-app analysis. For example, we were able to identify 19 distinct device families covering 122 different vendors and 139 apps that were using similar software within the family. As another example, we were able to detect 14 distinct device families covering 51 different vendors that were using similar hardware components within the family. Note that these families are not mutually exclusive; a device might share software components with one device and hardware components with another. The largest device family we identified includes 31 device vendors and the smallest device family includes only 2 device vendors. Figure 5 shows a more intuitive illustration of the device family map.

3.2 Results Validation

Our platform is solely based on code analysis of mobile companion apps, without requiring the physical devices or their firmware images. This is the key to a large-scale security analysis of smart home IoT devices. However, the drawback

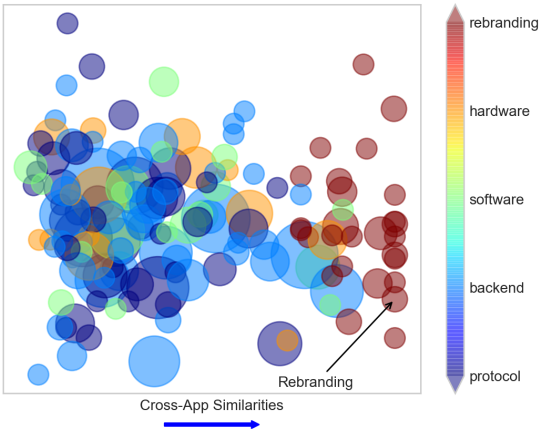

Figure 5: Device family map. Circle size indicates the number of device vendors in the family (the largest circle covers 31 vendors, while the smallest covers two).

of such approach is the accuracy of the result: the output from this analysis (e.g., a family of devices impacted by a particular vulnerability) is a conjecture that points to potential security issues that need to be validated with real devices.

In this paper, we validate and report some of the results we obtained from our analysis to demonstrate the value of the approach. We took a hybrid validation route, taking into consideration practical limitations such as the budget. First, we try to acquire the real device and test it in a local environment (Figure 6 shows the devices we purchased for validation). Second, if we do not have the device, we try to simulate, or in some cases statically analyze, device firmware stored in the Device Firmware Database (built from the method discussed in Section 2.5). Third, if neither of the first two methods applies, we search through online reports including vendor manuals and websites, bug reporting forums, IoT hacking communities and so on. Fourth, we work with the vendor and request their help in validating the results. We primarily used the second and third method, as the first method is very expensive and the fourth method is often a black hole (i.e., no responses from vendor). Upon confirming our findings, we also try to estimate the impact of the finding by searching the online presence of the device on *Shodan* [4].

Ethics. Testing vulnerabilities and scanning real-world devices often bring up serious ethical concerns. In our study, we pay special attention to not cross legal and ethical boundaries. For both real and simulated devices, we evaluate the device in a local network that only allows outbound connections. The device is brought offline immediately after the experiment to avoid being exploited and used as a bot. To evaluate the impact of a particular security issue, we collect data from existing results of *Shodan*, instead of scanning vulnerable

Figure 6: Smart home IoT devices for vulnerability validation

devices directly. In this way, we don't introduce extra network scans. Most importantly, we release our findings to all affected vendors, and refrain from including the real name of any device that is still un-patched or under investigation.

3.3 Results Overview

We present our findings from the perspective of threats, by showing how many smart home IoT devices are potentially impacted by a given vulnerability or security weakness. However, it is also possible to look at the findings from a device's perspective, i.e., for a specific device, what kind of vulnerability or security weakness it may suffer from. The results are encouraging: we identified 324 device models from 73 vendors that are potentially vulnerable to a number of security issues. For the devices that we can confirm or disapprove, about 91% are confirmed to be vulnerable. The total number of users of these devices is estimated to be over 11.1 million.

3.3.1 Vulnerable Software

To demonstrate how software vulnerabilities propagate across devices, we applied our analysis to five high profile vulnerabilities (shown in Table 3) that were reported in GoAhead web server which many smart home IoT devices utilize to provide a web-based interface. These vulnerabilities range from authentication bypass to backdoor account to remote code execution. We started from mobile companion app *object.liouzx.client* of *NEO Coolcam* IP Camera, which was known to be vulnerable to these vulnerabilities, and utilized the cross-app analysis to identify devices that might be similar in their software. Since these are relatively old vulnerabilities (reported in 2017), we expected fewer results. However, in total we still identified 72 device models belonging to 16 distinct vendors that share similar software as the vulnerable device. To validate the results, we utilized the methodology discussed in Section 3.2. We were able to confirm through online reports that 45 device models from four vendors are indeed vulnerable. Since these results are already publicly disclosed, we included them in Table 3. Additionally, we confirmed through manual firmware analysis that six device models from three IP camera vendors, *Vendor A*, *Vendor B* and *Vendor C*, are also impacted by these vulnerabilities. We have informed those vendors about the vulnerabilities but no patch is released yet. Furthermore, we confirmed through real device that one baby monitor device from *Vendor D* is also impacted by the vulnerabilities. Vendor D has asked us to refrain from including their names until further investigation is done on their side. In total, we confirmed the existence of the vulnerabilities on 52 device models from eight different device vendors, with *seven device models from four vendors newly discovered*.

While validating the results, we also encountered one case where our platform mistakenly flagged a device as vulnerable. The analysis results output by the platform show that three device models produced by *KGUARD* have very similar interfaces with other vulnerable devices. However, our manual validation on the real device as well as emulated firmware shows that the *KGUARD* devices are not vulnerable. We inspected the firmware, and found that the software configuration of *KGUARD* is indeed very similar to the vulnerable devices. Specifically, we found that 26 out of 31 CGI programs and web pages are in common. But the vulnerable code was removed. Our hypothesis is that since these devices are relatively new on market (i.e., after the vulnerability was reported), *KGUARD* may have customized the software configuration and patched the vulnerability before releasing the product to the market.

We also want to highlight two observations we made during the process of results validation. First, although the vulnerabilities are old, we are still seeing a large set of devices to be potentially vulnerable. A Shodan search shows that there are potentially 58,456 devices running in the wild still being vulnerable and the total number of app downloads is more than 282,000 times. This is the result of a dataset with merely $\sim$2K apps. With a large-scale analysis covering more device models and vendors, the problem can be even worse. This demonstrates a common issue in the smart home IoT market: the market is highly fragmented and many smaller vendors never bother taking care of the device after selling the device.

Another observation to highlight is the difficulty to validate the results, or rather the general challenge of evaluating the security of an IoT device. We were *not* able to validate findings on 17 device models from seven out of the 16 vendors, despite that the cross-app analysis tells us they might also be vulnerable. The validation was not successful for a number of reasons: some devices are not targeting U.S. market therefore we could not easily acquire, some devices do not provide firmware download therefore we could not evaluate, for devices that we could download firmware the encryption and packing render analysis difficult. In addition, the collaboration with device vendors have been very difficult. Many times,

Table 3: IoT devices impacted by vulnerable software and device rebanding

CVE	Impacted Vendor	Device Models	Validation Method	Mobile App	App Downloads	Confirmation Status
CVE-2017-8221 CVE-2017-8222 CVE-2017-8223 CVE-2017-8224 CVE-2017-8225	IP Camera Vendor A	4	Firmware	App A	100,000+	Newly Discovered
	IP Camera Vendor B	1	Firmware	App B	5,000+	
	IP Camera Vendor C	1	Firmware	App C	100,000+	
	Baby Monitor Vendor D	1	Device	App D	10,000+	
	Instar	2	Reports	camviewer.mobi.for_instar	1,000+	Independently Reconfirmed
	VStarcam	35	Reports	vstc.GENIUS.client	1,000+	
	Sricam	6	Reports	object.shazx1.client.yi object.smartmom.client	55,000+	
	Conceptronic	2	Reports/Firmware	camviewer.p2pwificam.client	10,000+	
	KGUARD	3	Device	object.kguard.client	10,000+	FP
	7 Vendors	17	N/A	7 apps	146,000+	Pending Confirmation

our email request about potential security issues went into a black hole (i.e., no responses ever received from the vendor). We believe this is also an artifact of market fragmentation as smaller vendors tend to care less about the security of their products.

3.3.2 Device Rebranding

Investigation of a more recent vulnerability, *CVE-2018-11560*, leads to another interesting finding of device rebranding. This vulnerability was initially reported in Insteon IP Camera 2864-222 (firmware 1.4.1.9), where the embedded web server on the device had a missing bounds check when parsing CGI parameters, resulting in a stack buffer overflow. We used the companion app of Insteon IP camera as the input to our Cross-App Analysis Engine to detect if any other devices might be vulnerable to the same vulnerability. To our surprise, we found that almost identical device interfaces are provided by a major IP camera vendor, *Foscam*. We initially suspected that the same web server might be used by both vendors, but later through research we found that Insteon IP camera 2864-222 is actually a rebranded version of Foscam IP Camera FI8918W—it is based on the exact same hardware and software but with a different brand name. Not surprisingly, early versions of Foscam IP camera also suffer from the same vulnerability, but no one has reported that.

The interesting part, however, is that Foscam actually patched the vulnerability *before* the vulnerability was reported in Insteon IP camera. We examined the firmware history of Foscam IP camera and found that the vulnerable code was shipped in over eight Foscam firmware versions before Jul. 2017, impacting at least 15 Foscam models. In firmware updates (2.x.1.120) of Jul. 2017, the vulnerability was patched. However, this patch never made it to the Insteon IP camera until the vulnerability was reported in 2018. We contacted Foscam about this issue, but their response neither confirmed nor denied the finding. Instead, we were advised to update to the newest version of firmware. This highlights another interesting issue about smart home IoT devices. Due to the fragmented market, smaller IoT vendors sometimes do not develop their own products. Instead, they customize IoT devices from OEMs and resell with their own branding. This complicates the security management of the product and puts customers in danger, as vulnerabilities in upstream vendors tend to propagate to a broader set of downstream vendors but security patches are not. Indeed, a Shodan search with the IP camera fingerprints (e.g., server type, time stamps) shows that although Foscam released patches as early as Jul. 2017, there are still 30.7% (10,210 out of 33,230) devices that are not patched to the secure version.

Additionally, our analysis shows that re-branding is indeed not uncommon. With a dataset of ∼2K apps, we identified 27 other re-branded device families not including Foscam example. Examples of these devices include smart plugs from *Bayit* and *Orvibo*, Wi-Fi sockets from *CHITCO* and *EDUP* and so on. Further validation is needed to confirm if these devices inherit any vulnerabilities from upstream vendors.

3.3.3 Vulnerable Hardware

Different IoT device vendors may rely on a common hardware module (e.g., Wi-Fi, Bluetooth), which, if vulnerable, could impact multiple devices. The challenge, however, is that IoT device vendors often do not publicize the hardware components in use. As a result, it is often difficult to decide if a device is vulnerable due to a vulnerable hardware component without tearing apart the physical device or unpacking the firmware to examine the driver code.

Through cross-app analysis, we identified a total of *166 devices belonging to 35 different vendors* that are potentially impacted by two recent security weaknesses found in hardware. In one example, a recent study [36] demonstrated that Hi-Flying Wi-Fi module (HF-LPB100, HF-LPT100, HF-LPB200) can be leveraged by an adversary to steal home network Wi-Fi credentials. The Hi-Flying Wi-Fi module is a self-contained 802.11b/g/n module used by a number of IoT devices to provide wireless interfaces. As an important feature, the module supports credential (e.g., SSID, password) provisioning from device companion app to IoT device via *SmartLink*. As reported by the study [36], the provisioning process may leak Wi-Fi credentials: an adversary could passively listen to the traffic and gather the home Wi-Fi network credentials without much effort. Through our cross-app analysis, we identified that 26 apps, covering 108 devices from 21

vendors are potentially impacted by this security weakness. These apps have been downloaded more than 158,000 times. In another example, ESP8622, a low-cost Wi-Fi microchip that appears in many cheap IoT devices (e.g., Wi-Fi controller, smart plug), was reported to have a similar vulnerability in its *ESP-Touch* provisioning protocol. In our analysis, we identified that 21 apps covering 58 devices from 14 distinct vendors are potentially impacted by the security weakness. In total, these apps have been downloaded more than 120,000 times.

Among the devices flagged by the platform, we were able to confirm that 67 devices from 16 vendors are indeed impacted by the security weaknesses (43 devices from eight vendors are confirmed through vendor response. 24 devices from eight vendors are confirmed through firmware emulation, real device or online reports.). Through vendor response, we were also able to identify that seven devices from two vendors were mistakenly flagged by the platform as vulnerable (i.e., ∼9% false positive rate). We manually examined the two apps to analyze the reason for the false positive. For one case, 14 devices supported by the *Revogi* app were flagged by the platform as potentially vulnerable. However, four of them (Power Plug SOW324, Power Strip SOW321 and SOW323, and Smart Light LTW311) were not actually using the vulnerable hardware. The issue was due to the imprecision of the static analysis performed by the platform. Since the app supports multiple devices from the same vendor, the code modules that control individual devices are not clearly distinguishable (i.e., some modules are shared across devices but others are independent). As a result, the platform was not able to attribute the network interfaces that correspond to the vulnerable hardware to a specific device. Instead, the platform outputs all the devices supported by the app as potentially vulnerable. For another case, three devices supported by the *smanos* app were flagged by the platform by mistake. The devices were found not to be using the vulnerable hardware, but the code module and the corresponding network interfaces that control the hardware was included in the app. This may be due to that the app developer built the app upon some open source templates that contain the hardware module, or maybe the device vendor changed their hardware configuration during the device development process, but the app code was never cleaned up. Nevertheless, the IPE method used by the platform is guided by static analysis to construct network interfaces as long as a code snippet is reachable from an Android *activity*, even though that activity may never be actually triggered by the real device.

3.3.4 Vulnerable Protocol

Similar to hardware components, IoT device vendors often do not publicize the protocols that a device speaks. These protocols range from more open and standard ones such as UPnP, mDNS and SSDP to proprietary ones such as TDDP[5] used for debugging, penetrating private networks and various other purposes. Not knowing which protocol a device can speak creates a great security challenge of managing the device, especially when the protocols are found to be vulnerable or can be leveraged by an adversary to launch attacks.

Through cross-app analysis, we can identify devices that speak the same protocol, thus may suffer from similar security problems. For example, previous research [37] showed that SSDP protocol can be abused by adversaries in order to launch DDoS attacks. SSDP queries such as "ssdp:all" and "upnp:rootdevice" may result in a response size orders of magnitude larger, thus if openly accessible to the Internet may serve as a reflector to amplify requests sent by the attacker. Through cross-app analysis, we identified *39 devices from 11 different vendors* that speak SSDP, despite that few of them clearly documented the protocol that their devices speak. As a result, once these devices are activated in the environment where a firewall is not configured to block incoming queries, they may act as reflectors for DDoS attacks. It's difficult to tell the exact number of devices that are exposed and vulnerable, but the total app downloads (over 10.2 million) indicate that a massive number of devices could possibly be harnessed by attackers.

We validated the results output by the platform. In total, we were able to confirm that 18 devices from six vendors are indeed speaking the SSDP protocol. One device, *Bixi* gesture controller, was mistakenly flagged by the platform. The case with *Bixi* gesture controller is interesting: the device itself does not speak SSDP, but its companion app does, therefore causing false positive for the platform. The reason is that the gesture controller is a device that allows users to control other devices via gesture. It does not speak SSDP but relies on its companion app to use SSDP to discover subsidiary devices for it to control. In this case, the network interface of the app is not an exact mirror of the device interface, causing false positives in the platform.

3.3.5 Vulnerable Backend Service

IoT devices may rely on the same IoT cloud backend service to relay command and control (e.g., to penetrate private home networks). When the backend service contains a security weakness, multiple IoT devices using the same service are impacted at the same time. However, without detailed knowledge of the registered customers of the cloud service, many of these impacted devices are left vulnerable until the problems are independently discovered.

Our cross-app analysis can help address this issue. In a particular case, the security weakness was initially reported on DeepSec 2017 [38], where an IoT cloud backend service is found to be using very short device IDs (i.e., only six digits) to register IoT devices. Consequently, any IoT device that is using the service to relay commands and control is vulnerable to device ID and password enumeration attacks. A successful

[5] TDDP stands for TP-Link Device Debug Protocol.

attack may enable attackers to authenticate to the device and abuse the device as a bot. We used the vulnerable device reported in DeepSec, *Yoosee*, as the seed for cross-app analysis and found *32 devices from 10 different vendors* also rely on the same vulnerable backend to relay command and control. While it is hard to estimate the actual number of devices in the wild that are vulnerable, the total amount of downloads of these apps is over 226,000 times.

Among the 32 devices flagged by the platform, we were able to confirm that 12 devices from seven vendors are indeed sending requests to the specific backend server, and the device IDs are indeed enumerable (i.e., 6-digits). We also found that four devices from one vendor, namely *secrui*, were mistakenly flagged by the platform. The reason is similar to the "dead code" issue we encountered while validating results for devices with vulnerable hardware: we found that *secrui* app embedded a self-contained app *com/jwkj* that talks to the problematic backend server and thus the app interfaces exhibit similarity with those that are vulnerable. However, the embedded app was never actually executed nor did the devices supported by *secrui* app actually talk to the problematic backend server.

3.4 Accuracy of Results

In total, the platform flagged 324 devices from 73 vendors as potentially vulnerable, and we were able to confirm that 164 devices from 38 vendors are indeed vulnerable. This accounts for roughly *50.6%* of all the devices flagged by the platform. During the process of validation, we were also able to identify that 15 devices from 5 vendors were mistakenly flagged by the platform as vulnerable. This accounts for *8.4%* of all the devices that we could either confirm or disapprove (i.e., false positives). Table 4 enumerates the reasons for the false positives and the number of instances of each reason. The first reason for the false positive is the existence of the patch. After vulnerabilities were disclosed, vendors may patch the device. In this case, the app-device interface may stay largely the same, but the device is no longer vulnerable. This is a fundamental limitation of the approach, as the platform is designed to only extract information from the app, not the device. Thus, if the patch does not have any impact on the app, the approach cannot differentiate a vulnerable device from a patched one. The second reason for false positive is the “dead code” inside of the apps. Sometimes the apps may contain code that was *not* actually being used by the device (legacy code, code adopted from elsewhere without cleaning and etc.). Statically, it is difficult to decide if the code will ever be triggered and executed at runtime. Our platform currently may mistakenly include analysis results from such "dead code" if the "dead code" exhibit similarities with other vulnerable devices, thus causing false positives. The third reason for the false positive is the imprecision of the static analysis. Currently, the static analysis techniques used by the platform are not precise enough to attribute network interfaces to individual devices if a single app supports multiple devices and these devices share much common control logic inside of the app. This issue, as well as the “dead code” issue listed above, are not a fundamental problem with the approach. Rather, they are an artifact of the static analysis techniques we used to analyze the apps in the platform. We are currently working on improvements to the techniques to improve the accuracy and precision of the static analysis. Finally, we encountered an exception case to our approach where the app interface is not an exact reflection of the device interface (i.e., the *Bixi* gesture controller). However, due to the nature of the device (i.e., a device that controls other devices), it is uncommon among the IoT devices. We are exploring the Google Play store to identify if there are any more devices of the similar kind.

Table 4: Reasons for the false positives

Reason	# of Devices	# of Vendors
Existence of the patch	3	1
Dead code inside of the apps	7	2
Fail to attribute interfaces to individual devices	4	1
Difference in device and app interface	1	1

4 Discussion

4.1 Miscellaneous Findings

During the process of analyzing the interfaces between apps and devices, we have some interesting observations, which are presented here.

Confusing Trust Model. We observed that IoT device developers sometimes have a confusing, if not conflicting, trust assumption regarding the local environment that their devices will run in. On one hand, they seem to assume that the local environment (i.e., consumer’s home network) is not trustworthy. They apply encryption and authentication to protect the communication between the app and the device. On the other hand, they place an excessive amount of trust on the app, e.g. they would embed the encryption keys and authentication credentials into the app. In such a scenario, an adversary within the local environment can easily bypass the protections that the device developer built, as long as the adversary has access to the Google Play Store and has a basic knowledge of reverse engineering an app. As an example, TP-Link Smart Plug (HS110) accepts commands from its mobile companion app (and potentially anywhere else from within the LAN) without authentication. The vendor seems to be concerned about local threats to this design and, therefore, encrypts the communication. However, the encryption key in use (i.e., integer 171 XOR message) is simple and static, and most importantly built into the app. Anyone with access to the app can thus forge the

communication easily. This problem is also reported by [50]. Another example is the D-Link water sensor. D-Link water sensor requires authentication from its mobile companion app. However, the credentials used to authenticate the app is fixed (i.e., not configurable by user) and built into the app. These examples highlight the confusing mindset of many IoT device developers and the lack of general understanding of security. While in this paper we do not intend to give solutions to the problem, we believe a more standard architecture developed with security in mind can help limit the freedom offered to developers thus improving the security.

"Convenient" Provisioning. Smart home IoT devices are often headless—they do not provide direct user interfaces (e.g., touch screen, keyboard). As a result, they often rely on mobile companion apps to provision the credentials of home Wi-Fi network, in order for them to join the network. Our observation through studying the device interfaces is that the provisioning method is evolving from more user interactive approaches (e.g., AP Mode, WPS and out-of-band channels such as Bluetooth) to a more automated and hands-off approach where users do not need to do anything except providing the credentials. This presumably provides convenience, but many times at the cost of security. These newer methods such as *Smart Config* [30] and *Sound Wave*[6] often artificially create a side channel between the app and the device, and rely on these channels to transfer Wi-Fi credentials. Unfortunately, these side channels are publicly observable therefore allowing the credentials to be leaked. In addition, even without considering the openness of side channels, securing a side channel can often be much more difficult than normal means of communication. This highlights the long-lasting problem of balancing usability with security.

4.2 Limitations and Future Work

The major limitation of the approach discussed in this paper is the accuracy of the analysis results. As we based our analysis solely on mobile companion apps, we are inherently limited to the information we can obtain from the app, and sometimes the information we can obtain may not be an accurate reflection of the device. For example, a device may have patched a vulnerability and the patch did not change the device interfaces at all. In this case, our analysis will still output the device as potentially vulnerable since our platform would have no clue about the existence of the patch by just inspecting the app. This, however, is a trade-off we have to make in order to study IoT device security at scale. We believe a multi-stage solution can help address this limitation where the first stage (i.e., our platform) narrows down the scope of analysis by identifying the potentially vulnerable devices, and the second stage automates the vulnerability confirmation with more targeted but rigorous analysis, e.g., dynamic/static analysis of firmware, device fuzzing.

Another limitation of the approach is that the network interface analysis can be rendered less effective in scenarios where IoT backend servers or cloud significantly decouple device interfaces from app interfaces. An example is the Google and Amazon devices where much of the management is done through the cloud. In this case, our approach can glean less information about the device software. However, information such as the Wi-Fi credential provisioning module and the backend services in-use are still available in the app, which allows the platform to predict security issues of these components.

This work could also benefit greatly from an automatic vulnerability collection system. Currently, this is a manual process: we manually collect the vulnerabilities and impacted devices that were reported publicly. We then propagate the vulnerability information to more devices through our platform. An automatic vulnerability collection system can help label the initial seed devices as well as evaluating security from a device's perspective (i.e., to find the set of security issues that a given device may have).

Another aspect to improve on is the dimension and granularity of the similarity analysis, as mentioned in Section 2.4. Further improvements to the App Analysis Engine may allow the platform to detect similarities in finer components of a device software stack (e.g., web server, PHP interpreter, web application, OS, driver) as well as other dimensions (e.g., similar developer, similar development toolchain). This would enable us to track vulnerability propagation more comprehensively and accurately. We leave the refinement of the App Analysis Engine for future work.

The general methodology, i.e., utilizing companion app analysis to study the device, also enables a number of interesting applications that we plan to explore for the future work. For example, from app analysis, we could potentially tell what types of sensors are on a device and what types of network traffic a device may produce. This would allow a home security gateway, which is shipped as a default component of many Wi-Fi routers on the market, to enforce an accurate protection profile and detect anomalous behaviors of IoT devices in real time. As another example, instead of similarities, the Cross-App Analysis Engine in our platform can detect differences between devices. Such differences may enable a more accurate fingerprinting method of the device.

5 Related Work

IoT device vulnerabilities. IoT devices are affecting increasing number of users in every aspect of their life. Meanwhile, various studies revealed that firmware of many devices is filled with vulnerabilities. For instance, Paleari [44, 45] reported that D-Link DIR-645 routers expose critical web pages to unauthenticated remote attackers, allowing them to extract

[6] Wi-Fi credentials are encoded in the sound wave and sent out directly by the phone. This method is being used by devices such as *360* and *Securenet* IP Cameras.

root credentials and take full control of the device; also multiple web interfaces are affected by stack-buffer overflow that leads to remote code execution. Cui et al. [18] found that attackers may inject malicious firmware modifications to a device while it's updated. With the vulnerabilities and the huge amount of insecure devices [31], there arise a series of large-scale attacks, such as Mirai [35], BASHLITE [1], etc. To better understand the events, researchers have conducted comprehensive study on both features of the known vulnerabilities (and malware) [15] and their propagation [6, 39].

Various analysis approaches have been proposed to identify vulnerable IoT devices. For instance, with a network scanner (i.e., *nmap*), Cui et al. [19] found that over 13% devices are publicly accessible by default credentials. Similarly, using an Internet-wide scanning, Heninger et al. [27] showed that a large amount of TLS and SSH servers on embedded devices are affected by weak certificates and keys. Online services, *Shodan* [4] for example, allow security researchers to identify vulnerable online web services and devices. Such works are effective to find devices with known vulnerabilities and evaluate their impacts, but in many cases fail to catch problems that also appear in other devices.

Further, researchers are utilizing different techniques [16, 20, 24, 25, 47, 49, 55] to statically identify vulnerabilities in the device firmware. A majority of the approaches fall into the broader category of *vulnerability search*: derive *signatures* from known vulnerabilities, and then use them to search in other firmware images. To name a few, Costin et al. [16] conducted a large-scale study by scanning 32k firmware images with simple signatures (e.g., certificates, unique keyword), which is difficult to cover vulnerabilities that are not bound to specific strings. To address the problem, following works that collect robust features from such as I/O behavior of the image binary [47] and control flow graphs [24, 25] were proposed. Instead of extracting signatures from firmware image, Xiao et al. [55] presented an approach that discovers unknown vulnerabilities based on the study of existing security patches. In addition, Davidson et al. [20] built a symbolic execution framework on top of KLEE [9] for detecting vulnerabilities in MSP430 microcontroller family, which is difficult to scale as it needs to customize for specific architectural features of an IoT device. Similarly, Shoshitaishvili et al. [49] showed how symbolic execution and other techniques (e.g., program slicing) work to find authentication bypass vulnerabilities in a firmware image. Corteggiani et al. [14] improved symbolic execution of firmware by incorporating source code semantics, etc. While these approaches are effective, they rely on the static analysis of the firmware images, and therefore are limited in cases where the images are not publicly available or cannot be unpacked. In contrast, our work focuses on finding potential vulnerabilities using analysis of the IoT companion apps, which turns to be scalable, especially for IoT devices of smaller companies.

Another effective approach is dynamic firmware analysis [10, 17, 42, 58]. Zaddach et al. [58] performed dynamic analysis by forwarding I/O access from an emulator to the actual hardware, and further, Muench et al. [42] presented how to orchestrate execution between multiple testing environments. Koscher et al. [34] used an FPGA bridge to allow the emulator full and real-time access to the hardware. While these approaches are accurate, they are not applicable to large-scale analysis because of the lack of budget to obtain the device, and the required effort to figure out the hardware interfaces. To address the problem, Costin et al. [17] built a QEMU-based emulation framework to discover vulnerabilities in web interfaces of an IoT device; Chen et. al [10] presented a full system emulation tool, FIRMADYNE, for Linux-based firmware in order to identify vulnerabilities. These works, however, also rely on feasibility of the firmware, and tend to be affected by the heterogeneous architectures of the firmware. Given the difficulty of fuzzing IoT devices directly [43], Chen et al. [11] proposed a testing method to detect memory corruptions in the device with the assistance from app analysis. Similarly, it requires presence of the physical device, and also fuzzing each individual device is time-consuming. Again, our work is more scalable since it is only based on static analysis of the companion apps, and leverages cross-app similarities as an indicator to find other potentially vulnerable devices.

Mobile app analysis. Dozens of static and dynamic techniques have been presented to analyze mobile apps. Among them, most related to our work are those [12, 32, 48, 60] proposed to collect runtime values in an Android app. These techniques may serve different purposes, e.g., collecting developer credentials [60], harvesting obfuscated/encrypted values for malware detection [48], extracting application imprints from network request [12] and reconstructing format of a protocol [32]. However, they have a basic idea in common: extracting parts of the application code (i.e., slices) that are related to the target (e.g., APIs, variables), and generating target value by only executing the slices. In this study, we designed the request construction with a similar *Instantiated Partial Execution* (IPE) as in [12]. Another related work is *Autoprobe* [56], which collects request probes from the malware, and then using the probes to fingerprint a remote malware server. *Autoprobe* is not applicable in our settings for several unique challenges. For instance, requests of an IoT device would not be triggered automatically because of the absence of the device; mobile companion apps often serve multiple devices, and thus it's difficult to pair the request and response and collect telemetry for each individual device. In our work, the interface analysis engine leverages several techniques that not only triggered the request, but also conducted a modularity analysis after locating the request/response pair.

6 Conclusion

In this paper, we present a platform to accelerate vulnerable device discovery in smart home IoT device market. Different

from previous approaches that examine real IoT devices or firmware images, our platform analyzes mobile companion apps of devices to indirectly detect device similarity and vulnerability propagation across devices, thus making it practical for large-scale analyses. By analyzing 2,081 mobile companion apps, our platform was able to discover 324 devices from 73 vendors that are potentially vulnerable to a number of security issues, out of which 164 devices from 38 vendors are confirmed to be indeed vulnerable.

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments that greatly helped us improve the paper. In particular, we want to thank our shepherd, Adwait Nadkarni, for his guidance and constructive feedback about this paper. This work is supported in part by NSF CNS-1527141, 1618493, 1801432, 1838083 and ARO W911NF1610127.

References

[1] Bashlite. https://github.com/anthonygtellez/BASHLITE.

[2] Binwalk: Firmware analysis tool. https://github.com/ReFirmLabs/binwalk.

[3] Google play. https://play.google.com/store/apps?hl=en.

[4] Shodan: the search engine for the internet of things. https://www.shodan.io.

[5] Smarthomedb. https://www.smarthomedb.com.

[6] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., ET AL. Understanding the mirai botnet. In *USENIX Security Symposium* (2017), pp. 1092–1110.

[7] BAPTISTE, D. Vulnerability in compiler leads to backdoor in software. https://2018.zeronights.ru/wp-content/uploads/materials/04-Vulnerability-in-compiler-leads-to-stealth-backdoor-in-software.pdf, 2018.

[8] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[9] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[10] CHEN, D. D., WOO, M., BRUMLEY, D., AND EGELE, M. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS* (2016).

[11] CHEN, J., DIAO, W., ZHAO, Q., ZUO, C., LIN, Z., WANG, X., LAU, W. C., SUN, M., YANG, R., AND ZHANG, K. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. *Proc. 2018 NDSS, San Diego, CA* (2018).

[12] CHEN, Y., YOU, W., LEE, Y., CHEN, K., WANG, X., AND ZOU, W. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 815–828.

[13] CORPORATION, T. R. Thermostat manufacturers. https://www.thermostat-recycle.org/thermostat_manufacturers.

[14] CORTEGGIANI, N., CAMURATI, G., AND FRANCILLON, A. Inception: system-wide security testing of real-world embedded systems software. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 309–326.

[15] COSTIN, A., AND ZADDACH, J. Iot malware: Comprehensive survey, analysis framework and case studies. *BlackHat USA* (2018).

[16] COSTIN, A., ZADDACH, J., FRANCILLON, A., BALZAROTTI, D., AND ANTIPOLIS, S. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium* (2014), pp. 95–110.

[17] COSTIN, A., ZARRAS, A., AND FRANCILLON, A. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016), ACM, pp. 437–448.

[18] CUI, A., COSTELLO, M., AND STOLFO, S. When firmware modifications attack: A case study of embedded exploitation. *Proc. 2018 NDSS, San Diego, CA* (2013).

[19] CUI, A., AND STOLFO, S. J. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 97–106.

[20] DAVIDSON, D., MOENCH, B., RISTENPART, T., AND JHA, S. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium* (2013), pp. 463–478.

[21] DIRECTORY, I. G. Ip cameras. https://directory.ifsecglobal.com/ip-cameras-code004823.html.

[22] EBAY. Smart bulbs. https://www.ebay.com/b/Smart-Bulbs/20706/bn_72322334.

[23] EBAY. Wi-fi smart plugs. https://www.ebay.com/b/Wi-Fi-Smart-Plugs/185061/bn_118504145.

[24] ESCHWEILER, S., YAKDAN, K., AND GERHARDS-PADILLA, E. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS* (2016).

[25] FENG, Q., ZHOU, R., XU, C., CHENG, Y., TESTA, B., AND YIN, H. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 480–491.

[26] FREY, B. J., AND DUECK, D. Clustering by passing messages between data points. *science 315*, 5814 (2007), 972–976.

[27] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium* (2012), vol. 8, p. 1.

[28] HOLCOMBE, J. Soho network equipment. https://www.securityevaluators.com/wp-content/uploads/2017/07/soho_techreport.pdf, 2017.

[29] HU, G., YUAN, X., TANG, Y., AND YANG, J. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 18.

[30] INSTRUMENTS, T. Simplelink wi-fi smartconfig. http://www.ti.com/sitesearch/docs/universalsearch.tsp?searchTerm=SmartConfig.

[31] INTERNET CENSUS. Port scanning /0 using insecure embedded devices. https://seclists.org/fulldisclosure/2013/Mar/166.

[32] KIM, J., CHOI, H., NAMKUNG, H., CHOI, W., CHOI, B., HONG, H., KIM, Y., LEE, J., AND HAN, D. Enabling automatic protocol behavior analysis for android applications. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (2016), ACM, pp. 281–295.

[33] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[34] KOSCHER, K., KOHNO, T., AND MOLNAR, D. {SURROGATES}: Enabling near-real-time dynamic analyses of embedded systems. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)* (2015).

[35] KREBSONSECURITY. New mirai worm knocks 900k germans offline. https://krebsonsecurity.com/2016/11/new-mirai-worm-knocks-900k-germans-offline/.

[36] LI, C., CAI, Q., LI, J., LIU, H., ZHANG, Y., GU, D., AND YU, Y. Passwords in the air: Harvesting wi-fi credentials from smartcfg provisioning. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2018), ACM, pp. 1–11.

[37] MAJKOWSKI, M. Stupidly simple ddos protocol (ssdp) generates 100 gbps ddos. https://blog.cloudflare.com/ssdp-100gbps/, Jun. 2017.

[38] MARTIN, B., AND BRAUNLEIN, F. Deepsec 2017 talk: Next-gen mirai botnet – balthasar martin & fabian braunlein. https://blog.deepsec.net/deepsec-2017-talk-next-gen-mirai-botnet-balthasar-martin-fabian-braunlein/, Sep. 2017.

[39] MARZANO, A., ALEXANDER, D., FONSECA, O., FAZZION, E., HOEPERS, C., STEDING-JESSEN, K., CHAVES, M. H., CUNHA, Í., GUEDES, D., AND MEIRA, W. The evolution of bashlite and mirai iot botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)* (2018), IEEE, pp. 00813–00818.

[40] MICROSOFT. Azure iot hub. https://azure.microsoft.com/en-us/services/iot-hub/.

[41] MIKHAIL KUZIN, YAROSLAV SHMELEV, V. K. New trends in the world of iot threats. https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/.

[42] MUENCH, M., NISI, D., FRANCILLON, A., AND BALZAROTTI, D. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* (2018), vol. 18, pp. 1–11.

[43] MUENCH, M., STIJOHANN, J., KARGL, F., FRANCILLON, A., AND BALZAROTTI, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA* (2018).

[44] PALEARI, R. Multiple vulnerabilities on d-link dir-645 devices. http://roberto.greyhats.it/advisories/20130801-dlink-dir645.txt.

[45] PALEARI, R. Unauthenticated remote access to d-link dir-645 devices. http://roberto.greyhats.it/advisories/20130227-dlink-dir.txt.

[46] PASCU, L. The iot threat landscape and top smart home vulnerabilities in 2018.

https://www.bitdefender.com/files/News/CaseStudies/study/229/Bitdefender-Whitepaper-The-IoT-Threat-Landscape-and-Top-Smart-Home-Vulnerabilities-in-2018.pdf.

[47] PEWNY, J., GARMANY, B., GAWLIK, R., ROSSOW, C., AND HOLZ, T. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 709–724.

[48] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS* (2016).

[49] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS* (2015).

[50] STROETMANN, L. Reverse engineering the tp-link hs110. https://www.softscheck.com/en/reverse-engineering-tp-link-hs110/.

[51] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (2010), IBM Corp., pp. 214–224.

[52] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 260–275.

[53] WERMKE, D., HUAMAN, N., ACAR, Y., REAVES, B., TRAYNOR, P., AND FAHL, S. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference* (2018), ACM, pp. 222–235.

[54] XIAO, C. Malware xcodeghost infects 39 ios apps, including wechat, affecting hundreds of millions of users. https://unit42.paloaltonetworks.com/malware-xcodeghost-infects-39-ios-apps-including-wechat-affecting-hundreds-of-millions-of-users/.

[55] XIAO, F., SHA, L.-T., YUAN, Z.-P., AND WANG, R.-C. Vulhunter: a discovery for unknown bugs based on analysis for known patches in industry internet of things. *IEEE Transactions on Emerging Topics in Computing* (2017).

[56] XU, Z., NAPPA, A., BAYKOV, R., YANG, G., CABALLERO, J., AND GU, G. Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 179–190.

[57] XUE, L., LUO, X., YU, L., WANG, S., AND WU, D. Adaptive unpacking of android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 358–369.

[58] ZADDACH, J., BRUNO, L., FRANCILLON, A., BALZAROTTI, D., ET AL. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS* (2014).

[59] ZHANG, Y., LUO, X., AND YIN, H. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 293–311.

[60] ZHOU, Y., WU, L., WANG, Z., AND JIANG, X. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks* (2015), ACM, p. 23.