IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing

Jiongyi Chen*, Wenrui Diao†, Qingchuan Zhao‡, Chaoshun Zuo‡, Zhiqiang Lin‡, XiaoFeng Wang§,
Wing Cheong Lau*, Menghan Sun*, Ronghai Yang*, and Kehuan Zhang*
*The Chinese University of Hong Kong
Email: {cj015, wclau}@ie.cuhk.edu.hk, mhsun@cse.cuhk.edu.hk, {yr013, khzhang}@ie.cuhk.edu.hk
†Jinan University, Email: diaowenrui@link.cuhk.edu.hk
‡The University of Texas at Dallas, Email: {qingchuan.zhao, chaoshun.zuo, zhiqiang.lin}@utdallas.edu
§Indiana University Bloomington, Email: xw7@indiana.edu

***Abstract*—With more IoT devices entering the consumer market, it becomes imperative to detect their security vulnerabilities before an attacker does. Existing binary analysis based approaches only work on firmware, which is less accessible except for those equipped with special tools for extracting the code from the device. To address this challenge in IoT security analysis, we present in this paper a novel automatic fuzzing framework, called IoTFuzzer, which aims at finding memory corruption vulnerabilities in IoT devices *without* access to their firmware images. The key idea is based upon the observation that most IoT devices are controlled through their official mobile apps, and such an app often contains rich information about the protocol it uses to communicate with its device. Therefore, by identifying and reusing program-specific logic (e.g., encryption) to mutate the test case (particularly message fields), we are able to effectively probe IoT targets without relying on any knowledge about its protocol specifications. In our research, we implemented IoTFuzzer and evaluated 17 real-world IoT devices running on different protocols, and our approach successfully identified 15 memory corruption vulnerabilities (including 8 previously unknown ones).**

I. Introduction

Recent years have witnessed the rapid progress of Internet of Things (IoT) technologies, with many of them already seeing wide adoption. Examples include smart plugs, smart door locks, smart bulbs and many others. According to a recent report [29], the number of IoT devices is projected to reach 20.4 billion in 2020, forming a global market valued $3 trillion. The booming of this IoT ecosystem inevitably attracts cyber criminals, who aim at compromising and controlling IoT devices. The loose protection of these devices and pervasiveness of vulnerabilities [15], [33], [8] in them actually present to the miscreants low-hanging fruits. For instance, from 2014 to 2016, more than 90 independent IoT attack incidents have been reported [48], with devastating consequences in some of them. A prominent example is the Mirai attack [32], which turns a large number of online IoT devices (e.g., IP cameras and home routers) into bots for launching DDoS attacks against online services. Given the pervasiveness of vulnerable devices, we strongly believe that these known attacks are nothing but a tip of the iceberg.

An important target of IoT attacks is implementation flaws (or security vulnerabilities) within a device's firmware. Systematic detection of these flaws needs to address a few challenges. The primary one is the difficulty in firmware acquisition because many vendors do not make their firmware images publicly available. Alternatively, we can dump images from the motherboard, which, however, needs the support from enabled debugging ports, which may not exist for many IoT devices, due to their simplicity. In addition, given the diversity of compression (even encryption) formats, how to unpack the obtained firmware is nontrivial as well.

When it comes to the security analysis of the files extracted from firmware, the main challenge comes from diverse underlying architectures (memory layout, instruction set, and so forth). Existing techniques mainly rely on emulation for certain architectures [23], [17], [13]. However, the programs running in the emulator will frequently crash due to unavailable NVRAM parameters. Some other related studies utilize symbolic execution to analyze firmware. This attempt is also impeded by the architecture issue. For example, FIE [21] only supports the security analysis of firmware images built on the TI MSP430 microcontroller family, and FirmUSB [31] only supports 8051 architecture.

Our Approach. Unlike traditional embedded devices, most IoT devices are controlled by users through mobile applications (*IoT app* for short). Such an IoT app is designed to act as its device's phone-side control panel, and therefore carries rich information about the device, particularly the way to talk to its firmware. Examples of such information include command (seed) messages, URLs, and encryption/decryption schemes that embedded in the app. Based on this observation, in this paper, we present IoTFuzzer, an automatic, black-box fuzzing framework designed specifically for detecting memory-corruption flaws in IoT firmware. A unique property of IoTFuzzer is that it runs a *protocol-guided* fuzz and utilizes the information carried by the IoT app *without reverse-*

Responsible Disclosure: All vulnerabilities described in this paper have been reported to the corresponding vendors.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23159
www.ndss-symposium.org

engineering the protocol or explicit recovering such knowledge from the app, as prior approaches [20], [10] do. Instead, it performs a dynamic analysis to identify the content inside the app that forms the messages to be delivered to the target device, and automatically mutates such content during the runtime so as to use the app's program logics to produce meaningful test cases for probing the target firmware. This approach turns out to be not only lightweight, avoiding heavyweight protocol analysis, but also reliable, capable of generating effective test payloads even in the presence of cryptographic protection (encryption, authentication, etc.).

On the other hand, IoTFuzzer is not designed to precisely locate software flaws [43], [44]. Like other fuzz tests, all it does is to report the presence of the problem, through a crash triggered by a test case, which is used to guide the follow-up security analysis to find out the root cause of the flaw.

In our research, we implemented a full-featured prototype of IoTFuzzer and deployed it in a real-world environment. To evaluate its effectiveness, we ran our implementation on 17 popular IoT devices. IoTFuzzer successfully discovered 15 serious memory vulnerabilities on 9 devices, and 8 of them were previously unknown. We have reported all these new vulnerabilities to the corresponding vendors, and some of them has released firmware updates.

Contributions. We summarize the contributions of the paper as follows:

- *New framework.* We present the first firmware-free fuzzing framework, IoTFuzzer, for security analysis of IoT devices. By utilizing the information carried by official mobile apps and their program logics, IoTFuzzer could automatically detect memory corruption vulnerabilities in IoT devices without direct access to the firmware.
- *New techniques.* We developed a set of new techniques to enable an automatic, blackbox IoT fuzzer, which includes protocol-guided fuzzing without protocol specifications, in-context cryptographic and networking function replay for message generation and delivery, and a lightweight mechanism to remotely monitor the target IoT device's status.
- *Implementation and findings.* We implemented a full-featured prototype of IoTFuzzer and evaluated it over 17 real-world IoT devices. Our study discovered 15 security-critical memory vulnerabilities, with 8 of them never reported before.

Roadmap. The rest of this paper is organized as follows. Section II gives the background of firmware analysis and our insights of IoTFuzzer with a running example. Section III presents the detailed design of IoTFuzzer. The evaluation results are summarized in Section IV. Section V discusses some limitations of the current design and the possible improvements. Section VI reviews the related work, and finally Section VII concludes this paper.

II. Background

In this section, we briefly introduce the typical IoT communication architecture in Section II-A and summarize the obstacles of performing security analysis on IoT devices in Section II-B. Then, we provide our motivation and insights to IoTFuzzer, as well as the practical challenges in developing IoTFuzzer in Section II-C. Finally, we discuss the testing scope and assumptions in Section II-D.

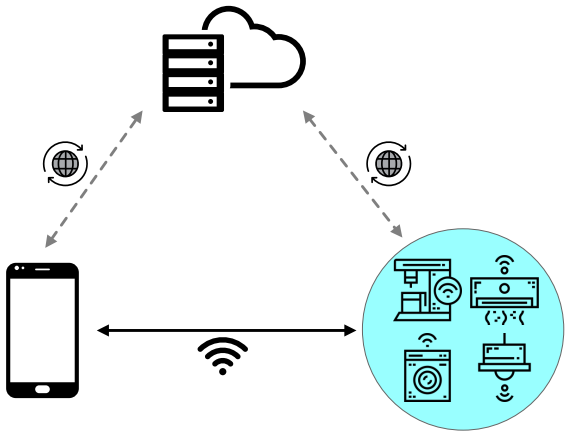

Fig. 1: Typical smart home communication architecture

A. Typical IoT Communication Architecture

In a typical IoT ecosystem (like the smart home environment, as shown in Figure 1), several smart devices are deployed by user for specific purposes. Generally, these devices are equipped with many sensors for external information collection, and a wireless connection module for data transmission. To facilitate the user's operations, IoT device vendors tend to provide the corresponding smartphone applications, called *IoT app*, as the control node. After pairing with IoT devices, such IoT app could send control commands to the IoT devices to change system settings or obtain running status data. The communication modes between an IoT device and an IoT app could be a direct connection (e.g., Wi-Fi, Bluetooth, and ZigBee) or a delegate connection through a wireless router (i.e., connected to the same Wi-Fi network). Also, some vendors provide extra cloud services to support remote device management and data storage. In such cloud-aided architecture, the device and/or IoT app will connect to the cloud server through the Internet.

In this paper, we focus on the local connection modes (direct connection and delegate connection) through Wi-Fi for universality, and our methodology can be extended to other communication channels (e.g., Bluetooth and Zigbee) easily.

B. Obstacles in Firmware Analysis

IoT devices typically run special software (also called *firmware*) providing system control, status monitoring, data collection, etc. The firmware is usually highly customized to suit the hardware chips with limited computing and memory resources. Due to its closed source and architectural diversity, it becomes extremely challenging to perform firmware analysis even with plenty of manual efforts. Here we summarize the obstacles of (manually or automatically) analyzing firmware.

- ***Firmware Acquisition.*** In some previous work [16], [13], researchers developed targeted web crawlers to collect firmware images from vendors' websites. However, in fact, many vendors do not explicitly provide images online for end users. On the other hand, hardware hackers try to dump firmware images or export a console using debugging port (e.g., JTAG interface). Nevertheless, more and more manufacturers tend to disable the debugging ports before production delivery to prevent hardware-based cracking. Therefore, if the vendors do not provide firmware or disable the debugging port on IoT devices, acquiring firmware images will be difficult. Naturally, all further firmware-based analysis would be in vain.

- ***Firmware Unpacking.*** Once we obtain the firmware images, the next step is to unpack them and extract the contained files. The immediate obstacle is the diversity of firmware formats. Moreover, the firmware images are usually compressed or even encrypted, Which makes things worse. Though the standard compression algorithms can be handled by Binwalk [22] or FRAK [18], researchers still have no clue to deal with proprietary compression algorithms or encryption algorithms without the secret keys.

- ***Executable Analysis.*** For the security analysis of the extracted files from firmware images, the main difficulty lays on the non-unified underlying architectures. To minimize the power consumption and material cost, different specifications of processors are adopted to develop specific IoT devices. As a result, the process of static binary executable analysis involves lots of manual, repetitive efforts, like adjusting load offsets and handling disassembling errors. On the other hand, for automated analysis, static bug search approaches in firmware with pattern matching [34], [26], [25] (e.g., memory safety bugs) can be quite inaccurate. Those approaches only search for similar functions and need a detailed description of the existing vulnerability. It can be difficult to describe memory corruptions because they may require points-to relations for pointers. In the aspect of dynamic analysis, one way is to emulate the program in the corresponding IoT device by separating the emulation with the real hardware [47], which requires the support of debugging ports (e.g., UART, JTAG). The other way is to run the program in an isolated emulator. However, it can be difficult without NVRAM parameters. Security analysts need to repetitively hijack certain functions to bypass exceptions so that the program can be executed. This process may not always be feasible, and it is very time-consuming.

Our Approach. In terms of discovering new vulnerabilities, the direct firmware analysis is very challenging. Thus we need to seek alternative approaches to automatically discover vulnerabilities in IoT devices. Our key observation is derived from the IoT communication architecture as illustrated in Figure 1: different from the traditional embedded devices, the IoT app plays an important role in interacting with IoT devices. Based on this observation, we find that the IoT app can be utilized as a client to test closed source IoT devices, and it is the node of sending testing messages. Such a design avoids the difficulties and tedious work in reverse engineering binary executables when discovering new vulnerabilities. As a result, this discovery motivates us to design a mobile app-based fuzzing framework.

Meanwhile, the preliminary results of our feasibility study also support the practicability of such design. We reviewed the official documents of randomly-selected 63 IoT devices on IOTLIST [2], and 61 of them are equipped with the corresponding mobile IoT apps (either iOS or Android).

C. *Challenges in* IoTFuzzer *Design*

In this paper, we present IoTFuzzer, an automated protocol-guided fuzzing framework which could send probing messages to IoT devices to trigger memory corruption vulnerabilities. Based on the fact that nowadays there is no popular and dominant protocol standard for IoT devices, IoT vendors tend to design customized data formats and protocols to meet the customized product requirements. Also, some of the vendors even use non-standard encryption functions to encrypt messages. Therefore, in order to generate protocol-guided and cryptographic consistent messages from IoT apps, we need to follow the protocol formats and encryption schemes of the corresponding devices. Though the existing technique, AutoForge [50], enables the forgery of cryptographic-consistent messages at client side by reimplementing cryptographic functions out-of-the-box, it does not suit for the IoT apps in our case. The reason is that it only works for standard protocols (like HTTP) and can not handle private cryptographic functions. To illustrate the challenges that motivate our solutions, we show a running example as follows.

A Running Example. To better understand the problem, we use a typical IoT app, TP-Link Kasa [6] as a running example. This IoT app is designed to control multiple smart home devices produced by TP-Link, including Wi-Fi smart plug, smart bulb, Wi-Fi extender, etc. Moreover, the variable names and class names are obfuscated in this IoT app to prevent reverse engineering. In the decompiled code snippet shown in Listing 1, there are two functions: one is used for constructing a request message to set the current location of the IoT device at line 2; the other function at line 17 is used for encrypting the output messages.

```
1  // Message construction
2  public final ControlResult a(...) {
3  ...
4  Object localObject = new com/tplink/
       smarthome/b/e;
5  ((e)localObject).<init>("system");
6  g localg = new com/tplink/smarthome/b/g;
7  localg.<init>("set_dev_location");
8  ...
9  localg.a("longitude", localDouble);
10 localDouble = Double.valueOf(paramDouble1);
11 localg.a("latitude", localDouble);
12 ...
13 return (ControlResult)localObject;
14 }
15
16 //Message encryption
17 public static byte[] a(byte[]
       paramArrayOfByte) {
```

```
18      ...
19      k = paramArrayOfByte[j];
20      i = (byte)(i ^ k);
21      paramArrayOfByte[j] = i;
22      i = paramArrayOfByte[j];
23      j += 1;
24      ...
25      return paramArrayOfByte;
26  }
```

Listing 1: Example Code from TP-Link IoT App

Particularly, the communication between this app and the smart device is encrypted by the non-standard encryption function `byte[] a(byte[])` at line 17, and the constructed plaintext message before passing to the encryption function is as below.

```
{"system":{"set_dev_location":{"longitude
    ":10.111213141,"latitude":51.617181920}}}
```

To achieve protocol-guided fuzzing of such a message, we wish to mutate the protocol fields (e.g., "`system`" and "`set_dev_location`") in the message before the message encryption. One possible approach is to fuzz the generated message following the target protocol specifications (e.g., HTTP with XML or JSON). However, such an approach cannot be generalized for unknown protocols. On the other hand, automatic extracting and replaying the cryptographic functions are difficult because we have to re-implement the functions and locate the (hardcoded or pre-shared) keys. When facing obfuscation, such a task will become more difficult.

Challenges. Therefore, in order to generate fuzzing messages without making assumptions on protocol formats, we need to solve the following challenges.

- ***Challenge 1: Mutating fields in networking messages.*** A fuzzer is "smart" if it has the knowledge of protocol format. The format distinguishes valid input from invalid input that is easily rejected by the program. However, nowadays IoT vendors use specific protocols in their products (as can be seen from the statistics in Table I). While it might be easy to fuzz messages with well-known protocols, it is challenging to fuzz messages with unknown ones. Therefore, we need to automatically recognize and fuzz the protocol fields for unknown protocols.
- ***Challenge 2: Handling encrypted messages.*** If the communication between an IoT device and an app is encrypted, our framework has to send the encrypted probing (mutated) messages in the same way (with the correct keys). While we could identify the cryptographic functions in the app, search for the key, and re-implement them out-of-box. It will become extremely difficult when facing complex obfuscated codes and non-standard encryption schemes (with library dependencies). Thus, we need a lightweight and flexible solution to reuse the message encryption functions in the app.
- ***Challenge 3: Monitoring crashes.*** Once a memory corruption is triggered by a mutated message, we do not know the real-time status of the IoT device, like whether the program has crashed, and which message has caused the crash. The reason is that we cannot locally monitor the running process in the system. Besides, the IoT devices have different application layer protocols with distinct exception handling mechanisms, so it is difficult to automatically identify the crash based on the semantics of the error messages. Even worse, on some devices, there are watchdogs to detect and recover the malfunctioned program from the crash. In order to identify the system crash and the corresponding probing message that triggers the crash, we need to design an effective mechanism to remotely and automatically monitor the device status.

Solutions. As previously mentioned, if we intend to perform protocol-guided fuzzing, we need to recognize the protocol fields and understand the protocol formats. However, for unknown formats, protocol reverse engineering may be quite challenging. Additionally, it is difficult to reuse the encryption functions due to various obfuscations. Fortunately, we have obtained the following insights to address the above challenges.

- ***Mutating protocol fields at data sources***. Since protocol reverse engineering for unknown protocols is expensive, we can mutate the data which is used in the protocol message (note that most of these data will be strings) at their sources (e.g., at data definition sites or some data use sites such as when passed as arguments to functions). Then correspondingly, following the original program logic, these mutated strings will eventually become protocol fields.
- ***Reusing cryptographic functions at runtime***. Since we have modified the data sources at the very beginning, the normal program execution will help us complete the message encryption procedure and generate ready-to-send messages. Therefore, we do not need to re-implement the complete encryption logic out-of-the-box.
- ***Detecting liveness with heartbeat mechanism***. Though we cannot monitor the status of the running device locally, we can infer whether the program or the system is alive by sending a heartbeat message. The heartbeat message can be any messages that query the status of the device.

D. Scope and Assumptions

The design goal of IoTFuzzer is to automatically generate protocol-aware fuzzing messages to the IoT devices from IoT apps and try to discover memory corruptions in firmware images. Therefore, we require the IoT devices under testing could be configured and controlled by the corresponding mobile apps. Also, currently the communication channel between them needs to be based on Wi-Fi, though our framework could be easily extended to other channels with some additional efforts. In addition, for the IoT apps, our framework currently focuses on the Android platform for its popularity and openness.

III. Detailed Design

In this section, we present the detailed design of IoTFuzzer as illustrated in Figure 2. At a high level, there are two phases in IoTFuzzer: app analysis phase and fuzzing phase. In the app analysis phase, it takes an IoT app as the input, analyzes the UI for network event triggering, and tracks the propagation of application protocol related fields. After those steps, IoTFuzzer has recorded all protocol fields and the corresponding functions for mutation. In the fuzzing phase, our framework mutates the interested protocol fields using dynamic instrumentation of the IoT app and dynamically monitors the crash of IoT devices. In the end, IoTFuzzer outputs alerts (i.e., crash messages) indicating potential memory corruptions. Specifically, there are four main steps to fuzz an IoT device:

1) ***UI analysis***: In the first step, the goal is to analyze the code of IoT app and discover all UI components that will lead to the sending of networking messages. With such information, we can trigger message sending events by driving the corresponding UI controls. The purpose of UI analysis is to facilitate data-flow analysis and fuzzing in the following steps.
2) ***Data-flow analysis***: In order to identify the program elements (e.g., string constant, input from system APIs, etc.) whose values are related to the content of the message to be sent to the IoT device, we track the data flows from a set of selected elements (Section III-B) to find those indeed affecting some message fields. Those program elements are then used to mutate the content of the fields for fuzzing the device. Note that unlike taint-based fuzzers [7], [28] looking for the *inputs* of a program that can reach a known vulnerable function (e.g., `printf`) inside the program, our approach utilizes the data-flow analysis to determine how to command the IoT app to generate meaningful test *outputs* for fuzzing its remote target.
3) ***Runtime mutation***: Once the protocol fields are recognized, according to the fuzzing policy we defined, IoTFuzzer mutates the original fields (e.g., original string) at their first use sites. Then, the IoT app will follow its normal execution logic to compute and build the message and send it to IoT device with the mutated data.
4) ***Response monitoring***: The final step is to monitor the running status of IoT device remotely and capture the triggered crash. For TCP-based communication, the connection between the IoT app and the IoT device will be interrupted, which is easy to detect. For UDP-based communication, we use a heartbeat mechanism to detect the crashes occurred at uncertain times.

A. UI Analysis

As the preparation for protocol field recognition and mutation, we only need to focus on the events that trigger network message delivery. Therefore, the goal of UI analysis is to determine the UI elements that eventually lead to the message delivery.

Generally, a typical IoT app works as follows: when we input to the text boxes, click the buttons on the UI, or enter another Activity, event handler (e.g., `onClick`, `onCreate`, `onResume`, etc.) will be invoked to handle the events; later, in the event handling functions, a background thread (i.e., `AsyncTask`) is created to build the output message and operate encryption/decryption and network functionalities. In order to determine the UI components that trigger network messages, we perform static analysis to associate the UI elements in different activities with targeted network APIs.

Call Path Construction. We first build the call graph of the app using Androguard [1]. Starting from the target network communication APIs (such as `URL.openConnection()` and `Socket.getOutputStream()`), we construct the backward code paths to UI event handlers. However, there are implicit control flow transitions, such as callback relation of `thread.start` and `thread.run`, and other event-driven calls. Therefore, we also list and add these implicit edges, which is obtained by system EdgeMiner [12]. The sinks of code paths are a set of event handlers which finally send network messages.

Activity Transition Graph Construction. In order to reach certain activities and trigger the network sending events during fuzzing, we need to construct the activity transition graph. To this end, we first use Monkeyrunner [3] to interact with the UI elements in each activity with a simple policy based on the order of event execution. For example, the submit button will be triggered after all the input fields are filled. Meanwhile, the text fields will be filled heuristically according to their types (e.g., a zip code, a telephone number, an address, etc.). After that, we obtain a sequence of UI events and the order of how to trigger them. Meanwhile, we also record the events that turn the current activity to another activity. For the events in each activity, we filter out the ones which will not lead to the network sending APIs according to the constructed call paths. In the end, we construct the activity transition graph with our recorded UI events. The node of activity transition graph is an activity with events (or the sequences of events) that trigger message sending APIs, and the edge is an event (or a sequence of events) that creates or resumes an activity.

In the following procedures, we will invoke the event handlers sequentially in our records based on the activity transition graph. The order of event handlers that lead to message sending APIs in each activity does not affect the following steps (i.e., data-flow analysis and fuzzing).

B. Data-flow Analysis

In order to mutate the protocol fields during the execution, we need to first recognize the protocol fields and record the functions that take protocol fields as arguments. Given the fact that command messages in IoT apps are usually constructed with hardcoded strings, user input, or system APIs, we use dynamic taint tracking with a modified version of TaintDroid [24] to recognize them.

While we could mutate the network message at network API based on known message formats, we find there is a much easier way of just mutating them at the first data use site without assuming message format. In particular, we can just mutate the protocol related data when it is first used as the argument of functions. As a result we record the functions that

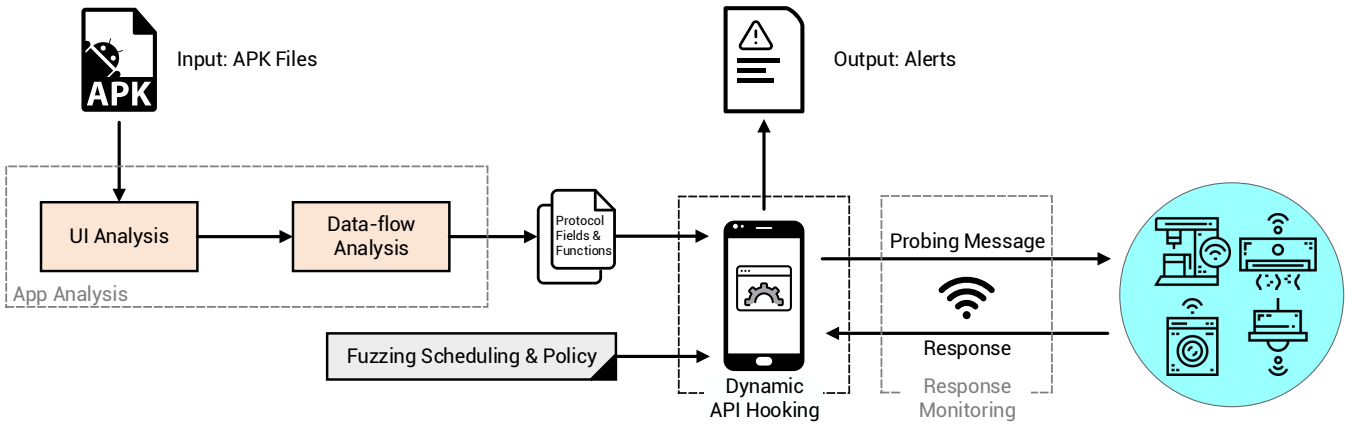

Fig. 2: Overview of IoTFuzzer

take the tainted data as argument in order to find out where to mutate these fields.

Furthermore, we do not propagate taint to encryption functions, since we focus on mutating the original data that are not encoded. Thus we perform cryptographic function identification first and then set one of the taint sinks as those cryptographic function's arguments, in addition to the sinks that are the arguments of standard network sending APIs. Also, we only record the taint propagation flows in the Dalvik VM at the variable level and do not go deeper to the native system libraries level (due to the limitation of TaintDroid). Since taint analysis has been widely used, we just describe how we customize and extend TaintDroid for our purpose.

- ***Taint Sources***. The taint sources of our analysis include all the strings in the IoT app (e.g., keywords and delimiters), the system APIs frequently used in messages (e.g., `WifiInfo.getMacAddress()`, `Location.getLatitude()`, `Location.getLongitude()`), and the user input from UI (e.g., `EditText.getText()`).
- ***Taint Propagation***. Given that the purpose of taint analysis in our case is different from that of the TaintDroid, and the current implementation of TaintDroid has limited number of sources, we therefore modified the propagation rules to enlarge more taint sources to meet our requirement. In particular, we use a dictionary to store the taint tag. Whenever there is a new tag created, we add it to our dictionary and track the dependencies of this tag. In this way, we can hold arbitrary number of taint tags, and exactly know how each tag is used and propagated.
- ***Taint Sinks***. The taint sinks are the data uses at networking APIs and encryption functions we identified. If there is no encryption involved in the network message, the taint sinks will be those standard networking APIs. If there are cryptographic functions used regardless whether it is public (developed by the general public) or private (developed by the app developers), we will perform a cryptographic function identification described below to identify them.

Cryptographic Function Identification. Identifying cryptographic functions during program execution is not a new problem. There is a significant amount of prior research in this direction. In our design, we just take the following lightweight approach that is similar to [9], [41]: cryptographic algorithms usually contain arithmetic and bitwise operations. First of all, we select the functions that contain arithmetic and bitwise operations. Even though there might be many candidate cryptographic functions, very few of them are called during the execution of message delivery. We then record the execution trace of a message sending event and refine the candidate functions based on position relative to network functions.

C. Runtime Mutation

With taint tracking, we recognize the protocol fields and the corresponding functions they pass through. In this step, we dynamically hook the recorded functions and mutate the protocol field parameters at runtime to generate probing messages. Such a design brings two benefits: (1) protocol fields can be fuzzed before they get encoded or encrypted; (2) the fields of unknown protocols can be fuzzed without protocol reverse engineering.

Function Hooking. During the app execution, the tagged variables are passed to a set of functions. In the data-flow analysis, we only need to record the first passed function as well as its calling context because mutation at the first function is enough to change the tagged variable's value in the succeeding program logic. Note that most of the protocol fields are hardcoded strings defined in the code, and developers rarely sanitize the passed parameters. Then we perform de-duplication and get the unique set of these recorded functions and dynamically hook them with Xposed [37] framework. After that, the original values passed to the hooked functions can be replaced by the mutated values.

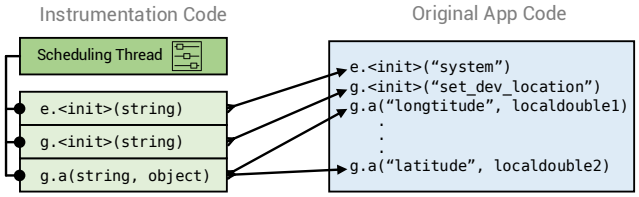

Fig. 3: Function Hooking of Original App Code

In the running example of Section II-C, after taint tracking, we tag the source string variables (most of the data sources are hardcoded strings) as follows:

```
"system", "set_dev_location", "longitude",
"latitude", 10.111213141, 51.617181920
```

The recorded functions related to the tag variables are (partially shows as below):

```
com.tplink.smarthome.b.e.<init>(String)
com.tplink.smarthome.b.g.<init>(String)
com.tplink.smarthome.b.g.a(String, Object)
```

Then we hook these functions and mutate their corresponding taint tagged arguments. Particularly, we require the hooked function to be context-sensitive (i.e., it is only triggered at certain context). For instance, if we hook Java APIs (e.g., `valueOf()`), other parts of the program will be affected if we do not consider the context.

When tagged variables are passed to the function with the parameter `Object`, we also hook this function and mutate the type of the incoming variable. For example, when `g.a("longtitude", 12.3)` is invoked (the function signature is `g.a(String, Object)`). We hook and fuzz it as: `g.a("longtitude", "a random string")`. Particularly, we can mutate the passed fields with different strategies based on the incoming values, even though the same function may be invoked many times for different protocol fields (e.g., `g.a(String, Object)`). Note that a protocol field may be fuzzed several times if it is passed to several hooked functions, and the same function may be hooked to fuzz several protocol fields (as shown in Figure 3). Nevertheless, it will never incur too much burden on the performance since mutating is absolutely based on function hooking.

Fuzzing Scheduling. When a hooked function mutates the field, we do not know whether the protocol field is already fuzzed. To schedule the fuzzing process, we assign mutations to function parameters using the algorithm as shown in Algorithm 1. Assigning of mutations is conducted before each event handler is triggered. For each message, we want to randomly select a subset of fields to mutate instead of mutating all the fields (because the message with all the fields being mutated can be easily rejected by the device). In particular, firstly, we extract the parameter set P from all hooked functions. Then we randomly generate the total mutation amount s which is a positive integer less than c (the number of identified fields in message M). Such a requirement is to decrease the chance of mutating all fields by limiting the number of mutations to less than the number of message fields for each execution. Finally, we calculate a group of valid solutions for equation $t_1 + t_2 + \cdots + t_n = s$ (t and s are integers), in which t_x is the mutation amount assigned to parameter field p_x. The solutions to that equation are not unique, and we just randomly select one valid group. Since the real-time amount of invoked functions is not available at runtime, picking a random valid group is a non-biased solution. This policy can be flexible to suit different scenarios such as emphasizing mutation times for certain functions or protocol fields.

Algorithm 1: Random Fuzzing Algorithm

Input: c: number of identified fields in message M
F: set of hooked functions
Output: T: number of mutations for each protocol field parameter of F

1 $P = \{p_1, p_2, \ldots, p_n\}$ = `extract_param`(F) ; `// get parameter set`
2 n = `count`(P) ; `// get the number of parameters`
3 s = `random_gen`(c) ; `// randomly generate s, (0<s<c)`
4 $T = \{t_1, t_2, \ldots, t_n\}$ = `get_solution`($t_1 + t_2 + \ldots + t_n$ = s) ; `// calculate one group of solutions`
5 `output` T

Fuzzing Policy. In general, there are mainly two types of fuzzing strategies [39]: generation-based fuzzing and mutation-based fuzzing. The former requires the understanding of protocol formats and generates inputs from scratch with structure information, to avoid being directly rejected. The latter only mutates the existing seed inputs with the type information. Since the runtime mutation of protocol field is inherently input structure aware, we can achieve a fuzzing strategy with structure and type information by adopting the following heuristic mutation rules for the fields[1]:

1) *Changing the lengths of strings for stack-based or heap-based overflow and out-of-bound access.* In our implementation, IoTFuzzer duplicates the original strings several times (from dozens to thousands) or appends a variable number of character `"A"` to the original string to construct malformed messages.
2) *Changing the integer, double or float values for integer overflow and out-of-bound access.* Therefore, we mutate the original values into boundary cases and very-large values. Also, to trigger the cases of miscounting of boundary conditions, we also generate the off-by-one values for potential off-by-one error.
3) *Changing the types, or providing empty values for misinterpretation of the value and uninitialized variable vulnerability.* For example, if a string value is replaced with an integer value, a null pointer dereference may be triggered (as the case of Section IV-E1). In the implementation of IoTFuzzer, we mutate the types of `Object` at the Java level of Android apps.

[1]Note that though the mutation is based on Java programming language, the mutated values are targeting at the memory corruptions in binary programs.

D. Response Monitoring

When the fuzzing message has been received by the target IoT device, we still need to know whether it triggers any abnormal behaviors (like system crashes). Therefore, in this last step, we focus on monitoring the running status of the device.

Given that we cannot monitor the system processes locally, the device status information can only be obtained or inferred from the responses answered by the IoT devices. Overall, the possible responses can be classified into the following categories:

1) *Expected Response*. In this case, the probing messages are handled properly by the IoT device, and no exception occurs. Such a situation is out of our interests.
2) *Unexpected Response*. The probing messages go beyond the intended logic of the program (e.g., reaching the input parameter boundary), and trigger untreated errors.
3) *No Response*. When no response is answered for a certain probing message, it may either trigger a DoS vulnerability, or it is just an error handled locally without replying, or running in a dead loop.
4) *Disconnection*. For connection-oriented communication protocol like TCP, the network connection will be interrupted when a system crash is triggered by the probing message.

In our IoTFuzzer framework, we focus on detecting memory corruptions that cause system crash. However, manual understanding of the response can be time-consuming and may not be able to monitor the status remotely. Therefore, based on the above listed four types of responses, we use a targeted crash detecting mechanism for different transport layer communications between the IoT device and the IoT app. More specifically:

- ***For TCP-based connection***, we simply infer whether the system crashed or not by looking at the connection status (since TCP is a connection-based protocol).
- ***For UDP-based connection***, if the program or the system is crashed, then there will be no response sent back to the IoT app. We need to determine that the no response is caused by a crash or it is just an internally handled error. To distinguish such two cases, we use a heartbeat insertion mechanism. In particular, we first extract a heartbeat message from the IoT app, which is used to detect whether the device is alive. Heartbeat messages are extracted based on the analysis about network functions as well as a differential analysis over network traffic with and without user's interactions with the paired app. Then, during the fuzzing, we insert a heartbeat message for every ten (note that this number can be easily changed, and it just affects how long we should wait for the heatbeat response) probing messages for a liveness detection. Under such configuration, ten probing messages can be sent within seconds, which is tolerable with the watchdog daemon. If there is no response to answer the heartbeat message, then we can confirm that a crash is triggered by one of previous ten probing messages. Then we will do a further check to locate the exact probing message that triggers the crash.

IV. Implementation and Evaluation

In this section, we present the prototype implementation of IoTFuzzer and analyze the evaluation results. Particularly, Section IV-A gives the implementation details, and Section IV-B introduces the experiments setup (the devices and the testing environment). In Section IV-C and Section IV-D, we discuss the effectiveness and efficiency of IoTFuzzer based on the evaluation results. Finally, two vulnerable devices are analyzed deeply to demonstrate how IoTFuzzer could help security analysts to locate the vulnerabilities in firmware.

A. Framework Implementation

We have implemented a full-featured prototype of IoTFuzzer with around 9,100 Java lines of code and 1,400 Python lines of code in total. Also, we integrated several open-source projects (e.g., Xposed and TaintDroid) into IoTFuzzer to avoid reinventing the wheel.

For the app analysis phase, we implement call path construction and automatic activity transition with Androguard [1], EdgeMiner [12] and Monkeyrunner [3]. Then we rely on Xposed Module [37] and Monkeyrunner to trigger network events for taint analysis and subsequent message delivery operations. During this step, Xposed provides an excellent tool with method hooking and replacing functionalities. Moreover, we implement taint tracking with TaintDroid [24] by extending its taint source and taint tracking policy (as described in Section III-B). Finally, the outputs (i.e., functions) of taint tracking are written into configuration files for the further usage.

For the fuzzing phase, the core functionalities (scheduling, mutation and crash monitoring) are implemented by creating an analysis thread in Xposed during app execution. Also, the configuration files are preloaded to provide the information of target functions. Note that the analysis thread is designed to schedule the fuzzing through assigning mutation quotas to each hooked function, and the mutation is still performed in the original threads of the app.

Crash Triage. It should be noted that some of the confirmed crashes are triggered by the mutated messages of the same seed messages with minor differences (different instances but could trigger the same bug). Therefore, in the implementation of IoTFuzzer, we also record the relationship of each seed message and mutated message pair, which could help us triage the crash messages. As a result, when we obtain a set of crash messages produced by IoTFuzzer, we could compare them with the corresponding seed messages to locate the mutated fields and further confirm the vulnerabilities in firmware images.

B. Experiment Setup

IoT Device Selection. We selected 17 representative IoT devices from different categories, especially smart home devices,

Fig. 4: IoT Devices Used for Our Experiments

including network-attached storage (NAS) device, IP camera, smart bulb, smart plug, printer, home router, etc. These devices are best-selling products offered by mainstream manufacturers, and rank top few in each category on Amazon. All of those devices could be operated by the official IoT mobile apps through local Wi-Fi network. There is no restriction on the communication protocols and the data transmission formats. Also, while we could have performed test with more devices, we have to note that each device costs our research budget and therefore only 17 devices were purchased due to limited resources.

The detailed specifications (type, vendor, model, and firmware version) of selected testing devices are described in Table I. In particular, we summarize their official IoT app information and communication protocols & formats (encryption or not). To deliver the direct impression of these devices, we also give their pictures in Figure 4.

Testing Environment. Our IoT UI analysis runs on a Ubuntu 14.04 PC with Intel Core i7 quad-core × 2.81 GHz CPU and 8 G RAM, and taint tracking runs on Google's Nexus 4.

For the fuzzing experiment environment, we configured the IoT devices under testing on a fully-controlled local Wi-Fi network setup by us, which avoids the interference of irrelevant traffic and unwanted package filtration of the firewall. After the device initialization, we paired these devices with the corresponding IoT apps installed on the smartphone. The smartphone is connected to the same wireless LAN.

C. Effectiveness

By performing fuzz testing on 17 IoT devices with our automated framework IoTFuzzer (each device is set to run for 24 hours), we found 15 serious vulnerabilities (memory corruptions) in 9 devices. As can be seen in Table II , these include 5 stack-based buffer overflows, 2 heap-based buffer overflows, 4 null pointer dereferences and 4 crashes that we further checked after they are identified by the IoTFuzzer.

All of these memory corruptions that we discovered with IoTFuzzer are of high impacts. They can either cause these best-selling devices out of service or be exploited and controlled by the attackers with one single message. Within those vulnerabilities that we have confirmed, seven (46.7%) of them are remotely exploitable. As a result, the attackers can make the Brother printer out of service (by only knowing the IP address) or crash the TP-Link Smart plug (with details in Section IV-E1).

Compared with the traditional approaches that discover vulnerabilities through firmware analysis with symbolic execution and firmware emulation respectively (like FIE [21] and Firmadyne [13]), IoTFuzzer adopts a firmware-free methodology and is much more efficient. After all, performing firmware emulation requires lots of manual efforts, and program analysis techniques could be inaccurate in identifying memory corruptions across different firmware architectures (as shown in Section IV-E1). Moreover, it is required to dig deep down to the program logic and the lowest level of binary code. In contrast, IoTFuzzer achieves an efficient protocol-guided fuzzing of inputs without heavy workload of firmware analysis.

D. Efficiency

Fuzzing Efficiency. We measure the efficiency of fuzzing in terms of crashes discovered over time and crashes over number of test cases. As can be seen in Table III, we compare IoTFuzzer with default configured open source network protocol fuzzers: Sulley [27] and BED [5]. Sulley is a framework which can mutate general request and watch the network, and BED is a program which is designed to check daemons for potential buffer overflows and format strings. Note that we believe both two fuzzers are fairly treated because they are both designed for general requests such as URL and HTTP. Additionally, they can both automatically identify the specifications of general protocols (e.g., HTTP headers). In the experiment, we feed mutation-based fuzzer Sulley with seed messages generated from the app. Since BED supports a set of standard protocols, we use BED to test general HTTP protocol. To compare the efficiency with IoTFuzzer, we reconfigured Sulley and BED to make them run for 24 hours. Additionally, when a crash is detected during the experiment, we resume the testing by resetting the IoT devices.

From Table III, we can see that IoTFuzzer generally surpasses popular open source fuzzers regarding both test time and the number of test cases. It identified 15 memory corruptions taking only 52.79 hours with 184159 fuzzing messages in total. In particular, IoTFuzzer discovered 3 null pointer dereferences by reusing original encryption functions in the app. Both Sulley and BED cannot handle such tough cases. However, IoTFuzzer has to take more time to test when the vulnerability is triggered by accessing a mutated URL. Furthermore, regarding detecting Buffer Overflow 3 and Crash 3, BED performed better than our framework because both vulnerabilities occur in HTTP headers and BED is designed for such vulnerability types.

In addition to that, in Table IV, the performance of UI analysis is presented. Based on such statistics, we could find, in most apps, the amounts of network events and activities are quite small, which means most IoT apps only have relatively

TABLE I: Summary of IoT Devices under Testing

Device Type	Vendor	Device Model	Firmware Version	Official Mobile App (Android[1])	Protocol and Format (Encrypted: Yes/No)
IP Camera	D-Link	DCS-5010L	1.13	com.dlink.mydlinkmyhome	HTTP, K-V Pairs (N)
Smart Bulb	TP-Link	LB100	1.1.2	com.tplink.kasa_android	UDP, JSON (Y)
	KONKE	KK-Light	1.1.0	com.kankunitus.smartplugcronus	UDP, String (Y)
Smart Plug	Belkin	Wemo Switch	2.00	com.belkin.wemoandroid	HTTP, XML (N)
	TP-Link	HS110	v1_151016	com.tplink.kasa_android	TCP, JSON (Y)
	D-Link	DSP-W215	1.02	com.dlink.mydlinkmyhome	HNAP, XML (N)
Printer	Brother	HL-L5100DN	Ver. E	com.brother.mfc.brprint	LPD & HTTP, URI (N)
NAS	Western Digital	My Passport Pro	1.01.08	com.wdc.wd2go	HTTP, JSON (N)
		My Cloud	2.21.126	com.wdc.wd2go	HTTP, JSON (N)
	QNAP	TS-212P	4.2.2	com.qnap.qmanager	HTTP, K-V Pairs (N)
IoT Hub	Philips	Hue Bridge	01036659	com.philips.lighting.hue	HTTP, JSON (N)
Home Router	NETGEAR	N300	1.0.0.34	com.dragonflow	HTTP, XML (N)
	Linksys	E1200	2.0.7	com.cisco.connect.cloud	HNAP, XML (N)
	Xiaomi	Xiaomi Router	2.19.32	com.xiaomi.router	HTTP, K-V Pairs (N)
Story Teller	Xiaomi	C-1	1.2.4_89	com.xiaomi.smarthome	UDP, JSON (Y)
Extension Socket	KONKE	Mini-K Socket	sva.1.4	com.kankunitus.smartplugcronus	UDP, String (Y)
Humidifier	POVOS	PW103	v2.0.1	com.benteng.smartplugcronus	UDP, String (Y)

Remarks: All IoT apps mentioned in this table could be obtained from Google Play.

TABLE II: Summary of Discovered Vulnerabilities

Device	Vulnerability Type	# of Issues	Remotely Exploitable?
Belkin WeMo (Switch)	Null Pointer Dereference	1	No
TP-Link HS110 (Plug)	Null Pointer Dereference	3	No
D-Link DSP-W215 (Plug)	Buffer Overflow (Stack-based)	4	Yes
WD My Cloud (NAS)	Buffer Overflow (Stack-based)	1	Yes
QNAP TS-212P (NAS)	Buffer Overflow (Heap-based)	2	Yes
Brother HL-L5100DN (Printer)	Unknown Crash	1	Not determined
Philips Hue Bridge (Hub)	Unknown Crash	1	Not determined
WD My Passport Pro (NAS)	Unknown Crash	1	Not determined
POVOS PW103 (Humidifier)	Unknown Crash	1	Not determined

TABLE III: Statistics on Fuzzing – Test Time and Number of Cases

Vulnerability	Device	IoTFuzzer	Sulley	BED
Null Dereference 1	TP-Link HS110	0.71 h (2517)	NA	NA
Null Dereference 2	TP-Link HS110	1.56 h (7068)	NA	NA
Null Dereference 3	TP-Link HS110	4.38 h (14839)	NA	NA
Null Dereference 4	Belkin WeMo	19.52 h (62424)	>24 h (309985)	>24 h (30274)
Buffer Overflow 1 (Stack-based)	D-Link DSP-W215	3.22 h (9392)	>24 h (314297)	>24 h (28131)
Buffer Overflow 2 (Stack-based)	D-Link DSP-W215	3.34 h (14696)	>24 h (314297)	>24 h (28131)
Buffer Overflow 3 (Stack-based)	D-Link DSP-W215	4.50 h (11110)	>24 h (314297)	0.87 h (1249)
Buffer Overflow 4 (Stack-based)	D-Link DSP-W215	10.85 h (42478)	>24 h (314297)	>24 h (28131)
Buffer Overflow 5 (Stack-based)	WD My Cloud	5.49 h (20323)	>24 h (333255)	>24 h (28493)
Buffer Overflow 6 (Heap-based)	QNAP TS-212P	2.95 h (10068)	>24 h (286552)	>24 h (29319)
Buffer Overflow 7 (Heap-based)	QNAP TS-212P	3.27 h (11811)	>24 h (286552)	>24 h (29319)
Crash 1	Brother HL-L5100DN	0.23 h (1021)	0.15 h (2034)	0.21 h (359)
Crash 2	Philips Hue Bridge	1.70 h (7415)	>24 h (308424)	>24 h (31810)
Crash 3	WD My Passport Pro	3.24 h (11016)	>24 h (323848)	0.28 h (453)
Crash 4	POVOS PW103	4.11 h (12832)	NA	NA

NA: not applicable for encrypted messages; >24 h: not found in 24 hours

simple logic. In Table V, we give the statistics of taint tracking (the numbers of identified fields, identified messages, and hooked functions). From this table, we could find the values for their devices are various due to different firmware and protocol implementations. For some apps (IoT devices), the number of hooked function is small because some functions are frequently used during message construction.

Fuzzing Accuracy. In Figure 5, we show the number of crashes reported by IoTFuzzer and the number of crashes we confirmed among them. This bar chart demonstrates that some IoT devices were actually poorly designed and network reliability cannot be fully guaranteed. As a result, the absence of heartbeat responses from the device or disconnection of TCP-based connections could be caused by unpredictable communication errors in the experiment. This phenomenon is the main cause of false positives existing in our results. To remove these false positives, we resent every non-responding mutated messages to the device to double confirm whether the target indeed crashed.

E. Case Studies

1) TP-LINK Wi-Fi Smart Plug: TP-Link corporation released the HS110, a Wi-Fi smart plug with energy monitoring for residential use. Through the official control app - TP-Link Kasa, the user can read the real-time power consumption

TABLE IV: Statistics of UI Analysis

IoT Devices	# of Identified Network Events	Total # of Identified Activities
D-Link DCS-5010L	16	5
TP-Link LB100	24	19
KONKE KK-Light	19	7
Belkin Wemo Switch	25	2
TP-Link HS110	32	20
D-Link DSP-W215	18	16
Brother HL-L5100DN	2	5
WD My Passport Pro	14	3
WD My Cloud	11	3
QNAP TS-212P	45	5
Philips Hue Bridge	19	15
NETGEAR N300	23	12
Linksys E1200	26	9
Xiaomi Router	42	34
Xiaomi Story Teller	11	6
Mini-K Socket	29	15
POVOS PW103	36	13

TABLE V: Statistics of Taint Tracking

IoT Device	# of Identified Fields	# of Identified Messages	# of Hooked Functions
D-Link DCS-5010L	258	11	23
TP-Link LB100	167	19	16
KONKE KK-Light	94	22	12
Belkin Wemo Switch	393	18	16
TP-Link HS110	649	39	15
D-Link DSP-W215	364	16	21
Brother HL-L5100DN	4	2	2
WD My Passport Pro	46	8	5
WD My Cloud	40	8	5
QNAP TS-212P	891	42	9
Philips Hue Bridge	933	24	13
NETGEAR N300	956	18	17
Linksys E1200	793	20	12
Xiaomi Router	1082	33	11
Xiaomi Story Teller	357	17	6
Mini-K Socket	104	19	9
POVOS PW103	89	25	8

data conveniently. This device can be accessed via local Wi-Fi network or the Internet. Even though such a device provides basic switch on/off functionality with energy monitoring and scheduling, it plays a safety-critical role: it determines the on/off status of any appliance that plugs on it. Attacker can power on/off the appliance through switching on/off the Wi-Fi smart plug.

We start the testing with UI analysis and taint tracking, then IoTFuzzer starts to perform runtime mutation. To show the original message and the fuzzed message, we record the original field and the mutated field at each hooked function. Also, we dump the plaintext of output message before the encryption function. After around 43 minutes, IoTFuzzer produced a crash with the following output message:

```
{"schedule":{"add_rule":{"stime_opt":4095,"
    wday":[1,0,0,1,1,0,0],"smin":"A...(Ax10)
    ...A","enable":254,"repeat":"A...(Ax100)
    ...A","":0,"name":0,"eactttttttttt
    ":-1,"":0,"sactttttt":1,"year":0,"
    longitude":"A...(Ax10)...A","day":0,"
    force":0,"latitude":"A...(Ax10)...A","
    emin":0},"set_overall_enable":{"enable
    ":1}}}
```

The original message is:

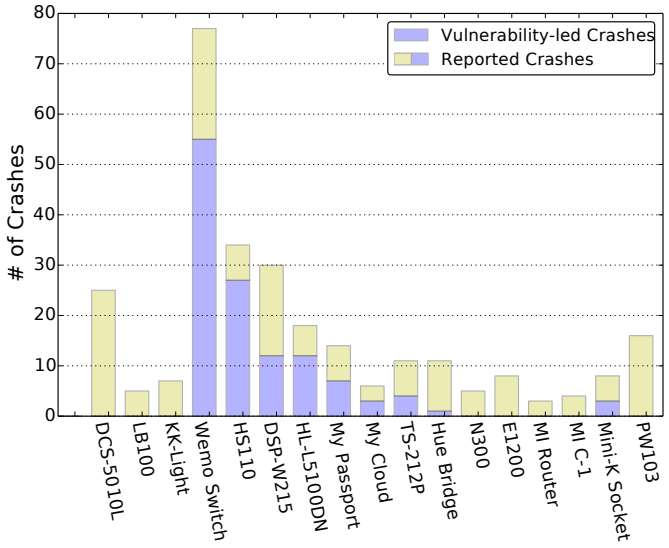

Fig. 5: Fuzzing Accuracy

```
{"schedule":{"add_rule":{"stime_opt":0,"wday
    ":[1,0,0,1,1,0,0],"smin":1014,"enable
    ":1,"repeat":1,"etime_opt":-1,"name":"
    lights on","eact":-1,"month":0,"sact":1,"
    year":0,"longitude":0,"day":0,"force":0,"
    latitude":0,"emin":0},"set_overall_enable
    ":{"enable":1}}}
```

When the mutated messages are delivered to the Wi-Fi smart plug, the device will blink in red and denies any valid messages. We further checked and confirmed that these identified vulnerabilities are triggered by the use of uninitialized pointers through complicated firmware analysis. The binary code is in MIPS and shown as below.

```
1  0x00423F14:  lui    $a1, 0x5C
2  0x00423F18:  lw     $v0, 0x14($v0)
3  0x00423F1C:  la     $t9, cJSON_GetObjectItem
4  0x00423F20:  sw     $v0, 8($s1)
5  0x00423F24:  la     $a1, aName_2    # "name"
6  0x00423F28:  jalr   $t9 ;
       cJSON_GetObjectItem
7  0x00423F2C:  move   $a0, $s4
8  0x00423F30:  lw     $gp, 0x38+var_28($sp)
9  0x00423F34:  beqz   $v0, loc_42469C
10 0x00423F38:  nop
11 0x00423F3C:  la     $t9, strncpy
12 0x00423F40:  lw     $a1, 0x10($v0)
13 0x00423F44:  addiu  $a0, $s1, 0x35
14 0x00423F48:  jalr   $t9 ; strncpy
15 0x00423F4C:  li     $a2, 0x20
16 0x00423F50:  lw     $a1, 0($s1)
17 0x00423F54:  lw     $gp, 0x38+var_28($sp)
18 0x00423F58:  beqz   $a1, loc_42469C
```

From the above binary code, we can see that standard C library function `strncpy()` is invoked (with register `a1` and `a0` as arguments) at line 14. Before the invocation of `strncpy()`, at line 12, argument `a1` is assigned by the

memory content at `v0` with offset `0x10`, which is a pointer points to the expected string in the `CJSON struct`. When we provide integer value 0 instead of a string (e.g. "lights on"), this pointer `[0x10($v0)]` is not initialized. Therefore, null pointer deference will be triggered in function `strncpy()`.

2) Belkin WeMo Switch: This is another interesting case that we apply IoTFuzzer to a Smart Wi-Fi Switch: the Belkin Wemo Switch. Users can conveniently configure the switch within local network or over the Internet. To test the switch, we pair and configure the switch with its IoT app. Then we follow the same procedures to run IoTFuzzer. It outputs a crash with the following fuzzing message:

```
<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.
    org/soap/envelope/" s:encodingStyle="http
    ://schemas.xmlsoap.org/soap/encoding/">
   <s:Body>
      <u:SetSmartDevInfo xmlns:u="urn:Belkin:
          service:basicevent:1">
      </u:SetSmartDevInfo>
   </s:Body>
</s:Envelope>
```

For the above test case, if no content is provided for the command message `"SetSmartDevInfo"`, the message would cause the switch to crash and automatically reboot. We first led wires from the motherboard and connected the debugging pins to a computer with a serial to USB converter (as shown in Figure 6). After we set this up, we resent the fuzzing message, and it printed the following message in the console, indicating that the message triggers an invalid read access from `0x00000000`.

```
19:34:50.036 stuntsx0x484f4 STUN client
    transaction destroyed sending SIGSEGV to
    wemoApp for invalid read access from
    00000000 (epc == 2b092804, ra == 2ab2cf48
    ) Cpu 0
$ 0   : 00000000 00000001 2b0927f0 0003a7f0
...
...
...
Call Trace:
Code: 00000000  2484ffff  24840001 <90820000>
     1040002d  00000000  5449fffc  248
thready: Destructor freeing name "ChildFDTask
    ".
Aborted thready: make_tname_key key now "0"
    GetLock (1938): Initializing Robust mutex
     for syslog, et al
thready:make_tname_key key now "0"
thready:Setting thread name to "main"
(tid:715849728)
```

We then confirmed the vulnerability in the firmware image by manually searching the keyword "SetSmartDevInfo". It turns out that the object `<SmartDevURL>` should be provided within the object `SetSmartDevInfo`. Without `<SmartDevURL>`, a pointer in the data structure is not initialized, and it then causes a read from `0x00000000`.

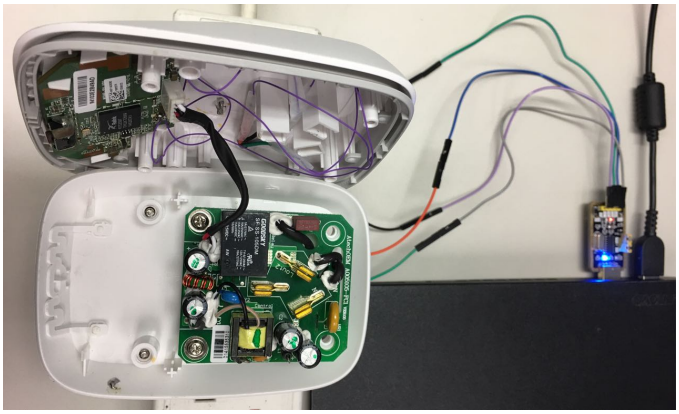

Fig. 6: The Belkin Wemo Switch. The debugging ports on the motherboard is connected to a computer with a serial to USB converter

V. Discussion and Limitations

Although our framework can discover memory corruptions in IoT devices efficiently, there are still some avenues for future improvements. In this section, we discuss the limitations existing in the current design and explore how they could be handled in the future.

Scope of Testing. While our IoTFuzzer achieves high specification coverage (for protocols), the code coverage of firmware and the coverage of attack surface are limited. The primary focus is *automatically identifying* memory corruptions from the data input channel of mobile apps. Memory corruptions could result in program crashes or abnormal program behaviors, which is always treated as a serious software safety threat. Discovering other vulnerability types such as authentication bypass [38] are remained as future work. For detecting (exploiting) such a memory vulnerability, IoTFuzzer (attackers) must input actual data with debugging (malicious) intent. For modern IoT devices, mobile apps provide the main data input channels for convenient device management. Though other data channels, like sensors or debug ports, may be exploited by attackers, the bar of exploiting is quite high due to the requirement of physical accessing, which makes attacks difficult to succeed in practice. Therefore, at this stage, we focus on the data input channel of IoT apps to design IoTFuzzer and leave other channels for future investigations.

Connection Mode. In the current implementation of IoTFuzzer, we only focus on the devices with Wi-Fi connection to the mobile app. However, there is no special technical obstacle for us to extend IoTFuzzer to other communication modes, like Bluetooth, Zigbee.

Cloud Relay. We do not consider the IoT devices which employ the cloud as a message relay/delegate. In such communication architecture, the request messages will be sent to the cloud set up by the vendor, then the cloud server relays the messages to the device, and vice versa (for response messages). Under such a scenario, the requests may be filtered by the cloud and trigger firewall alarms, thus breaking the work of IoTFuzzer. It is our next step to extending our techniques to include a cloud component.

Result Judgments. IoTFuzzer cannot generate memory corruption types and root causes directly. Similar to other fuzzers such as PEACH [4] and Sulley [27], the design target of IoTFuzzer is to automatically produce vulnerability alerts (i.e., crash message) to assist security analysts locating and confirming the root causes in firmware easily. For example, analysts could search the keywords mentioned in crash messages as the clue to locate the vulnerable binary code block, or utilizing existing emulation tools such as Firmadyne [13] and Avatar [47]. In fact, in general black-box testing, the final vulnerability confirmation always requires some kinds of manual efforts.

Result Accuracy. IoTFuzzer might present false positive and false negatives. For false positives, some crash messages reported by IoTFuzzer are derived from the inevitable and unpredictable network errors (see Section IV-D), especially for TCP-based connections. For false negatives, IoTFuzzer may miss some crashes if they occur in child processes and are handled by the corresponding parent processes properly. Note that, the thread crash will always affect the whole process and lead to process crash because a thread does not have its own address space. Also, memory corruptions will not result in crashes in some cases. For example, a buffer overflow leads to a corrupted local variable, and it does not corrupt the stack frame or the return address. Detection of such corruptions still remains an open problem for black-box fuzzing.

VI. Related Work

With the increasing prevalence of IoT devices, there are already efforts of detecting security vulnerabilities in those devices. In this section, we review these works from two categories: fuzz testing and IoT device security.

A. Fuzz Testing

Fuzz testing is a widely studied topic. Since IoTFuzzer belongs to the taint-based fuzzing techniques on Android apps, we categorize the literature into taint-based fuzzing and fuzzing on Android.

Taint-based Fuzzing. At a high level, IoTFuzzer adopts a taint-based fuzzing approach. Though taint-based fuzzing is not a new technique, how to apply it to different problem settings still tackles unique challenges. For instance, TaintScope [40] leverages taint propagation information to bypass checksum checks via control flow alteration and thus improves fuzzing coverage. Both Bekrar et al. [7] and Ganesh et al. [28] identify the untrusted input that leads to critical security functions using taint analysis and then fuzz the identified untrusted input. These works target at detecting the vulnerabilities of programs on the same platform and improve the efficiency through fuzzing only a subset of inputs. In contrast, our approach utilizes the data-flow analysis to determine how to command the IoT app to generate meaningful test outputs for fuzzing its remote target.

Fuzzing on Android. There exist some approaches utilizing Android apps for remote server vulnerability fuzzing. For instance, AutoForge [50] tries to generate cryptographically consistent messages to identify password brute-forcing in mobile services, by differing inputs to identify message fields in mobile applications. AuthScope [51] identifies the vulnerable authorizations by mutating the reverse engineered security tokens and resource IDs. Both AutoForge and AuthScope assume standard protocols such as HTTP and HTTPS. Compared with them, IoTFuzzer achieves protocol-guided fuzzing for unknown protocols by replaying cryptographic functions in context at runtime. Another similar work is SmartGen [49] which performs symbolic execution on mobile apps to expose harmful server URLs. However, IoTFuzzer targets at memory corruptions in IoT devices by triggering the network events and mutating the seed messages in IoT apps.

On the other hand, fuzzing has been widely used to test Android OS and apps. For example, AppsPlayground [36] is a framework that consists of intelligent GUI exploration and fuzz testing for vulnerability detection and malware analysis for Android apps. IntentFuzzer [45] could detect the violation of permission model by sending mutated Intents to various interfaces. Buzzer [11] fuzzes the Android system services by sending requests with malformed arguments to them. DroidFuzzer [46] targets at activities that process MIME data passed via a URI, and the activities are picked out by analyzing the Intent-filter tag in the `AndroidManifest.xml` file. However, the design goal (security analysis of IoT devices) of IoTFuzzer differ from those works significantly, which also bring several technical distinctions.

B. Embedded / IoT Device Security

Previous research on vulnerability detection for embedded devices focuses on certain attack surfaces in firmware images. The most related work to ours is RPFuzzer [42], a fuzzing framework for vulnerability detection in routers. RPFuzzer monitors routers by sending normal packets, keeping watch on CPU utilization and checking system logs. However, it requires monitoring the running process. In contrast, IoTFuzzer can perform testing without process monitoring but instead with networking connection monitoring. On the other hand, DrE [35] is a symbolic execution framework targeting at sensor input channel of an embedded system. It generates traces of sensor readings that will drive an MSP430-based embedded system to a chosen point in its code. In contrast, IoTFuzzer focuses on mobile apps because it is the major data input channel. Heninger et al. [30] studied the weak keys in embedded network devices through scanning the entire address space for listening hosts and retrieving keys from the programs. Costin et al. [17] studied the management web interface in embedded devices at a large scale. Also, some works proposed to apply program analysis techniques to the firmware to discover certain vulnerabilities. For example, FIE [21] improved symbolic execution techniques to suit for firmware-specific features. The result showed their tool could discover many memory bugs. Shoshitaishvili et al. [38] proposed a general model to describe backdoors in binary firmware and combined dynamic symbolic execution to identify them.

For the work of case studies which identify vulnerabilities in embedded systems, the authors usually had interesting discoveries, but their approaches are generally not general. For instance, Chen et al. [14] reverse engineered and exploited an Apple firmware update. They found that with custom keyboard firmware images, it was possible to persist a rootkit. Cui et al. [19] also found HP LaserJet printer was vulnerable to

firmware modification attacks. They did a deep case study on HP LaserJet printer's update mechanism and proposed several methods to detect and defend against attacks that were targeting at firmware update. However, these vulnerabilities are specific to certain models or products.

Meanwhile, there are also efforts concentrating on the scalable security analysis of embedded devices or developing the frameworks supporting multiple types of devices. Compared with those works, IoTFuzzer focuses on *automatically identifying* memory corruptions. For example, Costin et al. [17] manually discovered many vulnerabilities by application level emulation and static analysis. Chen et al. [13] extended this work by emulating the whole file systems of Linux-based firmware images with Qemu. Under their setup, the NVRAM emulation implementation did not work for all the firmware images because they could not emulate NVRAM-related functions. In addition, Avatar [47] is a framework that enables complex dynamic analysis of embedded devices by orchestrating the execution of an emulator together with the real hardware.

VII. Conclusion

We have presented the first IoT fuzzing framework IoTFuzzer, utilizing the official mobile app to detect memory corruptions of the corresponding IoT device. To enable efficient security testing, we have developed a set of novel techniques including in-context cryptographic and network function replay for message generation and delivery, runtime mutation of protocol fields without protocol specifications, as well as a lightweight monitoring mechanism to detect system crashes. By conducting experiments in real environment, we successfully identified 15 memory safety vulnerabilities among 17 IoT devices with IoTFuzzer.

Acknowledgements

We thank anonymous reviewers, Zhou Li, Zhe Zhou, and Wei You for their insightful comments. This work was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61572415, Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme/General Research Fund No. 24207815, No. 14217816 and Theme-based Research Scheme T23-407/13-N as well as the Mobile Technologies Centre (MobiTeC) of CUHK. Wenrui Diao was supported in part by the JNU Research Startup Grants. The authors in UT Dallas were supported in part by AFOSR FA9550-14-1-0119, FA9550-14-1-0173, and NSF CNS-1453011 and CNS-1516425. The IU author was supported in part by the NSF CNS-1527141, 1618493, ARO W911NF1610127, and a Samsung Research Gift fund.

References

[1] "Androguard: Reverse engineering, Malware and goodware analysis of Android applications," https://github.com/androguard/androguard, Accessed: November 2017.

[2] "Iotlist," http://iotlist.co/, Accessed: November 2017.

[3] "monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/index.html, Accessed: November 2017.

[4] "Peach Fuzzer," http://www.peachfuzzer.com/, Accessed: November 2017.

[5] "Penetration Testing Tool: BED," http://tools.kali.org/vulnerability-analysis/bed, Accessed: November 2017.

[6] T.-L. R. America, "Kasa for Mobile," https://play.google.com/store/apps/details?id=com.tplink.kasa_android, Accessed: November 2017.

[7] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A Taint Based Approach for Smart Fuzzing," in *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), Montreal, QC, Canada, April 17-21, 2012*, 2012.

[8] C. Brook, "Travel Routers, NAS Devices Among Easily Hacked IoT Devices," https://threatpost.com/travel-routers-nas-devices-among-easily-hacked-iot-devices/124877/, Accessed: November 2017.

[9] J. Caballero, P. Poosankam, C. Kreibich, and D. X. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, 2009.

[10] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS), Chicago, Illinois, USA, October 4-8, 2010*, 2010.

[11] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC), Los Angeles, CA, USA, December 7-11, 2015*, 2015.

[12] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 8-11, 2015*, 2015.

[13] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.

[14] K. Chen, "Reversing and exploiting an Apple firmware update," *Black Hat*, 2009.

[15] L. Constantin, "Hackers found 47 new vulnerabilities in 23 IoT devices at DEF CON," http://www.csoonline.com/article/3119765/security/hackers-found-47-new-vulnerabilities-in-23-iot-devices-at-def-con.html, Accessed: November 2017.

[16] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, 2014.

[17] A. Costin, A. Zarras, and A. Francillon, "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 (ASIACCS), Xi'an, China, May 30 - June 3, 2016*, 2016.

[18] A. Cui, "Embedded Device Firmware Vulnerability Hunting with FRAK," *DEF CON*, vol. 20, p. 9, 2012.

[19] A. Cui, M. Costello, and S. J. Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 24-27, 2013*, 2013.

[20] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto, "ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy (Oakland), Oakland, California, USA, 20-23 May 2007*, 2007.

[21] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *Proceedings of the 22th USENIX Security Symposium (USENIX Sec), Washington, DC, USA, August 14-16, 2013*, 2013.

[22] devttys0, "Binwalk: Firmware Analysis Tool," https://github.com/devttys0/binwalk, Accessed: November 2017.

[23] devttyS0, "Embedded Device Hacking," http://www.devttys0.com/category/reverse-engineering/, Accessed: November 2017.

[24] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[25] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.

[26] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable Graph-based Bug Search for Firmware Images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.

[27] Fitblip, "Sulley - a pure-python fully automated and unattended fuzzing framework," https://github.com/OpenRCE/sulley, Accessed: November 2017.

[28] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based Directed Whitebox Fuzzing," in *Proceedings of the 31st International Conference on Software Engineering (ICSE), Vancouver, Canada, May 16-24, 2009*, 2009.

[29] Gartner, "Internet of Things (IoT) Market," https://www.gartner.com/newsroom/id/3598917, February 2017.

[30] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices," in *Proceedings of the 21th USENIX Security Symposium (USENIX Sec), Bellevue, WA, USA, August 8-10, 2012*, 2012.

[31] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler, "Firmusb: Vetting USB device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[32] B. Herzberg, D. Bekerman, and I. Zeifman, "Breaking Down Mirai: An IoT DDoS Botnet Analysis," https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html, Accessed: November 2017.

[33] J. Lyne, "Uncovering IoT Vulnerabilities in a CCTV Camera," https://www.rsaconference.com/videos/demo-uncovering-iot-vulnerabilities-in-a-cctv-camera, Accessed: November 2017.

[34] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-Architecture Bug Search in Binary Executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 17-21, 2015*, 2015.

[35] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using Program Analysis to Synthesize Sensor Spoofing Attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS), Abu Dhabi, United Arab Emirates, April 2-6, 2017*, 2017.

[36] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic Security Analysis of Smartphone Applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY), San Antonio, TX, USA, February 18-20, 2013*, 2013.

[37] Rovo89, "Xposed Module Repository," http://repo.xposed.info/, Accessed: November 2017.

[38] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 8-11, 2015*, 2015.

[39] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[40] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland), Oakland, California, USA, May 16-19, 2010*, 2010.

[41] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic Reverse Engineering of Encrypted Messages," in *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009.*, 2009.

[42] Z. Wang, Y. Zhang, and Q. Liu, "RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing," *TIIS*, vol. 7, no. 8, pp. 1989–2009, 2013.

[43] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.

[44] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "POMP: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts," in *Proceedings of the 26th USENIX Security Symposium (USENIX-Sec), Vancouver, BC, Canada, August 1618, 2017*, 2017.

[45] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting Capability Leaks of Android Applications," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS), Kyoto, Japan, June 3-6, 2014*, 2014.

[46] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag," in *Proceedings of the 11th International Conference on Advances in Mobile Computing & Multimedia (MoMM), Vienna, Austria, December 2-4, 2013*, 2013.

[47] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 23-26, 2014*, 2014.

[48] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian, C. A. Gunter, K. Zhang, P. Tague, and Y. Lin, "Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be," *CoRR*, vol. abs/1703.09809, 2017.

[49] C. Zuo and Z. Lin, "SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution," in *Proceedings of the 26th International Conference on World Wide Web (WWW), Perth, Australia, April 3-7, 2017*, 2017.

[50] C. Zuo, W. Wang, R. Wang, and Z. Lin, "Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services," in *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.

[51] C. Zuo, Q. Zhao, and Z. Lin, "AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.