No More Companion Apps Hacking but One Dongle: Hub-Based Blackbox Fuzzing of IoT Firmware

Xiaoyue Ma
George Mason University
Fairfax, Virginia, USA
xma9@gmu.edu

Qiang Zeng
George Mason University
Fairfax, Virginia, USA
zeng@gmu.edu

Haotian Chi
Shanxi University
Taiyuan, Shanxi, China
htchi@sxu.edu.cn

Lannan Luo ✉
George Mason University
Fairfax, Virginia, USA
lluo4@gmu.edu

ABSTRACT

Given the massive difficulty in emulating IoT firmware, blackbox fuzzing of IoT devices for vulnerability discovery has become an attractive option. However, existing blackbox IoT fuzzers need much time and tedious effort to reverse engineer the IoT companion app (or manually collect test scripts) of *each* IoT device, which is unscalable when analyzing many devices. Moreover, fuzzing through a companion app is impeded by the input sanitization inside the app and limited to the manually revealed functions. We notice that IoT devices are typically able to connect a hub using standard wireless protocols (such as ZigBee, Z-Wave, and WiFi). We thus propose a uniform hub-based architecture for fuzzing various IoT devices, without reverse engineering any companion apps. It exploits the messages exchanged between a hub and an IoT device to automatically discover all the functions, and then launches systematic function-oriented message-semantics-guided fuzzing. It avoids sanitization imposed by a companion app. In addition, it conducts device state-sensitive fuzzing, which we find very effective in finding IoT bugs. We implement the system named HubFuzzer. The evaluation shows that HubFuzzer leads to much higher coverage than prior state of the art. We test 21 IoT devices and find 23 zero-day vulnerabilities. Six CVEs have been assigned.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

IoT security, fuzzing, vulnerability discovery

ACM Reference Format:
Xiaoyue Ma, Qiang Zeng, Haotian Chi, and Lannan Luo ✉. 2023. *No More Companion Apps Hacking but One Dongle*: Hub-Based Blackbox Fuzzing of IoT Firmware. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '23), June 18–22, 2023, Helsinki, Finland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581791.3596857

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
MobiSys '23, June 18–22, 2023, Helsinki, Finland
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0110-8/23/06.
https://doi.org/10.1145/3581791.3596857

1 INTRODUCTION

The global IoT market size is anticipated to rise from $478 billion in 2022 to $2,465 billion by 2029 [30]. Recent studies show that cyber attacks against IoT devices are growing at alarming rates—the first half of 2021 saw 1.5 billion attacks on smart devices, an increase from 639 million in 2020 [55]. These attacks frequently exploit vulnerabilities of IoT firmware. Thus, it is important to discover vulnerabilities in IoT firmware.

Several vulnerability-finding systems use static analysis to analyze IoT firmware [3, 18, 45, 50, 72], but tend to report many false positives. Others dynamically emulate firmware to support greybox or whitebox fuzzing. Despite much work [8, 12, 44, 70, 71], emulating IoT firmware is still an unresolved challenge, as the large number of custom and proprietary hardware components make the task of building an accurate emulator intractable [64].

Thus, blackbox fuzzing becomes an attractive option and receives growing attention. For example, `IoTFuzzer` [9] and `DIANE` [48] leverage IoT companion apps to send inputs to IoT devices. First, as obfuscation has become a standard practice in developing mobile apps [6, 14, 37] and strong obfuscation techniques are widely available [6, 65], it requires much effort to reverse engineer and modify an obfuscated companion app. Second, they combine static analysis and manual effort to identify routines that can be used to inject fuzzing messages, which is tedious, time consuming, and incomplete [19, 48]. Third, such fuzzing is impeded by input sanitization inside a companion app. Actually, the main contribution of `DIANE` [48] is to bypass certain, but not all, sanitization. `DIANE` mitigates the third limitation but does not address the other two. (`SNIPUZZ` [19] manually collects API-testing programs to fuzz IoT devices. But very few vendors disclose their API-testing programs and the disclosed test scripts do not necessarily cover all the functions of a device.)

Our Insight and Main Idea. To increase sales, IoT vendors typically build their devices to be able to connect various hubs (e.g., hubs of Google, Amazon, SmartThings, and Home Assistant) [10]. Moreover, most IoT devices utilize standard wireless protocols [66]. ZigBee and Z-Wave are among the most popular wireless protocols for home automation [63] and share a large market size [17, 29]. There are more than 100 million such devices in smart homes worldwide, which cover 70% of the smart home market [17].

We notice that, when a ZigBee or Z-Wave IoT device joins a hub, it generates a sequence of *setting-up* messages for handshaking [36]. These messages contain rich information about the device, such as the device id, address, manufacture code, and all the *supported functions* (e.g., unlock a door, turn on a camera). Surprisingly, none of the existing works exploit such messages for IoT fuzzing. We propose to make use of such messages to discover the functions of a device and then perform systematic function-oriented fuzzing. It is worth highlighting that we do not reverse engineer any IoT companion apps but automatically derive a complete list of supported functions of a device from setting-up messages.

Fuzzing Approach. Unlike prior work that hacks one companion app *per* device (or manually collects test scripts) to send fuzzing messages, we propose a uniform hub-based fuzzing architecture. Our system, named HubFuzzer, first declares itself as a hub to connect directly with the IoT device to be tested. HubFuzzer extracts useful information from the setting-up messages and learns the functions supported by the device. After that, HubFuzzer systematically generates testing messages for each of the functions and sends them, through an inexpensive USB dongle, to the IoT device. During the process, it captures inputs that trigger exceptions, such as crashes.[1]

Since companion apps are not used, our fuzzing is not hindered by various input sanitization inside companion apps. Moreover, our investigation finds that the device context (i.e., the device state before the current fuzzing message is sent) often has an impact on the testing result. We thus conduct *state-sensitive fuzzing*. Specifically, we set the device state to a special state before sending a testing message. This significantly helps vulnerability discovery.

We envision that HubFuzzer can be used by third-party security researchers, IoT vendors who may not have the resources to develop and maintain their white-box fuzzers, and organizations that emphasize security and want to test their purchased IoT devices.

We implement HubFuzzer and conduct an extensive evaluation. It is compared with a state-of-the-art work SNIPUZZ, which also does not need to hack companion apps (but relies on manually collected test scripts). The evaluation results show that HubFuzzer can achieve much higher coverage of IoT functions than SNIPUZZ. Specifically, HubFuzzer can achieve full function coverage for each tested device, while the coverage of SNIPUZZ is much lower ($\in [0\%, 66.7\%]$). As a result, HubFuzzer can detect more vulnerabilities. We test 21 IoT devices from various vendors and find 23 zero-day vulnerabilities (all are missed by SNIPUZZ). Six CVEs have been assigned: CVE-2022-47100, CVE-2023-24678, CVE-2023-29779, CVE-2023-29780, CVE-2023-34596, CVE-2023-34597. We make the following contributions.

- We propose a novel hub-based dynamic analysis architecture, and demonstrate its usefulness for blackbox IoT fuzzing. Compared to prior approaches, it does not need to reverse engineer any companion apps (or collect testing scripts). Moreover, our fuzzing is not impeded by various sanitization in companion apps.
- We present function-oriented blackbox fuzzing, which derives functions supported by IoT devices from setting-up messages and performs systematic function-oriented fuzzing.
- We extract knowledge about message structures and command/attribute value ranges from the protocol specifications. The knowledge can be reused to test other IoT devices. Our fuzzing is guided by the precise knowledge, while prior work, SNIPUZZ, by inaccurately inferred message structures.
- Device state-sensitive fuzzing is designed to detect vulnerabilities that can only be triggered when a particular device condition is met. Our findings show that state-sensitive fuzzing is very effective and helps find 18 vulnerabilities.
- We implement HubFuzzer[2] and conduct an evaluate on 21 popular IoT devices. We find 23 zero-day vulnerabilities with 6 CVEs assigned. HubFuzzer significantly outperforms prior state of the art, in terms of both coverage and discovered vulnerabilities.

2 RELATED WORK

2.1 Static Analysis-based Approaches

A set of approaches are based on static analysis, which requires the access to IoT firmware images [3, 7, 21–24, 33, 38, 53, 54, 56, 61, 62, 68]. For example, InnerEye [75] presents the first NLP-inspired deep learning approach to analyzing native code in the literature. However, as manufacturers often do not release their firmware, the applicability of these approaches is limited [9, 15, 48]. Moreover, static analysis tends to have a high false positive rate.

2.2 Dynamic Analysis-based Approaches

Dynamic analysis-based IoT vulnerability detection approaches can be divided into the following three groups.

Emulation. Some rely on emulation [8, 12, 16, 44, 64, 71]. Two major challenges for firmware emulation are the scalability and throughput [35, 68, 69, 73]. Although a lot of efforts have been made to improve the performance, how to precisely emulate IoT devices is still an open question [48, 64, 67].

Symbolic Execution. Another research line applies symbolic execution [13, 25, 39–41]. But the precise execution of IoT firmware needs to access various peripherals [46]. To symbolically execute IoT firmware, they consider all inputs from peripherals as symbolic, which causes imprecision, or uses imprecise emulation results.

Blackbox Fuzzing. Recently, blackbox IoT fuzzing gains much attention as an effective approach to finding vulnerabilities [51, 60]. Existing blackbox IoT fuzzers either rely on reverse engineering companion apps [9, 48] or manually collected testing programs [19].

For example, IoTFuzzer [9] and DIANE [48] hack companion apps to test IoT devices. For each IoT device, they *statically* analyze the companion app to locate and modify the code corresponding to IoT device functions. However, this requires enormous reverse engineering efforts especially for obfuscated apps [6, 14], and the static analysis may introduce false positives and false negatives in identifying code for IoT device functions [19, 48].

[1] The hub-based design is inspired by PFirewall [10], but PFirewall filters IoT data for enhancing privacy, which is very different from our purpose.

[2] https://github.com/iot-sec23/HubFuzzer.

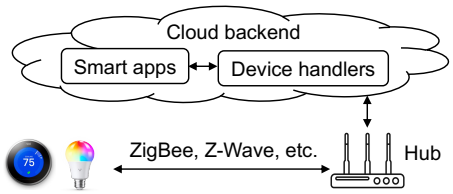


Figure 1: A typical IoT platform architecture.

SNIPUZZ [19] collects initial network messages using API-testing programs and then mutates these messages, which saves the effort to reverse engineer companion apps. However, it has multiple limitations. First, very few IoT vendors disclose their API-testing programs, Second, one needs to manually infer the message structure, which is tedious (it takes *5 man-hours of manual work per device* [19]) and inaccurate. Third, it can only test device functions covered by API-testing programs, resulting in false negatives. Finally, SNIPUZZ fails if network messages are encrypted, while most IoT communication protocols use encryption [9].

In sum, existing blackbox IoT fuzzers need much time and tedious effort to either reverse engineer the IoT companion app or manually collect test scripts for *each* IoT device, which is unscalable when analyzing many devices. Moreover, they have limited function coverage for testing devices.

3 BACKGROUND

3.1 IoT Platform Architecture

Figure 1 shows a typical IoT platform architecture, consisting of the following main components: *hub*, *smart apps*, *device handlers*, and *cloud backend*. The hub bridges the communication between IoT devices and the cloud backend and the device-hub connection uses various short/medium-range wireless radios (such as ZigBee, Z-Wave and WiFi). It plays a critical role in the interconnectivity and interoperability of heterogeneous IoT devices.

3.2 ZigBee and Z-Wave

ZigBee and Z-Wave are among the most popular wireless protocols in home automation [42, 49]. The global ZigBee home automation market has grown significantly over the past few years and will witness a 13% CAGR (compound annual growth rate) through 2027 [29]. Z-Wave devices cover 70% of the smart home market [17]. Nowadays, households increasingly use ZigBee and/or Z-Wave IoT devices as they consume less power and reduce energy costs. This work demonstrates our hub-based fuzzing approach on ZigBee and Z-Wave IoT devices, which we call **Z-based** devices. We discuss how the fuzzing approach can be generalized to testing WiFi, Bluetooth and Thread/Matter devices in Section 7.

4 OVERVIEW

4.1 Limitations of the State of the Art

SNIPUZZ represents the latest progress in blackbox IoT fuzzing. SNIPUZZ implements a snippet-based mutation strategy, which uses feedback from IoT devices to segment a network packet into snippets. To fuzz an IoT device, (1) SNIPUZZ needs to first collect seed messages using the API-testing programs. (2) Then, for each seed

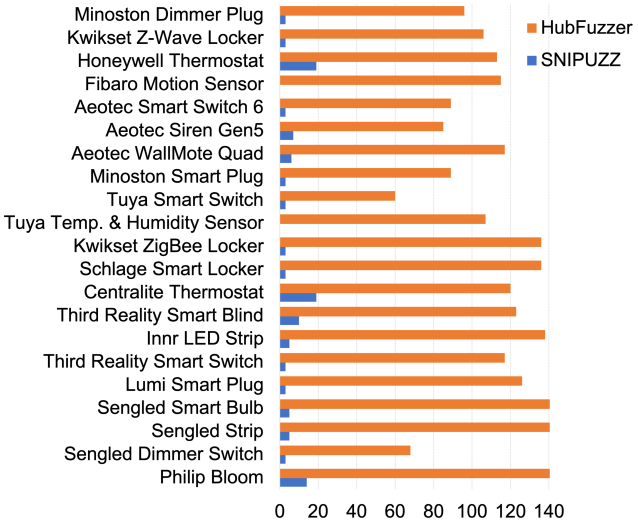


Figure 2: Commands covered by HubFuzzer vs. SNIPUZZ. Among the 21 devices in our evaluation, only two devices, *Philips Bloom* and *Sengled Smart Bulb*, release their API-testing programs. For the other devices, we enhance SNIPUZZ by regarding a device as its abstract device and consider the commands for the abstract device covered by SNIPUZZ.

message, it removes its bytes one by one to generate a set of probe messages, sends each probe message to the device, and collects the device responses. (3) It then categorizes and clusters the responses. (4) Finally, based on the clusters, it infers message snippets to guide mutation. However, SNIPUZZ has several limitations.

Manually Collecting Testing Programs. SNIPFUZZ needs the API-testing programs to obtain seed messages. However, very few IoT vendors disclose their API-testing programs. For example, only 2 out of the 21 devices used in our evaluation, *Philips Bloom* and *Sengled Smart Bulb*, release their API-testing programs.

Low Function Coverage. In Figure 2, the blue bars represent the number of commands covered by SNIPFUZZ, while the orange bars represent that of our tool. For those without API-testing programs, we improve SNIPFUZZ by regarding each device as its abstract device and count the commands of an abstract device as being covered by SNIPFUZZ. For example, a smart lock is abstracted into a *lock* with the `lock()` and `unlock()` commands. We can see that a large portion of commands are missed by SNIPFUZZ, which we call *hidden commands*. For example, for *Kwikset Smart Locker*, there are 133 hidden commands out of 136 commands in total; two examples are `toggle_door()`, which switches the device state from *on* to *off* and the other way around, and `set_pin_code()`, which sets the PIN for the device.

As SNIPUZZ cannot test the hidden commands, it has high false negative rate in vulnerability discovery. For instance, one vulnerability we discovered in *Sengled Smart Bulb* is related to the hidden command `Move_up`, which moves the 'current level' of the device up at a specified *rate*. This vulnerability is missed by SNIPUZZ.

Limited Message-Segmentation Capacity. The performance of SNIPUZZ heavily depends on the effectiveness of its snippet

determination algorithm. However, there are two main limitations. *First*, this algorithm works for devices where the message formats are JSON, SOAP, and Key-Value. In such cases, when it removes the bytes in a seed message one by one to generate probe messages, the removed byte does not affect the roles of the subsequent bytes in the message. However, for Z-based devices, the message is a byte stream consisting of fields. As a result, removing one byte results in shifting the subsequent bytes left by one, which changes the semantics of the subsequent bytes. *Second*, in order to segment a message correctly, `SNIPUZZ` requires rich information from the device responses. However, our evaluation finds that most Z-based devices report the errors in a uniform message (e.g., "*No response to 'x' command with seq id 'y'* "), making it hard for `SNIPUZZ` to accurately determine the message snippets.

Cannot Handle Encrypted Messages. `SNIPUZZ` avoids the effort to modify companion apps by directly mutating network packets. However, when the communication is encrypted, the approach fails, while most IoT devices use encrypted communication [9].

Ignoring Impacts of Device Context. We find that device context (e.g., the current device state) often has an impact on the triggering of device crashes, which is ignored by `SNIPUZZ`. For example, our experiments reveal that *only* when the current brightness level of *Sengled Smart Bulb* is the highest, can a crash be triggered.

4.2 Our Goals, Insight and Idea

We have the following goals in designing our system.

- **Easy to use.** Unlike prior work, our approach does not need to reverse engineer companion apps or manually collect testing programs.
- **High function coverage.** The system should have high function coverage; i.e., it should test (almost) all functions supported by an IoT device.
- **Generalizable.** It can be generalized to testing devices that use complex messages and encrypted communication.
- **State-sensitive fuzzing.** The design of the fuzzing strategy should take the impact of device states into consideration.

Our observation is that, to increase the sales, IoT vendors typically build their devices to be able to connect various hubs [10]. Thus, we propose a uniform hub-based architecture and build our fuzzer as a hub to directly talk with IoT devices by sending/receiving messages to/from IoT devices. For the same reason, most IoT devices support standard wireless protocols, such as ZigBee and Z-Wave [66].

Our insight is that when a Z-based IoT device joins a hub, it generates a sequence of setting-up messages to establish the connection. These messages contain rich information about the device, including device id, address, manufacture code, and supported functions. Based on this, ***our idea*** is to make use of the setting-up messages to get aware of the functions supported by IoT devices and then perform systematic function-oriented fuzzing. The approach is easy to use, has a high function coverage and can be generalized to devices that use complex messages and encryption.

Moreover, we extract knowledge about message command/attribute value ranges from the protocol specifications, and the knowledge is used to build our testing input mutation strategies, including state-sensitive fuzzing.

4.3 System Architecture

Figure 3 shows the architecture of our system, called HubFuzzer, containing four main components: *Device Connector*, *Function Extractor*, *Fuzzing Mutator*, and *Response Monitor*.

(1) The *Device Connector* component communicates with IoT devices directly via protocols, such as ZigBee and Z-Wave. It does three major things: i) pairing IoT devices; ii) sending testing messages to IoT devices; and iii) receiving messages from IoT devices. (The implementation details are presented in Section 6.1)

(2) When a Z-based IoT device joins the hub, it generates a sequence of setting-up messages. The *Function Extractor* component collects the setting-up messages and learns the functions supported by the IoT device (Section 5.1).

(3) Based on the supported functions, the *Fuzzing Mutator* component generates testing messages by mutating commands and attributes (Section 5.2, Section 5.3 and Section 5.4).

(4) Finally, the *Response Monitor* component monitors the status of the IoT device to capture crashes (Section 5.5).

The radio communication between IoT devices and HubFuzzer is supported by a USB Dongle. We next present the design of each of these components.

5 DESIGN OF HUBFUZZER

5.1 Learning Supported Functions

Clusters in ZigBee. As defined in the ZigBee Cluster Library (ZCL) [74], each *cluster* corresponds to a specific functionality and has an associated 2-byte cluster identifier (i.e., CID). For example, the *On/Off* cluster (with CID = 0x0006) allows a switch device to be put into the 'on' and 'off' states, and the *Level Control* cluster (with CID = 0x0008) allows control of the level of a physical quantity (e.g., heating output) on a device. A list of all available clusters can be found in the ZCL Specification [74].

A cluster is a collection of *commands* and *attributes*, which define an interface to a specific functionality. (1) An *Attributes* is a property of a device that can be stored as a state; e.g., a switch device has the *switch* attribute with two states, *on* and *off*. (2)A *Command* is a method that can control a device and manipulate attributes; e.g., the `lock()` (*resp.* `unlock()`) command in the *Lock* cluster can lock (*resp.* unlock) a door.

Command Classes in Z-Wave. Similar to ZigBee, Z-Wave abstracts device functionalities and groups related ones into *command classes* [52]. A command class is a collection of *commands* and *attributes*, where commands are used for controlling, querying, and reporting device attributes corresponding to specific functionality.

There are three kinds of commands for each command class: *Set*, *Get*, and *Report*. (1) A *Set* command is *sent to a device* to instruct the device to perform a specific task (which may change the device status). (2) A *Get* command is *sent to a device* to request the current status of a device. (3) A *Report* command is *sent from a device* to report its current device status if the status changes.

Note that the clusters in ZigBee and command classes in Z-Wave can be implemented in *any* application profile (e.g., home automation), and an IoT device that operates in an application profile must *implement and respond correctly to all the required clusters/command classes*. For example, a ZigBee light switch that operates in

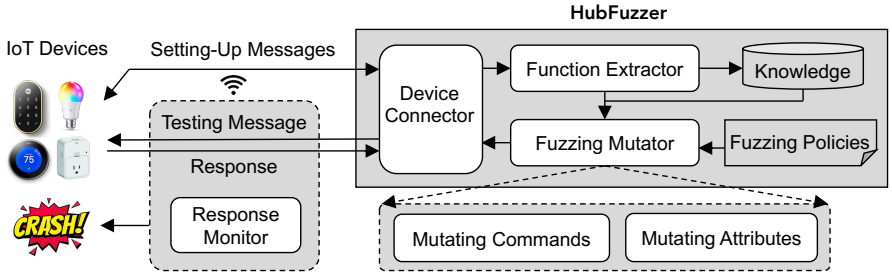


Figure 3: Architecture of HubFuzzer.

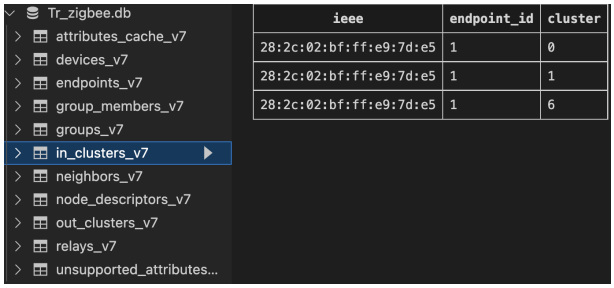


Figure 4: An example of setting-up messages.

the home-automation profile must correctly implement the *On/Off* cluster and other required clusters in order to interoperate with other home automation devices (e.g., hubs).

Although different names (i.e., *clusters* and *command classes*) are used, they actually have the same meaning. For the sake of presentation, we do not distinguish the two names and use *clusters* in the following sections.

Our Method. A sequence of setting-up messages are generated when a Z-based device joins a hub, which contains rich information about the device, including the device id, manufacture code, and supported functions. As an example, Figure 4 shows the information extracted from the setting-up messages for the *Third Reality smart switch* device, where the details of the most important ones, *in-clusters* (i.e., *in_clusters_v7*), are highlighted. Here, *incluster* means *the contained commands can control this device*. Note that there is another field named *outclusters* (i.e., *out_clusters_v7* in Figure 4), which means the current device can control other devices using the corresponding commands. As we aim to send testing messages to IoT devices to trigger device actions, we focus on *inclusters*.

Based on the reported *inclusters* and according to the ZCL [74], we can learn the functions supported by the device, and then determine (1) which *commands* can control this device, and (2) which *attributes* are stored in the device. For example, for the three *inclusters* in Figure 4, (1) the cluster CID = 0 is the *Basic* cluster, which is a mandatory cluster for all ZigBee devices. It includes 21 attributes, including the basic properties of a device, such as software and hardware versions, manufacturer name, and 1 command, which can send a '*Reset To Factory Defaults*' message to reset the device to its factory defaults. (2) The cluster CID = 1 is the *Power Configuration* cluster, which allows information to be obtained about the power sources of a device and voltage alarms to be configured. 58 attributes are defined, including the battery information and battery settings. (3) The cluster CID = 6 is the *On/Off* cluster. It includes 5 attributes, such as *OnTime* that specifies how long the light remains *on*, and 6 commands, which allow a device to be put into the 'on' and 'off' state for a time period or with an effect.

Extracting Knowledge to Support Fuzzing. The protocol specification [52, 74] provides a detailed description for each cluster, including the contained commands, arguments/attributes, and each argument's data type and value range, which are reflected in the corresponding protocol stack library. For example, *zigpy* [27] implements ZigBee standard specifications [11] as a Python library, and *Z-Wave JS* implements Z-Wave standard specifications [52]. We first extract the relevant information from these libraries using a script and store it in a database, which is then integrated in HubFuzzer (note that this is a one-time effort). Specifically, *zigpy* has a directory, `zigpy/zcl/clusters`, that contains a list of files implementing all the clusters, including information about the associated commands, attributes, and the value range and data type of each command's argument, which is extracted by our script. Similarly, *Z-Wave JS* provides a header file `ZW_classcmd.h` and an Excel file `List of defined Z-Wave Command Classes.xlsx`, from which we can extract relevant information about each cluster. When fuzzing a device, we first learn the supported clusters from the setting-up messages, and then retrieve the supported commands and attributes as well as the value range and data type of each command's argument from the database. The information then assists the testing message generation.

5.2 Packing Procedure

We generate various testing messages to trigger device actions and monitor crashes. A simple solution is to mutate the arguments of each supported command (learned from the setting-up messages). However, as the Z-based libraries used in our hub involve *input sanitization*, a command with an out-of-range argument value will get rejected, causing fewer testing messages to be sent.

As an example, the command `move_to_level()` in the *Level* cluster moves the 'current level' of the device to a target level over a specified time. The target level is set as an argument and must be within the range of 0x00 to 0xFE. If we set the target level to a value out of this range (e.g., 0xFF), this command will throw a "*failed to convert*" error, and no testing message will be generated.

To resolve this issue, our solution is to locate the procedure that packs messages, which we call the *packing procedure*. It is

invoked by each command request to generate messages to be sent to IoT devices. We then remove the input sanitization in the packing procedure and mutate the information for building a message (e.g., command argument type and value) to generate testing messages. Note unlike prior work that removes sanitization in the companion app of *each* device, our sanitization elimination is one-time effort.

ZigBee. Listing 1 shows the signature of the packing procedure, `request`, from *zigpy* [27], a library that implements the ZigBee protocol stack. The `request` function is invoked by each command to generate messages. To generate diverse testing messages that can be accepted by devices, HubFuzzer mutates two critical parameters, *cluster* and *data* (both are highlighted in blue). Specifically, *cluster* stores the current cluster ID and *data* stores the payload.

```
def request(device, profile, cluster, src_ep, dst_ep,
        sequence, data, expect_reply=True, use_ieee=False)
```

Listing 1: ZigBee packing procedure

There are two types of commands. We thus send two kinds of testing messages by invoking the two types of commands.

- ***Cluster commands*** are defined in each cluster. Each cluster may define zero or more cluster commands.
- A ***write-attribute command*** can modify one or more specified cluster attributes. Note that a write-attribute command is not specific to an individual cluster but shared by all clusters.

Figure 5 shows the ZCL message format for the two types of commands, where the sections that will be modified to generate diverse testing messages are highlighted in blue. Figure 5(a) shows the format of a message when a cluster command is invoked. The cluster ID is stored in the header. The payload includes the command type, the command ID, and the value of each command argument. If a cluster command is invoked, the command type is 0x01, while if a write-attribute command is invoked, the command type is 0x02. Figure 5(b) shows the format of a message when a write-attribute command is invoked. The cluster ID is also part of the header. The payload specifies the command type and one or more attributes. For each attribute, the message includes its attribute identifier, data type, and value. The commands and attributes are mutated to generate testing messages.

Z-Wave. For the Z-Wave protocol, we find its packing procedure, named `sendMessage`, from the *Z-Wave JS* library [5]. A Z-Wave cluster contains three types of commands, *Set*, *Get* and *Report* (see Section 5.1). As we aim to send testing messages to IoT devices to trigger their actions, we use the *Set* command (its command ID is 0x01) to generate testing messages. We follow a similar way to mutate the cluster ID and payload of packed messages to generate diverse testing messages.

Note that HubFuzzer mutates testing messages, which are then encrypted and sent out of the hub. Thus, HubFuzzer has no problems dealing with IoT devices that use encryption. It is worth noting that, although both prior work (DIANE [48]) and HubFuzzer need to locate the functions for preparing testing messages, DIANE needs to reverse-engineer and hack the companion app of *each* IoT device, while it is one-time effort with HubFuzzer to locate and modify the *packing procedures*.

5.3 Fuzzing Policies

We have the following fuzzing policies.

Policy 1: *Changing Argument Values.* There are 57 data types defined in the ZCL. We divide them into two categories. (1) *High risk* types have a higher probability of causing device crashes if argument values of these types are mutated; there are 26 high risk types (e.g., *OctetString*, *CharString*, *integer*, *float*, *Array*, *Set*, and *NoData*). (2) *Low risk* types are the rest (e.g., *Time of day*, *Date*).

Given a command with n arguments, we first mutate arguments of *high risk* types, and then those of *low risk*. Specifically, we (a) change the lengths of strings to trigger buffer overflow; (b) provide empty values to strings to cause uninitialized variable vulnerability or null pointer dereference; (c) mutate integer, double or float values into extreme values to cause integer overflow or out-of-range access; (d) provide `NULL` or only one element to arrays, sets, or bags to cause null pointer dereference or out-of-bounds access; and (e) provide a randomly generated value to arguments with the *NoData* type (*NoData* means no data should be provided).

Policy 2: *Changing Argument Types.* Given an argument supposedly with the data type t, we change its type to a randomly selected one t'. For example, for an argument with the *String* type, we change its type to the *integer* type by replacing a string value with an integer value, to check whether the device can handle the special "string".

We notice a special data type *Unknown*, which is not an actual data type and should not be used for attributes or command arguments. However, it is still defined in the ZCL for completeness to reserve the data type identifier for use where a data special type unknown is needed. We thus also change t to *Unknown* to check the device's ability to handle such special cases.

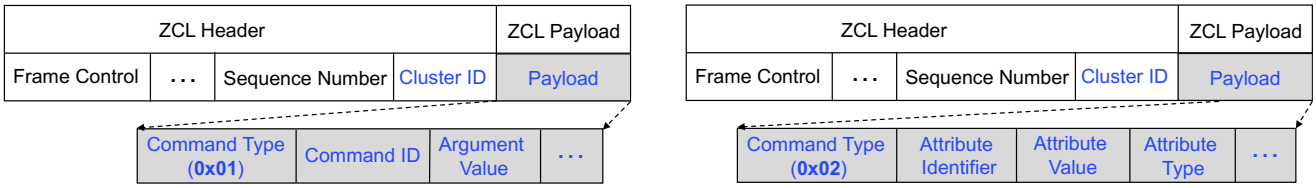
Policy 3: *Changing the Number of Arguments.* We provide more (or less) arguments to the current command. For example, given a command that requires n arguments, we provide $n + x$, $n - x$, or 0 arguments ($x \in \mathbb{Z}$).

Policy 4: *Trying Unsupported Clusters and Commands.* Besides the supported clusters, we also randomly select a few *unsupported clusters* given the full set in the ZCL [74] (our experiments randomly select 3 unsupported clusters). For each command in a selected unsupported cluster, we generate testing messages following the command definition. Through this, we can check whether *unexpected commands* can cause the device to crash.

Regarding the *write-attribute* command, its semantics is to modify the device attributes (Section 5.2). To test the device's ability to handle such a command, we use this command to modify all the attributes one by one; the attribute being changed becomes the argument of the command. Moreover, for an attribute specified in the write-attribute command, if the device response indicates that this attribute is "read-only", we skip this attribute (Policies 1 and 2 are not applied to such cases), and move to the next attribute.

5.4 State-Sensitive Fuzzing

Based on our experiments, we find that the device state has a significant impact on triggering crashes. For example, given a testing message that turns up the brightness of a smart bulb with a certain rate, only when the bulb's current brightness is the highest (i.e., 254), can the message cause the bulb to crash. We thus conduct



(a) Cluster command. **(b) Write-attribute command.**

Figure 5: Format of the ZCL frame.

state-sensitive fuzzing. Specifically, we first set the device state to a special state (e.g., setting the bulb's brightness as the highest/lowest) before sending a testing message that may *impact the state*.

We call a message that sets the device state to a special state a *reset message*. Given a command, we first determine which device attribute it tries to modify. For the write-attribute command, it is easy to determine as the attribute is specified by us. For a cluster command, we rely on the ZCL [74] and Z-Wave class specification [52] to find the attribute modified by this command. After that, we determine the value range of the attribute based on the specification [52, 74] (this information is first extracted and stored in a database, as discussed in Section 5.1). If a range is specified, we generate two reset messages by providing the maximum and minimum values to the argument related to this attribute. If no range is specified, a random value is used to generate a reset message.

We clarify that *not all* vulnerabilities, in order to be triggered, require the device to be set to a special state. Note that the influence of device state is not considered by the prior IoT fuzzing work [9, 19, 48], As a result, they failed to detect many vulnerabilities, which cause crashes only when a special condition is met.

5.5 Response Monitor

Figure 6 shows four scenarios about how an IoT device responds when receiving a message m, where the device does not crash in the two scenarios shown in Figure 6(a) and (b), but crashes in the other two in Figure 6(c) and (d).

As shown in Figure 6(a), when m is valid, the device i) accepts it and sends an `ACK` response to acknowledge receiving m, and then ii) executes the corresponding command and sends a `Report` response to report the updated device state (note that a device usually reports its updated state after successfully executing a command). Thus HubFuzzer receives two responses, `ACK` and `Report`.

As illustrated in Figure 6(b), when m is invalid, the device sends an `ACK` response to acknowledge receiving m, but then discards m (the device can handle invalid messages well, which is different from the two scenarios shown in Figure 6(c) and (d)). As a result, HubFuzzer receives only one response, `ACK`.

For the two scenarios illustrated in Figure 6(c) and (d), m is invalid, and the device crashes at different points. In Figure 6(c), the device crashes before sending an `ACK` response; as a result, no response is received by HubFuzzer. In Figure 6(d), the device crashes after sending an `ACK`; as a result, HubFuzzer receives an `ACK` response but no `Report` response.

Based on this observation, we monitor device crashes using the following rules.

- If no `ACK` response is received by HubFuzzer, it indicates a device crash has occurred. (Note that when the network connection is poor, a message may get lost, causing the `ACK` to be lost as well. However, if the missing `ACK` is due to a crash, the symptom can be reproduced in a reliable way using the same message. But if it is due to poor network connection, it cannot; plus, the device does not manifest a restart due to crash, such as LED blinks and sound alerts.)
- If an `ACK` is received, but no `Report` response is received, we cannot determine whether the device has crashed or not (see Figure 6(b) and (d)). We thus need to distinguish them by sending a *reset* message, which is known to be valid, for the next state-sensitive testing. If no `Report` response is received, it indicates that the device has crashed.

Moreover, our experiment results show an interesting and important phenomenon. Given a message that is known to cause a device crash, if the fuzzer sends the same message multiple times continuously, a device may enter a state worse than crashes; e.g., the device cannot re-join the network.

6 EVALUATION

This section presents the implementation of HubFuzzer and the evaluation results. Specifically, Section 6.1 gives the implementation details. Section 6.2 presents the experimental setup. Section 6.3 discusses the function coverage and and Section 6.4 the effectiveness of vulnerability discovery; we compare HubFuzzer with the state-of-the-art work. Section 6.5 presents the efficiency of HubFuzzer.

6.1 Implementation

We have implemented a prototype of HubFuzzer. It communicates with IoT devices directly through device-dependent protocols such as ZigBee and Z-Wave. Several open-source platforms, e.g., Home Assistant [26], openHAB [47], and WebThings [57], provide the device connector alike functions and allow developers to use add-ons for integrating various IoT devices.

Specifically, Home Assistant contains a ZigBee protocol stack, called *zigpy* [27], which implements ZigBee standard specifications [11] as a Python library. It allows ZigBee devices to connect directly to Home Assistant. Home Assistant also provides *Z-Wave JS* to support connection with Z-Wave devices [5]. Numerous Z-based IoT devices are supported by Home Assistant, including sensors (e.g., motion, door, and temperature sensors), lights, switches, buttons, covers, fans, climate control equipment, locks, and alarm

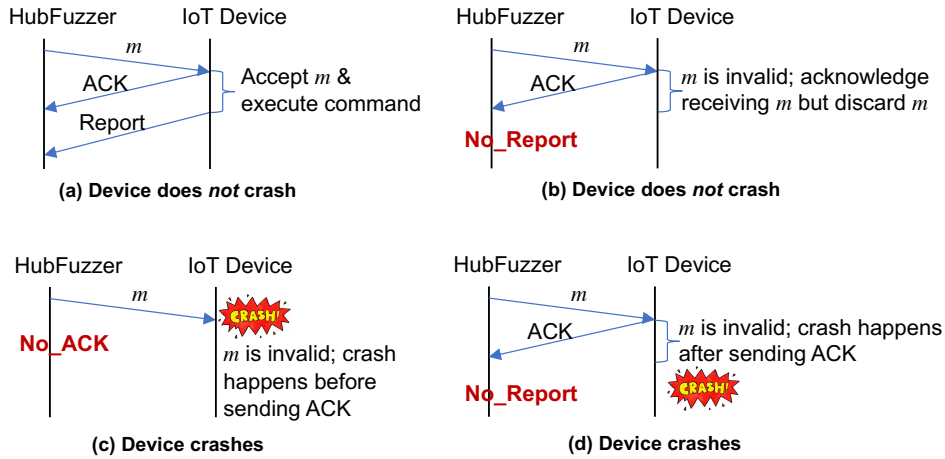


Figure 6: Different scenarios when an IoT device receives a message.

ID	Device Type	Vendor	Model	Firmware Version	Protocol
1	Thermostat	Centralite	Pearl	0x04075010	ZigBee
2	Lighting	Sengled Bulb	E11-N1EAW	0x00000024	ZigBee
3	Lighting	Innr	FL 120 C	0x28002162	ZigBee
4	Lighting	Philip Bloom	Bloom	1.88.1	ZigBee
5	Lighting	Sengled Strip	E1G-G8E	0x00000024	ZigBee
6	Blind	Third Reality	3RSB015BZ	1.00.54	ZigBee
7	Plug	Lumi	ZNCZ12LM	18	ZigBee
8	Locker	Schlage	BE468GBAK CAM 619	0.21.0	ZigBee
9	Locker	Kwikset	99140-139	0x40a32a10	ZigBee
10	Sensor	Tuya	B09TDQ9GP6	v1.0.10	ZigBee
11	Switch	Tuya	B09XCP7DN1	v1.1.0	ZigBee
12	Switch	Third Reality	3RSS009B	v1.01.10	ZigBee
13	Dimmer Switch	Sengled	E1E-G7F	v0.0.9	ZigBee
14	Plug	Minoston	MP21Z	v1.0.1	Z-Wave
15	Wall Switch	Aeotec	ZW130-A	v2.3	Z-Wave
16	Sensor	Aeotec	ZW080	v3.28	Z-Wave
17	Switch	Aeotec	ZW096-A	v1.7	Z-Wave
18	Motion Sensor	Fibaro	FGMS-001	v3.4	Z-Wave
19	Thermostat	Honeywell	TH6320ZW2003	v1.3	Z-Wave
20	Locker	Kwikset	98880-004	v4.79	Z-Wave
21	Dimmer Plug	Minoston	MP22Z	v7.13.9	Z-Wave

(a) Device details

(b) Photo of devices

Figure 7: IoT devices used in our experiments.

system devices. In our implementation, we utilize the ZigBee and Z-Wave add-ons of Home Assistant to connect with Z-based devices. The Device Connector is built with a Nortek Security & Control HUSBZB-1 USB dongle for the ZigBee and Z-Wave radio communication capabilities ($39.50 on Amazon [20]) .

6.2 Experimental Setup

IoT Devices Under Test. We have selected 21 popular consumer Z-based IoT devices from both online and offline markets, covering various well-known brands, such as Philips, Centralite, Third Reality, Sengled, Aeotec, and Lumi. The types of selected IoT devices include smart switch, plug, lighting, blind, locker, sensor, and thermostat. These devices are either recommended by Amazon or the best-selling products available in supermarkets. The details (type, vendor, version, protocol, etc.) of the devices are described in Figure 7(a). To deliver a visual impression of these devices, Figure 7(b) shows a photo of these devices.

Moreover, as we compare HubFuzzer with SNIPUZZ [19], we also selected the Z-based IoT device considered by SNIPUZZ: *Philip Bloom Lighting*, with ID=4 in Figure 7(a). Other devices tested by SNIPUZZ are WiFi based, so we did not include them.

Testing Environment. Our HubFuzzer runs on a Ubuntu 20.04 PC with 4.9 GHz Intel® Core(TM) i7 CPU and 32 GB RAM. We configured the Z-based devices in a fully-controlled network to avoid the interference of irrelevant traffic.

Baseline Method. SNIPUZZ [19] has proven to be more effective in vulnerability discovery in IoT devices than many other tools, such as IoTFuzzer [9] (a blackbox fuzzer for IoT devices), NEMESYS [34] (a

Table 1: Function Coverage Results. N/A: Z-Wave devices do not support write-attribute commands.

ID	HubFuzzer			SNIPUZZ		
	Cluster	Command	Attribute	Cluster	Command	Attribute
1	8(100%)	29(100%)	91(100%)	2(25%)	19(66.7%)	0(0%)
2	8(100%)	48(100%)	99(100%)	3(37.5%)	5(10.4%)	0(0%)
3	8(100%)	44(100%)	94(100%)	3(37.5%)	5(11.4%)	0(0%)
4	8(100%)	54(100%)	100(100%)	4(50%)	14(25.9%)	0(0%)
5	8(100%)	48(100%)	99(100%)	3(37.5%)	5(10.4%)	0(0%)
6	7(100%)	30(100%)	93(100%)	1(14.3%)	10(33.3%)	0(0%)
7	7(100%)	26(100%)	100(100%)	1(14.3%)	3(11.5%)	0(0%)
8	7(100%)	55(100%)	81(100%)	1(14.3%)	3(6%)	0(0%)
9	7(100%)	55(100%)	81(100%)	1(14.3%)	3(6%)	0(0%)
10	3(100%)	4(100%)	79(100%)	0(0%)	0(0%)	0(0%)
11	5(100%)	26(100%)	34(100%)	1(20%)	3(11.5%)	0(0%)
12	6(100%)	26 (100%)	91(100%)	1(16.7%)	3(11.5%)	0(0%)
13	5(100%)	26(100%)	42(100%)	1(20%)	3(11.5%)	0(0%)
14	14(100%)	89(100%)	N/A	1(7%)	3(3%)	N/A
15	19(100%)	117(100%)	N/A	3(16%)	6(5%)	N/A
16	16(100%)	85(100%)	N/A	3(19%)	7(8%)	N/A
17	18(100%)	89(100%)	N/A	1(5%)	3(4%)	N/A
18	20(100%)	115(100%)	N/A	0(0%)	0(0%)	N/A
19	21(100%)	113(100%)	N/A	2(9.5%)	19(16.8%)	N/A
20	15(100%)	106(100%)	N/A	1(6.7%)	3(2.8%)	N/A
21	15(100%)	96(100%)	N/A	1(6.7%)	3(3.1%)	N/A

Table 2: Vulnerability Discovery Results. (*UV*: Unknown Vulnerability. *DoS*: Denial of Service.)

ID	HubFuzzer		SNIPUZZ	
	Vul. Type	Number	Vul. Type	Number
1	UV/DoS	2	-	0
2	UV/DoS	8	-	0
3	-	0	-	0
4	-	0	-	0
5	UV/DoS	8	-	0
6	UV/DoS	2	-	0
7	-	0	-	0
8	-	0	-	0
9	-	0	-	0
10	-	0	-	0
11	-	0	-	0
12	-	0	-	0
13	DoS	1	-	0
14	-	0	-	0
15	UV	1	-	0
16	-	0	-	0
17	-	0	-	0
18	DoS	1	-	0
19	-	0	-	0
20	-	0	-	0
21	-	0	-	0

protocol reverse engineering tool), BooFuzz [32] (a network protocol fuzzer), DooNA [58] (a network protocol fuzzer). We thus consider SNIPUZZ as a baseline method and compare it with HubFuzzer.

6.3 Function Coverage

We first evaluate the function coverage when fuzzing with HubFuzzer, and compare it to that of the baseline tool SNIPUZZ. Since the function of a device is determined by the clusters that the device supports, we measure the *cluster* coverage. Moreover, as a cluster is a related collection of commands and attributes, we also conduct a more fine-grained comparison by measuring the *command* and *attribute* coverage. The results are shown in Table 1.

Cluster and Command Coverage. HubFuzzer makes use of setting-up messages to extract the supported clusters of a device (Section 5.1). A Z-based device relies on the setting-up messages to declare its functions to a hub. In other words, functions described in these messages fully cover a device's functions that can be used or tested. As shown in Table 1, we can see that HubFuzzer can achieve full coverage for clusters, commands, and attributes, while SNIPUZZ cannot. For a given device, SNIPUZZ uses its API-testing programs to obtain seed messages, which are mutated to generate testing messages. If the API-testing programs do not cover a command M, it is difficult (or impossible) to cover M by mutating the seed messages via blackbox fuzzing. As a result, SNIPUZZ can only test the clusters and commands included in the API-testing programs.

Attribute Coverage. For ZigBee devices, the write-attribute command is shared by all clusters and can be used to modify device attributes (Section 5.2). HubFuzzer can achieve full attribute coverage since we can generate testing messages to modify all the supported attributes via the write-attribute commands (i.e., the command type $0x02$). However, the write-attribute command is not covered by any API-testing programs. As a result, SNIPUZZ cannot generate testing messages corresponding to the write-attribute commands to modify device attributes.

For Z-Wave devices (IDs 14-21), they do not support the write-attribute command.

6.4 Vulnerability Discovery

To fuzz-test a device, the only manual effort is to pair it with HubFuzzer. We test 21 IoT devices and discover ***23 zero-day vulnerabilities in 7 devices*** as presented in Table 2. We have reported all the 23 newly discovered vulnerabilities to their vendors. Specifically, we find 2 vulnerabilities in *Centralite Pearl* (ID = 1) with CVE-2023-24678; 8 vulnerabilities in *Sengled Bulb E11-N1EAW* (ID = 2) with CVE-2022-47100; 8 vulnerabilities in *Sengled Strip E1G-G85* (ID = 5) with CVE-2022-47100; 2 vulnerabilities in *Third Reality 3RSB015BZ* (ID = 6) with CVE-2023-29780; and 1 vulnerability in *Sengled E1E-G7F* (ID = 13) with CVE-2023-29779; 1 vulnerability in *Aeotec ZW130-A* (ID = 15) with CVE-2023-34596; and 1 vulnerability in *Fibaro FGMS-001* (ID = 18) with CVE-2023-34597. To save CVE resources, given multiple vulnerabilities of a device that are related to a group of similar commands or exploit messages, only one CVE is requested.

6.4.1 Case Studies. Below we discuss several discovered vulnerabilities as case studies. The details of these vulnerabilities are summarized in Table 3.

Case 1: Centralite Pearl Thermostat (ID = 1). We discover two vulnerabilities: one is UV (*unknown vulnerability*) and the other DoS (Denial of Service). The two vulnerabilities have been assigned CVE-2023-24678. Both are related to the *hidden* command,

Table 3: *Some* of Discovered Vulnerabilities Details. (HD API: Hidden API, defined in Section 4.1.)

ID	HD API?	Command	Normal Mes. ⟶ Exploit (only *payload* is shown)	Required Device Initial State	Observations	CVE
1	✓	`Write_Battery_thres(uint8)`	0x02007d20 ⟶ 0x02007d42 ***Policy 2:*** Change argument type from `uint8` to `CharString`	Heating mode is set to 32 degrees Celsius	Device crashed & bricked	CVE-2023-24678
2	✓	`Move_up(uint8)`	0x01010003 ⟶ 0x01010000 ***Policy 1:*** Provide an invalid value 0x00 to the argument	Brightness is set to the lowest (0)	Device crashed	CVE-2022-47100
2	✓	`Move_down(uint8)`	0x01010102 ⟶ 0x01010100 ***Policy 1:*** Provide an invalid value 0x00 to the argument	Brightness is set to the highest (254)	Device crashed	CVE-2022-47100
5	✓	`Move_up_OnOff(uint8)`	0x01050007 ⟶ 0x01050000 ***Policy 1:*** Provide an invalid value 0x00 to the argument	Brightness is set to the lowest (0)	Device crashed	CVE-2022-47100
5	✓	`Move_down_OnOff(uint8)`	0x01050108 ⟶ 0x01050100 ***Policy 1:*** Provide an invalid value 0x00 to the argument	Brightness is set to the highest (254)	Device crashed	CVE-2022-47100
6	✗	`Down_close()`	0x0103 ⟶ 0x010303 ***Policy 3:*** Provide one or more arguments to the command	Any state	Device crashed	CVE-2023-29780
13	✓	`Set_short_poll_interval(uint16)`	0x01030009 ⟶ 0x01030000 ***Policy 1:*** Provide an invalid value 0x0000 to the argument	Any state	Device kept reporting state until battery was drained	CVE-2023-29779
15	✓	`Firmware_Update_Request_Get(uint8,uint8)`	0x7A038601 ⟶ 0x7A03ff ***Policy 3:*** Provide *only one* argument with an invalid value 0xff	Any state	Device crashed	CVE-2023-34596
18	✓	An invalid command in the `Clock` cluster	0x8101 ***Policy 4:*** Try unsupported cluster 0x81 and invalid command 0x01	Any state	Device crashed	CVE-2023-34597

`Write_Battery_thres (uint8)`, which can set a threshold for low battery alarms. This hidden command accepts one argument with the data type `uint8`.

Triggezring vulnerabilities. If we change the data type of this command to `CharString` (following the *fuzzing policy 2* in Section 5.3), and at the same time, the device heating mode is set to 32 degrees Celsius (the *required device initial state*), the generated testing message makes the device crash. Specifically, as shown in the fourth column in Table 3, the normal message is $0x02007d20$ (note that only the payload is shown here), where the last two digits $0x20$ represents the `uint8` data type. If we change it to $0x42$ representing the `CharString` data type, the message $0x02007d42$ causes the device to crash.

Observations. We have two observations indicating two kinds of vulnerabilities. (1) **UV:** If the exploit (i.e., $0x02007d42$) is sent *once*, the device loses the connection and reconnects automatically after around one second. (2) **Permanent DoS:** If the exploit is sent *multiple* times within a period of time (in our experiment, we send 50 commands in 100 seconds), the device loses the connection and cannot reconnect automatically anymore, allowing an attacker to conduct DoS attacks. Moreover, even if we reboot, manually factory reset, or manually pair the device, it still cannot rejoin the network anymore, indicating that the device is *completely bricked*.

The two vulnerabilities are probably because of null pointer dereference, buffer overflow, or memory leaks, which may be exploited to hijack the control flow.

Case 2: Sengled Bulb E11-N1EAW (ID = 2). We find eight vulnerabilities: four are UV and four DoS. These vulnerabilities have been assigned CVE-2022-47100. They are related to four *hidden* commands, `Move_up (uint8)`, `Move_down (uint8)`, `Move_up_OnOff (uint8)`, as well as `Move_down_OnOff (uint8)`, which can increase or decrease the brightness of the device with or without the *OnOff* effect at a certain *rate*. Each command accepts one argument with the data type `uint8`, which specifies the *rate* value.

Triggering vulnerabilities. If we set the device to the required initial state and then provide an invalid value $0x00$ to the argument (following the *fuzzing policy 1*), the testing message makes the device crash. Take `Move_up (uint8)` as an example (the third row in Table 3), the normal message is $0x01010003$, where the last two digits $0x03$ indicate the rate value. If we set the brightness of the device to the lowest (i.e., 0) and the rate to an invalid value $0x00$, the message $0x01010000$ causes the device to crash.

Observations. We have the following two observations for each command. (1) **UV:** If the exploit is sent *once*, the device flashes once, loses the connection, automatically changes to its factory status, and then rejoins the network after one second. (2) **Extended DoS:** If the exploit is sent *multiple* times within a period of time, the device cannot reconnect automatically until we manually pair it, allowing an attacker to conduct extended DoS attacks.

Case 3: Third Reality 3RSB015BZ (ID = 6). We detect two vulnerabilities, which have been assigned CVE-2023-29780. Both of them are related to the command, `Down_close()`, which can extend the smart blind to the maximum length. This command does not accept any argument. However, if we provide an argument to it (following the *fuzzing policy 3*), no matter what the current device state is, the generated testing message ***can trigger the vulnerabilities***.

Moreover, ***our observations are that*** if the message is sent once, the device loses the connection and rejoins automatically; however, if the message is sent multiple times in succession, the device loses connection and rejoins after around 30 seconds, allowing an attacker to conduct DoS attacks. *The two vulnerabilities are confirmed by the vendor.*

Case 4: Sengled E1E-G7F (ID = 13). We find one DoS vulnerability, which has been assigned CVE-2023-29779. This vulnerability is related to `Set_short_poll_interval (uint16)`, which can set the interval of the short poll. This command receives an argument with the data type `uint16`, which specifies the *interval* value. ***To trigger the vulnerability***, we provide an invalid value 0x0000 to the argument (it means that the interval is set to 0). ***Our observations are that*** the device keeps reporting its states with no interval and does not respond to any normal messages. More importantly, in our experiments, the device drained its battery finally as it kept reporting its status.

Case 5: Aeotec ZW130-A (ID = 15). We discover one UV vulnerability, which has been assigned CVE-2023-34596. This vulnerability is related to a *hidden* command, `Firmware_Update_Request_Get (uint8, uint8)`, which is used to initiate a firmware update. This command requires two arguments: `Manufacturer ID` and `Firmware ID`, both of which have the data type `uint8` and their valid values $\in [0, 254]$. ***To trigger the vulnerability***, we provide *only one* argument with an invalid value 0xff, which results in a device crash.

Case 6: Fibaro FGMS-001 (ID = 18). We reveal one DoS vulnerability, which has been assigned with CVE-2023-34597. The Fibaro FGMS-001 is designed to trigger various actions or events, such as detecting motion, changes in lighting, and temperature in the environment. ***To trigger the vulnerability***, an invalid command in an unsupported cluster, specifically the `Clock` cluster (0x81), is involved, which aims to synchronize the device clock with the controller system clock. We send a testing message containing the unsupported cluster (0x81) and an invalid command (0x01); the command is invalid because it does not exist in the cluster (0x81). ***Our observation is that*** if the testing message is sent multiple times within a period of time (in our experiment, we send it 120 times in around 60 seconds), the device crashes.

6.4.2 Comparison with Baseline Method. We consider `SNIPUZZ` as the baseline, which represents the state of the art in blackbox fuzzing of IoT devices [19]. We use it to test these devices. However, after 24 hours fuzz testing on each device, no crashes are found by `SNIPUZZ`. There are various reasons. First, `SNIPUZZ` needs the API-testing programs of IoT devices to collect seed messages, and can only test the commands that are covered by the API-testing programs. As a result, it cannot detect the vulnerabilities triggered by the *hidden* commands, which include all the vulnerabilities in Table 3, except those of *Third Reality Blind* (ID = 6).

Second, the vulnerabilities in *Third Reality Blind* (ID = 6), which is related to a *non*-hidden command, however, are still missed by `SNIPUZZ`. This is due to the ineffectiveness of its snippet determination algorithm—the algorithm determines that all the bytes in the seed message corresponding to the `Down_close()` command are in a *single* snippet, while the correct number of snippets is 15. **Because of the very inaccurate segmentation**, it fails to generate the messages to exploit the vulnerabilities.

We further evaluate the effectiveness of the snippet determination algorithm of `SNIPUZZ`. Specifically, for each discovered vulnerability, we provide `SNIPUZZ` with a message corresponding to the hidden command that can trigger the vulnerability. We then use the snippet determination algorithm to analyze these messages. The result shows that `SNIPUZZ` cannot accurately determine the snippets for any of them.

There are two main reasons that cause `SNIPUZZ` to fail. *First*, `SNIPUZZ` removes the bytes in a seed message one by one to generate probe messages. So it works for devices where the message formats are JSON, SOAP, and Key-Value—the removed byte usually do not impact the roles of the subsequent bytes in the message. However, for Z-based devices, the message format is a byte stream of fields. As a result, removing one byte causes to shift the subsequent bytes left by one, which changes the semantics of these bytes. *Second*, as discussed in `SNIPUZZ`, the effectiveness of its snippet determination algorithm depends on how much information could be obtained from the device responses. However, most of our devices report the errors with a uniform message, making it infeasible for `SNIPUZZ` to accurately conduct message segmentation.

The discussion above indicates that even if API-testing programs are upgraded to include the hidden commands and `SNIPUZZ` modifies the API-testing programs to inject testing messages in order to overcome the barrier of encrypted communication, `SNIPUZZ` still fails to detect the vulnerabilities. Moreover, `SNIPUZZ` does *not* leverage state-sensitive fuzzing, while we make use of it to find most most of the vulnerabilities (18 out of 23).

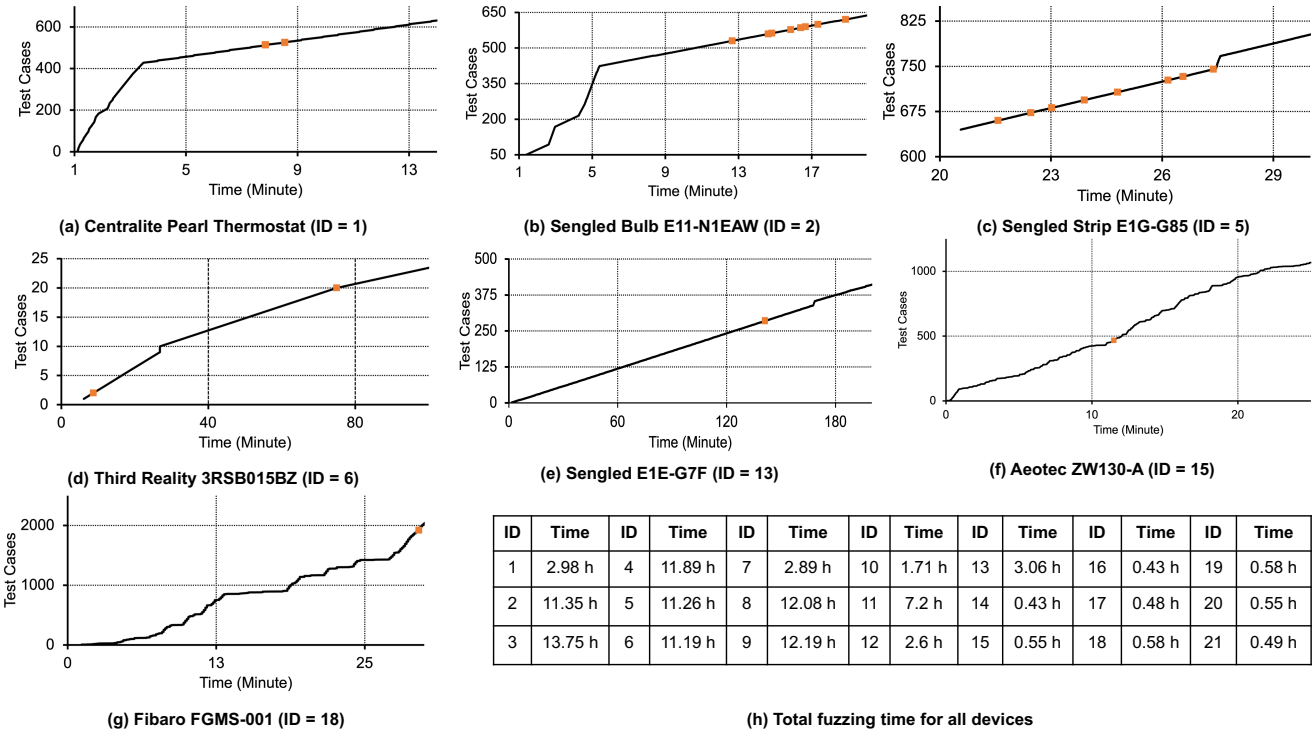
6.5 Efficiency

We measure the efficiency of fuzzing in terms of vulnerabilities discovered over time and over the number of testing messages. The results are shown in Figure 8(a)-(g). We can see that HubFuzzer can efficiently discover vulnerabilities. For example, for *Centralite Pearl Thermostat* (ID = 1) in Figure 8(a), within 9 minutes and less than 600 test messages, the two vulnerabilities are found. For *Sengled Bulb E11-N1EAW* (ID = 2) in Figure 8(b), all the eight vulnerabilities are detected within 20 minutes.

In Figure 8(h), we present the total time taken by HubFuzzer for fuzz testing each of the 21 devices. Since we can obtain the functions supported by IoT devices from the setting-up messages, we test all of them. For ZigBee devices (ID from 1 to 13), the longest fuzz testing time is 13.75 hours for the device ID = 3. For Z-Wave devices (ID from 14 to 21), the fuzz testing time is about half an hour due to relatively fewer commands supported by Z-Wave devices.

7 DISCUSSION

Testing WiFi and Bluetooth Devices. We showcase the proposed idea and techniques on ZigBee and Z-Wave IoT devices, which have a large market size [17, 29]. Essentionally, the fuzzing idea leverages the local IoT control channel, which enables an IoT device to be controlled locally. A device supports one or more local control channels as long as it is compatible with HomeKit [31], and many IoT vendors support HomeKit. According to the HomeKit protocol,



ID	Time	ID	Time	ID	Time	ID	Time	ID	Time	ID	Time	ID	Time
1	2.98 h	4	11.89 h	7	2.89 h	10	1.71 h	13	3.06 h	16	0.43 h	19	0.58 h
2	11.35 h	5	11.26 h	8	12.08 h	11	7.2 h	14	0.43 h	17	0.48 h	20	0.55 h
3	13.75 h	6	11.19 h	9	12.19 h	12	2.6 h	15	0.55 h	18	0.58 h	21	0.49 h

(h) Total fuzzing time for all devices

Figure 8: Runtime performance. (a)-(g) show the vulnerabilities (red dots) discovered over time and over the number of testing messages. (h) shows the total fuzzing time for testing each of the 21 devices.

a HomeKit-enabled device not only pairs with a HomeKit hub but also declares its device type and functions [4]. Our future work will exploit the HomeKit framework for building a hub to test WiFi and Bluetooth devices. The work certainly involves new technical challenges, but it is promising. For example, some tutorials (Home-Assistant [59], HomeBridge [28]) introduce how to build your own opensource HomeKit hub, which can be a starting point.

Testing Matter Devices. Matter and Thread are two industry-wide standards designed to improves smart home device interoperability and connectivity [2]. Matter standardizes the application layer, and Thread the lower layers. They are developed by Google, Amazon, Apple and more than a hundred other leading technology companies. This provides great research opportunities. Multiple vendors target December 2022 to release devices that implement Matter/Thread [1]. A key feature of Matter is that IoT devices can be controlled locally [43]. Specifically, a Matter-compatible device must be accessible to smart home control centers in the home network, without a detour via the Internet. The control center communicates with IoT devices at home directly, which in principle is similar to the HomeKit architecture. Thus, a promising future work is to apply the hub-based fuzzing idea to testing Matter devices.

8 CONCLUSION

Blackbox fuzzing of IoT firmware gains growing attention, as it delivers promising results. Different from prior work that reverse engineers companion apps or manually collects test scripts, we propose another direction: hub-based IoT fuzzing, where the fuzzer declares itself as a hub to connect IoT devices. HubFuzzer exploits the setting-up messages to discover functions supported by IoT devices and then performs systematic function-oriented fuzzing. The mutation is guided by knowledge extracted from IoT protocol specifications. State-sensitive fuzzing is conducted, which is effective in finding vulnerabilities. Our fuzzing is not constrained by input sanitization of companion apps and can deal with encrypted communication. We have implemented HubFuzzer and conducted an extensive evaluation with 21 popular IoT devices. We discovered 23 zero-day vulnerabilities, significantly outperforming prior state of the art. Six CVEs have been assigned. We advocate that, because of the imminent popularity of Matter/Threat devices and their emphasis on interoperability, the hub-based fuzzing is worth further exploration.

ACKNOWLEDGEMENTS

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-2309550, CNS-2310322, CNS-2309477, and CNS-2304720. The authors would like to thank the anonymous reviewers for their valuable comments.

APPENDIX

The research artifact accompanying this paper is available via https://doi.org/10.5281/zenodo.7924626.

REFERENCES

[1] 2022. Matter (standard). https://en.wikipedia.org/wiki/Matter_(standard).

[2] 2022. Your home is getting more helpful with Matter and Thread. https://store.google.com/intl/en/ideas/articles/matter-thread-for-your-smart-home/.

[3] Zafeer Ahmed, Ibrahim Nadir, Haroon Mahmood, Ali Hammad Akbar, and Ghalib Asadullah Shah. 2020. Identifying mirai-exploitable vulnerabilities in IoT firmware through static analysis. In *Proc. IEEE International Conference on Cyber Warfare and Security (ICCWS)*.

[4] Apple. 2023. Developing Apps and accessories for the home. https://developer.apple.com/apple-home/.

[5] Home Assistant. 2022. Z-Wave JS. https://www.home-assistant.io/integrations/zwave_js/.

[6] Lina Berzinskas. 2020. Obfuscating Android Apps: Do you know your choices for protection? https://proandroiddev.com/obfuscation-is-important-do-you-know-your-options-30b3ef396dfe.

[7] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference (ACSAC)*.

[8] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for Linux-based embedded firmware. In *Network and Distributed System Security Symposium (NDSS)*.

[9] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering memory corruptions in IoT through App-based fuzzing. In *Network and Distributed System Security Symposium (NDSS)*.

[10] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. 2021. PFirewall: Semantics-aware customizable data flow control for home automation systems. In *Network and Distributed System Security Symposium (NDSS)*.

[11] Connectivity Standards Alliance. 2022. Building the foundation and future of the IoT. https://csa-iot.org.

[12] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proc. ACM Asia Conference on Computer and Communications Security (ASIACCS)*.

[13] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium (USENIX Security)*.

[14] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *Security and Privacy in Communication Networks*.

[15] Jianhe Du, Xin Luo, Libiao Jin, and Feifei Gao. 2022. Robust tensor-based algorithm for UAV-assisted IoT communication systems via nested PARAFAC analysis. *IEEE Transactions on Signal Processing* 70 (2022), 5117–5132.

[16] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. 2022. AflIot: Fuzzing on Linux-based IoT device with binary-level instrumentation. *Computers & Security* 122 (2022), 102889.

[17] Z-Wave explained: What is Z-Wave and why is it important for your smart home? 2022. https://www.the-ambient.com/guides/zwave-z-wave-smart-home-guide-281.

[18] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[19] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[20] GoControl CECOMINOD016164 HUSBZB-1 USB Dongle. 2023. https://www.amazon.com/GoControl-CECOMINOD016164-HUSBZB-1-USB-Hub/dp/B01GJ826F.

[21] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. 2020. FIRMCORN: Vulnerability-oriented fuzzing of IoT firmware via optimized virtual execution. *IEEE Access* 8 (2020), 29826–29841.

[22] Zhijie Gui, Hui Shu, and Ju Yang. 2020. FIRMNANO: Toward IoT firmware fuzzing through augmented virtual execution. In *Proc. IEEE International Conference on Software Engineering and Service Science (ICSESS)*.

[23] Daojing He, Hongjie Gu, Tinghui Li, Yongliang Du, Xiaolei Wang, Sencun Zhu, and Nadra Guizani. 2020. Toward hybrid static-dynamic detection of vulnerabilities in IoT firmware. *IEEE Network* 35, 2 (2020), 202–207.

[24] Daojing He, Xiaohu Yu, Tinghui Li, Sammy Chan, and Mohsen Guizani. 2022. Firmware vulnerabilities homology detection based on clonal selection algorithm for IoT devices. *IEEE Internet of Things Journal* 9, 17 (2022), 16438–16445.

[25] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. 2017. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[26] Home Assistant. 2022. Home Assistant. https://www.home-assistant.iol.

[27] Home Assistant. 2022. Zigbee Home Automation. https://www.home-assistant.io/integrations/zha/.

[28] Homebridge. 2021. Homebridge HomeKit Control. https://github.com/minamoanes/homebridge-homekit-control.

[29] Global Market Insights Inc. 2021. IoT Gateway Devices Market Revenue to Cross USD 20 Bn by 2027. https://www.prnewswire.com/news-releases/iot-gateway-devices-market-revenue-to-cross-usd-20-bn-by-2027-global-market-insights-inc-301336087.html.

[30] Fortune Business Insights. 2022. Internet of Things Market Size [2022-2029] Worth USD 2465.26 Billion. https://www.globenewswire.com/en/news-release/2022/04/04/2415728/0/en/Internet-of-Things-Market-Size-2022-2029-Worth-USD-2465-26-Billion-Exhibiting-a-CAGR-of-26-4.html.

[31] Yan Jia, Bin Yuan, Luyi Xing, Dongfang Zhao, Yifan Zhang, XiaoFeng Wang, Yijing Liu, Kaimin Zheng, Peyton Crnjak, Yuqing Zhang, et al. 2021. Who's in control? On security risks of disjointed IoT device management channels. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[32] Joshua Pereyda. 2017. Boofuzz: Network protocol fuzzing for humans. https://boofuzz.readthedocs.io/en/stable/.

[33] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards large-scale emulation of IoT firmware for dynamic analysis. In *Annual Computer Security Applications Conference (ACSAC)*.

[34] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *USENIX Workshop on Offensive Technologies*.

[35] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *International Conference on Software Engineering (ICSE)*.

[36] Xiaopeng Li, Qiang Zeng, Lannan Luo, and Tongbo Luo. 2020. T2Pair: Secure and usable pairing for heterogeneous IoT devices. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[37] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. 2018. Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization. In *Proc. IEEE International Conference on Information and Communications Security (ICICS)*.

[38] Lannan Luo and Qiang Zeng. 2016. SolMiner: Mining distinct solutions in programs. In *International Conference on Software Engineering (ICSE)*.

[39] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2017. System service call-oriented symbolic execution of Android framework with applications to vulnerability discovery and exploit generation. In *Proc. ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[40] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. 2019. Tainting-assisted and context-migrated symbolic execution of Android framework for vulnerability discovery and exploit generation. *IEEE Transactions on Mobile Computing* 19, 12 (2019), 2946–2964.

[41] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. 2021. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart Apps on IoT cloud platforms. In *Annual Computer Security Applications Conference (ACSAC)*.

[42] Global Z-Wave Automation Market. 2022. https://www.prophecymarketinsights.com/market_insight/Global-Z-Wave-Automation-Market-4593.

[43] Matter-Smarthome. 2023. Benefits of Matter #1: Local connection. https://matter-smarthome.de/en/benefits/benefits-of-matter-1-local-connection/.

[44] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *Proc. IEEE Symposium on Security and Privacy (S&P)*.

[45] Ibrahim Nadir, Zafeer Ahmad, Haroon Mahmood, Ghalib Asadullah Shah, Farrukh Shahzad, Muhammad Umair, Hassam Khan, and Usman Gulzar. 2019. An auditing framework for vulnerability analysis of IoT system. In *IEEE European Symposium on Security and Privacy Workshops*.

[46] Anh TV Nguyen and Mizuhito Ogawa. 2022. Automatic stub generation for dynamic symbolic execution of ARM binary. In *International Symposium on Information and Communication Technology*.

[47] OpenHAB. 2022. openhab–features–introduction. http://www.openhab.org/features/introduction.html.

[48] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. DIANE: Identifying fuzzing triggers in Apps to generate under-constrained inputs for IoT devices. In *Proc. IEEE Symposium on Security and Privacy (S&P)*.

[49] Sandler Research. 2016. ZigBee Home Automation Market to Grow at 26% CAGR to 202. https://prn.to/3jLYLmk.

[50] Vinay Sachidananda, Suhas Bhairav, and Yuval Elovici. 2020. OVER: Overhauling vulnerability detection for IoT through an adaptable and automated static analysis framework. In *Proc. ACM Symposium on Applied Computing (SAC)*.

[51] Zhan Shu and Guanhua Yan. 2022. IoTInfer: Automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference. *IEEE Internet of Things Journal* 9, 22 (2022), 22737–22751.

[52] Smartthings. 2022. Z-Wave Command Classes. https://graph.api.smartthings.com/ide/doc/zwave-utils.html.

[53] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. Firmfuzz: Automated IoT firmware introspection and

analysis. In *International ACM Workshop on Security and Privacy for the Internet-of-Things*.

[54] Pengfei Sun, Luis Garcia, Gabriel Salles-Loustau, and Saman Zonouz. 2020. Hybrid firmware analysis for known mobile and IoT security vulnerabilities. In *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[55] IoT World Today. 2021. IoT Cyberattacks Escalate in 2021, According to Kaspersky. https://www.iotworldtoday.com/2021/09/17/iot-cyberattacks-escalate-in-2021-according-to-kaspersky/.

[56] Junzhe Wang and Lannan Luo. 2022. Privacy leakage analysis for colluding smart Apps. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*.

[57] Webthings. 2022. https://webthings.io.

[58] wireghoul. 2019. Doona. https://github.com/wireghoul/doona.

[59] Wltd. 2020. How to Easily Use a Raspberry Pi as a Homekit Hub. https://wltd.org/posts/how-to-easily-use-a-raspberry-pi-as-a-homekit-hub.

[60] Zelin Xu, Wei Huang, Wenqing Fan, and Yixuan Cheng. 2022. FIoTFuzzer: Response-based black-box fuzzing for IoT devices. In *Proc. IEEE/ACIS International Conference on Computer and Information Science (ICIS)*.

[61] Yao Yao, Wei Zhou, Yan Jia, Lipeng Zhu, Peng Liu, and Yuqing Zhang. 2019. Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution. In *European Symposium on Research in Computer Security*.

[62] Qidi Yin, Xu Zhou, and Hangwei Zhang. 2021. FirmHunter: State-aware and introspection-driven grey-box fuzzing towards IoT firmware. *Applied Sciences* 11, 19 (2021), 9094.

[63] Z Wave Vs ZigBee: Which Is Better For Your Smart Home? 2022. https://thesmartcave.com/z-wave-vs-zigbee-home-automation/.

[64] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*.

[65] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized Android application repackaging detection using logic bombs. In *Proc. ACM International Symposium on Code Generation and Optimization (CGO)*.

[66] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home Apps from encrypted traffic. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[67] Yu Zhang, Nanyu Zhong, Wei You, Yanyan Zou, Kunpeng Jian, Jiahuan Xu, Jian Sun, Baoxu Liu, and Wei Huo. 2022. NDFuzz: A non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity* 5, 1 (2022), 1–21.

[68] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *USENIX Security Symposium (USENIX Security)*.

[69] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[70] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic firmware emulation through invalidity-guided knowledge inference. In *USENIX Security Symposium (USENIX Security)*.

[71] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. 2022. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[72] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. 2019. FIoT: Detecting the memory corruption in lightweight IoT device firmware. In *IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ IEEE International Conference On Big Data Science And Engineering*.

[73] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

[74] ZigBee. 2022. ZigBee Cluster Library Specification. https://zigbeealliance.org/wp-content/uploads/2019/12/07-5123-06-zigbee-cluster-library-specification.pdf.

[75] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Network and Distributed System Security Symposium (NDSS)*.