Z-Fuzzer: Device-agnostic Fuzzing of Zigbee Protocol Implementation

Mengfei Ren
mengfei.ren@mavs.uta.edu
The University of Texas at Arlington
Arlington, Texas, USA

Xiaolei Ren
xiaolei.ren@mavs.uta.edu
The University of Texas at Arlington
Arlington, Texas, USA

Huadong Feng
huadong.feng@mavs.uta.edu
The University of Texas at Arlington
Arlington, Texas, USA

Jiang Ming
jiang.ming@uta.edu
The University of Texas at Arlington
Arlington, Texas, USA

Yu Lei
ylei@cse.uta.edu
The University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

With the proliferation of the Internet of Things (IoT) devices, Zigbee is widely adopted as a resource-efficient wireless protocol. Recently, severe vulnerabilities in Zigbee protocol implementations have compromised IoT devices from different manufacturers. It becomes imperative to perform security testing on Zigbee protocol implementations. However, it is not a trivial task to apply the existing vulnerability detection techniques such as fuzzing to Zigbee protocol implementations. In particular, it remains a significant obstacle to deal with low-level hardware events. Many existing protocol fuzzing tools lack a proper execution environment for the Zigbee protocol, which communicates via a radio channel instead of the Internet.

To bridge the above gap, we develop a device-agnostic fuzzing platform named *Z-Fuzzer* to detect security vulnerabilities in Zigbee protocol implementations. Z-Fuzzer provides a software simulation environment with pre-defined peripherals and hardware interrupts configurations to simulate Zigbee protocol execution on real IoT devices. We first extend the existing protocol fuzzing framework's capabilities with a proxy server to bridge communication with the Zigbee protocol execution. Second, we generate more high-quality test cases with code-coverage heuristics. We compare Z-Fuzzer with advanced protocol fuzzing tools, BooFuzz and Peach fuzzer, on top of Z-Fuzzer's simulation platform. Our results show that Z-Fuzzer can achieve higher code coverage in a mainstream Zigbee protocol implementation called Z-Stack. Z-Fuzzer has detected more vulnerabilities using fewer test cases than BooFuzz and Peach. Three of them have been assigned CVE IDs with high CVSS scores (7.5~8.2).

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

WiSec '21, June 28–July 2, 2021, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-8349-3/21/06.
https://doi.org/10.1145/3448300.3468296

KEYWORDS

IoT network, Zigbee protocol, Fuzzing

ACM Reference Format:
Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. 2021. Z-Fuzzer: Device-agnostic Fuzzing of Zigbee Protocol Implementation. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), June 28–July 2, 2021, Abu Dhabi, United Arab Emirates.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3448300.3468296

1 INTRODUCTION

With the increasing popularity of the Internet of Things (IoT) devices, Zigbee protocol [4] has become a dominant wireless protocol to reduce the power and memory cost of IoT devices. It has been widely adopted in home automation, green power, and manufacturing [8, 27, 42]. The Zigbee protocol transmits data through the radio channel instead of the Internet; thus, it has a close dependency on an embedded device's hardware configuration. The Zigbee protocol is defined by the Zigbee Alliance [5] with various security services that offer a range of options within a Zigbee network [6]. Recent works have disclosed the weaknesses and vulnerabilities of different Zigbee stack implementations [11, 17, 36, 43, 50]. These vulnerabilities can be exploited to launch a DDoS attack or a remote code execution to Philips lighting system that utilizes the Zigbee protocol. Even though some previous security vulnerabilities have been addressed in the Zigbee protocol's latest version, an automated bug-finding tool is still missing. Therefore, detecting security vulnerabilities from Zigbee protocol implementations is necessary and of great practical significance.

Fuzz testing (fuzzing) [46] is a popular security testing technique to discover software vulnerabilities by executing the target program with random inputs. AFL [53] is a widely used grey-box fuzzing platform that leverages code-coverage heuristics to guide test input generation. It can either instrument source code of software under test if available or execute a closed-source binary file with QEMU emulation platform [14] to obtain dynamic instrumentation output. Many advanced fuzzing approaches have extended AFL capabilities [15, 23, 39]. However, it is not a trivial task to apply these fuzzing approaches to Zigbee protocol implementations. Firstly, AFL-based fuzzers could violate compiler examination when injecting instrumentations to the Zigbee protocol's source code if available. Zigbee protocol vendors generally develop the protocol for specific embedded devices using a particular development

toolchain [24]. In terms of protocol availability, vendors deploy compiler checks in their implementations to prevent compilers that are not in the supported list, especially the general compilers (e.g., GCC and LLVM) used by most AFL-based fuzzers.

Moreover, when the source code of software under test is unavailable, some existing fuzzing methods can fuzz the software's binary file in a simulation environment [22, 34, 44, 56]. QEMU[14] is a widely adopted software emulation platform by these fuzzers. However, those approaches cannot provide a proper execution environment for the Zigbee protocol. The Zigbee protocol is usually executed in the embedded devices that are system-on-chip (SoC) and baremetal containing a single control loop for scheduling tasks and handling events [24]. Hence, it is not feasible to fuzz the Zigbee protocol on the simulation platforms that requires the presence of a Linux kernel or an abstraction layer [34, 37, 44, 52, 56]. Though some QEMU-based embedded fuzzer [22] also supports baremetal programs and various embedded CPU types, they currently do not support devices that can execute different Zigbee protocol implementations required by vendors [40]. As the Zigbee protocol is developed for specific devices by different vendors, the protocol binary file cannot even boot on QEMU if the required devices are not supported. Additionally, the Zigbee protocol stack interacts with events triggered by particular peripheral interrupts, which are not supplied in existing solutions [16]. The same peripheral can also be configured differently on various devices with different interrupts [22]. It requires significant engineering efforts for the existing simulation platform to add support for all device-specific peripherals and new embedded chips, if not impossible.

In this paper, we develop *Z-Fuzzer*, a device-agnostic protocol simulation platform to enable the fuzz testing of Zigbee protocol implementations. Z-Fuzzer consists of a fuzzing engine and a test harness for executing the protocol stack. Z-Fuzzer firstly aims to generate high-quality test cases that satisfy the protocol packet format. Thus, we augment grammar-based fuzzing [25] with coverage-guided feedback to generate high-quality test cases and prioritize them to favor finding vulnerabilities.

The test harness simulates the execution environment for the Zigbee protocol stack and analyzes the code coverage feedback. Most Zigbee protocol manufacturers have released their protocol stack source code to encourage the IoT development community contributions. Given the implementation's source code, we analyze it to construct an execution environment with the required system configuration in the test harness. To execute the Zigbee protocol stack, we utilize an industrial embedded device development platform, IAR Embedded Workbench [47], to simulate the embedded device. Many Zigbee protocol vendors use the IAR toolchain, which provides a compiler, a linker, and a device simulator. The software simulator supports most embedded devices in the current IoT market across various device vendors (e.g., Samsung, Toshiba, and Texas Instruments). The IAR Workbench has also pre-defined a set of device-specific hardware interrupt/peripheral configurations to fully simulate the embedded device. As the simulator does not provide a network interface to transmit test messages, we also develop a proxy server to bridge the communication between the fuzzing engine and the simulator. The test harness monitors the simulator execution result to collect memory crashes and calculates the code coverage for every execution. Z-Fuzzer can detect explicit memory crashes that are returned with crash call stacks from the simulator.

We have implemented Z-Fuzzer and evaluated its effectiveness in detecting security vulnerabilities. In terms of fuzzing strategy, we select two state-of-art protocol fuzzing platforms, BooFuzz [38] and Peach [49], as our comparative tools. BooFuzz is the successor of industry-standard protocol fuzzer Sulley [21], and Peach [49] is a commercial protocol fuzzer that has been widely used. We run BooFuzz and Peach on top of our Zigbee protocol simulation platform and compare them with Z-Fuzzer by fuzzing Z-Stack [29], a mainstream Zigbee protocol implementation developed by Texas Instruments (TI), for which the source code is available. The results indicate that Z-Fuzzer effectively increases code coverage and detects security vulnerabilities. Z-Fuzzer has identified six unique previously unknown vulnerabilities in Z-Stack implementation with fewer test cases than BooFuzz and Peach. We have reported all of the new vulnerabilities to TI. Three of these vulnerabilities have been assigned CVE IDs with high CVSS scores (7.5~8.2) [20] at the time of writing, while others are still under review. Our work sheds light on detecting the Zigbee protocol vulnerabilities in a software simulation environment without accessing a physical device. Overall, we make the following contributions.

- Z-Fuzzer provides a device-agnostic fuzzing platform for Zigbee protocol implementations. Z-Fuzzer uses a full software simulator and includes a proxy server that facilitates communication between the fuzzing engine and the simulator.
- We improve the test generation process of the grammar-based fuzzing by leveraging code coverage information as feedback to generate higher-quality test cases. Compared to BooFuzz and Peach, Z-Fuzzer can generate fewer test cases while achieving higher code coverage.
- We have discovered six previously unknown vulnerabilities in Z-Stack. Three of them have been assigned CVE IDs with high-severity ratings.

Open source. To facilitate the reproducibility of the research results, we release Z-Fuzzer's source code, which is publicly available at https://github.com/zigbeeprotocol/Z-Fuzzer.

2 RELATED WORK

As Z-Fuzzer is a fuzzing platform of the Zigbee protocol, in this section, we discuss related work in fuzz testing and security analysis of Zigbee.

2.1 Fuzz Testing

Fuzz testing is a widely used technique to detect vulnerabilities. In recent years, fuzzing with AFL [53] and its extension tools like AFL++ [23] has become famous for automated security analysis. The fuzzers utilize code-coverage heuristics to guide their mutation process. Unfortunately, they fail to test the source code of the Zigbee protocol [4]. The Zigbee protocol is developed by different protocol vendors using some particular toolchains. They enforce embedded-system-specific compiler examinations in their implementations, which prevent instrumentations of AFL-like fuzzers using general compilers such as GCC and LLVM. In contrast, Z-Fuzzer utilizes the embedded compiler of the IAR Workbench [47]

to inject instrumentations in the source code, which most Zigbee protocol vendors support.

Fuzzing on IoT embedded devices is also challenging due to the strong dependency on the device hardware configuration. Many existing researches [22, 34, 37, 44, 56] integrate an emulator to emulate IoT firmware with their fuzzing tools. A notable emulator is QEMU [14] which provides user-mode emulation and full emulation for a variety of embedded devices. AFL QEMU mode [54], Frankenstein [44] and BaseSAFE [34] utilize QEMU in user-mode, which require presence of Linux kernel or an abstract layer. Some researches [22, 37, 56] propose hybrid solutions that combine user-mode and full emulation together.

However, none of them can be directly applied to the Zigbee protocol execution due to the insufficient simulation environment of QEMU for the Zigbee protocol. First, the Zigbee protocol is usually executed on ARM-based Systems on Chip (SoC) devices with a baremetal system. It is infeasible to simulate the protocol firmware with the fuzzers in QEMU user-mode that requires embedded OS. Although QEMU also supports the baremetal program, few embedded devices supported by QEMU are compatible with the Zigbee protocol. For example, QEMU currently supports two ARM Cortex M3 microcontrollers from Texas Instruments (TI) [41]. TI also develops a Zigbee protocol implementation, called Z-Stack [29], for some particular ARM-M3 embedded boards which do not contain the devices supported by QEMU. Due to the strong dependency on hardware configurations (e.g., peripheral interrupts and off-site sensors), even the same SoC on different machines can still vary further. As the required embedded platforms by protocol vendors for Zigbee are not supported, QEMU could fail to boot the Zigbee system image [41]. In contrast, the IAR simulator supports various embedded devices from different vendors that can execute Zigbee system images. It has also pre-defined a set of device-specific hardware interrupt/peripheral configurations to properly simulate the embedded device for the Zigbee protocol.

2.2 Zigbee Security

Various approaches to analyze the security problems of the Zigbee protocol exist. Z3Sec [36] by Morgner et al. and Snout [35] by Mikulskis et al. are penetration platforms to assess existing vulnerabilities in Zigbee by packet replaying and spoofing. IoTcube [31] and beSTORM [45] are developed to analyze the security of the Zigbee protocol on particular embedded devices. Akestoridis et al. [3] also proposed a security analysis tool, called Zigator, to analyze encrypted Zigbee packets for selective jamming and spoofing attacks. These security analysis applications are black-box approaches that monitor and manipulate Zigbee network traffic to detect security issues in Zigbee.

Ronen et al. [43] proposed an attack model that exploits an existing bug in a particular protocol implementation to perform over-the-air device update with malicious firmware. The proposed model can launch massive DDoS attacks in smart lighting systems. In terms of fuzzing Zigbee, Cui et al. proposed two fuzzing platforms for Zigbee based on a finite state machine [18] and a genetic algorithm [19]. Unfortunately, both of these fuzzers are not open source, so we cannot compare their performance.

Compared to the above security exploitation works, Z-Fuzzer aims to detect unknown vulnerabilities in the Zigbee protocol implementations' source code rather than the real-time Zigbee network. It does not require physical devices and specific knowledge of the underlying hardware design. Our experiment results show that the protocol stack's high-layer vulnerabilities can also lead to catastrophic failures and risks in the IoT application's functionalities.

3 BACKGROUND

In this section, we first provide the background information necessary to understand the Zigbee protocol in Section 3.1. Then, we introduce the status quo of protocol fuzzing techniques in Section 3.2 and their limitations, which motivate us to develop our fuzzing platform. Finally, we discuss the problem scope and assumptions in Section 3.3.

3.1 Zigbee Protocol

Zigbee is a low-cost, low-power-consumption, two-way wireless communication protocol [6]. The Zigbee Alliance defines the operation of a Zigbee network and the protocol specification [5].

The Zigbee protocol stack, as shown in Figure 1a, is designed as a 4-layer stack on top of the IEEE 802.15.4 standard. The Zigbee Alliance defines the upper two layers, i.e., Application Layer (APL) and Network Layer (NWK). The IEEE 802.15.4 standard defines Medium Access Control Layer (MAC) and Physical Layer (PHY). They aim to support packet transmission via the radio channel in a Zigbee network. The APL is responsible for the application-level functionalities, whereas the NWK layer manages the Zigbee network and forwards packets.

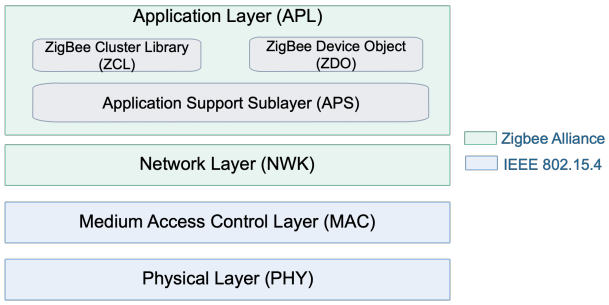
Figure 1b shows a prototype of a message exchange between two Zigbee devices. The manufacturer's application in the controller can initiate a service request with commands in Zigbee Cluster Library (ZCL), which are defined to perform device functionalities. The ZCL then sends the request to the sub-layers. The message is transmitted over the air. After receiving the message, the ZCL in the end device processes and passes it to the upper application's request for a response.

From the user's perspective, the ZCL is an application layer protocol and the Zigbee protocol stack's main library to perform all of the device's functionalities. Therefore, we use ZCL as a case study for fuzzing the Zigbee protocol implementation in this paper.

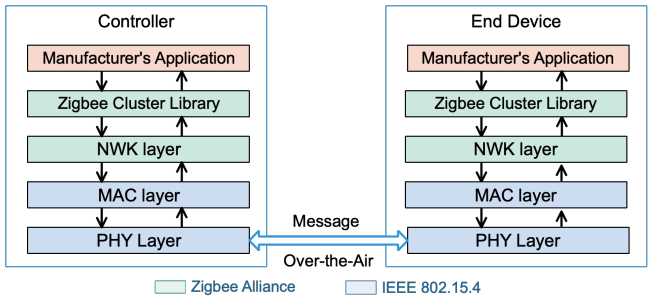
3.2 Protocol Fuzzing

Many black-box protocol fuzzing approaches are proposed and developed to generate high-structured packets that conform to network protocol format requirements. These fuzzing approaches (e.g., SPIKE [2], Sulley [21], BooFuzz [38], AutoFuzz [26], and SNOOZE [13]) widely adopt *grammar-based fuzzing* [25]. They construct test inputs from scratch according to the input specifications that define data format and integrity constraints.

These protocol fuzzers generate protocol frames with abstract representation *blocks*, which is also called *block-based protocol representation* [1]. A *block* is an abstract set organizing several primitive data or nested blocks which conforms to protocol format. With the format definition script, the fuzzer represents the protocol message with primitive data following their placements.



(a) Zigbee protocol stack overview [6].



(b) Zigbee protocol message exchange [28].

Figure 1: Zigbee protocol communication.

ZCL Header				ZCL Payload
Frame Control	Manufacturer Code	Transaction Sequence Number	Command Identifier	Frame Payload

(a) ZCL frame format [7].

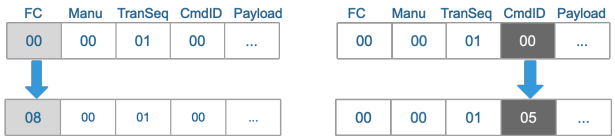
```
1  s_initialize("ZCLMessage")
2  s_group("frame_control", values=<USER_GIVEN_VALUES>)
3  with s_block("manuCode", dep="frame_control", dep_values =
       <USER_GIVEN_VALUES>):
4    s_word(0, endian='<', name="manu")
5  s_byte(1, name="tranSeq")
6  s_group("commandId", values=<USER_GIVEN_VALUES>)
7  with s_block("payload", dep="commandId", values =
       <USER_GIVEN_VALUES>):
8    .....
```

(b) Message construction script.

Figure 2: Example of ZCL message construction with block-based representation. Each message field is generated in the order of their placement in the message format.

Figure 2b shows a practical example of representing the ZCL message with the ZCL frame format displayed in Figure 2a. The entire ZCL message is initialized as a block with the name `ZCLMessage` (line 1 in Fig 2b). Each message field is represented as primitive data based on their types (line 2, 4, 5, 6, and 8 in Fig 2b). If a message field has user-specific values, then it is represented as a `group` primitive data, e.g., the field *frame control* is defined as line 2 in Fig 2b with the user-given values. Besides, some message fields may depend on another field. Such a field is represented within a `block` primitive data that specifies the field's dependency constraint (line 3 and 7 in Fig 2b). For example, the field *Manufacture Code* depends on the field *Frame Control*. It is generated in a test message only if the value of the field *Frame Control* satisfies the user-specific requirement. Then it is represented within the `block` primitive data `frame_control` (line 3 in Fig 2b).

Although these protocol fuzzers with block-based representation defines and generates highly structured input formats, they have a disadvantage. The quality of test input generation is low due to the lack of feedback [12, 33]. They do not prioritize any test cases to improve fuzzing performance. Many interesting test cases are discarded for further fuzzing, which would have achieved a new



(a) Test Case 1 (b) Test Case 2

Figure 3: Example of mutation in protocol fuzzing.

code coverage in previous executions. Here we use an example running the ZCL message on an existing protocol fuzzing framework to explain the limitations.

In a fuzzing iteration, the fuzzer only mutates a single message field. Figure 3 shows that test case 1 is generated by fuzzing *Frame Control* (FC) with the actual value `08`, which has triggered new code. Once the mutation on this field completes, the fuzzer resets it to its original value `00`. Then, the fuzzer continues to mutate the following fields; test case 2 is generated by fuzzing *Command Identifier* (CmdID), in which the actual value `05` executes another new line of the statement.

Following this example, we make an observation. Messages with the actual value `08` or `05` have increased code coverage, and they should have been retained for further fuzzing, which may exercise more statements and paths in the target program. Based on our static analysis of the Zigbee protocol source code, we found the message with these two interesting values (`08` & `05`) executes a particular `if` condition in the code. However, such a test case is missing during the fuzzing process. The common limitation of grammar-based protocol fuzzing tools is that they do not consider heuristic feedback [12, 33].

As motivated by the above observations, we leverage code-coverage heuristics to augment the existing protocol fuzzing process. When a test case executes a new code, we store it as a favored test case with the interesting value that increases the coverage. In a new fuzzing cycle, we mutate the favored test cases first by keeping the interesting values and mutating other fields to generate more valuable test cases. We will explain more details in Section 4.

3.3 Problem Scope and Assumptions

This paper aims at discovering vulnerabilities in the Zigbee protocol implementations without the real IoT devices; thus, the IoT

Algorithm 1: Z-Fuzzer Protocol Fuzzing Algorithm

Input : Input format script $\mathcal{S}$, Program under test $\mathcal{P}$
Output: Seeds that crash the program *crash*,
The cumulative code coverage *cumCoverage*

```
1  crash ← ∅
2  blocks ← Initialize(S)
3  top_rated ← ∅
4  cumCoverage ← 0
5  while true do
6      if top_rated not empty then
7          favored ← Select(top_rated)
8          seed ← Mutate(favored)
9          if favored.was_fuzzed then
10             top_rated ← top_rated − favored
11     else
12         testcase ← Choose(blocks)
13         seed ← Mutate(testcase)
14     end
15     coverage, result ← RunTarget(P, seed)
16     if isInteresting(coverage, result) then
17         top_rated ← top_rated ∪ seed
18         cumCoverage ← CalCoverage(coverage)
19         crash ← crash ∪ result
20     end
21 end
22 return crash, cumCoverage
```

application manufacturers can take corresponding actions before or during their development phase to avoid security risks. Therefore, we require the source code of the target protocol implementation, which is easy to obtain from the protocol stack vendor in practice. Currently, we focus on generating high-quality ZCL messages for fuzzing the Zigbee protocol due to its significance in performing device functionalities. As we will show, this topic is challenging enough in its own right. Given the protocol specification, our framework can be extended to test other types of Zigbee messages with small engineering efforts.

4 DESIGN AND IMPLEMENTATION

In this section, we present the details of the design and implementation of Z-Fuzzer. We first discuss the challenges of Z-Fuzzer design and our solutions. Next, we introduce Z-Fuzzer's protocol fuzzing algorithm. Then, we present detailed implementations of core components in Z-Fuzzer to address the challenges.

4.1 Challenges in Z-Fuzzer Design

The goal of Z-Fuzzer is to detect vulnerabilities in the Zigbee protocol implementations without the real embedded devices; that is, simulating the execution of the Zigbee protocol in a proper software environment. Most existing IoT firmware simulation applications encounter obstacles to execute the Zigbee protocol due to the diverse underlying hardware and system configurations. The Zigbee protocol interacts with the events triggered by peripheral interrupts varying in different embedded devices. Unfortunately, existing embedded simulators have insufficient knowledge to simulate all of the peripheral interrupts. Besides, the Zigbee protocol is usually executed in a baremetal embedded device. The system can be customized based on particular embedded devices required by manufacturers that are not supported by most existing simulators. Therefore, we need to develop a proper software execution environment to simulate the peripheral interrupts without considering the underlying hardware of specific embedded devices.

Moreover, we design our framework based on grammar-based fuzzing with block-based representation that has been widely used in existing protocol fuzzing frameworks [2, 13, 21, 26]. This approach aims to construct test messages, which satisfy the protocol frame format requirements. However, it has a limitation on the quality of test inputs. It does not prioritize test cases with execution feedback for further fuzzing, which could cover the target program's more execution paths. To effectively detect vulnerabilities in the protocol implementations, we need to consider such feedback from the protocol execution and generate more valuable test inputs.

Solutions. To tackle these challenges, we design Z-Fuzzer with two main components: a test harness and a mutation engine. The test harness consists of an execution engine to run the Zigbee protocol stack with the generated test cases in a simulator and a coverage report parser to calculate cumulative coverage information. We leverage the coverage feedback to retain the interesting test cases for further fuzzing. Additionally, we develop a proxy server in the execution engine to bridge the communication between the simulator and the mutation engine without forming an entire Zigbee network.

4.2 Protocol Fuzzing Algorithm

The fuzzing engine of Z-Fuzzer adopts the grammar-based fuzzing using the block-based protocol representation. The overall fuzzing process is displayed in Algorithm 1.

With a message format script, Z-Fuzzer constructs a list of *Blocks* containing all message fields' representations with their constraints (line 2). Initially, the fields are selected from this list to generate a test case (line 13). We now use an additional list of *top_rated* to record favored test cases that increase code coverage in previous executions. If a favored test case is waiting to be mutated, we prioritize the favored test case for the following mutations (line 7). The selected favored test case is the one that has covered the most number of edges in the previous executions.

The message fields that are selected to generate a test case are mutated according to their selection sequence. When a favored test case is selected, the interesting values in this test case that result in coverage increment are retained. Z-Fuzzer then mutates other field values in the test case in sequential order of their placements during the initialization phase. If a message field is defined with user-specific values, Z-Fuzzer sequentially selects these values for mutation. Otherwise, the fuzzer mutates it with the pre-defined fuzzing dictionary. If all of the message fields of a favored test case are completely mutated, we label the favored test case as *was_fuzzed* and remove it from *top_rated* list (lines 9 - 10). Z-Fuzzer completes the entire fuzzing process when no favored test

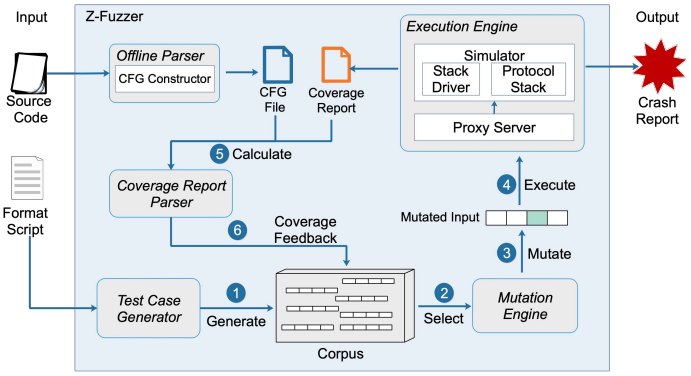


Figure 4: The workflow of Z-Fuzzer framework.

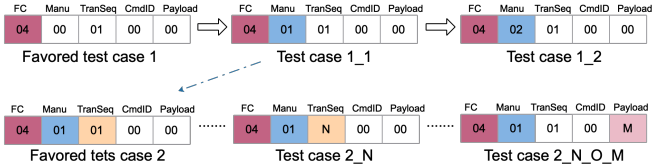


Figure 5: Example of mutation on a favored test case

cases are pending in *top_rated*, and all of the message fields in *Blocks* have been fuzzed.

A test case is evaluated based on code coverage, including *line coverage* and *control-flow edge coverage*. If the test case leads to the code coverage improvement, we save it in the *top_rated* list with the associated interesting values that increase coverage for future mutations (line 18). Otherwise, the test case is ignored. Z-Fuzzer also monitors the execution results and records the test cases that result in an execution error.

4.3 Implementation Details

Figure 4 presents the workflow of Z-Fuzzer framework. It consists of five components: *an offline parser*, *a test case generator*, *a mutation engine*, *an execution engine*, and *a coverage report parser*.

4.3.1 ***Test Case Generation and Mutation.*** We use Zigbee Cluster Library (ZCL) as a case study to demonstrate our framework. The format script represents ZCL message format defined in the Zigbee protocol specification, as displayed in Figure 2a.

All of the message fields of a ZCL frame are represented as primitive data, e.g., bit, byte, integer, string, or random data, in the format script. Some message fields are defined without user-specific types and values. We represent such fields as string primitive data, e.g., a variant attribute data field in the ZCL payload. For other message fields, we represent them based on their defined length and values, such as bit, byte, and word. All of the representations are saved in a list of primitive data. The test case generator then constructs a test case by selecting corresponding primitive data from the list based on the format definition and the constraints (❶ in Figure 4).

If favored test cases are pending for mutation, Z-Fuzzer selects one for the following fuzzing; otherwise, it selects a test case that is generated with the primitive data list (❷ in Figure 4). The selected favored test case has covered the most number of edges in a previous execution and has not been fully mutated. Here an edge is a connection between two basic blocks in a control flow graph (CFG) of the target program. We add a flag `skip_mutation` to the primitive data in the favored test case in which the interesting value increases the code coverage. With this flag, the primitive data will be retained during the following mutation process.

All of the message fields selected to generate a test case are mutated according to their selection sequence (❸ in Figure 4). When a favored test case is selected, the mutation engine will skip the mutation of the interesting values if the flag `skip_mutation` is present. Moreover, other primitive data representing the following fields are mutated in sequential order. Z-Fuzzer fuzzes primitive data assembling a typical test case based on the primitive data's selection order. If a user defines a message field with a list of possible values in the format script, Z-Fuzzer will sequentially select these values for mutation. Otherwise, the mutation engine mutates it with the pre-defined fuzzing library. The favored test case is removed from the corpus when all its message fields are entirely mutated. Z-Fuzzer completes the entire fuzzing process when no favored test cases are pending, and all of the test cases in the corpus have been fuzzed. The mutated input is then sent to the execution engine for testing at runtime (❹ in Figure 4).

Example. Figure 5 shows an example to explain the mutation of the favored test cases. Suppose the favored test case 1 is generated when we fuzz the field *FC* to the value `04`. The test case 1_1 is then generated based on this favored test case, which results in new code coverage. It also exercises more edges than the favored test case 1 and therefore becomes the new favored test case 2. Both interesting values `04` and `01` are recorded. We now fuzz the favored test case 2 on its following fields: *TransSeq*, *ComdID*, and *Payload*. Assume *TranSeq* has N possible values, *CmdID* has O values, and *Payload* has M values in their fuzzing libraries. We will generate $(N + O + M)$ new test cases in total because we mutate a single primitive data in each fuzzing iteration. The mutation of favored test case 2 is regarded as completed once all of those values have been rendered. If no more favored test cases are better than the favored test case 1, we resume its previous mutation process to continue generating test case 1_2 rather than starting from scratch. This process is repeated until all of the message fields are fuzzed.

4.3.2 ***Execution Engine.*** The execution engine is responsible for executing the Zigbee protocol stack with the test cases, consisting of a local proxy server and a simulator. The local proxy server is used to bridge the communication between the mutation engine and the simulator through a socket connection. It also saves the received message in a file for later processing by the protocol stack. We also develop a stack driver to initialize proper system configuration based on the source code of target protocol implementation. We compile the driver with the protocol stack as a single binary file and execute it in the simulator.

Embedded Device Simulator. We utilize the simulator from IAR Embedded Workbench [47] to fully simulate a physical embedded device, which supports different microcontroller architectures. We choose the ARM version since most IoT devices are built on this architecture. The IAR Workbench contains a development toolchain, particularly for IoT devices, including a specific compiler, linker, debugger, and simulator. Currently, the IAR Workbench for ARM

Listing 1: Example of interrupt setting in a macro file and an interrupt handler in the stack driver.

```
1  /* Interrupt Settings in Macro File*/
2  execUserSetup(){
3    //Read the incoming message from the file
4    _fileHandle __openFile("file\\location","r");
5    //Set up interrupt
6    _interruptID __orderInterrupt("UARTR_VECTOR"
         ,100000,60000,0,1,0,100);
7    //Set up the immediate breakpoint
8   _breakID __setSimBreak("SBUF","R","Access()");
9  }
10
11 Access(){
12   _var _msg;
13   if(__readFile(_fileHandle,&_msg)==0){
14     SBUF=_seedData;
15   }
16 }
17
18 //The interrupt handler in the stack driver.
19 #pragma vector=UARTR_VECTOR
20 __interrupt __root void UartReceiveHandler(void){
21   uint32 data;
22   //Save the value from the serial data buffer
23   data=SBUF;
24   ....
25 }
```

architecture supports 50 different ARM CPUs and hundreds of devices from 42 IoT manufacturers [47], which are not supported by a generic simulator such as QEMU. Most embedded devices required by different Zigbee protocol vendors for their implementations are supported in the IAR device list. We also observe that IAR provides diverse device-specific description files, including memory layout, hardware, and peripheral interrupts. We can simulate the embedded device to execute the Zigbee protocol with the pre-defined device description files without considering the underlying hardware design.

Before executing the Zigbee protocol stack, we first build the stack driver with the protocol stack implementation as a single binary file using the IAR compiler and the linker. The IAR C-SPY Debugger communicates with the simulator through a built-in simulator driver [48]. The IAR Workbench also defines various flash loader configurations to download the executable file for all of the supported embedded devices. According to the device description file and the flash loader configuration, the simulator loads the binary file to the corresponding RAM location for execution. The proxy server invokes the C-SPY debugger as a child process to run the Zigbee protocol stack. Additionally, the C-SPY debugger also provides several plugin modules, such as a coverage report and call stack, which we can leverage to guide our fuzzing process.

Stack Driver. The Zigbee protocol is usually executed in an environment that handles events triggered by peripheral interrupts. Though the execution environment can be customized by different protocol vendors, some system properties defined in the protocol specification are mandatory for all implementations. We analyze the sample project provided in the source code of the target protocol implementation. Then we develop a stack driver to initialize the protocol stack system, including memory initialization and basic functionalities of a simulated embedded device. The stack driver then invokes the target protocol implementation with the received message for execution.

In practice, the Zigbee protocol handles the system events when an on-chip communication peripheral interrupt (e.g., UART) is triggered. Hence, we also develop an interrupt handler in the stack driver to simulate the UART interrupt by reading the incoming message from a file using the C-SPY Macro System in conjunction with immediate breakpoints [48]. We set up a repeatable interrupt and an immediate read breakpoint in the macro file according to the device description files. Whenever the interrupt is triggered, the breakpoint temporarily suspends the execution and reads a value from the file, storing an incoming message from the proxy server. The interrupt will be disabled if no values are available in the file. Note that different devices may configure a different register for the interrupt; Z-Fuzzer can set the correct register in the handler based on the device description file.

We present an example of an interrupt setting in the macro file and the interrupt handler in Listing 1. This example simulates the UART interrupt on an embedded device, CC2538, a popular device for IoT application development. The function *execUserSetup()* is a built-in function in the Macro System that is called when the system starts up (line 2). Inside this function, we set a file handler to read the incoming message (line 4), a UART interrupt with the function *__orderInterrupt()* (line 6), and an immediate read breakpoint with the function *__setSimBreak()* (line 8). The interrupt will be activated after 100000 system cycles and repeat every 60000 cycles. When the interrupt is triggered, the immediate breakpoint is enabled on *SBUF*, which is a data buffer to save the data received from UART. Rather than collecting data from the actual peripheral device, we simulate the operation by reading the incoming message from the saved file by the proxy server (line 11-16). Besides, we define the interrupt handler in the stack driver with the keyword *vector=UARTR_VECTOR*, which is the same interrupt variable configured in the macro file (see lines 6 and 19). The handler can directly access the UART's data buffer (*SBUF*) to read the data and save it to a variable for further use. In practice, the name *UARTR_VECTOR* of the UART peripheral device and its data buffer *SBUF* will be configured differently on various embedded devices.

4.3.3 ***Coverage Report Analysis.*** We evaluate test cases in terms of line coverage and edge coverage. A test case is saved as a favored test case if it increases code coverage. The C-SPY debugger can generate a coverage report for the current execution. Unfortunately, the coverage report does not provide adequate information. Thus, we developed an *offline parser* and a *coverage report parser* to calculate cumulative coverage results.

Offline Parser. The offline parser is a static code analysis tool to generate a control flow graph (CFG) data from the protocol implementation's source code. It is used later by the coverage report parser. The offline parse only executes once before the entire fuzzing iterations. The coverage report only records the uncovered statements in functions in a single execution, which is insufficient for calculating cumulative line coverage and edge coverage. Hence, we leverage the CFG information, including statements, basic blocks, and branches of every function, to calculate cumulative code coverage. We assign every basic block with a random number with hashing to obtain edge coverage information when analyzing the CFG information. The random number acts as the label of every basic block. These analysis results are saved as formatted data in a

Table 1: Total number of crashes and unique vulnerabilities detected by BooFooz, Peach and Z-Fuzzer.

Fuzzer	Total # of Crashes (median)	Unique Vulnerabilities
BooFuzz	62	2
Peach	3	3
Z-Fuzzer	223	6

file for the coverage report parser to compute detailed line coverage and edge coverage.

Coverage Report Parser. The coverage report parser analyzes the coverage report and the CFG file to calculate cumulative line coverage and edge coverage (❺ in Figure 4). We use two lists, *line_hits* and *edge_hits*, to record lines of code and edges that have been covered in the previous executions. The value of *line_hits*[*i*] means the total executed times of the statement in line *i*. The value of *edge_hits*[*i*] is the total accessed times of the *i*th edge, in which *i* is calculated by the *XOR* operation on the labels of the source and destination basic blocks. A label is a random number assigned by the offline parser.

The parser firstly scans a coverage report to collect functions that have been accessed in the last execution. The uncovered lines of code in the accessed function are saved into a list. All the statements contained by a basic block are also extracted from the CFG file to a list. Then we compare these two lists to check whether the current basic block is covered in the last execution. If a basic block is accessed, we also record the covered edge between the block and its source block to the list *edge_hits*. After completing parsing the entire coverage report, we calculate the non-zero values in the list *line_hits* and the list *edge_hits* to find out if any new lines and edges have been added. If so, we consider the current test case as a favored one and put it in the pending favored queue for a further mutation (❻ in Figure 4).

5 EVALUATION

In this section, we evaluate Z-Fuzzer through multiple experiments. The experiments are designed to answer the following research questions:

- **RQ1**: Can Z-Fuzzer detect more vulnerabilities in comparison with the state-of-the-art fuzzers? (Section 5.1)
- **RQ2**: Can Z-Fuzzer achieve higher coverage rate in comparison with the state-of-the-art fuzzers? (Section 5.2)

The target of the protocol fuzzing approach is to generate more high-quality test inputs that conform to the protocol frame format. Thus, we demonstrate the novelty of Z-Fuzzer in comparison with two baseline protocol fuzzers, BooFuzz [38] and Peach [49]. BooFuzz is the successor of industry-standard protocol fuzzer Sulley [21], and Peach fuzzer is a model-based commercial fuzzer. Both of them have been widely used in existing research papers [51, 55]. BooFuzz and Peach initially do not target IoT wireless protocols like the Zigbee protocol. Thus, we incorporated them into our simulation platform to communicate with the Zigbee protocol. We specifically compared the number of vulnerabilities and code coverage exposed in 24-hour fuzzing experiments. All of our experiments were performed on a machine with 8 cores (Intel® Core™ i7-6700 CPU @ 3.40GHz) and 32 GB memory running the Windows 10 Pro operating system and IAR Embedded Workbench for ARM 8.3. We tested a widespread Zigbee protocol implementation, Z-Stack [29], which has been deployed in the most Zigbee compliant platforms [9]. Z-Stack is developed by Texas Instruments with various sample project codebases and its source code is available.

5.1 Vulnerability Detection Capability

To answer **RQ1**, we measure the number of detected crashes and the number of unique vulnerabilities discovered by all fuzzers. We repeated experiments 10 times on fuzzers and present the result in Table 1.

Unique Vulnerabilities. We leveraged information in *call stack* to de-duplicate detected crashes. The simulator returns a call stack trace for a memory crash, which contains the executed functions, the line number of particular statements in the functions, and the memory address of the statement. We hashed the memory address and its function name and line number as an identifier of a detected crash. Stack hashing may result in bug overcounting [32]. In our case, we manually check function call trace in the source code for every unique vulnerability to avoid the overcounting issue. The experiment result is displayed in Table 1; it indicates that Z-Fuzzer can discover more crashes and unique vulnerabilities than the other two fuzzers. We also cross-checked all detected vulnerabilities. Only one vulnerability can be reproduced with the test cases generated by BooFuzz. All of the vulnerabilities can be reproduced with test cases generated by Peach fuzzer and Z-Fuzzer. We reported all detected vulnerabilities to the CVE database and vendors, and three of them have been assigned CVE IDs with high CVSS scores (7.5~8.2) [20].

Test Cases vs. Vulnerabilities. We measure the number of detected vulnerabilities over the generated test cases for BooFuzz, Peach fuzzer, and Z-Fuzzer, as shown in Table 2. The vulnerability ID in the table is used to identify each vulnerability in other experiments, which does not present the detection order during the experiment. The result indicates that Z-Fuzzer can generate more test cases and detect more vulnerabilities in the protocol implementation. We noticed that only CVE-2020-27892 is detected in every fuzzing round over ten times by all fuzzers. Other bugs are discovered in some particular rounds. All fuzzers can detect CVE-2020-27891 and CVE-2020-27892, while Z-Fuzzer can generate more unique test cases for detection. BooFuzz failed to discover other 4 vulnerabilities, especially the function *zclParseInReadRspCmd* and *zclParseInReportCmd* found by Z-Fuzzer with specific test cases. Compared to BooFuzz, Peach fuzzer can instead discover the vulnerable function *zclProcessInWriteCmd* with a particular test case. According to our analysis of these vulnerabilities, most crashes occurred in a deeper location of vulnerable functions caused by some long malformed string values at the end of the message payload field. Before processing these values, the function performs several condition checks on other preceding primitive data. With the coverage feedback, some interesting values are retained to generate specific test cases to satisfy such condition checks.

Coverage vs. Vulnerabilities. We also analyze the relationship between line coverage and the number of detected vulnerabilities. Figure 6 presents the max cumulative number of vulnerabilities detected over line coverage. The symbols are the vulnerability

Table 2: Summary of new vulnerabilities detected by BooFuzz, Peach and Z-Fuzzer.

#	Vulnerabilites	Severity	Total # of Test Cases Triggering a Vulnerability		
			BooFuzz	Peach	Z-Fuzzer
1	CVE-2020-27891 (High 7.5)	Improper Input Validation	57	1	10
2	CVE-2020-27892 (High 7.5)	Improper Memory Allocation	10	4	219
3	CVE-2020-27890 (High 8.2)	Improper Input Validation	-	-	96
4	zclParseInReportCmd	Out-of-bound read	-	-	2
5	zclParseInReadRspCmd	Out-of-bound read	-	-	3
6	zclProcessInWriteCmd	Null pointer reference	-	1	231

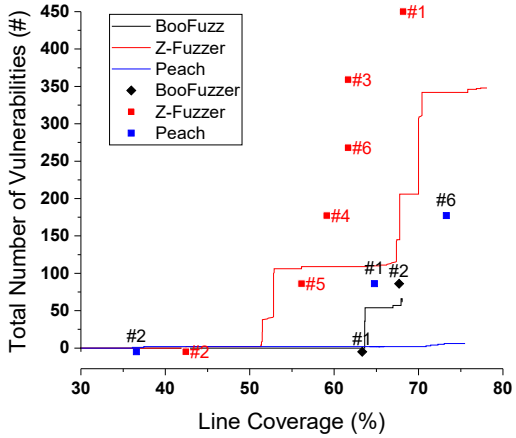


Figure 6: The relationship between line coverage and the number of detected protocol crashes in 10 runs. X-axis: line coverage on average, Y-axis: the max cumulative number of vulnerabilities. The symbols are the vulnerability ID displayed in Table 2's first column; they represent the minimum line coverage when the corresponding vulnerability is detected.

identifiers displayed in Table 2 and represent the minimum line coverage that detects the corresponding vulnerability. We can see that Z-Fuzzer can detect more vulnerabilities by exercising fewer lines of source code. Peach and Z-Fuzzer first detected CVE-2020-27892 at the earlier fuzzing stage, while BooFuzz found the same vulnerability at the end of the fuzzing process. We notice that some crashes are caused by some abnormal values of the message payload field with a particular value of a preceding field, which may exercise new code. BooFuzz and Peach fuzzer fails to generate such test messages since they consider the message payload field and its preceding field independent during fuzzing. The particular value of the preceding field is not retained when the message payload field is mutated. However, Z-Fuzzer can generate such a test case once the line coverage is changed. Therefore, Z-Fuzzer improves the effectiveness and efficiency of vulnerability discovery by boosting code coverage.

Vulnerabilities on Real Embedded Devices. We also verify the detected vulnerabilities on real embedded devices. We used two Texas Instruments CC2538 devices with the SmartRF06 Evaluation

Figure 7: Message transmission on TI CC2538 with the evaluation board. Z-Stack is deployed to CC2538 for execution. Evaluation board powers CC2538 and displays the debugging information on the LED. We use one as a coordinator and another as an end device, and the ZCL messages are transmitted between them.

Board to form a real IoT network. TI CC2538 is a wireless microcontroller System-on-Chip (SoC) for high-performance ZigBee applications [30] and has been widely adopted in the IoT market.

As shown in Figure 7, one device acts as a coordinator that sends the crash messages we found in the simulator; another acts as an end device that receives the coordinator's messages. We added debugging information in the test harness to print device status on the LED display. The entire protocol stack with the test harness is built as a single binary file and flashed to CC2538. The coordinator initiates the network formation, and the end device joins the network.

We executed test cases that triggered vulnerabilities on the physical devices. Table 2 shows that all fuzzers can detect vulnerabilities in the function *zcl_HandleExternal* and the function *zclParseInDiscCmdsRspCmd* in the simulation environment. However, the vulnerability in the function *zcl_HandleExternal* cannot be reproduced with the test cases generated by BooFuzz and Peach. Instead, we could detect those two crashes with the test cases generated by Z-Fuzzer on the real device. The embedded device was frozen when processing the received crashing messages. In addition to these two vulnerabilities, we can also verify the vulnerable function *zclParseInWriteCmd* with the test cases generated by Z-Fuzzer. We notice that memory corruption occurred when the device processes the received messages. The Z-Stack implementation has captured the crash; however, it does not perform further operations and report

Listing 2: Source code of CVE-2020-27892

```
1  static void *zclParseInDiscCmdsRspCmd(
       zclParseCmd_t *pCmd)
2  {  .....
3    pDiscoverRspCmd=(zclDiscoverCmdsCmdRsp_t*)
4    zcl_mem_alloc(sizeof(zclDiscoverCmdsCmdRsp_t) +
5    (numCmds* sizeof(uint8)));
6    if(pDiscoverRspCmd != NULL)
7    {  ......
8      for(i = 0;i < numCmds;i++)
9      {
10       pDiscoverRspCmd->pCmdID[i] = *pBuf++;
11     }
12   }
13   return ( (void *)pDiscoverRspCmd );
14 }
```

the crash. From the user's perspective, the processing is successful since a success status code is returned to the end device. Nevertheless, the attribute value is not updated. We have reported all of the six detected vulnerabilities to the protocol vendor, Texas Instruments. Three vulnerabilities have been confirmed at the time of writing, and others are still under review.

Case Study. We use CVE-2020-27892 as a case study to explain more details of our observations. This vulnerability is triggered by two specific valid command identifiers in the ZCL header. When the command identifier is set to `0x12` or `0x14`, which indicates a *Discover Commands Received Response* message or a *Discover Commands Generated Response* message, it crashes the protocol stack when parsing payload values of such message. The end device is frozen and fails to respond to any operations unless we restart the board.

We examined this crash on both the simulator and the real device. The root cause is an incorrect memory allocation for a structure variable. The source code is showing in Listing 2. The struct variable `pDiscoverRspCmd` is a pointer that contains an attribute `pCmdID` pointing to an array. In standard C programs, `pCmdID` is assigned to a valid memory address when the system allocates memory space for `pDiscoverRspCmd`. As the code shown in line 4, Z-Stack calls its memory allocation method rather than using the C standard API. However, the self-implemented memory allocation method fails to assign a valid address to `pCmdID`. Suppose the memory address of `pDiscoverRspCmd` is `0x20005B80` and the size of this structure type is 4 bytes and `numCmds` equals 1, then `pCmdID` should point to the address `0x20005B85`. In practice, it points to the content of that address, which is `0xCDCDCDCD` and an invalid memory address. Thus, an out-of-bounds write vulnerability is triggered when code in line 10 is executed. Similar memory issues like memory copy also lead to other vulnerabilities.

We observe that most protocol vendors develop their customized APIs to replace the standard functions in the C library. The main reason is that an embedded device has limited memory resources and computing power, which is hard to support all C standard API libraries like PC software. Besides the bugs in the protocol implementation itself, this customization may bring potential security risks. Currently, the protocol vendors bear responsibility for the vulnerabilities of the Zigbee protocol. The mitigation of security problems entirely depends on whether the vendors are proactive or not to the reported issues [10]. The IoT application developers may not be aware of those potential issues until they complete the entire production. This observation also motivates us to propose Z-Fuzzer for developers to acknowledge the Zigbee protocol stack's potential issues at the earlier development stage; thus, they can take corresponding actions to avoid such problems without waiting for the protocol vendor's feedback.

Table 3: Evaluation results on Z-stack in 10 runs. We report the line coverage and edge coverage on average.

Fuzzer	Total # of Unique Test Cases	Line Coverage		Edge Coverage	
		total	%	total	%
BooFuzz	16,756	912	73.80%	680	73.82%
Peach	18,271	850	68.71%	628	67.58%
Z-Fuzzer	61,386	971	78.52%	769	82.30%

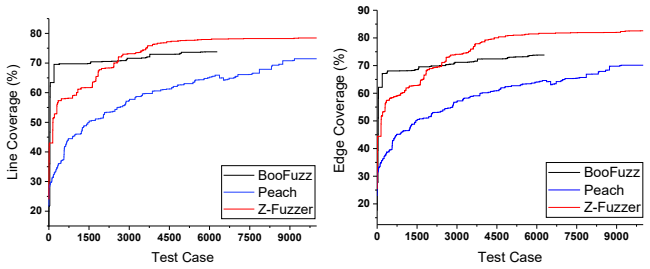
5.2 Code Coverage

To answer **RQ2**, we examined the ability of fuzzers to improve code coverage in 24h fuzzing, which is a widely accepted and evaluated metric in existing research [32]. We performed a set of experiments on each fuzzer to observe their line coverage and edge coverage variation over time. Here an edge is a connection between two basic blocks in CFG. We inputted the same protocol frame format script to all fuzzers. Therefore, their fuzzing process was initialized with the same valid protocol frame. Given the frame format script, the fuzzers generate test cases with the user-specific or pre-defined fuzzing dictionary, for which the total number of test cases is finite. Results are presented in Table 3. From the results, we observe that Z-Fuzzer is significantly more effective than BooFuzz and Peach.

We first analyze the uniqueness of test cases generated by three fuzzers. As shown in Table 3, Z-Fuzzer can generate 6 times more unique test cases than the other two fuzzers. Moreover, according to the Zigbee protocol specification, we categorize test cases by the field *command identifier* in the ZCL header to distinguish the difference among fuzzers on test case generation. Z-Fuzzer generated 308 different types of test cases in total, in which 35 of those types can be generated by BooFuzz and Peach. In addition, many test cases result in coverage increments, and therefore they are retained as favored test cases for further mutation. For example, Z-Fuzzer can generate the message in Figure 3 during the fuzzing process when the value `08` of the field frame control is retained.

Moreover, we measure the code coverage of Z-Fuzzer in comparison with BooFuzz and Peach. Without our Zigbee protocol simulation platform, BooFuzz and Peach cannot directly test Z-Stack implementation. Therefore, we replaced our mutation engine with the other two fuzzers' fuzzing engines to compare their performance. The experiment result is presented in Table 3 and Figure 8. Table 3 indicates that Z-Fuzzer can achieve higher line coverage and edge coverage. Currently, we focus on generating high-quality test cases that satisfy the message format of the Zigbee protocol specification. Therefore, we cannot cover exception handling code and reach full code coverage. As Section 5.1 indicates, Z-Fuzzer can discover more vulnerabilities than the other two fuzzers though it does not achieve full code coverage.

From Figure 8, we can see that BooFuzz and Z-Fuzzer proliferated at a very early phase. As the Zigbee protocol performs several checks on the ZCL header first when processing a message, minor



(a) Line Coverage **(b) Edge Coverage**

Figure 8: Line coverage and edge coverage achieved by fuzzers over 10 runs. The X-axis represents the median number of test cases. The Y-axis represents the percentage of line coverage and edge coverage on average.

changes in the header can lead to a significant difference in executed code and path. Both of the two fuzzers start fuzzing from the field *Frame Control* (the first field in the ZCL header shown in Figure 2a). It is the reason that code coverage rapidly increased in BooFuzz and Z-Fuzzer at the early phase. Instead, Peach randomly mutated a message field, and therefore its code coverage increased slowly. Even though BooFuzz achieved its maximum code coverage with fewer test cases, it terminated the fuzzing process after generating about 6, 200 test cases. BooFuzz uses fewer values for each primitive data to prevent an inevitable combinatorial explosion in the number of possible mutation values. These values are specified by the protocol specification or a pre-defined fuzzing dictionary of values. All values are static over the fuzzing time. Thus, BooFuzz generated fewer test cases and terminated the fuzzing process earlier than the other two fuzzers. For better result presentation, we plot the coverage trend of the first 10, 000 test cases generation in Figure 8. On the other hand, Z-Fuzzer and Peach fuzzer kept executing more code and edges and generated more test cases. We also examine the differences in accessed code and edges. Z-Fuzzer can exercise more different code and edges that BooFuzz or Peach does not execute.

In summary, Z-Fuzzer achieves a higher code coverage rate than BooFuzz and Peach with the coverage-guided test case generation. The interesting values are recorded with the coverage feedback and guide the fuzzing process to generate more high-quality test cases to access more in-depth code. We observe that many functions in ZCL process the message payload value for the upper-level application object. They could require a test case to satisfy some particular condition checks to execute more in-depth code in those functions. During BooFuzz's and Peach's mutation process, the values of specific message fields, which may satisfy such a dependency constraint, are neglected during the fuzzing. In contrast, Z-Fuzzer can infer such a correlation with the runtime coverage feedback. The current mutant primitive data and all of the preceding fields are retained for further fuzzing, satisfying those particular conditions and covering more code and edges.

6 CONCLUSION AND FUTURE WORK

We have presented the first device-agnostic fuzzing framework, *Z-Fuzzer*, to detect security vulnerabilities in Zigbee protocol implementations. Z-Fuzzer integrates a software simulator to simulate real IoT devices combining the pre-defined hardware interrupts and peripheral configurations. We also develop a test harness to provide a proper execution environment for the Zigbee protocol stack, including a proxy server facilitating the communication between the simulator and the mutation engine. Z-Fuzzer outperforms the state-of-the-art work by detecting more deep vulnerabilities with fewer test cases. We have identified six unique vulnerabilities, and three of them have been assigned CVE IDs with high-severity scores.

Currently, we already integrated the fuzzing engine of BooFuzz and Peach on top of our simulation environment. We plan to integrate our simulation environment in our future work, including the proxy server with other state-of-art fuzzers; they can transmit their test cases through the Internet to Zigbee protocol execution. Additionally, we can extend some embedded fuzzers like HALucinator [16] with our test case generation engine. Then they can be aware of the Zigbee protocol frame format and test Zigbee protocol implementations.

ACKNOWLEDGMENTS

We would like to thank our shepherd Kevin Butler and the anonymous paper reviewers for their helpful feedback. This work is partly supported by a research grant (70NANB18H207) from Information Technology Lab of National Institute of Standards and Technology (NIST) and the National Science Foundation (NSF) under grant CNS-1850434.

REFERENCES

[1] Dave Aitel. 2002. The Advantages of Block-based Protocol Analysis for Security Testing. *Immunity Inc., February* 105 (2002), 106.
[2] Dave Aitel. 2002. An Introduction to SPIKE, the Fuzzer Creation Kit. BlackHat USA.
[3] Dimitrios-Georgios Akestoridis, Madhumitha Harishankar, Michael Weber, and Patrick Tague. 2020. Zigator: Analyzing the Security of Zigbee-enabled Smart Homes. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'20)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3395351.3399363
[4] Zigbee Alliance. 2015. Zigbee. https://zigbeealliance.org/.
[5] Zigbee Alliance. 2016. Zigbee Alliance Accelerates IoT Unification with 20 Zigbee 3.0 Platform Certifications. https://bit.ly/2Cxo62D.
[6] Zigbee Alliance. August 5, 2015. Zigbee Specification. https://bit.ly/3hKpKMW.
[7] Zigbee Alliance. Jan 14, 2016. Zigbee Cluster Library Specification. https://zigbeealliance.org/wp-content/uploads/2019/12/07-5123-06-zigbee-cluster-library-specification.pdf.
[8] Zigbee Alliance. January 3, 2017. The Zigbee Alliance to Unveil Universal Language for the IoT from CES 2017 Making it Possible for Smart Objects to Work Together on Any Network. https://bit.ly/2ZTcgbD.
[9] Zigbee Alliance. [online]. Zigbee Compliant Platforms. https://bit.ly/2WLvePC.
[10] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based Iot Deployments. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, Piscataway, NJ, USA, 1362–1380. https://doi.org/10.1109/SP.2019.00013
[11] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, Vancouver, BC, 1093–1110. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis
[12] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. arXiv:2005.11498 [cs.SE]
[13] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward A Stateful Network Protocol Fuzzer. In *Proceedings of the 9th International Conference on Information Security (ISC'06)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 343–358.
[14] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. USENIX Association, Vancouver, BC, 46.

[15] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*. IEEE, Piscataway, NJ, USA, 711–725. https://doi.org/10.1109/SP.2018.00046

[16] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, Berkeley, CA, USA, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[17] Common Vulnerabilities and Exposures. [online]. Zigbee CVEs. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=zigbee.

[18] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. 2014. A Novel Fuzzing Method for Zigbee based on Finite State Machine. *International Journal of Distributed Sensor Networks* 10, 1 (2014), 762891.

[19] Baojiang Cui, Ziyue Wang, Bing Zhao, and Xiaobing Liang. 2016. CG-Fuzzing: A Comprehensive Fuzzy Algorithm for ZigBee. *International Journal of Ad Hoc and Ubiquitous Computing* 23, 3-4 (2016), 203–215.

[20] National Vulnerability Database. 2021. NVD Vulnerability Severity Ratings. https://nvd.nist.gov/vuln-metrics/cvss.

[21] Ganesh Devarajan. 2007. Unraveling SCADA Protocols: Using Sulley Fuzzer. Defon 15 Hacking Conference.

[22] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, Berkeley, CA, USA, 1237–1254. https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[23] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT'20)*. USENIX Association, Berkeley, CA, USA. https://www.usenix.org/conference/woot20/presentation/fioraldi

[24] Drew Gislason. 2008. *Zigbee Wireless Networking, 1st Edition*. Newnes, London, UK.

[25] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[26] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated Network Protocol Fuzzing Framework. *International Journal of Computer Science and Network Security (IJCSNS)* 10, 8 (2010), 239.

[27] Fortune Business Insights. July, 2019. Internet of Things (IoT) Market Analysis. https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307.

[28] Texas Instruments. 2006. Z-Stack 3.0 Developer's Guide. https://bit.ly/2EitbvY.

[29] Texas Instruments. 2018. A fully compliant ZigBee 3.x solution: Z-Stack. http://www.ti.com/tool/Z-STACK.

[30] Texas Instruments. [online]. CC2538. http://www.ti.com/product/CC2538.

[31] IoTcube. 2021. Blackbox-testing zfuzz. https://iotcube.net/userguide/manual/zfuzz.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[33] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation. In *Proceedings of the 57th Annual Design Automation Conference (DAC'20)*. ACM/IEEE, Piscataway, NJ, USA, 1–6. https://doi.org/10.1109/DAC18072.2020.9218603

[34] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. BaseSAFE: Baseband Sanitized Fuzzing through Emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'20)*. Association for Computing Machinery, New York, NY, USA, 122–132. https://doi.org/10.1145/3395351.3399360

[35] John Mikulskis, Johannes K Becker, Stefan Gvozdenovic, and David Starobinski. 2019. Snout - An Extensible IoT Pen-Testing Tool. Poster presented at: the 26th ACM SIGSAC Conference on Computer and Communications Security.

[36] Philipp Morgner, Stephan Mattejat, Zinaida Benenson, Christian Müller, and Frederik Armknecht. 2017. Insecure to the Touch: Attacking ZigBee 3.0 via Touchlink Commissioning. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'17)*. Association for Computing Machinery, New York, NY, USA, 230–240.

[37] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *In Workshop on Binary Analysis Research.*, Vol. 18. Network and Distributed System Security Symposium, San Diego, CA, USA, 1–11.

[38] Joshua Pereyda. 2020. Boofuzz: Network Protocol Fuzzing for Humans. https://boofuzz.readthedocs.io/en/latest/.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST'20)*. IEEE, Piscataway, NJ, USA, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[40] QEMU. 2018. QEMU ARM Guest Support. https://wiki.qemu.org/Documentation/Platforms/ARM#Supported_in_qemu-system-arm.

[41] QEMU. [online]. QEMU Wiki. https://wiki.qemu.org/Documentation/Platforms.

[42] Sandler Research. September 13, 2016. ZigBee Home Automation Market to Grow at 26% CAGR to 2020. https://prn.to/3jLYLmk.

[43] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. 2017. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Proceedings ot the 38th IEEE Symposium on Security and Privacy (S&P'17)*. IEEE, Piscataway, NJ, USA, 195–212. https://doi.org/10.1109/SP.2017.14

[44] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. 2020. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, Berkeley, CA, USA, 19–36. https://www.usenix.org/conference/usenixsecurity20/presentation/ruge

[45] Beyond Security. 2021. Dynamic, Black Box Testing on the ZigBee. https://beyondsecurity.com/dynamic-fuzzing-testing-zigbee.html?cn-reloaded=1.

[46] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, London, UK.

[47] IAR System. [online]. IAR Embedded Workbench. https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/.

[48] IAR Systems. [online]. C-SPY Debugging Guide for Amr cores. https://bit.ly/2P1a7ob.

[49] Peach Tech. [online]. Peach Fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/.

[50] Common Vulnerabilities and Exposures. 2020. CVE-2020-6007. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6007.

[51] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. Association for Computing Machinery, New York, NY, USA, 2525–2527. https://doi.org/10.1145/3319535.3363247

[52] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *Proceedings of the 21st Network and Distributed Systems Security Symposium (NDSS'14)*. Network and Distributed Systems Security Symposium, San Diego, CA, USA.

[53] Michal Zalewski. 2015. American fuzzy lop.

[54] Michal Zalewski. 2015. Technical whitepaper for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.

[55] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. 2019. SRFuzzer: An Automatic Fuzzing Framework for Physical SOHO Router Devices to Discover Multi-Type Vulnerabilities. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. Association for Computing Machinery, New York, NY, USA, 544–556. https://doi.org/10.1145/3359789.3359826

[56] 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 *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*. USENIX Association, Santa Clara, CA, USA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng