Fuzz Testing of Zigbee Protocol Implementations

Mengfei Ren

FUZZ TESTING OF ZIGBEE PROTOCOL IMPLEMENTATIONS

by

MENGFEI REN

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2023

To my dear mother, stepfather and stepsister Onna

who support me to explore this wonderful world.

Acknowledgements

Reflecting on the past 25 years, it feels as though my journey began just yesterday when I stepped foot into elementary school. This incredible voyage has encompassed the pursuit of Bachelor's, Master's, and Ph.D. degrees, presenting numerous challenges along the way. In particular, the decision to return to academia for a Ph.D. after spending several years in the industry was a formidable one. I vividly remember the internal struggle I faced in making that pivotal choice. I am profoundly grateful to my beloved parents and grandparents who wholeheartedly supported my aspirations, enabling me to embark on this transformative path of self-discovery. Despite the countless difficulties and obstacles encountered over the past six years, I have no regrets about pursuing advanced education. In its entirety, my Ph.D. journey has been a profoundly fortunate and blessed experience. I have been fortunate to receive invaluable guidance and support from esteemed professors, dedicated teammates, and members of my church community. Their collective influence has brought me to this very moment of writing my dissertation.

First and foremost, I would like to express my deepest gratitude to my supervising professor, Dr. Lei, for his unwavering patience, guidance, encouragement, and profound inspiration throughout the entirety of this dissertation. Without his boundless support, this work would not have come to fruition. Dr. Lei's mentorship has not only fostered my growth as an independent thinker, problem solver, and decision maker but has also been an immense honor to be associated with.

Furthermore, I extend my heartfelt thanks to my co-supervisor, Dr. Ming, for his exceptional patience, guidance, and unwavering encouragement throughout my research journey. Collaborating with him for five years in the capacity of a GTA for the course ”Information Security” has been an enlightening experience. Specifically, his staunch encouragement to pursue a faculty position post-graduation, along with the abundant opportunities and guidance he provided, played an instrumental role in securing a tenure-track faculty job offer. Additionally, I wish to express my gratitude to my esteemed committee members, Dr. Christoph Csallner and Dr. Hao Che, for their profound interest in my research and invaluable feedback.

I would also like to extend my appreciation to all my current and former colleagues from the Software Engineering Lab and Information Security Lab. Dr. Feng Duan, Dr. Huadong Jack Feng, Michael Hou, Xiaolei Ren, Sunny Shree, Qiping Wei, Fadul Sidker, Krishna Khadka, Dr. Haotian Zhang, and Erika Leal have been instrumental in providing support, assistance, and valuable feedback throughout the years.

Lastly, I wish to convey my deepest respect and gratitude to my stepsister Onna and my parents for their unwavering support and selfless sacrifices. They have been my unwavering pillar, enabling me to explore uncharted territories to my heart's content. I express my heartfelt gratitude to my stepsister for her unwavering presence and guidance throughout my journey, particularly for providing solace during the challenging times of homesickness caused by the epidemic, when returning to my home country was not possible during pandemic. Moreover, I am grateful to my mother for her lighthearted conversations that help alleviate anxiety, restlessness, and unhappiness. Similarly, I express my gratitude to my father for his profound insights into life and unyielding support for my decisions. Without their unwavering backing, I have no doubt that it would have been arduous to dedicate myself wholeheartedly to my Ph.D. studies.

July 10, 2023

Abstract

FUZZ TESTING OF ZIGBEE PROTOCOL IMPLEMENTATIONS

MENGFEI REN, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professor: Dr. Yu Lei

In recent years, we have witnessed the increasing of the Internet of Things (IoT) devices deployed by many areas, such as home automation, healthcare, manufacture, and smart vehicle. Among the numerous IoT wireless standards available, Zigbee stands out as one of the most globally popular choices, with major companies like Amazon, Samsung, IKEA, Huawei, and Xiaomi incorporating it into their products. Notably, Zigbee has even been utilized in NASA's Mars mission, where it serves as the communication radio between the flying drone and the Perseverance rover.

However, with the rapid growth of Zigbee's global market presence, the incentive for cybercriminal attacks has also escalated. Recent incidents have highlighted severe vulnerabilities in Zigbee protocol implementations, compromising IoT devices from multiple manufacturers. Consequently, conducting security testing on Zigbee protocol implementations has become an imperative task. Nevertheless, applying existing vulnerability detection techniques like fuzzing and data flow analysis to Zigbee protocols is nontrivial, especially when dealing with vendor-specific requirements and low-level hardware events. Additionally, many existing protocol fuzzing tools lack an appropriate execution environment for Zigbee, as it relies on radio communication rather than internet connectivity.

This dissertation aims to address the aforementioned gaps by proposing comprehensive fuzzing solutions tailored to the security testing of Zigbee protocol implementations. The goal is to assist IoT application manufacturers and protocol vendors in mitigating security risks during their development process. The dissertation makes the following contributions: (i) **Z-Fuzzer**: A device-agnostic fuzzing platform that utilizes code coverage feedback to detect security issues of the Zigbee protocol implementations. It leverages a commercial embedded device simulator with pre-defined peripherals and hardware interrupt setups to interact with the fuzzing engine. It also addresses the communication gap between the fuzzing engine and the simulator to make fuzzing applicable to Zigbee protocol stack. (ii) **TaintBFuzz**: An intelligent Zigbee protocol fuzzing solution via constraint-field dependency inference. It utilizes static taint analysis to infer the dependency between the protocol message field and path constraints, which augment the mutation process during the fuzzing. (iii) **CT-BFuzz**: A fuzzing platform with combinatorial approach of Zigbee protocol implementation. It utilizes static taint analysis and fuzzing to identify important message fields and their representative values for dynamically generating combinatorial testing model. While combinatorial testing helps fuzzing generate cover important combination values of message fields that may explore uncovered execution paths.

This dissertation is presented in a monograph based format and includes three research articles. The first article introduces our work of *Z-Fuzzer* that is the first device-agnostic fuzzing tool making fuzzing applicable to detect security problems of Zigbee protocol implementation. The second article reports the work of *TaintBFuzz* that uses constraint-field dependency inference to augment test input mutation in fuzzing Zigbee protocol implementation. The third article presents *CT-BFuzz* that optimizes the Zigbee protocol fuzzing via combinatorial test generation to generate test cases for efficiently covering combination values of important message fields. The first two papers have been accepted at peer-reviewed venues, while the third one is currently in press.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

The global market has witnessed a rapid surge in the popularity of Internet of Things (IoT) devices. These devices have found applications in diverse fields such as home automation, healthcare, manufacturing, and smart vehicles. According to a recent survey report [1], the global market value of IoT is projected to reach hundreds of millions of dollars by 2023. However, this exponential growth in the market also attracts the attention of cybercriminals who seek to exploit vulnerabilities in IoT devices. Various cyberattacks [2,3,4,5] have been demonstrated, targeting IoT firmware, mobile applications, and network communications.

Among the numerous wireless communication protocols, Zigbee protocol stands out as one of the most widely used. It is employed by millions of devices and favored by renowned companies such as Amazon, Samsung, IKEA, Huawei, and Xiaomi. Notably, Zigbee protocol was even utilized in NASA's Mars mission for communication between the flying drone and Perseverance rover [6]. According to a recent report by the Zigbee Alliance, it is estimated that approximately four billion Zigbee devices will be sold globally by 2023 [7]. The Zigbee protocol transmits data through radio channels rather than the internet, making it closely tied to the hardware configuration of embedded devices.

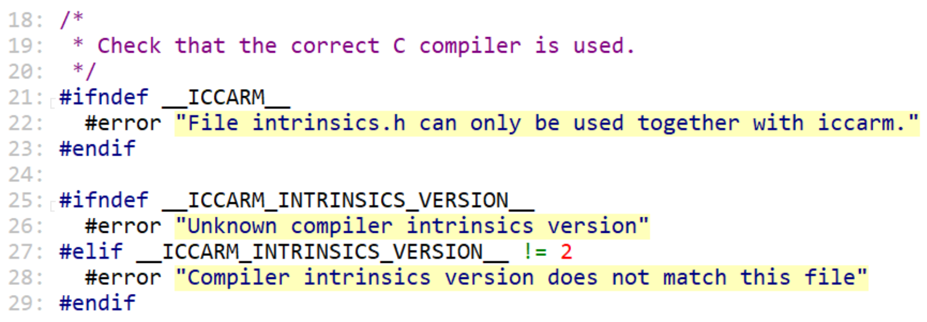
While IoT applications and protocol vendors have implemented security measures based on specification requirements, several research studies [8, 9, 10, 11] have uncovered security vulnerabilities in the Zigbee protocol. Exploiting these vulnerabilities could lead to distributed denial-of-service (DDoS) attacks or remote code execution on Zigbee-based systems such as the Philips lighting system. Despite efforts to address previous security flaws in the latest Zigbee protocol version, it still remains relatively understudied in the research community. Consequently, the detection of security vulnerabilities in Zigbee protocol implementations is crucial and holds significant practical implications.

Fuzz testing [12], also known as fuzzing, is a widely used and effective technique for detecting security vulnerabilities. It involves running the target program with random inputs to identify potential weaknesses. Conventional protocol fuzzing approaches, such as Spike [13], Boofuzz [14], and Peach [15], rely on constructing protocol messages according to the protocol specifications and then randomly mutating them to explore the program's input space. While these approaches generate test inputs that conform to the format requirements of the program, they often struggle to trigger vulnerabilities that lie deep within the program. This limitation arises from a lack of consideration for execution feedback and program structure. In other words, these conventional approaches overlook the valuable information obtained during program execution and fail to leverage the underlying structure of the program. As a result, they may miss certain vulnerabilities that require specific combinations of inputs to manifest.

Coverage-guided grey-box fuzzing (CGF) has demonstrated promising results in bug discovery [16, 17, 18, 19]. AFL [16], a widely used grey-box fuzzing platform, utilizes code-coverage heuristics to guide the generation of test inputs. It achieves this by instrumenting the source code of the software under test, if available, or by executing a closed-source binary file on the QEMU [20] emulation platform to obtain dynamic instrumentation output. Many advanced coverage-guided fuzzing approaches have built upon AFL's solution [21, 22, 23]. Furthermore, in scenarios where the source code of the software under test is unavailable, certain existing fuzzing tools [17, 18, 19, 24, 25, 26] are capable of fuzzing the binary file of the software within a simulation environment.

However, the efficiency of CGF solutions is often hindered by the large search space of inputs. To address this challenge, numerous approaches have been proposed to enhance the efficiency of CGF, with one prominent optimization method being the inference of the relationship between input bytes and path constraints. Existing solutions have found data flow analysis, such as dynamic taint analysis, to be beneficial for such optimizations. VUzzer [27] and GREYONE [28] utilize dynamic taint analysis to determine where and how to mutate inputs. REDQUEEN [29] focuses on solving magic values and checksums in fuzzing by "coloring" an input seed, replacing each input byte with the maximum number of random bytes possible. Angora [23] employs dynamic taint analysis to depict the pattern of input bytes relevant to path constraints. PATA [30] proposes a path-aware taint analysis to identify and mutate critical bytes, thereby addressing path constraints. These approaches leverage data flow analysis, particularly dynamic taint analysis, to gain insights into the relationship between input bytes and path constraints. By doing so, they enhance the effectiveness and efficiency of CGF by enabling targeted and informed mutations of inputs.

Despite the impressive performance exhibited by state-of-the-art fuzzers in various applications, applying them to Zigbee protocol implementations presents unique challenges. Firstly, these fuzzers encounter restrictions when attempting to compile the source code of Zigbee protocols for injecting instrumentation to collect code coverage data. Zigbee protocol vendors typically develop the protocol specifically for embedded devices using particular development toolchains [31]. These vendors incorporate compiler validation into their implementations to restrict the use of compilers that are not on the supported list, especially general-purpose compilers like GCC, LLVM, and Clang that are commonly used by existing fuzzing tools.



```
/*
 * Check that the correct C compiler is used.
 */
#ifndef __ICCARM__
  #error "File intrinsics.h can only be used together with iccarm."
#endif

#ifndef __ICCARM_INTRINSICS_VERSION__
  #error "Unknown compiler intrinsics version"
#elif __ICCARM_INTRINSICS_VERSION__ != 2
  #error "Compiler intrinsics version does not match this file"
#endif
```

Figure 1.1: Compiler Validation Example in A Popular Zigbee Protocol Stack.

Though these state-of-the-art fuzzers have shown good performance in general applications, it is not a trivial task to apply them to Zigbee protocol implementations. Firstly, these fuzzers are not allowed to compile Zigbee protocol's source code when injecting instrumentation for code coverage collection. Zigbee protocol vendors generally develop the protocol for specific embedded devices using a particular development toolchain [31]. In terms of protocol availability, vendors deploy compiler validation in their implementations to prevent compilers that are not in the supported list, especially the general compilers (e.g., GCC, LLVM, and Clang) used by many existing fuzzing tools. For example, Figure 1.1 illustrates how Texas Instruments (TI) has implemented specific compiler validation in their Zigbee protocol stack, Z-Stack. This prevents the usage of general open-source compilers such as GCC, Clang, and LLVM for compiling the protocol stack. Therefore, fuzzing tools that rely on these general-purpose compilers face limitations when it comes to fuzzing Zigbee protocol implementations due to the compiler restrictions imposed by the protocol vendors.

Another challenge in applying existing fuzzing approaches to Zigbee protocol implementations lies in the lack of a suitable simulation environment that can accommodate the specific hardware configuration required by Zigbee protocol vendors. The execution of the Zigbee protocol typically occurs on embedded devices, which are system-on-chip (SoC) devices running bare-metal programs that consist of a single control loop for task scheduling and event handling [31]. Consequently, fuzzing the Zigbee protocol on simulation platforms [17,18,19,25,26] that rely on the presence of a Linux kernel or an abstraction layer becomes infeasible. While some QEMU-based embedded fuzzers, like P2IM [24], do support bare-metal programs and various embedded CPU types, they currently lack support for devices capable of executing the different Zigbee protocol implementations required by protocol vendors [32]. Since 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.

Furthermore, the Zigbee protocol stack interacts with events triggered by specific peripheral interrupts, which are not accounted for in existing solutions [33]. Moreover, the same peripheral may be configured differently on various devices with different interrupts [24]. Incorporating support for all device-specific peripherals and new embedded chips into existing simulation platforms would require significant engineering efforts, and in some cases, it may not be feasible. Consequently, these limitations hinder the direct deployment of state-of-the-art fuzzing methods on Zigbee protocol implementations, as they are unable to provide a proper simulation environment that aligns with the particular hardware configuration and peripheral interactions of the Zigbee protocol.

Furthermore, existing protocol fuzzing solutions have limitations when it comes to efficiently mutating test cases to generate new ones. Conventional protocol fuzzers typically generate test cases from scratch based on the protocol specification. They either sequentially mutate message fields (e.g., Boofuzz [14]) or randomly mutate a single message field (e.g., Peach [15]). However, these approaches often overlook the constraints between multiple message fields. In our observations, we have noticed that most vulnerabilities in communication protocols are triggered by specific messages that satisfy branch conditions, leading to the execution of vulnerable code paths.

These path constraints often require specific combinations of values across multiple message fields. Therefore, protocol fuzzers that only mutate a single message field at a time are likely to generate test cases that fail to satisfy these conditions effectively. As a result, a significant amount of computing power is wasted on exploring the vast test input space, yielding suboptimal results. Hence, there is a need for improved protocol fuzzers that take into account the inter-field dependencies and can efficiently generate test cases that satisfy the required path constraints, thereby optimizing the fuzzing process.

To address the aforementioned gaps, this dissertation proposes several comprehensive fuzzing solutions specifically designed to detect security vulnerabilities in Zigbee protocol implementations. The primary objective of these solutions is to aid IoT application developers in assessing the security risks associated with the Zigbee protocol during the development of their applications. By applying these solutions, developers will be able to identify potential security threats and weaknesses within their Zigbee-based systems. First, Chapter 2 will introduce the basic background knowledge of the Zigbee protocol and state-of-the-art approaches of Zigbee security analysis, fuzz testing can combinatorial testing. In Chapter 3, I will present a prototype of **a device-agnostic fuzzing platform** that makes fuzzing applicable to Zigbee protocol implementation. It leverages grammar-based fuzzing with code coverage heuristics to generate high-quality test cases for detecting security issues in the Zigbee protocol implementations. In Chapter 4, I will propose **an intelligent fuzzing solution** to improve Zigbee protocol fuzzing performance by utilizing static taint analysis to infer the relationship between message field and path constraints. The inference result then guides the mutation process during fuzzing to generate more diversified test cases efficiently. In Chapter 5, I will report **an optimized test generation method** for Zigbee protocol fuzzing to reduce test input space and redundant test cases generation with combinatorial test generation. CT-BFuzz utilizes static taint analysis and fuzzing to identify the important message fields and their representative values for dynamically generating CT test models, while the CT test set helps fuzzing generate more diversified test cases to cover important combination values of message fields that may explore uncovered program branches. Finally, I will summarize this dissertation in Chapter 6.

In addition to their practical impact, the proposed research solutions are expected to yield several academic publications that will be shared with the research community through academic channels. These publications will document the methodologies, findings, and contributions of the research, allowing other researchers to build upon and further advance the field of security analysis in Zigbee protocol implementations. The dissemination of these research publications will contribute to the collective knowledge and understanding of security issues in Zigbee-based systems. By sharing the proposed solutions and their outcomes, protocol vendors and IoT application developers will have access to valuable insights that can help them proactively address and mitigate potential security risks during the development phase. And I believe these publications will foster collaboration and engagement within the research community, enabling researchers from different backgrounds to exchange ideas, provide feedback, and collectively work towards enhancing the security of Zigbee protocol implementations.

Chapter 2

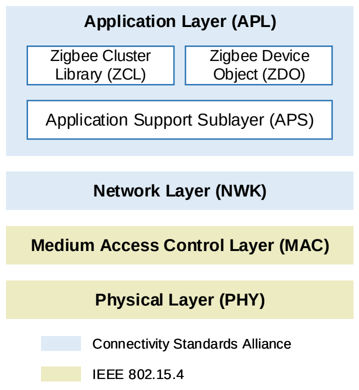
Related Work

This chapter will provide a comprehensive overview of the background, current research, and methodologies relevant to the security analysis of the Zigbee protocol. I first introduce background knowledge of the Zigbee protocol. Then I will summarize the current security analysis work on Zigbee protocol. I will also discuss related work of fuzz testing, including conventional protocol fuzzing, coverage-guided fuzzing, and taint inference based fuzzing. Finally, I will discuss related work of utilizing combinatorial testing to detect security problems.

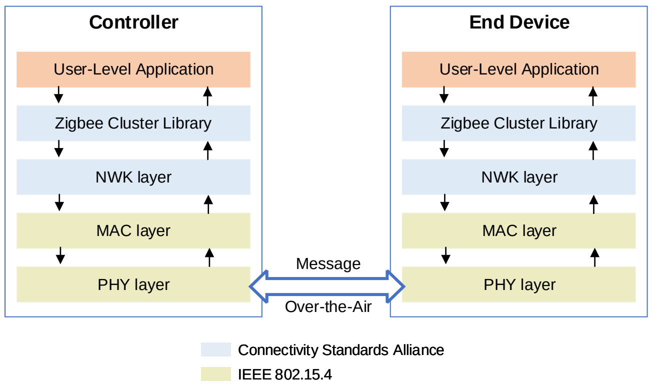
2.1 Zigbee Protocol

The Zigbee protocol is a low-cost, low-power-consumption, two-way wireless communication protocol [34]. The Connectivity Standards Alliance (previously called Zigbee Alliance) defines the operation of a Zigbee network and the protocol specification.

The Zigbee protocol stack, as shown in Figure 2.1a, is designed as a four-layer stack on top of the IEEE 802.15.4 standard. The Connectivity Standards Alliance defines the upper two layers, i.e., Application Layer (APL) and Network Layer (NWK). The IEEE 802.15.4 standard defines the Medium Access Control Layer (MAC) and Physical Layer (PHY). The MAC and PHY 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. The Zigbee protocol also provides security services on its NWK and APS


Application Layer (APL)
Zigbee Cluster Library (ZCL)
Zigbee Device Object (ZDO)
Application Support Sublayer (APS)
Network Layer (NWK)
Medium Access Control Layer (MAC)
Physical Layer (PHY)
Connectivity Standards Alliance
IEEE 802.15.4

(a) Overview of Zigbee Protocol Stack [34].



(b) Zigbee Protocol Message Exchange [35].

Figure 2.1: Zigbee Protocol Communication.

layers by using AES-128 algorithms for the packet traffic encryption. In a Zigbee network, there are two types of encryption keys. One is shared across all devices, which is referred as the network key. Another one shared only between two paired devices is referred to as link keys.

Figure 2.1b 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 the Zigbee Cluster Library (ZCL), which are defined to perform device functionalities. The ZCL then sends the request to the lower layers. The message is transmitted over the air. After receiving the message, the ZCL in the end device processes the message and passes the request to the upper application to make 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, I use ZCL as a case study for fuzzing the Zigbee protocol implementation in this dissertation.

2.2 Security Analysis on Zigbee Protocol

Since the standardization of the Zigbee protocol in 2003, numerous research works have been published to analyze the security risks associated with the protocol. Previous studies have primarily focused on the security of Zigbee network transmission.

Z3Sec [9], and Snout [36] mploy penetration testing techniques to assess vulnerabilities in Zigbee networks. IoTcube [37] and beSTORM [38] have been developed to analyze the security of the Zigbee protocol on specific embedded devices. Akestoridis et al. [39] propose Zigator, a tool that analyzes encrypted Zigbee packets to detect selective jamming and spoofing attacks. Wang et al. [10] introduce VEREJOIN, an automated verification tool based on model checking, to evaluate the Zigbee network rejoin procedure. Ronen et al. [8] demonstrate the potential damage caused by a worm that targets all Zigbee-enabled lamps, affecting smart lighting in an entire city. Recently, Ma et al. [40] also proposed a hub-based fuzzing solution for IoT devices, enabling the discovery of vulnerabilities without relying on companion apps. They capture the setting-up message sequences between an IoT device and a hub (e.g., a gateway device to manage all other IoT devices and connect them to the Internet) and identify the supported functions for fuzzing.

Most of these solutions are considered black-box approaches as they monitor and manipulate Zigbee network traffic to identify security issues. Additionally, Cui et al. proposed two fuzzing approaches, namely FSM-Fuzzing [41] and CG-Fuzzing [42], to detect security risks in Zigbee. FSM-Fuzzing is based on a finite state machine, while CG-Fuzzing relies on a genetic algorithm. However, both of these approaches are closed-source and thus cannot be directly compared with the proposed fuzzing solution presented in this dissertation. Recently, Wang et al. [43] also evaluated new threat models of Zigbee network with real IoT devices, in which the adversaries are outside the network. They also utilize semantic fuzzing to generate test packets that have higher chance to produce meaningful results.

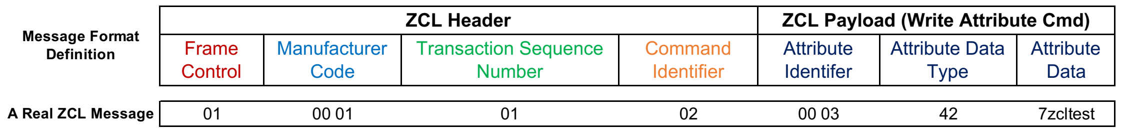
Unlike the aforementioned vulnerability exploitation works that focus on analyzing network traffic or targeting real IoT devices, the approaches proposed in this dissertation aim to identify unknown vulnerabilities in Zigbee protocol implementations themselves, rather than in real-time Zigbee networks. These fuzzing platforms do not rely on physical devices or specific knowledge of the underlying hardware design. By leveraging fuzzing techniques, my proposed approaches generate and mutate test cases for the Zigbee protocol, aiming to trigger potential security vulnerabilities within the protocol implementation. This allows for a systematic exploration of the protocol's code paths and the identification of security issues that may have been overlooked during the development phase. The fuzzing platforms developed in this research can be applied independently of real devices, providing a cost-effective and efficient means of detecting vulnerabilities in Zigbee protocol implementations.

2.3 Fuzz Testing

Fuzz testing is a widely used technique to detect vulnerabilities. The basic idea is to execute a program under test with random inputs and monitor execution failures that can be used for further analysis. Many techniques have been proposed to improve the fuzzing performance.

2.3.1 Conventional 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 [13], Sulley [44], Boofuzz [14], AutoFuzz [45], and SNOOZE [46]) employ *grammar-based fuzzing* [47] to generate well-structured



	ZCL Header				ZCL Payload (Write Attribute Cmd)		
Message Format Definition	Frame Control	Manufacturer Code	Transaction Sequence Number	Command Identifier	Attribute Identifer	Attribute Data Type	Attribute Data
A Real ZCL Message	01	00 01	01	02	00 03	42	7zcltest

(a) ZCL Frame Format Definition [48].

```
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) Example of Message Format Script.

Figure 2.2: Example of ZCL Message Construction with Block-based Representation.

packets that adhere to network protocol format requirements. These approaches construct test inputs based on input specifications, which define the data format and integrity constraints, enabling effective fuzzing of network protocols.

Block-based protocol representation, also known as abstract representation *blocks*, is utilized by protocol fuzzers [49]. In this representation, a *block* represents a set of abstracted data or nested blocks that conform to the protocol format. By organizing the protocol frames into blocks, the fuzzers can generate test inputs that conform to the format definition, allowing for format validation without early rejection during runtime. An example of generating a ZCL message using the block-based representation is illustrated in Figure 2.2b, where the format definition script specifies the placement of primitive data within the ZCL message. A ZCL message is initialized as a block with the name `ZCLMessage` (line 1 in Figure 2.2b). With the format definition script, the protocol fuzzers represent the protocol message with primitive data following their placements. The generated test inputs could satisfy the format validation without early rejection during the execution time.

Although these protocol fuzzers with block-based representation define and generate highly structured input formats, they have a disadvantage of the quality of test inputs generation. For example, as shown in Figure 2.2a, the fuzzer first mutates the field *Frame Control*. Once the mutation on this field finishes, the fuzzer resets the field to its initial value. The fuzzer mutates a single field at a time. Then the fuzzer moves to the following field *Manufacturer Code* for mutation. The field *Attribute Data* would be the last for mutation. Thus, if a message consists of M fields and each field has N possible values, the fuzzer can generate $(M * N)$ new test cases in total. Without considering execution feedback and program structure, those protocol fuzzers suffer from large input space and fail to explore deeper code of the target program.

Unlike the mentioned protocol fuzzers, our proposed fuzzing solution, *Z-Fuzzer* integrates grammar-based fuzzing with code coverage heuristics for testing Zigbee protocol implementations.Z-Fuzzer starts by generating an initial test corpus using a provided Zigbee message format script, ensuring that the test cases pass the pre-check of the target program. It then prioritizes test cases that explore new execution paths, allowing for further mutation.

2.3.2 Coverage-guided Fuzzing

Some researchers have recognized the importance of code coverage in guiding protocol fuzzing to improve performance. AFL [16] and its derivatives, such as AFL++ [21], have gained popularity in automated security analysis. However, they face limitations when it comes to compiling the source code of the Zigbee protocol. Zigbee protocol vendors typically use specific development toolchains for their protocol stack, which restricts the use of general compilers like GCC and LLVM, commonly used in AFL and its derivatives, for building the protocol stack.

Fuzzing on IoT embedded devices poses additional challenges due to the reliance on specific hardware configurations. Several existing research works [17,18,19,24,26] address this challenge by integrating emulators into their fuzzing tools. A notable emulator is QEMU [20] which provides user-mode emulation and full emulation for a variety of embedded devices. However, using QEMU in user-mode typically requires a Linux kernel or a hardware abstraction layer (HAL) to execute the target program. AFL QEMU mode [50], Frankenstein [18] and BaseSAFE [19] utilize QEMU in user-mode, which require presence of Linux kernel or an abstract layer. Some researches [17,24,26] propose hybrid solutions that combine user-mode and full emulation together.

However, the execution environment required by Zigbee protocol implementations, particularly on specific chipsets and embedded devices, poses a challenge for existing simulation platforms like QEMU. The Zigbee protocol is typically executed on system-on-chip (SoC) devices and bare-metal systems, which may not be compatible with the Linux kernel or hardware abstraction layers used in simulation platforms. Additionally, vendor-specific embedded devices often have unique hardware and peripheral interrupt configurations that are not supported by existing simulators. For example, the Zigbee protocol stack *Z-Stack* from Texas Instruments can only be executed in three embedded devices, which instead are not supported by current simulators. This lack of support for the specific configurations required by Zigbee protocol vendors makes it difficult, or even impossible, for existing simulation platforms to provide an appropriate execution environment for Zigbee protocol stacks.

2.3.3 Taint Inference Based Fuzzing

Indeed, symbolic execution-based approaches, like Driller [51] and QSYM [52], have been proposed to address the challenge of generating test inputs that satisfy complex path constraints. These techniques use symbolic execution to explore different program paths and generate inputs that satisfy specific conditions. However, symbolic execution can suffer from scalability issues and slow execution speed, especially when dealing with large and complex applications. The path explosion problem, where the number of possible execution paths grows exponentially, further limits the scalability of symbolic execution-based approaches. As a result, while these techniques can be effective in certain scenarios, they may not be suitable for fuzzing large-scale applications due to their limitations in scalability and execution speed.

In order to efficiently resolve path constraints, several lightweight solutions have been proposed to efficiently resolve path constraints in fuzzing by inferring the relationship between input bytes and constraints. These approaches aim to guide seed mutation to generate test cases that satisfy specific path constraints. VUzzer [27] focuses on passing magic value validations by using taint analysis to identify critical bytes that need to be mutated to satisfy path constraints. Angora [23] locates input bytes that flow into path constraints using byte-level taint tracking and mutates them with a gradient descent algorithm to satisfy the constraints. REDQUEEN [29] aims to solve magic values and checksums by coloring an input seed, replacing every input byte with as many random bytes as possible while preserving the execution path. Matryoshka [53] explores nested branches for fuzzing based on both control flow and taint flow, allowing for deeper exploration of program paths. GREYONE [28] utilizes taint analysis to locate critical input bytes and determines how to mutate them effectively during the fuzzing process. PATA [30] proposes a path-awareness taint analysis for fuzzing, inferring taints based on control flow and value changes to guide mutation. TRUZZ [54] infers the relationship between input bytes and validation checks, preventing those bytes from being mutated during fuzzing to avoid violating constraints.

Though these fuzzers have shown good performance on general applications, they are hard to directly deploy on the Zigbee protocol implementation due to the vendor-specific requirements of compiler and underlying hardware configuration. Most of these fuzzers develop their approaches with general compilers such as LLVM or Clang for dynamic taint analysis, which are prevented by many Zigbee protocol vendors in their protocol stacks. Compared to these fuzzers, our proposed solution *TaintBFuzz* utilizes vendor-specific compiler to pre-process the protocol source code for static taint analysis. It also customizes static analysis application to parse vendor-specific syntaxes that are not intially supported.

2.4 Combinatorial Testing

Combinatorial Testing (CT), which is also referred to as t-way testing, is a popular testing method for examining interaction between input parameters that affect software execution [55, 56]. The key insight of this approach is that most execution failures are triggered by a single input parameter or combinations of several relative parameter values. Assume that a program under test P has n input parameters, $P = \{p_1, p_2, ..., p_n\}$, and each parameter can take values from a finite set V_i, for $1 \leq i \leq n$. Then a CT *test model* is consists of parameters P and value domains $V = \{V_1, V_2, ..., V_n\}$, depicting the test input space of the target program. CT can efficiently generate a covering array for *any* t (out of n) parameters, in which every t-way combination is covered at least once. It aims to achieve a good balance between test input space and the efficiency of failure discovery.

Combinatorial testing has been widely applied in various domains to detect security issues and improve test generation. Wang et al. [57] introduced Tance, a specification-based testing approach that leverages combinatorial testing to efficiently generate test inputs for external parameters, aiming to detect buffer overflow vulnerabilities. Chandrasekaran et al. [58] applied combinatorial testing to Deep Neural Network-based autonomous driving systems, effectively detecting safety-critical bugs before deployment. To efficiently test RESTful APIs, RESTCT [59] was developed as a systematic approach for testing RESTful APIs, using combinatorial testing to generate operation sequences and test different combinations of operations. Feng et al. [60] proposed MagicMirror, which integrates combinatorial testing with fuzzing to test smart contracts, effectively exploring function parameter interactions and critical values. These studies demonstrate the effectiveness of combinatorial testing in different contexts for security analysis and improving test generation.

Compared to these solutions, our proposed approach CT-BFuzz, makes fuzzing and combinatorial testing interact to improve test case generation for testing Zigbee protocol implementation. It utilizes static taint analysis and fuzzing to identify the important message fields and their representative values for dynamically generating CT test models. At the same time, the CT test set helps the fuzzing process generate more diversified test case, particularly the combination values of critical fields that have higher probability to explore uncovered execution paths.

Chapter 3

Device-Agnostic Fuzzing of Zigbee Protocol Implementation

The content of this chapter is based on a paper published in the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), in June 2021[1] and an article published in ACM journal Digital Threats: Research and Practice (DTRAP), in March 2023[2].

3.1 Overview

In this chapter, we present the first device-agnostic fuzzing framework of Zigbee Protocol Implementation *Z-Fuzzer*. 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

[1]Copyright © 2021 Association for Computing Machinery. Reprinted, with permission, from Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-Fuzzer: Device-agnostic Fuzzing of Zigbee Protocol Implementation. In Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21). Association for Computing Machinery, New York, NY, USA, 347–358. `https://doi.org/10.1145/3448300.3468296`

[2]Copyright © 2023 Association for Computing Machinery. Repritned, with permission from Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. 2023. Security Analysis of Zigbee Protocol Implementation via Device-agnostic Fuzzing. Digit. Threat.: Res. Pract. 4, 1, Article 9 (March 2023), 24 pages. `https://doi.org/10.1145/3551894`

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 [13, 44, 45, 46]. 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.

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.

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 [14] and Peach [15], as our comparative tools. BooFuzz is the successor of industry-standard protocol fuzzer Sulley [44], and Peach [15] 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 [61], 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) 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.

3.2 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

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 current cumulative code coverage $current\_coverage$

1  crash ← ∅
2  blocks ← Initialize($\mathcal{S}$)
3  top_rated ← ∅
4  current_coverage ← 0
5  repeat
6      if top_rated is not ∅ 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         tescase ← Choose(blocks)
13         seed ← Mutate(testcase)
14     end
15     coverage, result ← RunTarget($\mathcal{P}$, seed)
16     if isInteresting(coverage, result) then
17         top_rated ← top_rated ∪ seed
18         current_coverage ← CalculateCoverage(coverage)
19         crash ← crash ∪ result
20     end
21 until top_rated is ∅ and all fields in blocks are fuzzed
22 return crash, current_coverage
```

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

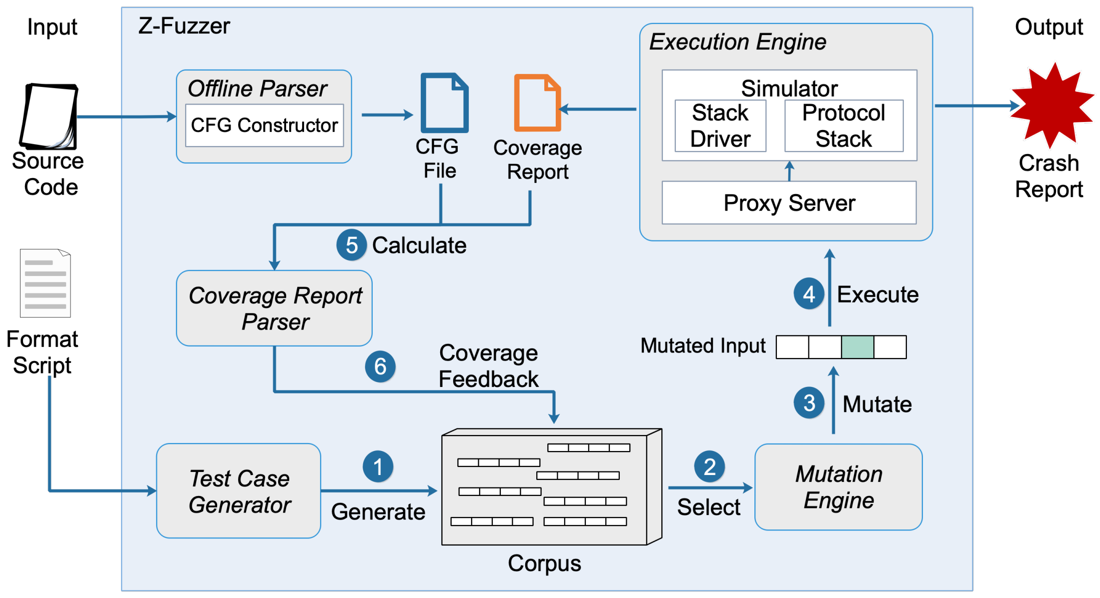


Figure 3.1: The workflow of Z-Fuzzer framework.

process when no favored test 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.

3.3 Implementation Details

Figure 3.1 presents the workflow of Z-Fuzzer framework[3]. It consists of five components: an offline parser, a test case generator, a mutation engine, an execution engine, and a coverage report parser.

[3]The source code of Z-Fuzzer is avaliable at `https://github.com/zigbeeprotocol/Z-Fuzzer`.

3.3.1 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 2.2a.

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 3.1).

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 3.1). 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 3.1). 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

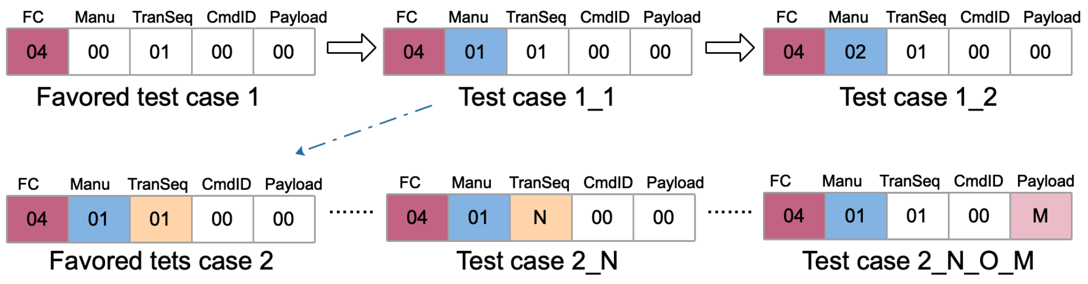


Figure 3.2: Example of mutation on a favored test case

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 3.1).

Example. Figure 3.2 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.

3.3.2 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 [62] 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 architecture supports 50 different ARM CPUs and hundreds of devices from 42 IoT manufacturers [62], 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 [63]. 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 [63]. 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

Listing 3.1: 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 }
```

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 3.1. 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.

3.3.3 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 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 Fig 3.1). 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 ith edge. We utilize the coverage measurement used in AFL[4] to calculate edge coverage. The calculation is shown below,

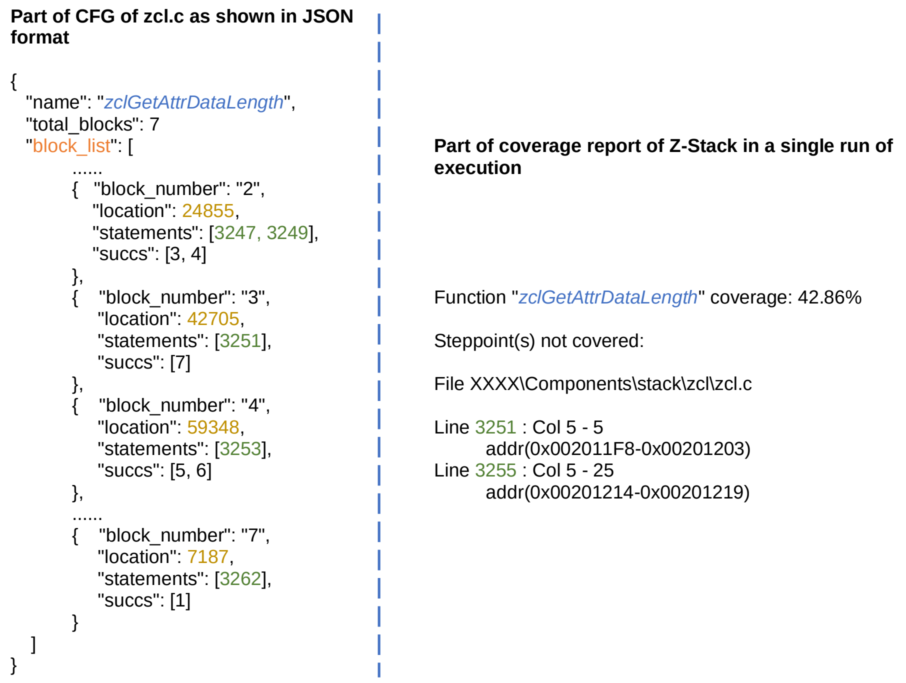
$$cur\_location = < RANDOM\_NUMBER >;$$

$$edge\_hits[cur\_location \ \oplus \ prev\_location] + +;$$

$$prev\_location = cur\_location >> 1;$$

The *cur_location* value is generated randomly for each basic block when the offline parser generates the CFG of source code. The result of XOR operation records a hit for a particular edge tuple. Though there might be a risk of collision caused by the XOR operation if the branch count becomes larger in complex programs [50], we observed that it could not be an issue in our scenario because the branch size in our target is relatively small. The shift operation is to update the previous location,

[4]More coverage measurement details have been explained in the AFL technical paper [50].



Part of CFG of zcl.c as shown in JSON format

```
{
  "name": "zclGetAttrDataLength",
  "total_blocks": 7
  "block_list": [
        ......
        {  "block_number": "2",
           "location": 24855,
           "statements": [3247, 3249],
           "succs": [3, 4]
        },
        {   "block_number": "3",
            "location": 42705,
            "statements": [3251],
            "succs": [7]
        },
        {   "block_number": "4",
            "location": 59348,
            "statements": [3253],
            "succs": [5, 6]
        },
        ......
        {   "block_number": "7",
            "location": 7187,
            "statements": [3262],
            "succs": [1]
        }
  ]
}
```

Part of coverage report of Z-Stack in a single run of execution

```
Function "zclGetAttrDataLength" coverage: 42.86%

Steppoint(s) not covered:

File XXXX\Components\stack\zcl\zcl.c

Line 3251 : Col 5 - 5
        addr(0x002011F8-0x00201203)
Line 3255 : Col 5 - 25
        addr(0x00201214-0x00201219)
```

Figure 3.3: Example of coverage report of Z-Stack execution.

which also preserves the directionality of tuples, e.g., (X >> 1) ⊕ Y distinguishes from (Y >> 1) ⊕ X.

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 3.1).

Example [64]. Fig 3.3 displays an example of the coverage report of Z-Stack in a single run of execution along with CFG information of file zcl.c. We use these two files to calculate the code coverage of function *zclGetAttrdataLength*. The function coverage is shown in the right side of Fig 3.3. When scanning this function, we first extract the uncovered statements from the coverage report to a list *uncovered_stmt*, which contains `3251` and `3255`. Then we compare this list with the variable *statements* of each basic block in the CFG file (see the left side of Fig 3.3). A basic block is labeled as executed if all statements are not in the list *uncovered_stmt*. Then we increase the value of $statement\_hits[i]$, where i is the line number of the statement contained in the covered basic block. In this case, when the *block 2* is covered in the current execution, we increase the values of $statement\_hits[3247]$ and $statement\_hits[3249]$ to record the executed statements. We then record the covered edge that is a path from the source block to *block 2*. The label of *block 2* is saved as *location* variable in the CFG file. Assume the source block's label is 0. We then increase the value of $edge\_hits[24855 \oplus 0]$ to record the executed edge.

Now we continue checking statements in block 2's successors: basic block 3 and basic block 4. Compare the statements of block 3 and block 4 with the list *uncovered_stmt*, we can derive that block 4 is executed after block 2 in the current execution since the statement `3251` of block 3 is in the list *uncovered_stmt*. The same process is repeated until the last block. The parsing process of function `zclGeAttrDataLength` has completed when the basic block 1 is triggered, which indicates the function has returned. Finally, all of the statements and edges that have been executed are recorded.

3.4 Evaluation of Z-Fuzzer

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 3.4.1)
- **RQ2**: Can Z-Fuzzer achieve higher coverage rate in comparison with the state-of-the-art fuzzers? (Section 3.4.2)

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 [14] and Peach [15]. BooFuzz is the successor of industry-standard protocol fuzzer Sulley [44], and Peach fuzzer is a model-based commercial fuzzer. Both of them have been widely used in existing research papers [65, 66]. 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 [61], which is developed by Texas Instruments with various sample project codebases and its source code is available.

Table 3.1: 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

3.4.1 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 3.1.

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 [67]. 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 3.1; 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).

Table 3.2: 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

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 3.2. 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.

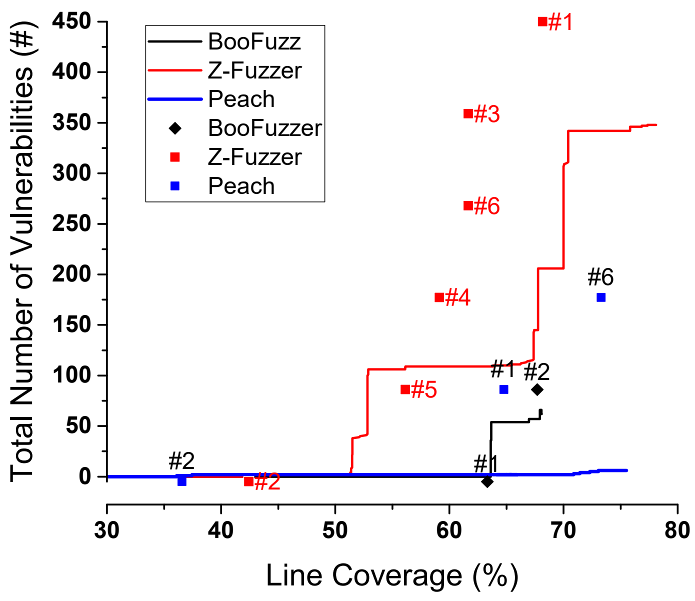


Figure 3.4: The relationship between line coverage and the number of detected protocol crashes in 10 runs.

Coverage vs. Vulnerabilities. We also analyze the relationship between line coverage and the number of detected vulnerabilities. Figure 3.4 presents the max cumulative number of vulnerabilities detected over line coverage. X-axis presents line coverage on average and Y-axis presents the max cumulative number of vulnerabilities. The symbols are the vulnerability identifiers displayed in Table 3.2 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.

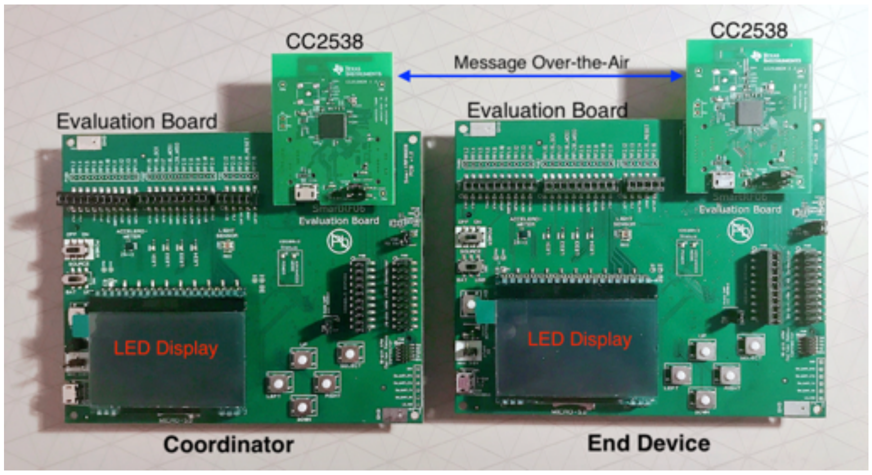


Figure 3.5: Message transmission on TI CC2538 with the evaluation board.

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 Board to form a real IoT network. TI CC2538 is a wireless microcontroller System-on-Chip (SoC) for high-performance ZigBee applications [68] and has been widely adopted in the IoT market.

As shown in Figure 3.5, 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 3.2 shows that all fuzzers can detect vulnerabilities in the function *zcl_HandleExternal* and the function *zclParseInDiscCmdsRspCmd* in the simulation environment. How-

Listing 3.2: Source code of CVE-2020-27892

```
static void *zclParseInDiscCmdsRspCmd(zclParseCmd_t *pCmd)
{ .....
  pDiscoverRspCmd=(zclDiscoverCmdsCmdRsp_t*)
  zcl_mem_alloc(sizeof(zclDiscoverCmdsCmdRsp_t) +
  (numCmds*sizeof(uint8)));
  if(pDiscoverRspCmd != NULL)
  { ......
    for(i = 0;i < numCmds;i++)
    {
      pDiscoverRspCmd->pCmdID[i] = *pBuf++;
    }
  }
  return ( (void *)pDiscoverRspCmd );
}
```

ever, 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 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 3.2. 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 [69]. The IoT application developers may not be aware of those potential issues until they complete the entire production.

Table 3.3: Evaluation results on Z-stack in 10 runs.

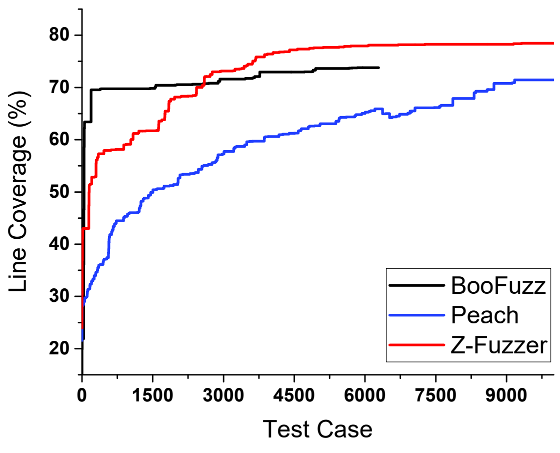
Fuzzer	Total # of Unique Test Cases	Line Coverage		Edge Coverage	
		total	%	total	%
BooFuzz	16,756	912	73.80%	680	73.82%
Peach	18,271	850	68.71%	628	67.58%
Z-Fuzzer	61,386	971	**78.52%**	769	**82.30%**

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.

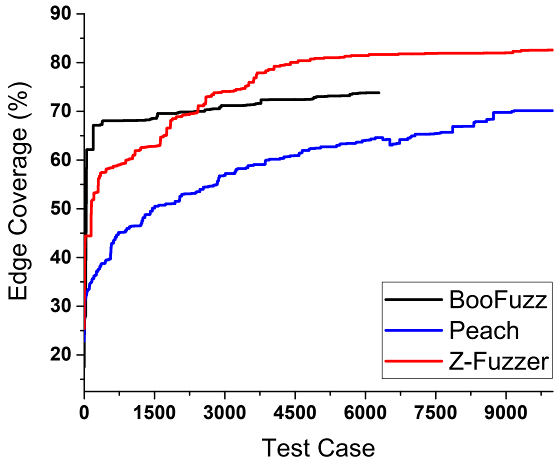
3.4.2 Code Coverage

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 [67]. 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 3.3. We report the line coverage and edge coverage on average. 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 3.3, Z-Fuzzer can generate 6 times more unique test cases than the other two fuzzers. Moreover, according to the Zigbee protocol specification, we cate-



(a) Line Coverage



(b) Edge Coverage

Figure 3.6: Line coverage and edge coverage achieved by fuzzers over 10 runs.

gorize 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.

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 3.3 and Figure 3.6. Table 3.3 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 3.4.1 indicates, Z-Fuzzer can discover more vulnerabilities than the other two fuzzers though it does not achieve full code coverage.

From Figure 3.6, 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 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 2.2a). 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 3.6. 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.

3.5 Conclusion

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.

Chapter 4

Intelligent Zigbee Protocol Fuzzing via Constraint-Field Dependency Inference

The content of this chapter is based on a paper [70] just accepted in European Symposium on Research in Computer Security (ESORICS), in April 2023[1].

4.1 Overview

Though Z-Fuzzer has shown promising results for finding security vulnerabilities in Zigbee protocol implementation, it still suffers from a large search space of inputs by ignoring the target program structure. Many existing fuzzers [23, 27, 28, 30, 51] apply various techniques to infer the relationship between input bytes and path constraints for generating test inputs efficiently, which can explore the deeper code of the target program. Data flow analysis (e.g., dynamic taint analysis) is one of the most adopted methods for dependency inference. However, it is not a trivial task to directly deploy those fuzzers to Zigbee protocol implementations. First, these fuzzers use general compilers like LLVM and Clang for dynamic taint analysis. As explained in Figure 1.1 in Chapter 1, those general compilers are prevented from compiling the Zigbee protocol stack by many protocol vendors. Second, these fuzzers, which utilize QEMU for program execution simulation, cannot provide a proper simulation environment due to the particular hardware configuration required by the Zigbee protocol vendors.

[1]Reproduced with permission from Springer Nature. Mengfei Ren, Haotian Zhang, Xiaolei Ren, Jiang Ming and Yu Lei, Computer Security – European Symposium on Research in Computer Security 2023 (ESORICS). *Just Accepted.*

To address these limitations, in this chapter, we propose *TaintBFuzz*, an intelligent Zigbee protocol fuzzing with constraint-field dependency inference. We leverage static taint analysis to infer the relationship between the message field and the path constraints, while also satisfying the protocol vendors' requirement of the specific compiler. The dependency inference then guides the fuzzing engine to prioritize the critical message fields for further mutation, which have a higher chance to exercise unvisited branches.

The fuzzing engine of TaintBFuzz is designed based on Z-Fuzzer's fuzzing engine. It constructs the initial test seeds based on the message format script from scratch. To execute the Zigbee protocol stack in a simulation environment, we use an industrial embedded device development platform, IAR Embedded Workbench [62], to interact with the fuzzing engine of TaintBFuzz. The IAR is used by many Zigbee protocol vendors, such as TI, Samsung, and Toshiba, and provides a particular compiler and a software simulator. The IAR simulator also supports many vendor-specific embedded devices with pre-defined hardware interrupt/peripheral configurations. We also develop a stack driver and a proxy server to bridge the communication gap between the IAR simulator and the fuzzing engine.

We implemented a prototype of TaintBFuzz and evaluated its effectiveness in security vulnerability detection on Z-Stack [61], a mainstream Zigbee protocol stack developed by Texas Instruments. We compare TaintBFuzz with three state-of-the-art protocol fuzzing tools, Peach [15], BooFuzz [14], and Z-Fuzzer [71]. Peach and BooFuzz are conventional protocol fuzzers widely used in academia and industry. Our experiment results show that TaintBFuzz outperforms those fuzzers by 27% and 25% in terms of the number of unique edges found and statements covered. TaintBFuzz has also identified eight unique vulnerabilities in Z-Stack, of which two are previously undiscovered.

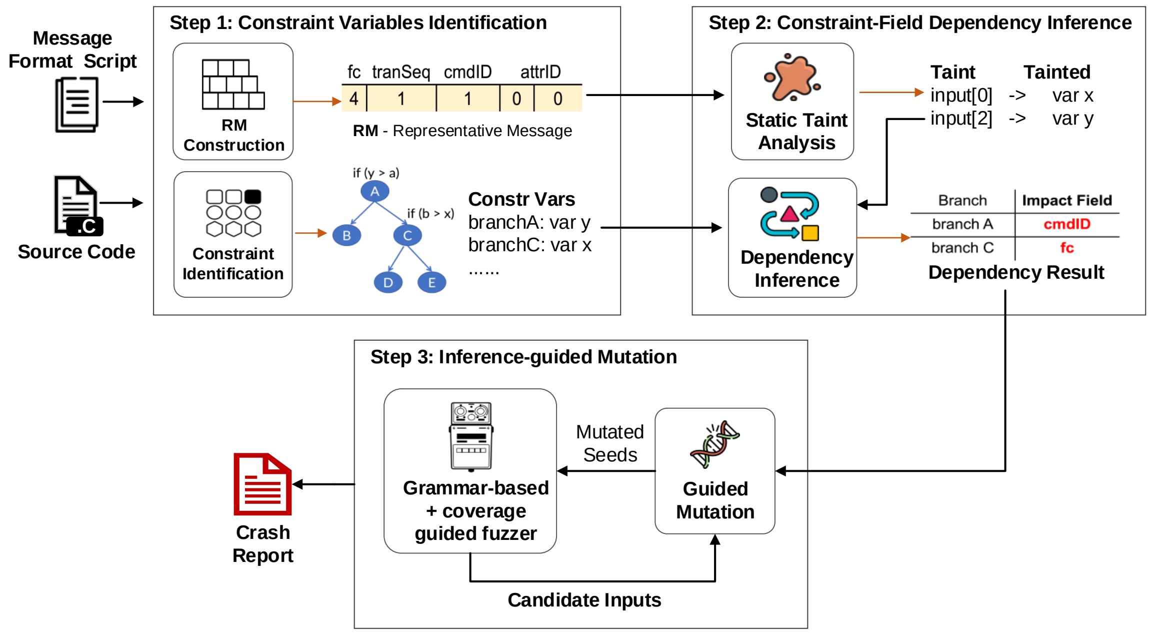


Figure 4.1: Overall design of TaintBFuzz.

4.2 Design of TaintBFuzz

Figure 4.1 presents the overall design of TaintBFuzz, which contains three main steps: (1) Constraint Variable Identification, (2) Constraint-Field Dependency Inference, and (3) Inference-guided Mutation. The black arrows in figure mean the main workflow of TaintBFuzz. The red arrows mean the intermediate results generated by the related components. As the ZCL is the core library of Zigbee protocol stack to implement an IoT device's functionalities, we will use it to present the details of each step in the following subsections.

4.2.1 Constraint Variable Identification

The first challenge of TaintBFuzz design is to identify the constraint variables reasonably. A constraint variable consists of a set of program variables used in a path constraint. To address this challenge, TaintBFuzz collects program variables used in all constraints based on the AST analysis of the program. A program variable can directly or indirectly influence a constraint. Notably, a temporary variable saves an intermediate result that can be used in the following constraints, e.g., in the statements $temp = Function\_A(x, y); if(temp)...$, the result of a function call is saved as a temporary variable that impacts the IF condition. In addition to the regular conditional constraint statements like *IF, LOOP, and SWITCH*, TaintBFuzz also collects program variables used in every function call to address the temporary variable propagation. Accordingly, a *constraint variable* is defined as a tuple (V, t, loc), where V is a set of program variables, $t \in T$ that T is a set of pre-defined constraint types (*IF, LOOP, SWITCH, CALL*), and *loc* is a statement line number of a constraint. A path constraint can be parsed as several sub-constraints during the AST analysis; thus, we save *loc* to assemble a completed dependent fields list during the following inference phase.

Additionally, TaintBFuzz constructs a set of Representative Messages (RM) based on the given protocol message format script, in which the message format is defined as Fig 2.2a in Chapter 2. An RM is defined as a tuple $(F, Len, data)$, where $F = (F_1, ..., F_n)$ is a set of message fields defined in the script, $Len = (L_1, ..., L_n)$ is the length of every message field, and *data* is a real ZCL message. Each RM represents a unique type of ZCL message. The generated RMs will be used for taint analysis to identify the critical fields that impact program variables.

4.2.2 Constraint-Field Dependency Inference

The second challenge of TaintBFuzz is inferring the relationship between the message fields and the path constraints. A standard solution is utilizing dynamic taint analysis (DTA) to identify which input bytes are used in branch instructions. However, it could fail to compile the Zigbee protocol because of the vendor-specific compiler requirement as shown in Fig 1.1 in Chapter 1. To tackle this challenge, TaintBFuzz performs static taint analysis on a preprocessed source code compiled by the protocol vendor-specific compiler to distinguish the dependency between message fields and path constraints.

Algorithm 2 illustrates the primary process for the dependency inference. First, we track an external input's impact on the program execution through static taint analysis. For each RM, we taint each message value (e.g., input[0] whose value is 4 as shown in Figure 4.1) and perform static taint analysis to collect the tainted variables (lines 4-6). After collecting the taint analysis result, we perform dependency inference based on the constraint variables collected from Step 1 and the taint analysis result. For each constraint variable, we first identify if its program variable exists in the tainted variables (line 10).

If a variable is a tainted variable, then we collect its tainted record (line 11) including the tainted label like `input[0]` in Step 2 and the message value like the array `[4,1,1,0,0]` in Step 1. Then the tainted record is used to search the corresponding message field in the set of RMs (line 12). Finally, we gather all message fields related to the program variables used in a path constraint, e.g., constraint A is impacted by the message field `cmdID` as shown in Figure 4.1. The collected result is saved as a map where the key is the constraint, and the value is the message fields influencing the constraint. As a path constraint could consist of several sub-constraints, we combine all constraint-field dependencies based on the constraint's *loc* value as the final dependency inference result and pass it to the mutation engine (line 17).

4.2.3 Inference-guided Mutation

The main challenge of TaintBFuzz is effectively leveraging dependency analysis results, which implicates inference-guided mutation. Our objective is to enhance the mutation process through dependency inference when a fuzzer is hard to explore more

Algorithm 2: Constraint-Field Dependency Inference

```
Input  : A set of representative message: $\mathcal{R}$,
         A set of constraint variables: $\mathcal{P}$,
         Preprocessed source code: $\mathcal{S}$
Output: Hashmap(constraint → fields): $Deps$
1  $tainted \leftarrow \emptyset$
2  $Deps \leftarrow \emptyset$
3  foreach $rs \in \mathcal{R}$ do
4      $taint \leftarrow$ taintField ($rs$)
5      $taint\_vars \leftarrow$ taintAnalysis ($\mathcal{S}, taint$)
6      $tainted \leftarrow tainted \cup (taint, taint\_vars, rs.data)$
7  end
8  foreach $constraint \in \mathcal{P}$ do
9      foreach $var \in constraint.V$ do
10         if isTainted ($var, tainted$) then
11             $tainted\_record \leftarrow$ getTainted ($var, tainted$)
12             $field \leftarrow$ searchField ($\mathcal{R}, tainted\_record$)
13             $Deps[constraint] \leftarrow Deps[constraint] \cup field$
14         end
15     end
16 end
17 $Deps \leftarrow$ assembleDependency ($Deps$)
```

paths of a program. Remarkably, we use coverage-guided fuzzing (CGF) in our main fuzzing engine because it is low-cost and efficiently covers the majority of easy-to-cover branches. Only for hard-to-cover branches, we introduce the constraint-field dependency to augment the mutation process and generate diversified seeds. Algorithm 3 shows the primary process of coverage-guided fuzzing with constraint-field dependency inference. A *threshold* is a pre-defined value of the number of mutations since the last updated code coverage, indicating when to utilize the constraint-field dependency for mutation on a particular path to explore more uncovered branches.

Algorithm 3: Fuzzing with Constraint-Field Dependency Inference

```
Input  : Input seed: $s$, Inference result: $Infer$,
         Control flow graph: $\mathcal{G}$, Timeout: $timeout$
         Program for coverage tracking: $\mathcal{P}$,
         Program for inference tracking: $\mathcal{P}'$
Output: Detected crash: $crash$

$execPath \leftarrow \emptyset$
$crash \leftarrow \emptyset$
$threshold \leftarrow$ user_predefined_value
def main():
    while $not\ timeout$ do
        $cov, execPath, crash \leftarrow$ execCheckCoverage $(s, \mathcal{P})$
        if $noUpdate\ (cov, threshold)$ then
            $s \leftarrow$ mutateWithInfer $(s, cov, execPath)$
        else
            $s \leftarrow$ mutate $(s)$
    end
def mutateWithInfer $(s, cov, execPath)$:
    $pid \leftarrow$ len $(execPath)$
    $uncovered \leftarrow$ checkPath $(cov, execPath, pid, \mathcal{G})$
    $inferFields \leftarrow$ getInferFields $(uncovered, Infer)$
    while $pid \geq 0$ do
        foreach $f \in inferFields$ do
            $s', mutated \leftarrow$ mutate $(s, f)$
            if $mutated$ then
                break
        end
        $cov', path', crash \leftarrow$ executeGetCovered $(s', \mathcal{P}')$
        if $hasCovered\ (uncovered, cov')$ then
            return $s'$
        else if $callStackChanged\ (execPath, path')$ then
            $inferFields \leftarrow$ updateFieldState $(s, execPath, inferFields)$
        else if $not\ mutated$ then
            $pid \leftarrow pid - 1$
            $uncovered \leftarrow$ checkPath $(cov', execPath, pid, \mathcal{G})$
            $inferFields \leftarrow$ getInferFields $(uncovered, Infer)$
        end
    end
```

4.2.3.1 Grammar Based with Coverage Guided Fuzzing

TaintBFuzz uses a grammar-based fuzzer with coverage-guided feedback as its fuzzing engine. We generate the initial seed corpus based on the given protocol message script from scratch so that each seed would satisfy the sanity check of message processing. If a new edge is discovered, the seed is saved as a favored test case with higher prioritization in the following mutations. The fuzzer also monitors the protocol stack execution result and reports any detected crashes. If the code coverage has not been updated after several seed mutations (*threshold*), we utilize the inference result for mutation optimization.

4.2.3.2 Mutation with Dependency Inference

Once no more new codes are explored after the pre-defined threshold, we mutate the seed based on the constraint-field dependency of the current execution path. Assume a sample input's message fields are $[fc, manu, seqID, cmd, attrId, type, data]$ and a covered basic block sequence is $[B_1, B_2, B_4, B_6, B_7]$. In order to explore deeper of the path, TaintBFuzz backtracks the block sequence to identify the last uncovered block in the current path by examing the control flow graph and coverage feedback (lines 12-14), e.g., B_6 is the predecessor block of B_7 that contains a condition check and has an uncovered block B_8. Then TaintBFuzz searches the corresponding constraint of B_6 in the dependency inference result. For example, we find the fields [fc and *cmd*] that influence the constraint. TaintBFuzz sequentially mutates each field to generate new inputs (lines 16-20), and executes the program with the new inputs (line 21). If the block B_8 has been accessed (lines 22-23) indicating the code coverage is increased, then we return to regular coverage-guided fuzzing with the new input.

A mutation on the dependent field may change the predecessor block sequence of the previously uncovered block. For example, the predecessor block sequence of B_8 is $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$. A new value of the inferred field *cmd* leads to a new execution path that does not exercise B_6 anymore. Then, TaintBFuzz first tries other candidate values of the field *cmd* and checks if the previously predecessor block sequence can be reaccessed (line 25). The worst case is that all candidate values of the field never explore the uncovered branch. In that case, TaintBFuzz restores the original value of this field and filters out this field from the inferred fields list without further mutation. Furthermore, suppose mutations on all dependent fields of a constraint fail to access the uncovered branch, i.e., the variable *mutated* is FALSE, indicating the completed mutation on the fields (line 26). In that case, TaintBFuzz then backtraces to the next uncovered block in the path to mutate with the inferred fields until all blocks in the block sequence have been traversed (lines 27-29).

4.3 Implementation of TaintBFuzz

The purpose of TaintBFuzz is to assist Zigbee protocol vendors and IoT application manufacturers in avoiding security risks during their development phase. Thus, the Zigbee protocol message format and related IoT device configuration are assumed to be aware and configured in the format script. As Fig 4.1 shows, the constraint variables identifier, the constraint-field dependency inferrer, and the inference-guided mutator are the three main components of TaintBFuzz. We illustrated the details of each component as follows. And the source code of TaintBFuzz is avaliable at `https://github.com/zigbeeprotocol/TaintBFuzz`.

The representative message constructor is implemented using the message generator of Boofuzz [14] with a pre-defined message format script that conforms to protocol format definition [72]. The constraint variables identifier and taint analysis tool are developed based on Frama-C [73]. Frama-C is an open-source platform dedicated to source-code analysis of C software and performs static analysis based on an abstract syntax tree (AST). The constraint variable collector is performed with a pre-processing file of the source code that is compiled with the IAR compiler to avoid compiler restriction. We modify Frame-C to analyze preprocessed code with the vendor-specific syntaxes that are not initially supported (e.g., *__intrinsic, __nounwind, #Pragma rtmodel* and so on) for AST analysis. We also implement a script using Ocaml to analyze AST and collect the constraint variables used in IF, LOOP, SWITCH, and CALL statements.

The constraint-field dependency inferrer implements Algorithm 2. According to the generated RMs and taint analysis result, it maps message fields to several message fields that could impact the condition decision. The inference-guided mutator implements Algorithm 3. Suppose no more new edges are explored after several mutations (a threshold). In that case, it evaluates each input seed along its execution path and collects constraint variables helpful in exploring new branches. Then it mutates the critical fields to generate new seeds to explore the deeper of the path. We currently set up the threshold as 50 based on our experiment results.

Moreover, several message fields in Zigbee are enumerated types with predefined values defined in the Zigbee protocol specification. The protocol checks if such a field has a particular value that requires a specific handling process. Existing protocol fuzzers mutate such a field by the following methods:

(1) randomly selecting values (e.g., selecting any value between [0, 255] if the field is byte type),

(2) enumerating all possible values based on the field size,

(3) selecting values based on their fuzzing dictionary defined according to human heuristics.

Such mutation methods lead to ineffective fuzzing performance. To tackle this problem, we customize the fuzzing dictionary of those message fields by considering their pre-defined values in the protocol specification along with several negative values to reduce the searching space.

The coverage-guided fuzzing engine is developed based on Z-Fuzzer's fuzzing engine that considers the code coverage feedback. We integrate our inference-guided mutator with its fuzzing engine. We utilize the embedded device simulator C-SPY [63] of IAR Workbench to execute the Zigbee protocol stack. We also create a proxy server to enable the connection between the fuzzing engine and the simulator, as the simulator lacks a network interface for sending test messages. According to the static analysis result, we noticed that some functions do not have any callers, which would be used depending on the IoT application vendor's device feature requirements. Thus, we also add corresponding handlers in the source code to fuzz these corner cases.

4.4 Evaluation of TaintBFuzz

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

- **RQ1**: Can TaintBFuzz achieve better fuzzing performance compared to state-of-the-art protocol fuzzers?
- **RQ2**: How efficient is TaintBFuzz at detecting vulnerabilities compared to state-of-the-art protocol fuzzers?

We illustrate the novelty and efficiency of TaintBFuzz in compariosn with three basedline protocol fuzzers, Peach [15], Boofuzz [14], and Z-Fuzzer [71]. Boofuzz is the successor of Sulley [44], an industry-standard protocol fuzzer more actively maintained than Sulley. Both Peach and Boofuzz are open source and have been used in existing research papers [74, 75]. Boofuzz and Peach do not initially work with the

Table 4.1: Fuzzing performance of all fuzzers on Z-stack in 10 runs.

Fuzzer	Unique Test Cases	Stmt Coverage		Edge Coverage	
		total	%	total	%
TaintBFuzz	12,493	1111	**68.88%**	800	**74.42%**
Z-Fuzzer	61,386	971	63.18%	769	71.53%
Boofuzz	16,756	912	59.33%	680	63.26%
Peach	18,271	850	55.30%	628	58.42%

Zigbee protocol. Hence, we incorporated them with our proxy server and simulation platform to send test inputs for Zigbee protocol execution.

All of our experiments were performed on a machine with eight 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 use a widespread Zigbee protocol implementation Z-Stack [61] as the target program, developed by Texas Instruments with various sample project codebases, and its source code is available. From the user's point of view, the ZCL is a protocol that runs at the application layer and serves as the core library for the Zigbee protocol stack. We employ ZCL as a case study in our evaluation. We ran each fuzzer on Z-Stack over 24 hours. All experiments were repeated ten times. We also set the threshold for inference-guided mutation as 50 when compared with other protocol fuzzers.

4.4.1 Fuzzing Performance

To answer **RQ1**, we performed a set of fuzzing experiments on each fuzzer to examine their generated test cases, statement coverage, and edge coverage. The fuzzers produce test cases with the given message format script using the user-specific or predefined fuzzing dictionary, for which the total number of test cases is finite. During our evaluation, we noticed that existing research has incorrect percentage calculations on state-of-the-art fuzzers. Thus, we recalculate them and show in Table 4.1. We report the total number of unique test cases generated by each fuzzer. The average statement coverage and average edge coverage of each fuzzer are also presented in the table. The results show that TaintBFuzz is significantly more effective than state-of-the-art protocol fuzzers.

4.4.1.1 Test Case Generation

We examine the uniqueness of the test cases produced by all fuzzers. TaintBFuzz can achieve higher code coverage than other fuzzers with fewer test cases, especially with five times fewer test cases than Z-Fuzzer, due to the reduced input space of several message fields with the customized fuzzing dictionary. In addition, to differentiate between different fuzzers on test case creation, we classify test cases according to the Zigbee protocol standard using the field *Command Identification* in the ZCL header. TaintBFuzz generated 194 distinct types of test cases in total, of which only 34 of them can be generated by other fuzzers. More than half of these distinct types are generated after mutating the dependent fields in the constraint-field dependency inference.

We also measure how the constraint-field dependency inference impacts the test case generation, i.e. when to consider the dependency inference to augment mutation for generating more diversified test cases. We defined a threshold as the number of mutation times since the last updated code coverage. If the coverage has not been updated after the threshold, TaintBFuzz looks up the constraint-field dependency inference to select an appropriate field for further mutations. The ideal threshold is set to 1, i.e., the constraint-field dependency is considered for each mutation. However, it could result in performance issues when the execution path is very long, which costs much time for TaintBFuzz to search the critical fields for every mutation. In

Table 4.2: Test cases generated by TaintBFuzz for different inference threshold.

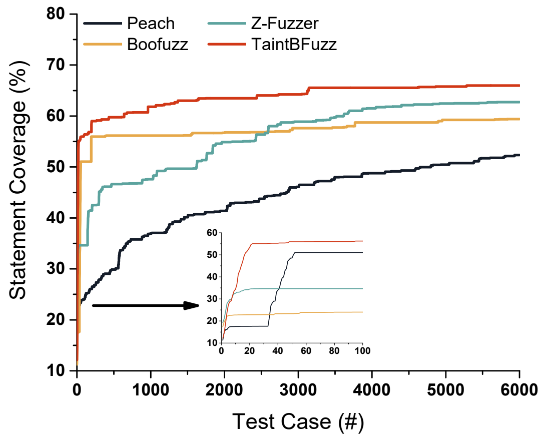
	Threshold=10	**Threshold=25**	**Threshold=50**
Favored Test Cases	50	49	**52**
Test Case Types	36	22	**57**
Type Difference	29	35	(base)

our evaluation, we compared three different threshold values *10, 25, 50* for and found that it performs better when setting the threshold as 50.

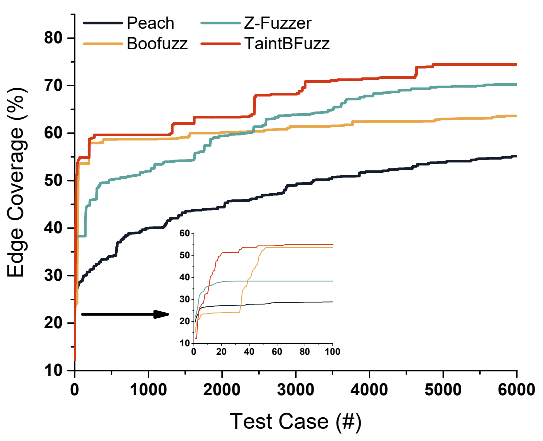
Table 4.2 presents the comparison results. We present the number of the favored test cases and the test cases types categorized based on the field *Command Identifier*. The type difference show the unique types generated in the base set but not in the other two sets. We can see there are more favored test cases generated when the threshold is 50, which provides more candidates for TaintBFuzz to explore new paths in the target program. We also categorized all generated test cases based on the field *Command Identifier*. The result inidicates that there are more different test cases types when threshold setting to 50, in which 29 are not generated by threshold 10 and 35 are not generated by threshold 25, while threshold 50 can generate all types in other two sets. The diversity of generated test cases also provides the fuzzer more probability to access more codes and paths in the target program. Therefore, we also use threshold 50 for the TaintBFuzz's mutation when comparing the fuzzing performance with state-of-the-art fuzzers.

4.4.1.2 Code Coverage

We measure the code coverage on all fuzzers. Peach and Boofuzz cannot directly work with Z-Stack execution, so we integrated them with our protocol simulation platform via the proxy server. As shown in Table 4.1, TaintBFuzz can achieve higher statement coverage and edge coverage with fewer test cases. As we reduced the



(a) Statement Coverage



(b) Edge Coverage

Figure 4.2: Statement coverage and edge coverage achieved by fuzzers over 10 runs.

searching space of several message fields with pre-defined values in the Zigbee protocol specification, TaintBFuzz can efficiently generate test seeds with dependency inference to explore more paths in the target program. Our primary focus is on effectively creating test cases that conform to the Zigbee protocol specification's message format and exploring more normal execution paths. As a result, we cannot fully cover the exception-handling code in the protocol implementation.

Fig 4.2 presents the variation of code coverage of fuzzing in all fuzzers. The X-axis represents the median number of test cases. The Y-axis represents the percentage of statement coverage and edge coverage on average. For better result presentation, we plot the coverage trend of the first 6000 test cases generation to show in Fig 4.2. The zoomed-in graph in the lower left corner display more details about how the code coverage varies in the first 100 test cases. It shows that Boofuzz, Z-Fuzzer and TaintBFuzz quickly proliferated at an early phase. Minor changes in the header can significantly impact the code and path that is performed since the Zigbee protocol first validates a ZCL header before processing any other fields of the message. Peach slowly increased its code coverage because it randomly fuzzed a message field. The other three fuzzers started mutation from the first message field resulting in the rapid code coverage increment in the early phase.

Notably, the coverage increment of TaintBFuzz is the fastest due to the guidance from the constraint-field dependency inference. Boofuzz mutated a single field at a time based on their placement order in the format script, in which the field is reset to the initial value after mutation completes. Therefore, it can enumerate a limited number of ZCL header types. Though Z-Fuzzer leverages code coverage to prioritize the favored test cases for further mutation, it is hard to consider all possible header values by only considering the coverage feedback. For example, a test case whose *Command Identifier* is `0x05` triggers a new edge and is saved as a favored test case for further mutation. In contrast, the field *Frame Control* is reset to the initial value `0x00`. Z-Fuzzer continues fuzzing succeeding fields of *Command Identifier*, which does not explore any new codes. However, a path constraint requires a particular value of *Frame Control* to trigger another branch. With the guidance from the constraint-field dependency inference, TaintBFuzz efficiently generates such a test case to explore the uncovered branch.

Summary. TaintBFuzz's constraint-field dependency inference allows it to attain a greater code coverage rate than Peach, Boofuzz, and Z-Fuzzer. We observed that many ZCL functions handle the message payload value for the higher-level application object. To run more in-depth code in those functions, they could need a test case to meet specific branch conditions. The values of specific message fields, which may meet such a dependence condition, are neglected throughout the fuzzing process by Peach, Boofuzz, and Z-Fuzzer. TaintBFuzz, on the other hand, can deduce such a correlation from the constraint-field dependency inference. The inferred message fields have higher priority for the further mutation to generate test cases, which satisfy those specific requirements and covering more codes and edges.

Table 4.3: Unique vulnerabilities detected all fuzzers over ten fuzzing.

Vulnerability	**Peach**	**Boofuzz**	**Z-Fuzzer**	**TaintBFuzz**
CVE-2020-27890	✗	✗	96	103
CVE-2020-27891	1	57	71	17
CVE-2020-27892	4	10	47	10
zclParseInReportCmd	✗	✗	2	3
zclParseInReadRspCmd	✗	✗	3	2
zclProcessInWriteCmd	2	✗	5	2
zcl_SendReadReportCfgCmd	✗	✗	✗	**2**
zcl_SendCommand	✗	✗	✗	**2**
Total	7	67	224	141

4.4.2 Vulnerability Detection

We measure the number of unique vulnerabilities discovered by all fuzzers to answer **RQ2**. On each fuzzer, we performed the experiments ten times and presented the result in Table 4.3. We present the total amount of test cases triggering the vulnerability on average. The vulnerabilities are distinguished by comparing the call stack and performing manual analysis.

As shown in Table 4.3, TaintBFuzz can detect the known vulnerabilities and two new crashes. We cross-checked the vulnerabilities detected by all fuzzers. Though Z-Fuzzer has generated more test cases for discovering CVE-2020-27891 and CVE-2020-27892 than TaintBFuzz, only 11% of them can be manually reproduced. Instead, most test cases generated by TaintBFuzz for the detected vulnerabilities are reproducible. For CVE-2020-27892, TaintBFuzz has fewer test cases than Z-Fuzzer because we reduced the input space of several message fields in the ZCL payload by customizing the fuzzing dictionary with pre-defined values in the protocol specification. Z-Fuzzer regards these fields as a regular byte or word variable and mutates it with a more extensive fuzzing dictionary, in which many test cases instead have no impact on path exploration and bug detection.

Table 4.4: Dependent constraints and fields for each vulnerability.

Vulnerability	**Constraints & Fields**
CVE-2020-27890	cmdID == 0x05
CVE-2020-27891	fc == 0x08 ∧ cmdID == 0x09
CVE-2020-27892	cmdID ∈ [0x12, 0x14]
zclParseInReportCmd	cmdID == 0x0A ∧ (attrID ∈ [0x7fff, 0x7ff7])
zclParseInReadRspCmd	cmdID == 0x01 ∧ attrID == 0x7ff9
zclProcessInWriteCmd	cmdID == 0x02
zcl_SendReadReportCfgCmd	cmdID == 0x08
zcl_SendCommand	cmdID == 0x08 ∧ (hdr.fc.type == 0x00)

Moreover, TaintBFuzz has detected two new crashes in functions *zcl_SendReadReportCfgCmd* and *zcl_SendCommand*, which are corner cases that have not been tested before in previous research. The root cause is the long list of attribute identifiers whose value is random. In practice, an IoT device may have a few defined features (e.g., less than 20), each having a unique attribute identifier to perform the device functionalities. The protocol vendor usually customized their memory management functions rather than using functions from the standard C library, e.g., Z-Stack use *zcl_mem_alloc()* instead of *malloc()* from `libc`, due to the limited hardware resources on IoT devices. When the attribute list is too long, the protocol stack requires more memory space to process them, which results in memory corruption when allocating space using the above self-implemented memory function. We have also reported these two new crashes to the protocol vendor, which are under review when writing this paper.

We also evaluate how the constraint-field dependency inference assists TaintBFuzz in detecting the vulnerabilities. The constraints and corresponding message fields are shown in Table 4.4, in which fc represents the field *Frame Control*, cmdID represents the field *Command Identifier*, attrID represents the field *Attribute Identifier* as shown in Fig 2.2a in Chapter 2. All vulnerabilities are triggered by messages with random payload values, which also satisfy the listed constraints. We noticed that all detected vulnerabilities are influenced by the message field *Command Identifier*, which is reasonable since the Zigbee protocol takes different message parsers and processors based on the *Command Identifier*. Moreover, for the two newly discovered bugs, mainly the vulnerable function *zcl_SendCommand*, there is a constraint to validate the device operation based on the ZCL message type, which returns failure if not satisfied. TaintBFuzz can generate proper test cases satisfying the constraint with the constraint-field dependency inference, which guides the fuzzer to mutate the field *Frame Control*.

Summary. TaintBFuzz can efficiently discover vulnerabilities compared to state-of-the-art protocol fuzzers for known vulnerabilities and new crashes in Z-Stack. We notice that most vulnerabilities are caused by the memory allocation function developed by the Zigbee protocol vendors, which takes the place of the C library's standard functions. It is difficult for resource-efficient IoT devices to support all C standard APIs because of the hardware and computing power limitation. Such customized system APIs from protocol vendors may bring more potential security risks during the IoT application development, which the developers may not be aware of before releasing their applications. The mitigation of potential security risks now depends on whether the vendors are active or not for the reported issues [69]. This situation is what inspired us to propose this approach to help IoT application developers identify possible security issues in advance during the development phase.

4.5 Conclusion

This chapter presents TaintBFuzz, an intelligent Zigbee protocol fuzzing with constraint-field dependency inference. It first identifies the path constraint variables and generates representative messages based on the Zigbee protocol format specification. Then it leverages static taint analysis to infer which critical message field impacts the constraint variables. Finally, with the constraint-field dependency inference, TaintBFuzz precisely mutates the critical field of constraint variables to explore the uncovered statements. In terms of code coverage, TaintBFuzz outperforms several state-of-the-art protocol fuzzers on a mainstream Zigbee protocol implementation called Z-Stack developed by Texas Instruments. Particularly, TaintBFuzz can identified eight unique vulnerabilities in Z-Stack, two of them are previously unknown.

Chapter 5

Fuzzing Zigbee Protocol Implementation with Combinatorial Testing

5.1 Overview

In practical scenarios, exploiting vulnerabilities in the Zigbee protocol typically necessitates following a specific execution path that involves multiple path constraints. These path constraints are often influenced by combinations of values assigned to various message fields. Fuzz testing [12], also known as fuzzing, is a widely used and effective technique for detecting security vulnerabilities. It involves running the target program with random inputs to identify potential weaknesses. Regrettably, existing fuzzing approaches have shown limited attention to the importance of these input parameter combinations.

Conventional protocol fuzzers [14, 15, 44, 74] generally focus on sequentially or randomly mutating individual message fields. Consequently, they may overlook critical combination values that have a high likelihood of triggering execution failures. Coverage-guided fuzzing approaches [16, 21] only leverage code coverage heuristics to prioritize test cases that exercise new program paths, without taking inter-parameter dependency and program structure into consideration for mutation. Taint-based fuzzing approaches like TaintBFuzz [70] utilize taint analysis to deduce the relationship between input bytes and path constraints, then decide how to mutate those bytes. Although this inference aids in exploring uncharted execution paths, it is unlikely to effectively provoke failures caused by specific combinations of message field values.

The use of Combinatorial Testing (CT) as a prevalent method for testing parameter interactions that influence software behavior has been widely acknowledged [55, 56]. The fundamental concept behind CT is that, for any given set of t parameters in a target program, it is possible to cover every combination of values for these t parameters at least once [55]. The objective is to achieve a favorable balance between the size of the test input space and the efficacy of failure detection. However, when the number of input parameters is substantial, the combinatorial explosion problem can arise, posing a challenge [60]. Consequently, a more intelligent approach is required to generate combination values for significant input parameters, as opposed to exhaustively enumerating all possible combinations.

In this chapter, we introduce CT-BFuzz, a fuzzing platform specifically designed for the Zigbee protocol implementation. CT-BFuzz utilizes combinatorial testing to efficiently generate test cases that cover important combinations of message field values. The main challenge lies in identifying the significant message fields and their corresponding values for combinatorial testing. To address this, we employ static taint analysis to identify message fields that can impact branch conditions. Additionally, we leverage coverage-guided fuzzing to filter out less critical fields, focusing on those that have a higher chance of exploring unvisited branches. We also consider dependent fields defined in the message script, based on the Zigbee protocol specification used for generating the initial valid test corpus. The test seeds generated through combinatorial testing are then prioritized for further mutation.

Furthermore, the fuzzing engine of CT-BFuzz is based on Z-Fuzzer. It generates initial test cases with a given message format script from scratch and employs code coverage heuristics to prioritize test cases that exercise unexplored program paths. By integrating combinatorial testing and coverage-guided fuzzing, CT-BFuzzaims to effectively detect vulnerabilities in Zigbee protocol implementations.

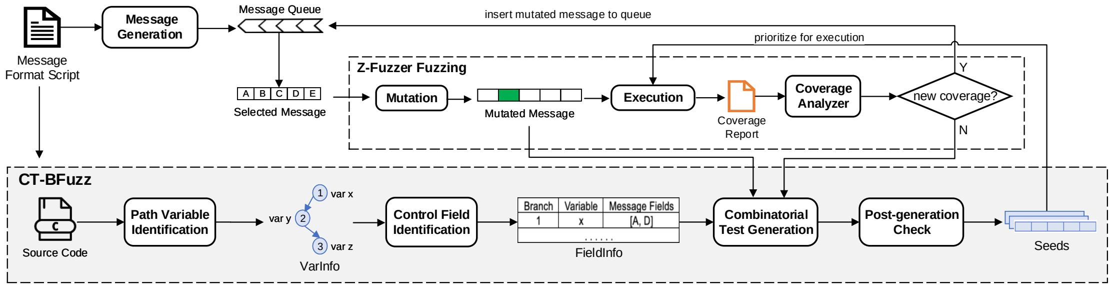


Figure 5.1: Overall design of CT-BFuzz.

We implemented a prototype of CT-BFuzz and evaluated its effectiveness in security vulnerability detection and fuzzing performance on Z-Stack [61]. We compare CT-BFuzz with five state-of-the-art protocol fuzzing tools, Peach [15], Boofuzz [14], Boofuzz with CT mode, Z-Fuzzer [71] and TaintBFuzz [70]. Peach and Boofuzz are conventional protocol fuzzers that have been widely used in existing research articals. The latest version Boofuzz also implements a plugin of combinatorial testing. The experiment results show that CT-BFuzz outperforms other protocol fuzzers. Especially, CT-BFuzz can detect vulnerabilities faster than other fuzzers with fewer test cases.

5.2 Design of CT-BFuzz

The overall design of CT-BFuzz is illustrated in Figure 5.1, comprising four major components. Initially, path variable identification and control field identification are conducted on the Zigbee protocol stack, prior to the fuzzing process. These components are responsible for identifying critical message fields that are essential for combinatorial testing, as they influence the execution of the target program along specific paths. When the fuzzer reaches a point where no new execution paths are being explored over a certain period of time, CT-BFuzz employs CT generation to create new test seeds. These seeds contain critical message fields that possess a high probability of uncovering new paths. Simultaneously, fuzzing assists CT in identifying critical fields and their representative values for dynamically constructing the CT test model. The CT test set is subsequently prioritized for execution, with a focus on maximizing code coverage. Only test cases that have traversed previously unexplored program branches are added to the message queue for further mutation. To provide a more comprehensive understanding of CT-BFuzz's design, we utilize the ZCL (Zigbee Cluster Library) as a case study in the subsequent sub-sections, where we delve into the specific details.

5.2.1 Path Variable Identification

The initial challenge that CT-BFuzz tackles involves identifying the important message fields to be used in combinatorial testing. Not all message fields have an impact on program execution. In practical scenarios, triggering a failure in the execution of the Zigbee protocol necessitates a test message that satisfies a specific execution path, comprising multiple path constraints. These constraints are typically influenced by the values assigned to specific message fields, either individually or in combination. For the purpose of this discussion, we refer to these fields as *control* fields. Selecting appropriate values for these *control* fields becomes critical in order to successfully trigger an execution failure.

To construct the list of *VarInfo*, CT-BFuzz initially collects the program variables used in all path constraints based on code analysis of the target program. A path constraint typically consists of one or more program variables, referred to as *path variables.* These variables can be explicitly or implicitly contained within a path constraint. Explicit variables are directly presented in regular conditional statements such as *IF*, *LOOP*, and *SWITCH.* On the other hand, implicit variables appear in function signatures, where the result of the function call is utilized in a branch condition. For example, in the following statement sequence: *result = FunctionCall(a, b, c); ......; if (result == SUCCESS) ......* , variables a, b, and c are implicit variables associated with the *IF* constraint. During the identification process, CT-BFuzz considers both explicit and implicit variables, ensuring their inclusion in the construction of the *VarInfo* list.

5.2.2 Control Field Identification

Once the path variables have been collected, CT-BFuzz proceeds to identify the *control* fields, i.e., the message fields that influence specific path variables. While existing approaches often rely on dynamic taint analysis (DTA) to pinpoint critical bytes within the test input, direct application of these approaches to Zigbee protocol implementations poses challenges. This is due to the utilization of general compilers like LLVM and Clang, which do not adhere to vendor-specific requirements, as depicted in Figure 1.1. To overcome this limitation, CT-BFuzz employs static taint analysis using a vendor-specific compiler to accurately locate the *control* fields. By leveraging the capabilities of the specific compiler mandated by the Zigbee protocol implementation, CT-BFuzz is able to perform effective static taint analysis and successfully identify the relevant *control* fields.

To identify the relationship between the input values and program variables, CT-BFuzz follows a series of steps. Firstly, a set of ZCL messages is generated as taint sources, using a provided message format script (refer to Figure 2.2b). Instead of generating messages with all possible values, representative messages are created to cover each type of ZCL message. Next, each field in these messages is tainted, and taint analysis is performed. This analysis helps determine the relationship between the input values and program variables. By analyzing the taint propagation, CT-

BFuzz can establish how changes in the input values affect the values of program variables. Based on the collected information in the *VarInfo* data structure, a list of *control* fields is constructed, along with their corresponding impact on specific path constraints. For example, in Table *FieldInfo* illustrated in Figure 5.1, it is evident that message fields *A* and *D* influence the execution of branch *1*. These steps enable CT-BFuzz to effectively identify the *control* fields and their relationships with specific path constraints, facilitating further analysis and testing of the Zigbee protocol implementation.

5.2.3 Fuzzing with Combinatorial Testing

A critical challenge in combinatorial testing lies in the selection of appropriate *control* fields and their representative values. This selection is crucial for generating new test seeds that effectively cover the desired combinations. However, when the number of fields and their potential values is large, combinatorial testing encounters the problem of combinatorial explosion. For instance, in the case of ZCL message format, which encompasses 30 different fields with variable lengths, considering just two possible values for each field would result in an exhaustive combination of 2^{30} test cases. This vast number of test cases often leads to redundancy, as many of them exhibit similar behavior.

To address this challenge, CT-BFuzz implements an adaptive strategy that dynamically generates CT test models for new test seed generation. This strategy takes into account the list of identified *control* fields and the execution results. By employing this adaptive approach, CT-BFuzz can intelligently generate test seeds with appropriate combinations of *control* field values. Furthermore, the CT test set collaborates with the fuzzing process to generate new test cases that explore untouched execution paths. This synergy between combinatorial testing and fuzzing enhances

Algorithm 4: Fuzzing with Combinatorial Testing

```
Input  : Message format script: $\mathcal{S}$, Timeout: $timeout$,
         Mutation threshold: $threshold$, Program under test: $\mathcal{P}$
Output: Detected crash: $crash$
1  $execPath \leftarrow \emptyset$
2  $cov \leftarrow 0$
3  $crash \leftarrow \emptyset$
4  $queue \leftarrow$ MessageGeneration($\mathcal{S}$)
5  $fieldInfo \leftarrow$ ControlFieldIdentification($\mathcal{S}, \mathcal{P}$)
6  while not timeout do
7      $message \leftarrow$ Select($queue$)
8      $mutated \leftarrow$ Mutate($message$)
9      $cov, execPath, crash \leftarrow$ ExecCheckCoverage($mutated, \mathcal{P}$)
10     if noCovUpdate ($cov, threshold$) then
11         $script \leftarrow$ GenerateScript($fieldInfo, mutated, execPath$)
12         $seeds \leftarrow$ CTGeneration($script$)
13         $seeds \leftarrow$ PostCheck($seeds$)
14         $cov, crash \leftarrow$ Execution($seeds, \mathcal{P}$)
15         if $seeds' \leftarrow$ newCoverage($cov, seeds$) then
16             $queue \leftarrow queue \cup seeds'$
17         end
18     end
19     else if isInteresting($cov$) then
20         $queue \leftarrow queue \cup mutated$
21     end
22 end
23 return $crash$
```

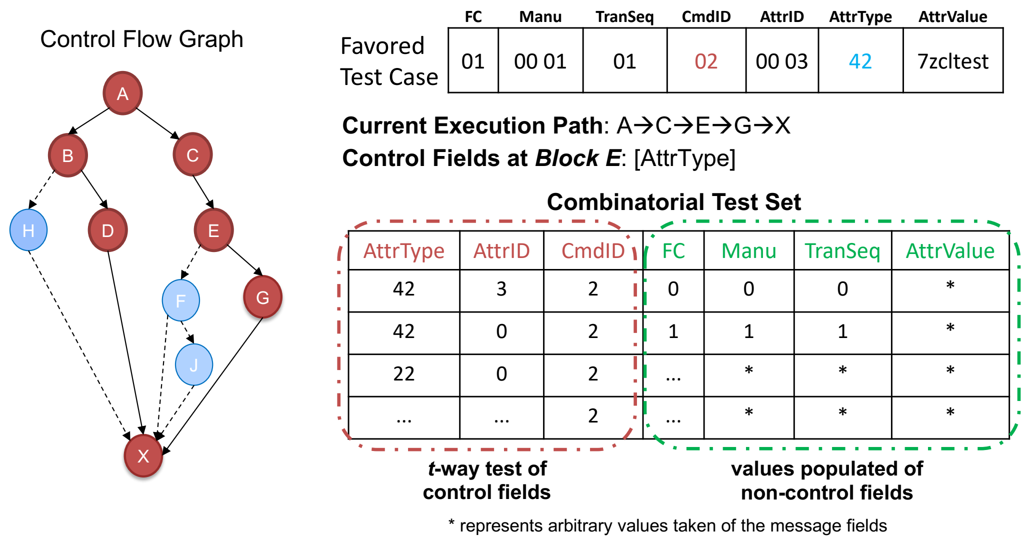
the overall effectiveness of test case generation and aids in exercising unexplored program behaviors.

Algorithm 4 outlines the process of testing Zigbee protocol implementation by leveraging both coverage-guided fuzzing and combinatorial testing. Coverage-guided fuzzing is employed initially to cover easily reachable branches (lines 6-9 and 19-21). If the fuzzing engine fails to explore any new execution paths within a predefined threshold of mutation attempts (lines 10-18), the algorithm switches to combinatorial testing. During combinatorial testing, the coverage-guided fuzzing component assists in selecting suitable message fields and their representative values for constructing a test model. This involves identifying a message field and its corresponding value from a favored test case that triggers new code coverage.

Combinatorial testing aids the coverage-guided fuzzing component in covering combinations of important message fields that are challenging to achieve through random mutation. The generated combinatorial testing (CT) test seeds are prioritized for execution based on their code coverage. It is important to note that not every CT test case requires mutation, as some may exhibit similar execution behaviors. Only test cases that cover new program branches are added to the message queue for further mutation (lines 15-17). This selective approach optimizes the use of resources by focusing on test cases that yield new program coverage.

5.2.3.1 Combinatorial Test Generation

The process of creating CT test models for each category of the ZCL message format can be time-consuming and labor-intensive. To address this challenge, CT-BFuzz leverages coverage-guided fuzzing to dynamically generate test models based on the current favored test case, execution path, and the identified *control* fields. By analyzing the favored test case, CT-BFuzz identifies the *control* fields associated with the current execution path covered by the test case. These *control* fields are then selected for combinatorial testing, as they play a crucial role in influencing the program's behavior. Furthermore, there are inter-field dependencies within the ZCL message format, which can impact the execution path. The Zigbee protocol specification already defines several field dependencies for each message format category [48], and these dependencies have been incorporated into the message format script used


Control Flow Graph
A
B
C
H
D
E
F
G
J
X
Favored Test Case
Current Execution Path: A→C→E→G→X
Control Fields at Block E: [AttrType]
Combinatorial Test Set
t-way test of control fields
values populated of non-control fields
* represents arbitrary values taken of the message fields

	FC	Manu	TranSeq	CmdID	AttrID	AttrType	AttrValue
Favored Test Case	01	00 01	01	02	00 03	42	7zcltest

AttrType	AttrID	CmdID	FC	Manu	TranSeq	AttrValue
42	3	2	0	0	0	*
42	0	2	1	1	1	*
22	0	2	...	*	*	*
...	...	2	...	*	*	*

Figure 5.2: An example of message fields used for combinatorial testing.

by CT-BFuzz. Consequently, CT-BFuzz includes the dependent fields of the current favored test case as critical fields for combinatorial testing as well.

Figure 5.2 illustrates an example of how CT-BFuzz selects message fields for combinatorial testing based on the current fuzzing result. Let's consider a scenario where the fuzzer is currently fuzzing the message field `AttrValue` of the current favored test case, but has not discovered any new execution paths after a certain period of time. The favored test case exercises a specific execution path A→C→E→G→X, where each character represents a basic block and X denotes the exit block.

To explore deeper along this execution path, CT-BFuzz performs the following steps: (1) Identifying the last uncovered block: CT-BFuzz analyzes the control flow graph and coverage report to determine the last covered block (excluding the exit block X) that contains an uncovered branch. In the example, *Block G* is the last covered block, and it has an uncovered branch leading to *Block F*. (2) Searching for control fields: CT-BFuzz searches for the *control* fields of the predecessor block E of *Block F* in the list of *control* fields. In this case, the message field `AttrType` is identified as a *control* field. (3) Continuing the search: CT-BFuzz continues searching for the *control* fields of the successor blocks of *Block F* until it reaches the exit block *X* (e.g., *Block J* in the example). This process helps to collect a list of *control* fields of potential paths rather than an individual uncovered basic blocks.

In addition to the above steps, CT-BFuzz examines the message field format of the current favored test case. If there are any *dependent* fields whose current values must be retained, such fields are also included in the combinatorial test generation. For instance, in the example, the message field `CmdID` is a *dependent* field specified in ZCL specification that decides the ZCL payload. Therefore it is also regarded as *control* field and included in the generation of combinatorial tests.

For each identified message fields, CT-BFuzz will use one of the following strategies to determine its value domains, i.e., the set of possible values that can take:

- **Specification**: using values described in the Zigbee protocol specification. Specifically, if a message field is associated with an `enum` type, the corresponding values specified in message format script are directly used as its value domain.
- **Dependent**: using specified values according to current favored test case and the execution path. This value domain could be either defined in protocol specification (e.g., a single value or small sub-set of enumeration values), or selected during the mutation process to generate current favored test cases.
- **Random**: generating four different values at random as the value domain. This strategy will be used for message field like `AttrValue` in Figure 5.2 whose type is usually *string* type.
- **No Values**: using a small range of numbers based on the defined length. This strategy will be used if no concret values are specified in the protocol specification. For example, `AttrID` is defined with `word` type. Rather than enumerating $2^1 6$ values of this field, a small set of values including current value of the fa-

vored test case, the first 10 numbers and several boundary numbers, are used as its value domain.

Once the *control* fields and their value domains are identified, CT-BFuzz applies *t-way* combinatorial testing to these fields. The goal is to generate combinations of values for the *control* fields, as combinations of these fields are more critical than non-control fields in triggering specific program behaviors. In the example shown in Figure 5.2, the message fields `AttrType`, `AttrID`, and `CmdID` are selected for *2-way* combinatorial test generation. To construct complete test seeds, CT-BFuzz adds the representative values of the non-control fields to the combinatorial test set. These values ensure that the generated test cases cover the complete message format. The order of the message fields is adjusted based on the message format script to maintain consistency and accurately reflect the structure of the message.

5.2.3.2 Post-Generation Check

The limitation of combinatorial testing (CT) in dynamically generating message fields based on dependent fields is a significant challenge. Currently, most CT applications only support simple arithmetic-related constraints. Once the message fields are defined in the testing script from scratch for CT, all of them are used to generate the test set. To overcome this limitation, CT-BFuzz incorporates a post-generation check process to ensure the validity of the generated test set. After the combinatorial test set is generated, CT-BFuzz performs a post-generation check on each test case to reconstruct valid ZCL messages based on the given message format script. This check takes into account the dependencies between message fields and dynamically includes/excludes fields as required.

For example, in Figure 2.2a, the message field `Manufacturer Code` is only included in a ZCL message if a particular bit of the message field `Frame Control` is set to 1 [48]. Otherwise, it should be omitted. The post-generation check identifies such dependencies and ensures that the generated test cases conform to the message format definition. By performing this post-generation check, CT-BFuzz guarantees that the test set consists of valid ZCL messages that adhere to the message format specifications. This ensures the accuracy and reliability of the generated test cases for further testing and evaluation.

5.3 Implementation of CT-BFuzz

In this section, we discuss some major decisions in the implementation of CT-BFuzz, including static taint analysis using Frama-C [76], CT test generation using ACTS [77] and fuzzing engine using Z-Fuzzer [71]. The source code of CT-BFuzz is avaliable at `https://github.com/zigbeeprotocol/ctbfuzz`.

We use Frama-C, an open-source platform designed for source-code analysis of C software to analyze the source code of target Zigbee protocol stack. To identify the path variables, we first preprocess the protocol stack's source code using IAR compiler, which is a commercial compiler of IAR Embedded Workbench [62] and used by many Zigbee protocol vendors. We also customize Frama-C by implementing a new plugin script using Ocaml for AST analysis and path variable collection. Particularly, the new plugin script can handle IAR-specific and architecture-specific syntax that are not recoginzed by existing analysis tools.

Frama-C is also used for static taint analysis to distinguish the *control* fields. We first generate a set of ZCL messages in which each message represents a unique type of ZCL messages using the given message format script. The script is manually constructed with *block-based protocol representation* used by many protocol fuzzers like Boofuzz [14]. Then every message field of generated test messages is labled as a taint source for static taint analysis. Based on the path variable result and taint analysis result, we finally identify the *control* fields that could influence the program execution.

We utilize ACTS (Automated Combinatorial Testing for Software) [77], a popular testing tool to generate *t-way* combinatorial test sets. ACTS offers efficient algorithms and techniques to handle the combinatorial explosion problem. By leveraging ACTS, CT-BFuzz can intelligently select appropriate *control* fields and their representative values for combinatorial testing, ensuring coverage of important field combinations. When generating the combinatorial test sets, the default strength of *control* fields is set to 2 that has been applied by many existing CT applications [55, 56, 59].

The fuzzing engine is implemented based on Z-Fuzzer's fuzzing engine that leverages grammar-based fuzzing with code coverage heuristics to generate high-quality test cases. It is also a device-agnostic fuzzing platform that satisfies underlying hardware requirements from most Zigbee protocol vendors to simulate the protocol stack execution by using IAR simulator [62]. We modify its fuzzing process to launch combinatorial testing when the fuzzer cannot explore any new execution path after a period of time. Here we define a threshold of 50 mutation times to represent this timeout.

5.4 Evaluation of CT-BFuzz

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

- **RQ1**: Can CT-BFuzz achieve better fuzzing performance compared to state-of-the-art protocol fuzzers?
- **RQ2**: How efficient is CT-BFuzz at detecting vulnerabilities compared to state-of-the-art protocol fuzzers?

We illustrate the efficiency of CT-BFuzz in compariosn with five basedline protocol fuzzers, Peach [15], Boofuzz [14], Boofuzz with CT mode, Z-Fuzzer [71] and TaintBFuzz [70]. Boofuzz is the successor of Sulley [44], an industry-standard protocol fuzzer more actively maintained than Sulley. The latest version of Boofuzz also implements a plugin of combinatorial testing [78]. According existing CT applications [55, 56, 59], we set the default strength of CT in Boofuzz and CT-BFuzz to 2. Boofuzz and Peach do not initially work with the Zigbee protocol. Hence, we incorporated them with our proxy server and simulation platform to send test inputs for Zigbee protocol execution.

All of our experiments were performed on a machine with eight 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 use a widespread Zigbee protocol implementation Z-Stack [61] as the target program, developed by Texas Instruments with various sample project codebases, and its source code is available. From the user's point of view, the ZCL is a protocol that runs at the application layer and serves as the core library for the Zigbee protocol stack. We employ ZCL as a case study in our evaluation. We ran each fuzzer on Z-Stack over 24 hours. All experiments were repeated ten times. We also set a threshold as 50 of mutation times to represent the period of time for calling CT when the fuzzer cannot explore any new execution path.

5.4.1 Fuzzing Performance

To answer **RQ1**, we performed a set of fuzzing experiments on each fuzzer to examine their generated test cases, statement coverage, and edge coverage. The fuzzers produce test cases with the same message format script constructed based on ZCL specification. Table 5.1 provides the results of the fuzzing experiments comparing

Table 5.1: Fuzzing performance of all fuzzers on Z-stack in 10 runs.

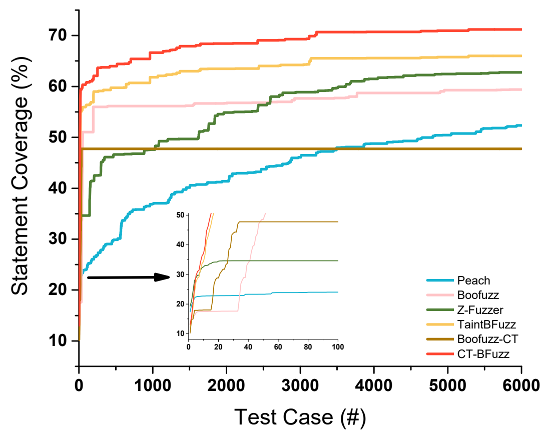
Fuzzer	Unique Test Cases	Statement Coverage		Edge Coverage	
		total	%	total	%
CT-BFuzz	10,584	1147	71.13%	816	75.91%
TaintBFuzz	12,493	1111	68.88%	800	74.42%
Z-Fuzzer	61,386	971	63.18%	769	71.53%
Boofuzz-CT	111	761	49.51%	511	47.53%
Boofuzz	16,756	912	59.33%	680	63.26%
Peach	18,271	850	55.30%	628	58.42%

CT-BFuzz with other state-of-the-art protocol fuzzers. It is observed that CT-BFuzz outperforms the other state-of-the-art protocol fuzzers in terms of effectiveness.

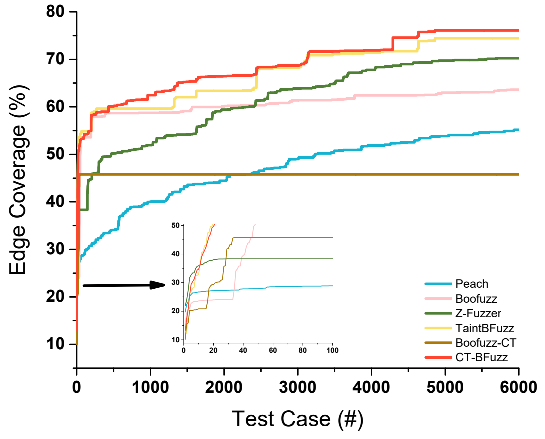
Test Case Generation. We examine the uniqueness of the test cases produced by all fuzzers. CT-BFuzz can achieve higher code coverage than other fuzzers with fewer test cases, especially with five times fewer test cases than Z-Fuzzer. With combinatorial testing, it efficiently generates test cases to covering important combination values of message fields rather than enumerating all possible comibinations.

In addition, to differentiate between different fuzzers on test case creation, we classify test cases according to the Zigbee protocol standard using the field *Command Identification* in the ZCL header. CT-BFuzz generated 13 new types of test cases that are not covered by other fuzzers. Particularly, half of the new types are new combinations of the message fields *Attribute Identifier* and *Attribute Type*, which exercises new execution paths.

The superior performance of CT-BFuzz can be attributed to the combination of combinatorial testing and fuzzing techniques. By leveraging combinatorial testing, CT-BFuzz intelligently selects critical message fields and their representative values, allowing for effective exploration of the input space. The integration of coverage-



(a) Statement Coverage



(b) Edge Coverage

Figure 5.3: Statement coverage and edge coverage achieved by fuzzers over 10 runs.

guided fuzzing further enhances the test case generation process by prioritizing inputs that are likely to uncover new execution paths.

Code Coverage. We measure the code coverage on all fuzzers. Peach and Boofuzz cannot directly work with Z-Stack execution, so we integrated them with our protocol simulation platform via the proxy server. The code coverage results in Table 5.1 indicate that CT-BFuzz achieves higher statement coverage and edge coverage compared to the other fuzzers, even with fewer test cases.

Figure 5.3 provides a visual representation of the code coverage trends for each fuzzer. The X-axis represents the median number of test cases generated, while the Y-axis represents the percentage of statement coverage and edge coverage on average. For better result presentation, we plot the coverage trend of the first 6000 test cases generation to show in Fig 5.3. The zoomed-in graph in the lower left corner display more details about how the code coverage varies in the first 100 test cases. It shows that Boofuzz, Z-Fuzzer TaintBFuzz and CT-BFuzz quickly proliferated at an early phase. Minor changes in ZCL header can significantly impact the execution path that is performed since the Zigbee protocol first validates a ZCL header before processing any other fields of the message. Peach slowly increased its code coverage because it randomly fuzzed a message field, . The other four fuzzers started mutation from the first message field resulting in the rapid code coverage increment in the early phase.

Peach focuses on generating test cases that conform to the message format definition specified in the protocol specification. However, its random mutation strategy and lack of consideration for program execution heuristics result in lower code coverage compared to CT-BFuzz. Similarly, Boofuzz, in its original version, also exhibits lower code coverage due to its sequential mutation of a single message field. Moreover, in theory, the CT mode in Boofuzz should provide better code coverage by incorporating combinatorial testing. However, the CT mode of Boofuzz has limitations in considering inter-field dependencies defined in the message format script. As a result, it generates duplicate and invalid test cases that are rejected by the protocol stack at an earlier execution stage. We have report this issue to Boofuzz's developers for review.

While Z-Fuzzer prioritizes test cases that explore new execution paths, it lacks consideration for the target program structure. This may result in a less systematic exploration of the execution paths and potentially limit its code coverage. On the other hand, TaintBFuzz utilizes taint analysis to infer the relationship between message fields and path constraints, allowing it to guide fuzzing towards unexplored execution paths. However, similar to Z-Fuzzer, it focuses on mutating a single message field at a time and may not fully explore the combinations of multiple message fields.

In contrast, CT-BFuzz places a strong emphasis on the combinations of *control* fields through combinatorial testing. By considering the interdependencies of message fields and applying combinatorial testing techniques, CT-BFuzz generates diversified test cases that can explore a greater number of unexplored execution paths. This

Table 5.2: Number of messages generated by all fuzzers for triggering vulnerabilities.

Vulnerability	**Peach**	**Boofuzz**	**Boofuzz-CT**	**Z-Fuzzer**	**TaintBFuzz**	**CT-BFuzz**
CVE-2020-27890	✗	✗	✗	96	103	97
CVE-2020-27891	1	57	✗	71	17	33
CVE-2020-27892	4	10	1	47	10	21
zclParseInReportCmd	✗	✗	✗	2	3	6
zclParseInReadRspCmd	✗	✗	✗	3	2	3
zclProcessInWriteCmd	2	✗	✗	5	2	2
zcl_SendReadReportCfgCmd	✗	✗	✗	✗	2	5
zcl_SendCommand	✗	✗	✗	✗	2	4
Total	7	67	1	224	141	171

approach leads to higher code coverage even with fewer test cases compared to Z-Fuzzer and TaintBFuzz.

5.4.2 Vulnerability Detection

We measure the number of unique vulnerabilities discovered and time consumption by all fuzzers to answer **RQ2**. On each fuzzer, we performed the experiments ten times and presented the result in Table 5.2. We present the total amount of test cases triggering the vulnerability on average. The vulnerabilities are distinguished by comparing the call stack and performing manual analysis.

Detected Vulnerability. As shown in Table 5.2, CT-BFuzz has shown effectiveness in detecting existing vulnerabilities with a higher number of test cases compared to other fuzzers. We cross-checked the vulnerabilities detected by all fuzzers. Though Z-Fuzzer has generated more test cases for discovering CVE-2020-27891 and CVE-2020-27892 than CT-BFuzz, only 11% of them can be manually reproduced, indicating a higher false positive rate. On the other hand, a significant portion of the test cases generated by CT-BFuzz for the detected vulnerabilities are reproducible, demonstrating the reliability of the generated test cases. In the case of CVE-2020-27892, CT-BFuzz generated fewer test cases compared to Z-Fuzzer because it focuses on generating combinations of important message fields, while Z-Fuzzer uses a more

Table 5.3: Time consumption (mins) of triggering vulnerabilities by all fuzzers.

Vulnerability	**Peach**	**Boofuzz**	**Boofuzz-CT**	**Z-Fuzzer**	**TaintBFuzz**	**CT-BFuzz**
CVE-2020-27890	-	-	-	102	93	85
CVE-2020-27891	69	259	-	713	91	100
CVE-2020-27892	74	352	2	1044	309	300
zclParseInReportCmd	-	-	-	141	127	98
zclParseInReadRspCmd	-	-	-	259	274	169
zclProcessInWriteCmd	673	-	-	64	78	81
zcl_SendReadReportCfgCmd	-	-	-	-	235	120
zcl_SendCommand	-	-	-	-	234	123

extensive fuzzing dictionary to mutate the fields, resulting in many irrelevant test cases that do not contribute to path exploration and bug detection. We also observe that CT-BFuzz exhibits the capability to generate a wider variety of test cases than other protocol fuzzers, particularly when categorizing them based on the message fields *Command Identifier*. For example, for the vulnerability in function *zclParseInReportCmd*, CT-BFuzz generate two unique combinations of the message fields *Attribute Identifier* and *Attribute Data Type* that are not generated by Z-Fuzzer and TaintBFuzz.

Time Consumption. Additionally, we evaluate the time consumption of all fuzzers for triggering each vulnerability. The result is presented in Table 5.3. We report the minium spending time of every fuzzer for detecting the vulnerabilities over ten fuzzing runs. As the execution engine needs to be restarted for each test case to generate the coverage report, this execution time is also included in the spending time.

Peach and Boofuzz-CT outperform other fuzzers in terms of time consumption for CVE-2020-27892, which can be triggered when the field *Command Identifier* field is set to `0x12` or `0x14`. Specifically, Boofuzz-CT detects this vulnerability in just 2 minutes. This exceptional performance is attributed to the specific mutation strategy employed by Boofuzz-CT, where it retains the *Frame Control* field and mutates only the *Command Identifier* field when the CT strength is set to 2. This strategy targets the specific conditions required to trigger the vulnerability and leads to faster detection.

Apart from CVE-2020-27892, CT-BFuzz demonstrates superior efficiency in detecting most vulnerabilities compared to other protocol fuzzers. Instead of mutating every CT test case, CT-BFuzz executes all CT test cases together and evaluates their code coverage. Only the test cases that cover new program branches are selected for further mutation and insertion into the message queue. This approach saves time by focusing on generating new and impactful test cases. Differently, Boofuzz, Z-Fuzzer, and TaintBFuzz evaluate every test case individually in terms of code coverage and decide how to mutate them. This process can be more time-consuming compared to CT-BFuzz's approach of selectively mutating test cases that contribute to new code coverage. Consequently, CT-BFuzz achieves faster detection of vulnerabilities compared to Z-Fuzzer and TaintBFuzz due to its efficient test case selection and mutation strategy.

5.5 Conclusion

In summary, this chapter introduces the CT-BFuzz approach, which combines combinatorial testing and fuzzing to effectively test Zigbee protocol implementations. By leveraging static taint analysis and fuzzing techniques, CT-BFuzz identifies important message fields and their representative values, which are then used to generate CT test models. These models are further utilized to generate diversified test cases, with a focus on exploring the combination values of *control* fields that have a higher likelihood of uncovering unexplored execution paths.

The evaluation of CT-BFuzz on the widely used Zigbee protocol implementation, Z-Stack, showcases its effectiveness and efficiency compared to state-of-the-art protocol fuzzing tools. The experimental results demonstrate that CT-BFuzz outperforms other fuzzers in terms of code coverage, vulnerability detection, and time consumption. By systematically generating diversified test cases and focusing on combination values of important message fields, CT-BFuzz improves the overall testing quality and efficiency for Zigbee protocol implementations.

Overall, the CT-BFuzz approach presents a valuable contribution to the field of protocol testing, specifically for Zigbee protocols, and showcases the benefits of integrating combinatorial testing and fuzzing techniques to enhance testing effectiveness and efficiency.

Chapter 6

Conclusion

In this dissertation, the main objective was to tackle the challenges in security analysis of Zigbee protocol by applying fuzz testing to Zigbee protocol implementations. Three different approaches were presented to address this goal: Z-Fuzzer, TaintBFuzz, and CT-BFuzz.

The first approach, Z-Fuzzer, introduced a device-agnostic fuzzing platform specifically designed for Zigbee protocol implementations. It utilized grammar-based fuzzing techniques along with code coverage heuristics to effectively discover vulnerabilities. Experimental results demonstrated the effectiveness and efficiency of Z-Fuzzer compared to existing protocol fuzzing tools. Notably, six unique vulnerabilities were discovered in a mainstream Zigbee protocol stack, with three of them assigned CVE IDs and evaluated as high severity.

The second approach, TaintBFuzz, presented an intelligent fuzzing solution that focused on inferring the relationship between message fields and path constraints. By understanding the constraint-field dependency, TaintBFuzz could precisely mutate critical message fields and explore unexplored program branches. Experimental results showed that TaintBFuzz outperformed state-of-the-art protocol fuzzers in terms of code coverage and vulnerability detection. Additionally, it identified two new crashes in a mainstream Zigbee protocol stack.

The third approach, CT-BFuzz, introduced a fuzzing approach that systematically generated diversified test cases to cover important combination values of message fields. This approach utilized static taint analysis and fuzzing techniques to identify significant message fields and their representative values, enabling the dynamic generation of Combinatorial Testing (CT) test models. The CT test set enhanced the diversity of the fuzzing process, particularly targeting combination values of *control* fields to explore unexplored execution paths. Evaluations conducted on a widely used Zigbee protocol implementation demonstrated the superiority of CT-BFuzz over existing protocol fuzzing tools.

The work presented in this dissertation could be extended along in several directions. (1) **Dynamic Combinatorial Testing on IoT Wireless Protocols**: The existing combinatorial testing applications are hard to generate test cases, in which some input parameters are dynamically constructed depending on another input parameter's value. It requires further studies to generate adoptive method including such dynamic insertation or deletion constraints for efficiently construct CT test models. (2) **Fuzzing on Other IoT Wireless Protocols**: Besides the Zigbee protocol, the proposed fuzzing solutions would be applicable to other IoT wireless protocols used for resource constrained devices. It would be interesting to investigate the technical challenges to fuzzing those protocol implementations, such as Z-Wave, NB-IoT, and LoRa. In addition, many IoT devices have extended multi-protocols wireless MCUs, like BLE with Zigbee, WiFi with Zigbee, and BLE with Thread. It would be another possible research area to detect security problems in mult-protocols using fuzz testing.

Overall, the dissertation presented three novel approaches that addressed the challenges in security analysis of Zigbee protocol through fuzz testing. These approaches showcased the effectiveness, efficiency, and superior performance of the proposed solutions compared to existing state-of-the-art protocol fuzzing tools. The discoveries of multiple vulnerabilities and crashes in mainstream Zigbee protocol implementations further validated the importance and impact of the research.

References

[1] A. M. Research, “IoT Device Market Expected to Reach $413.7 Billion By 2031,” https://www.globenewswire.com/news-release/2022/08/08/2493893/0/en/IoT-Device-Market-Expected-to-Reach-413-7-Billion-By-2031-Allied-Market-Research.html, 2022.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, “Understanding the Mirai Botnet,” in *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[3] C. Cimpanu, “Over 65,000 Home Routers Are Proxying Bad Traffic for Botnets, APTs,” https://www.bleepingcomputer.com/news/security/over-65-000-home-routers-are-proxying-bad-traffic-for-botnets-apts/, April 12, 2018.

[4] D. Lodge, “Steal your Wi-Fi key from your doorbell? IoT WTF!” https://www.pentestpartners.com/security-blog/steal-your-wi-fi-key-from-your-doorbell-iot-wtf/, 2016.

[5] E. Turjeman, “Threat Spotlight: IoT application vulnerabilities leave IOT devices open to attack,” https://blog.barracuda.com/2019/01/24/threat-spotlight-iot-application-vulnerabilities/, January 24, 2019.

[6] T. Ricker, “ZIGBEE ON MARS!” https://www.theverge.com/2021/5/20/22445330/zigbee-on-mars-ingenuity-helicopter-perseverance-rover, 2021.

[7] BusinessWire, "Analysts Confirm Half a Billion Zigbee Chipsets Sold, Igniting IoT Innovation; Figures to Reach 3.8 Billion by 2023," https://www.businesswire.com/news/home/20180807005170/en/Analysts-Confirm-Half-a-Billion-Zigbee-Chipsets-Sold-Igniting-IoT-Innovation-Figures-to-Reach-3.8-Billion-by-2023, 2018.

[8] E. Ronen, C. O'Flynn, A. Shamir, and A.-O. Weingarten, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," in *Proceedings ot the 38th IEEE Symposium on Security and Privacy (S&P '17).* Piscataway, NJ, USA: IEEE, 2017, pp. 195–212.

[9] P. Morgner, S. Mattejat, Z. Benenson, C. Müller, and F. Armknecht, "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).* New York, NY, USA: Association for Computing Machinery, 2017, p. 230–240.

[10] J. Wang, Z. Li, M. Sun, and J. C. Lui, "Zigbee'sNetwork Rejoin Procedure for IoT Systems: Vulnerabilities and Implications," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '22).* New York, NY, USA: Association for Computing Machinery, 2022.

[11] C. Vulnerabilities and Exposures, "Zigbee CVE Records," https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=zigbee, 2022.

[12] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery.* London, UK: Pearson Education, 2007.

[13] D. Aitel, "An Introduction to SPIKE, the Fuzzer Creation Kit," BlackHat USA, 2002.

[14] J. Pereyda, "Boofuzz: Network Protocol Fuzzing for Humans," https://boofuzz.readthedocs.io/en/latest/, 2020.

[15] P. Tech, "Peach Fuzzer: Discover unknown vulnerabilities," https://www.peach.tech/, [online].

[16] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl, 2015.

[17] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19).* Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1099–1114.

[18] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20).* Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 19–36.

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

[20] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41. Vancouver, BC: USENIX Association, 2005, p. 46.

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT'20).* Berkeley, CA, USA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[22] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A Greybox Fuzzer for Network Protocols," in *Proceedings of the IEEE 13th International Conference*

on Software Testing, Validation and Verification (ICST'20). Piscataway, NJ, USA: IEEE, 2020, pp. 460–465.

[23] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P '18)*. Piscataway, NJ, USA: IEEE, 2018, pp. 711–725.

[24] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[25] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices," in *Proceedings of the 25th Annual Network and Distributed Systems Security Symposium (NDSS'18)*. San Diego, CA, USA: Network and Distributed Systems Security Symposium, 2018.

[26] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, *et al.*, "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)*. San Diego, CA, USA: Network and Distributed Systems Security Symposium, 2014.

[27] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *Proceedings of the 24th Network and Distributed Systems Security Symposium (NDSS '17)*, vol. 17. San Diego, CA, USA: Network and Distributed Systems Security Symposium, 2017, pp. 1–14.

[28] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20).* Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 2577–2594.

[29] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence." in *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS '19)*, vol. 19. San Diego, CA, USA: Network and Distributed Systems Security Symposium, 2019, pp. 1–15.

[30] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "PATA: Fuzzing with Path Aware Taint Analysis," in *Proceediings of the 43rd IEEE Symposium on Security and Privacy (S&P '22).* Piscataway, NJ, USA: IEEE, 2022, pp. 154–170.

[31] D. Gislason, *Zigbee Wireless Networking, 1st Edition.* London, UK: Newnes, 2008.

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

[33] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20).* Berkeley, CA, USA: USENIX Association, Aug. 2020, pp. 1201–1218.

[34] Z. Alliance, "Zigbee Specification," https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf, August 5, 2015.

[35] T. Instruments, "Z-Stack 3.0 Developer's Guide," https://software-dl.ti.com/simplelink/esd/plugins/simplelink_zigbee_sdk_plugin/1.60.00.14/docs/zigbee_user_guide/html/zigbee/developing_zigbee_applications/z_stack_developers_guide/z-stack-overview.html, 2006.

[36] J. Mikulskis, J. K. Becker, S. Gvozdenovic, and D. Starobinski, "Snout - An Extensible IoT Pen-Testing Tool," Poster presented at: the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19), 2019.

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

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

[39] D.-G. Akestoridis, M. Harishankar, M. Weber, and P. Tague, "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)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 77–88. [Online]. Available: https://doi.org/10.1145/3395351.3399363

[40] X. Ma, Q. Zeng, H. Chi, and L. Luo, "No More Companion Apps Hacking but One Dongle: Hub-Based Blackbox Fuzzing of IoT Firmware," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services (MobiSys '23)*, ser. MobiSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 205–218. [Online]. Available: https://doi.org/10.1145/3581791.3596857

[41] B. Cui, S. Liang, S. Chen, B. Zhao, and X. Liang, "A Novel Fuzzing Method for Zigbee based on Finite State Machine," *International Journal of Distributed Sensor Networks*, vol. 10, no. 1, p. 762891, 2014.

[42] B. Cui, Z. Wang, B. Zhao, and X. Liang, "CG-Fuzzing: A Comprehensive Fuzzy Algorithm for ZigBee," *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 23, no. 3-4, pp. 203–215, 2016.

[43] X. Wang and S. Hao, "Don't kick over the beehive: Attacks and security analysis on zigbee," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).* New York, NY, USA: Association for Computing Machinery, 2022, p. 2857–2870. [Online]. Available: https://doi.org/10.1145/3548606.3560703

[44] G. Devarajan, "Unraveling SCADA Protocols: Using Sulley Fuzzer," Defon 15 Hacking Conference, 2007.

[45] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated Network Protocol Fuzzing Framework," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 10, no. 8, p. 239, 2010.

[46] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: Toward A Stateful Network Protocol Fuzzer," in *Proceedings of the 9th International Conference on Information Security (ISC'06).* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 343–358.

[47] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08).* New York, NY, USA: Association for Computing Machinery, 2008, p. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[48] Z. Alliance, "Zigbee Cluster Library Specification," https://zigbeealliance.org/wp-content/uploads/2019/12/07-5123-06-zigbee-cluster-library-specification.pdf, Jan 14, 2016.

[49] D. Aitel, "The advantages of block-based protocol analysis for security testing," *Immunity Inc., February*, vol. 105, p. 106, 2002.

[50] M. Zalewski, "Technical whitepaper for afl-fuzz," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2015.

[51] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing through Selective Symbolic Execution," in *Proceedings of the 23rd Network and Distributed Systems Security Symposium (NDSS '16)*, no. 2016. San Diego, CA, USA: Network and Distributed Systems Security Symposium, 2016, pp. 1–16.

[52] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*. Berkeley, CA, USA: USENIX Association, 2018, pp. 745–761.

[53] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing Deeply Nested Branches," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 499–513.

[54] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States," *Proceedings of the 44th International COnference on Software Engineering (ICSE '22)*, 2022.

[55] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing.* CRC press, 2013.

[56] C. Nie and H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.

[57] W. Wang, Y. Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. Kacker, and R. Kuhn, "A Combinatorial Approach to Detecting Buffer Overflow Vulnerabili-

ties," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN '11)*. IEEE, 2011, pp. 269–278.

[58] J. Chandrasekaran, Y. Lei, R. Kacker, and D. Richard Kuhn, "A Combinatorial Approach to Testing Deep Neural Network-based Autonomous Driving Systems," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '21)*, 2021, pp. 57–66.

[59] H. Wu, L. Xu, X. Niu, and C. Nie, "Combinatorial Testing of RESTful APIs," in *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 426–437. [Online]. Available: https://doi.org/10.1145/3510003.3510151

[60] H. Feng, X. Ren, Q. Wei, Y. Lei, R. Kacker, and D. S. Kuhn, "MagicMirror: Towards High-Coverage Fuzzing of Smart Contracts," in *Proceeding of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST '23)*. IEEE, 2023, pp. 141–152.

[61] T. Instruments, "A fully compliant ZigBee 3.x solution: Z-Stack," http://www.ti.com/tool/Z-STACK, 2018.

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

[63] I. Systems, "C-SPY Debugging Guide for Amr cores," https://wwwfiles.iar.com/arm/webic/doc/EWARM_DebuggingGuide.ENU.pdf, [2015].

[64] M. Ren, X. Ren, H. Feng, J. Ming, and Y. Lei, "Security Analysis of Zigbee Protocol Implementation via Device-Agnostic Fuzzing," *Digital Threats*, vol. 4, no. 1, mar 2023. [Online]. Available: https://doi.org/10.1145/3551894

[65] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing IoT Firmware via Multi-stage Message Generation," in *Proceedings of the 26th ACM SIGSAC*

Conference on Computer and Communications Security (CCS'19). New York, NY, USA: Association for Computing Machinery, 2019, p. 2525–2527. [Online]. Available: https://doi.org/10.1145/3319535.3363247

[66] Y. Zhang, W. Huo, K. Jian, J. Shi, H. Lu, L. Liu, C. Wang, D. Sun, C. Zhang, and B. Liu, "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).* New York, NY, USA: Association for Computing Machinery, 2019, p. 544–556. [Online]. Available: https://doi.org/10.1145/3359789.3359826

[67] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18).* New York, NY, USA: Association for Computing Machinery, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[68] T. Instruments, "CC2538," http://www.ti.com/product/CC2538, [online].

[69] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: Security Evaluation of Home-Based Iot Deployments," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19).* Piscataway, NJ, USA: IEEE, 2019, pp. 1362–1380.

[70] M. Ren, H. Zhang, X. Ren, J. Ming, and Y. Lei, "Intelligent Zigbee Protocol Fuzzing via Constraint-Field Dependency Inference," in *Proceeding of the 28th European Symposium on Research in Computer Security (ESORICS '23).* London, United Kingdom: Springer Nature, 2023 (Just Accepted).

[71] M. Ren, X. Ren, H. Feng, J. Ming, and Y. Lei, "Z-fuzzer: Device-agnostic fuzzing of zigbee protocol implementation," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec*

'21). New York, NY, USA: Association for Computing Machinery, 2021, p. 347–358. [Online]. Available: https://doi.org/10.1145/3448300.3468296

[72] Boofuzz, "Boofuzz Protocol Definition," https://boofuzz.readthedocs.io/en/stable/user/protocol-definition.html, 2020.

[73] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A Software Analysis Perspective," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 573–609, 2015.

[74] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation," in *the 57th ACM/IEEE Design Automation Conference (DAC '20).* New York, NY, USA: ACM/IEEE, 2020, pp. 1–6.

[75] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing IoT Firmware via Multi-Stage Message Generation," in *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2525–2527. [Online]. Available: https://doi.org/10.1145/3319535.3363247

[76] D. Bühler, P. Cuoq, B. Yakobowski, M. Lemerre, A. Maroneze, V. Perrelle, and V. Prevosto, *Eva - The Evolved Value Analysis plug-in*, 24th ed., CEA-List, Université Paris-Saclay Software Safety and Security Lab, Gif-sur-Yvette, France, 2021.

[77] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A Combinatorial Test Generation Tool," in *Proceedidng to the 6th IEEE International Conference on Software Testing, Verification and Validation (ICST'13).* Piscataway, NJ, USA: IEEE, 2013.

[78] J. Pereyda, "Boofuzz with combinatorial testing (v0.4.0)," https://boofuzz.readthedocs.io/en/stable/user/changelog.html, [online].

Biographical Statement

Mengfei Ren was born in China in 1989. She received her Bachelor of Computer Science and Technology from China University of Petroleum in 2011. She then joined the University of Texas at Arlington for graduate studies in Computer Science and earned her Master's degree in 2013. After working in the industry for three years, she went back to UT Arlington in 2017 for further study and earned her Ph.D. in Computer Science in 2023. She serverd as Graduate Teaching Assistant in the Department of Computer Science and Engineering at UT Arlington from 2018 - 2023. She is a recipient of STEM fellowhsip from 2017 - 2023 and a Dissertation Fellowship recipient in 2023. Her research interests lie at the intersection of cybersecurity and software engineering. Her PhD thesis focuses on combining practical software testing techniques to detect security vulnerabilities in IoT wireless protocols. Her work has detected several critical vulnerabilities in a mainstream Zigbee protocol stack. She hopes to continue working on security analysis of IoT wireless protocols with advanced software testing techniques after the completion of her doctoral program.