Intelligent Zigbee Protocol Fuzzing via Constraint-Field Dependency Inference

Mengfei Ren[1,3⋆], Haotian Zhang[1], Xiaolei Ren[1], Jiang Ming[2], and Yu Lei[1]

[1] University of Texas at Arlington, Arlington TX 76019, USA
{haotian.zhang, xiaolei.ren}@mavs.uta.edu,
ylei@cse.uta.edu
[2] Tulane University, New Orleans, LA 70118, USA
jming@tulane.edu
[3] University of Alabama in Huntsville, Huntsville, AL 35899, USA
mengfei.ren@uah.edu

Abstract. Zigbee is one of the global most popular IoT standards widely deployed by millions of devices and customers. Its fast market growth also incentivizes cybercriminals. Inference-guided fuzzing has shown promising results for security vulnerability detection, which infers the relationship between input bytes and path constraints. However, deploying such a technique on Zigbee protocol implementation is not a trivial task because of the vendor-specific requirements and particular hardware configuration. In this paper, we propose *TaintBFuzz*, an intelligent Zigbee protocol fuzzing by inferring the dependency between message fields and path constraints. We then use the inference to prioritize the corresponding fields in the mutation process and generate inputs that could explore untouched branches. We implemented a prototype of TaintBFuzz and evaluated it on a mainstream Zigbee protocol implementation called Z-Stack. Compared with state-of-the-art protocol fuzzing tools, including Boofuzz, Peach, and Z-Fuzzer, TaintBFuzz outperforms them in code coverage with the assistance of constraint-field dependency inference. Notably, TaintBFuzz efficiently identifies eight distinct vulnerabilities, of which two are previously unidentified.

Keywords: Fuzzing · Taint Analysis · IoT Wireless Protocols · Zigbee

1 Introduction

Due to the new sensors and more reliable mobile connectivity, the Internet of Things (IoT) device market is projected to reach hundreds of millions of dollars by 2023 [1]. Zigbee protocol [2] is one of the dominant wireless communication protocols deployed in resource-efficient IoT devices. According to the recent market report from Connectivity Standards Alliance, about four billion Zigbee

⋆ This research work is completed when the author takes her Ph.D. degree at University of Texas at Arlington.

devices are expected to be sold globally by 2023 [3]. This fast market growth of Zigbee also incentivizes cybercriminals. Several recent research works have revealed the security issues on Zigbee protocol [4,5,6,7]. These detected vulnerabilities could be exploited for DDoS attacks and remote malicious execution. Therefore, discovering security issues in Zigbee protocol implementations is necessary and practical.

Fuzz testing has shown promising results for finding security vulnerabilities. Many fuzzers [8,9,10,11,12,13] 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. VUzzer [10], and GREYONE [11] utilize it to determine where and how to mutate inputs. REDQUEEN [14] aims to solve magic values and checksum in fuzzing, which colors an input seed by replacing each input byte with the largest number of random bytes possible. Angora [12] uses it to depict the pattern of input bytes related to path constraints. PATA [13] proposes a path-aware taint analysis to identify and mutate critical bytes to solve path constraints.

However, it is not a trivial task to directly deploy those fuzzers to Zigbee protocol implementations. First, they have difficulty compiling the Zigbee protocol. For example, as shown in Fig 1, Texas Instruments (TI) deploys specific compiler check in its Zigbee protocol stack Z-Stack. It prevents general compilers (e.g., GCC, Clang, and LLVM) compiling the full protocol stack, which are instead widely used by the existing fuzzing solutions [16].

```
18: /*
19:  * Check that the correct C compiler is used.
20:  */
21: #ifndef __ICCARM__
22:   #error "File intrinsics.h can only be used together with iccarm."
23: #endif
24:
25: #ifndef __ICCARM_INTRINSICS_VERSION__
26:   #error "Unknown compiler intrinsics version"
27: #elif __ICCARM_INTRINSICS_VERSION__ != 2
28:   #error "Compiler intrinsics version does not match this file"
29: #endif
```

Fig. 1: An example of compiler check deployed in a system library used by Z-Stack [15], a popular Zigbee protocol stack developed by Texas Instruments.

Moreover, those fuzzing approaches cannot provide a proper simulated execution environment for the Zigbee protocol due to the particular hardware configuration required by the Zigbee protocol vendors. The Zigbee protocol stack is usually executed in particular system-on-chip (SoC) devices and a bare-metal program containing a single control loop for scheduling tasks and handling events [17]. Existing fuzzers with simulation platforms (e.g., QEMU) not only require a Linux kernel or an abstraction layer for execution but also support limited embedded devices which are not satisfied the Zigbee protocol vendors' device requirements [16]. The vendor-specific devices also have particular peripheral interrupts not supplied in existing simulation solutions [18]. The current simulation platform cannot, or only with significant engineering effort, provide support for all device-specific hardware configurations required by the Zigbee protocol vendors. Hence, these limitations prevent those state-of-the-art fuzzing methods from directly deploying on the Zigbee protocol implementations.

In this paper, we propose *TaintBFuzz*, an intelligent Zigbee protocol fuzzing with constraint-field dependency inference. Our solution intends to assist IoT application developers in evaluating the security threats associated with the Zigbee protocol implementation in developing their applications. We leverage static taint analysis to infer the relationship between the message field and the path constraints. The dependency inference then guides the fuzzing engine to prioritize the critical message fields for further mutation, which have higher chance to exercise unvisited branches.

The fuzzing engine of TaintBFuzz is designed based on grammar-based fuzzing with code coverage heuristics. 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 [19], 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 [15], a mainstream Zigbee protocol stack developed by Texas Instruments. We compare TaintBFuzz with three state-of-the-art protocol fuzzing tools, Peach [20], Boofuzz [21], and Z-Fuzzer [16]. Peach and Boofuzz are conventional protocol fuzzers widely used in academia and industry. Z-Fuzzer is a recently proposed coverage-guided protocol fuzzer specialized for the Zigbee protocol implementation. Our experiment results show that TaintBFuzz outperforms those fuzzers 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. We have also reported the detected two crashes to the protocol vendor, which are under review when writing this paper. To summarize, we make the following contributions:

- We propose a framework to infer the relationship between the message fields and path constraints by leveraging static taint analysis, which specifically addresses the Zigbee protocol vendor-specific compiler requirement.
- We propose an intelligent mutation strategy utilizing the constraint-field dependency inference to tune the direction of fuzzing, able to prioritize which message field for mutation.
- We implement a prototype of TaintBFuzz and evaluate it on a mainstream Zigbee protocol stack, showing that it outperforms several state-of-the-art protocol fuzzers in terms of code coverage. TaintBFuzz discovers eight vulnerabilities in Z-Stack, two of which are previously unknown.

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

2 Related Work

In this section, we first introduce background knowledge of the Zigbee protocol. As TaintBFuzz is a protocol fuzzer based on taint inference for the Zigbee protocol, we then discuss related work in the security analysis of the Zigbee protocol and fuzz testing with the taint analysis technique.

2.1 Zigbee Protocol

The Connectivity Standard Alliance standardizes the Zigbee protocol as a resource-efficient two-way wireless communication protocol for IoT devices [23]. The protocol stack is shown in Fig 2. The alliance defines the Application Layer (APL) and the Network Layer (NWK) on top of the IEEE 802.15.4 standard, which defines the Medium Access Control Layer (MAC) and the Physical Layer (PHY). The MAC and PHY support packet transmission through the 2.4GHz radio channel. While the NWK layer administers the Zigbee network and forwards packets, the APL is in charge of application-level functionality. Zigbee Cluster Library (ZCL) is a core component in the APL, providing essential API for the manufacturers to implement the device functionalities. 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 device's functionalities. Therefore, we will use ZCL as a case study in the following subsections and remaining parts of this paper.

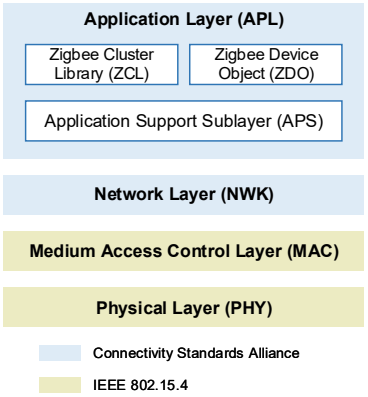


Fig. 2: Zigbee protocol stack overview [22].

2.2 Security Analysis on Zigbee

Since the Zigbee protocol was standardized in 2003, various research work has been published to analyze the security risks of the Zigbee protocol. Specifically, prior work [5,6,24,25,26,27,4] focuses on the security of the Zigbee network transmission. Z3Sec [5], and Snout [24] utilize penetration testing to assess existing vulnerabilities in a Zigbee network. To analyze the security of the Zigbee protocol on specific embedded devices, IoTcube [28], and beSTORM [26] have also been developed. Akestoridis et al. [27] proposed Zigator to analyze encrypted Zigbee packets for selective jamming and spoofing attacks. Wang et al. [6] developed an automated verification tool VEREJOIN via the model checking technique to evaluate the Zigbee network rejoin procedure. Ronen et al. [4] demonstrated

that a worm affecting all Zigbee-enabled lamps might damage the smart lighting in a city. Most of these solutions are black-box solutions that monitor and manipulate Zigbee network traffic to detect security issues.

One of the most well-liked vulnerability identification techniques is fuzz testing (e.g., AFL [29]), which is widely used and researched in the community. Cui et al. proposed two fuzzing approaches to detect security risks on Zigbee: FSM-Fuzzing, which is based on a finite state machine [30], and CG-Fuzzing, which is based on a genetic algorithm [31]. However, both are closed sources thus we failed to compare with state-of-the-art protocol fuzzers. The most recent approach closest to our method is Z-Fuzzer [16], which leverages the code coverage heuristic to guide the fuzzing process on a mainstream Zigbee protocol stack. However, it still has limitations in efficiently exploring the target program's deeper code by ignoring the path constraints' structure.

Compared to the prior work, our work target security issues in Zigbee protocol implementation rather than the real-time Zigbee network. Specifically, our work leverages the relationship between the message field and the path constraints via the static taint analysis technique to efficiently guide the mutation of test inputs, which could explore deeper code of the target program.

2.3 Taint Inference Based Fuzz Testing

A significant drawback of mutation-based fuzzers is efficiently generating test input satisfying complex path constraints. Many fuzzers, such as Driller [8] and QSYM [32], utilize symbolic execution to resolve the complicated branch condition constraints. However, they are not scalable to the extensive application due to the slow execution speed and path explosion issue.

In order to efficiently resolve path constraints, more lightweight solutions are proposed, which infer the relationship between input bytes and path constraints to guide seed mutation. VUzzer [10] focuses on generating test cases to pass magic value validations. It uses taint analysis to identify critical bytes mutated to satisfy the path constraints. Angora [12] locates input bytes that flow into path constraints based on byte-level taint tracking. It then mutates these bytes with a gradient descent algorithm to satisfy the path constraints. REDQUEEN [14] aims to solve magic values and checksum in fuzzing. While reserving the execution path, it colors an input seed by replacing every input byte with as many random bytes as possible. Matryoshka [33] explores nested branches for fuzzing based on both control flow and taint flow. GREYONE [11] utilizes taint analysis to locate the critical input bytes and decides how to mutate them. PATA [13] proposes a path-awareness taint analysis for fuzzing inferring taints based on control flow and value changes. TRUZZ [34] infers the relationship between input bytes and validation checks and prevents those bytes being mutated during the fuzzing.

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 [16]. Most of these fuzzers develop their approaches with general compilers

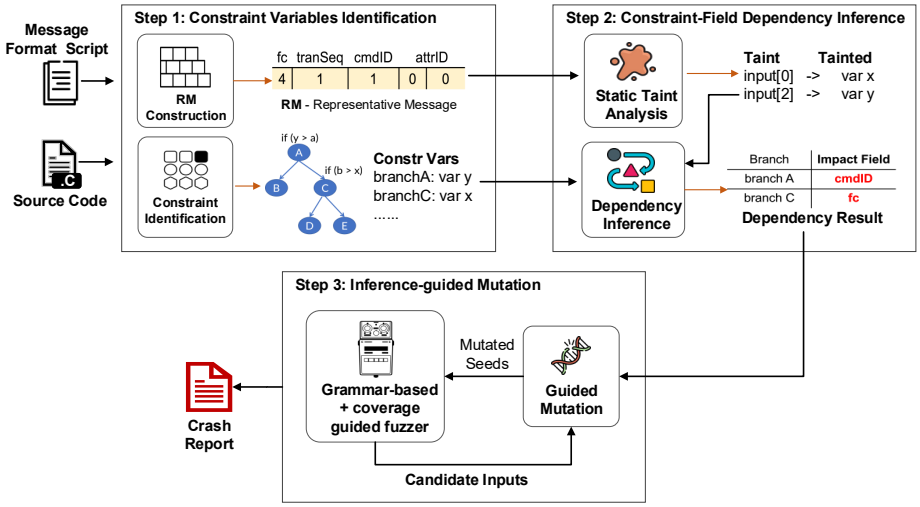


Fig. 3: Overall design of TaintBFuzz. The black arrows mean the main workflow of TaintBFuzz. The red arrows mean the intermediate results generated by the related components.

such as LLVM or Clang, which are not supported by many Zigbee protocol vendors in their protocol implementations. Compared to these fuzzers, our method first pre-process the Zigbee protocol implementation with the compiler specified by the protocol vendor. The pre-processed code is then parsed and type-checked for the further taint analysis.

3 Design of TaintBFuzz

Figure 3 presents the overall design of TaintBFuzz. As the ZCL is the core library of Zigbee protocol stack to implement an IoT device's functionalities, we will deploy it to present the details of each step in the following subsections.

3.1 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

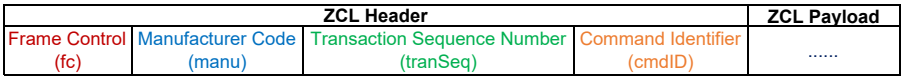


Fig. 4: ZCL frame format [22].

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 [1]. 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.

3.2 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. To tackle this challenge, TaintBFuzz performs static taint analysis on a pre-processed source code compiled by the protocol vendor-specific compiler to distinguish the dependency between message fields and path constraints.

Algorithm 1 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 3) 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 3. The collected result is saved as a map where the key is the constraint, and the value is

[1] An example of the message format script is presented in Appendix A.

Algorithm 1: 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 ← ∅
2  Deps ← ∅
3  foreach rm ∈ R do
4  |   taint ← taintField (rm)
5  |   taint_vars ← taintAnalysis (S, taint)
6  |   tainted ← tainted ∪ (taint, taint_vars, rm.data)
7  end
8  foreach constraint ∈ P do
9  |   foreach var ∈ constraint.V do
10 |   |   if isTainted (var, tainted) then
11 |   |   |   tainted_record ← getTainted (var, tainted)
12 |   |   |   field ← searchField (R, tainted_record)
13 |   |   |   Deps[constraint] ← Deps[constraint] ∪ field
14 |   |   end
15 |   end
16 end
17 Deps ← assembleDependency (Deps)
```

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).

3.3 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 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 2 shows the primary process of coverage-guided fuzzing with constraint-field dependency inference. A *threshold* is a predefined 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.

Grammar Based with Coverage Guided Fuzzing. TaintBFuzz uses a grammar based fuzzer with coverage-guided feedback as its fuzzing engine. We gen-

Algorithm 2: Fuzzing Process 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

erate 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.

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 examining 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 any more. Then, TaintBFuzz first tries other candidate values of the field *cmd* and checks if the previously predecessor block sequence can be re-accessed (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 back-traces 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 Implementation

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 3 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.

The representative message constructor is implemented using the message generator of Boofuzz [21] with a pre-defined message format script that conforms to protocol format definition [35]. The constraint variables identifier and

taint analysis tool are developed based on Frama-C [36]. Frama-C is an open-source platform dedicated to source-code analysis of C software and perform static analysis based on 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 check problem. We modify Frame-C to analyze pre-processed code with the vendor-specific syntax 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 collects the constraint variables used in IF, LOOP, SWITCH, and CALL statements.

The constraint-field dependency inferrer implements Algorithm 1. 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 2. 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 pre-defined 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: (a). randomly selecting values (e.g., selecting any value between [0, 255] if the field is byte type), (b). enumerating all possible values based on the field size, (c). 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 [37] 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.

5 Evaluation

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 comparison with three baseline protocol fuzzers, Peach [20], Boofuzz [21], and Z-Fuzzer [16]. Boofuzz is the successor of Sulley [38], an industry-standard protocol fuzzer more actively maintained than Sulley. Both are open source and have been used in existing research papers [39,40]. Z-Fuzzer is a device-agnostic fuzzing tool for Zigbee protocol implementation that leverages code coverage heuristic on grammar-based fuzzing. 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 [15] as the target program, developed by Texas Instruments with various sample project code bases, 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.

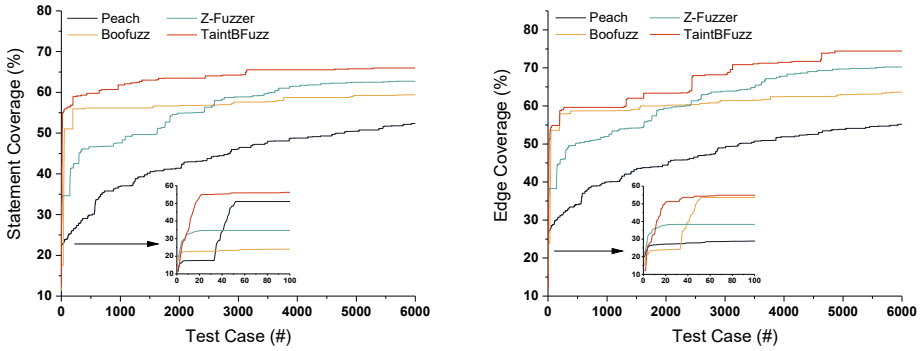
5.1 Fuzzing Performance

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 pre-defined fuzzing dictionary, for which the total number of test cases is finite. Results[2] are presented in Table 1. The results show that TaintBFuzz is more effective than state-of-the-art protocol fuzzers.

Table 1: Evaluation results of fuzzing performance of all fuzzers on Z-stack in 10 runs.

Fuzzer	Unique Test Cases	Statement 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%

[2] 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 1.



(a) Statement Coverage (b) Edge Coverage

Fig. 5: Statement coverage and edge coverage achieved by fuzzers over 10 runs. The X-axis represents the median number of test cases. The Y-axis represents the percentage of statement coverage and edge coverage on average. We also display a zoomed-in graph in the left corner of the coverage variation in the first 100 test cases for each coverage graph.

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. The result is shown in Appendix B

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 1, TaintBFuzz can achieve higher statement coverage and edge coverage with fewer test cases. As we reduced the searching space of several message fields with predefined 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 5 presents the variation of code coverage of fuzzing in all fuzzers. For better result presentation, we plot the coverage trend of the first 6000 test cases generation to show in Fig 5. 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.

5.2 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 2. The vulnerabilities are distinguished by comparing the call stack and performing manual analysis.

As shown in Table 2, 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.

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.

Table 2: Summary of unique vulnerabilities detected all fuzzers over ten fuzzing runs in 24 hours. We present the total amount of test cases triggering the vulnerability on average.

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

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 3. 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*

Table 3: The constraints and message fields assisting TaintBFuzz to trigger the vulnerabilities, in which fc represents the field *Frame Control*, cmdID represents the field *Command Identifier*, attrID represents the field *Attribute Identifier* as shown in Fig 4.

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)

Identifier, which is reasonable since the Zigbee protocol takes different message parser and processor 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 API because of the hardware and computing power limitation. Such customized system API 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 [41]. 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.

6 Conclusion

This paper 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.

Acknowledgments. We would like to thank the anonymous paper reviewers for their insight and helpful feedback. This work was supported by the National Science Foundation (NSF) under grant CNS-2128703. Jiang Ming was also supported by Carol Lavin Bernick Faculty Grant.

References

1. Allied Market 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. The Connectivity Standards Alliance. Zigbee: The Full-Stack Solution for All Smart Devices. https://csa-iot.org/all-solutions/zigbee/, 2015.
3. 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.
4. Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *Proceedings ot the 38th IEEE Symposium on Security and Privacy (S&P '17)*, pages 195–212, Piscataway, NJ, USA, 2017. IEEE.
5. Philipp Morgner, Stephan Mattejat, Zinaida Benenson, Christian Müller, and Frederik 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)*, page 230–240, New York, NY, USA, 2017. Association for Computing Machinery.
6. Jingcheng Wang, Zhuohua Li, Mingshen Sun, and John C.S. 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, 2022. Association for Computing Machinery.
7. Common Vulnerabilities and Exposures. Zigbee CVE Records. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=zigbee, 2022.
8. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In *Proceedings of the 23rd Network and Distributed Systems Security Symposium (NDSS '16)*, pages 1–16, San Diego, CA, USA, 2016. Network and Distributed Systems Security Symposium.
9. Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. Intriguer: Field-level Constraint Solving for Hybrid Fuzzing. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pages 515–530, New York, NY, USA, 2019. Association for Computing Machinery.
10. Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Network and Distributed Systems Security Symposium (NDSS '17)*, volume 17, pages 1–14, San Diego, CA, USA, 2017. Network and Distributed Systems Security Symposium.
11. Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, pages 2577–2594, Berkeley, CA, USA, August 2020. USENIX Association.
12. Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P '18)*, pages 711–725, Piscataway, NJ, USA, 2018. IEEE.
13. Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *Proceediings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*, pages 154–170, Piscataway, NJ, USA, 2022. IEEE.

14. Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the 26th Network and Distributed Systems Security Symposium (NDSS '19)*, volume 19, pages 1–15, San Diego, CA, USA, 2019. Network and Distributed Systems Security Symposium.
15. Texas Instruments. A fully compliant ZigBee 3.x solution: Z-Stack. http://www.ti.com/tool/Z-STACK, 2018.
16. 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)*, page 347–358, New York, NY, USA, 2021. Association for Computing Machinery.
17. Drew Gislason. *Zigbee Wireless Networking, 1st Edition*. Newnes, London, UK, 2008.
18. Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, pages 1201–1218, Berkeley, CA, USA, August 2020. USENIX Association.
19. IAR System. IAR Embedded Workbench. https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/, [online].
20. Peach Tech. Peach Fuzzer: Discover unknown vulnerabilities. https://www.peach.tech/, [online].
21. Joshua Pereyda. Boofuzz: Network Protocol Fuzzing for Humans. https://boofuzz.readthedocs.io/en/latest/, 2020.
22. Zigbee Alliance. Zigbee Specification. https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf, August 5, 2015.
23. BusinessWire. ZigBee Alliance Accelerates IoT Unification with 20 ZigBee 3.0 Platform Certifications From Eight Silicon Providers. https://www.businesswire.com/news/home/20161206005020/en/ZigBee-Alliance-Accelerates-IoT-Unification-with-20-ZigBee-3.0-Platform-Certifications-From-Eight-Silicon-Providers, 2016.
24. John Mikulskis, Johannes K Becker, Stefan Gvozdenovic, and David Starobinski. Snout - An Extensible IoT Pen-Testing Tool. Poster presented at: the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19), 2019.
25. Philipp Morgner, Stephan Mattejat, and Zinaida Benenson. All your bulbs are belong to us: Investigating the Current State of Security in Connected Lighting Systems. *CoRR*, abs/1608.03732, 2016.
26. Beyond Security. Dynamic, Black Box Testing on the ZigBee. https://beyondsecurity.com/dynamic-fuzzing-testing-zigbee.html?cn-reloaded=1, 2021.
27. Dimitrios-Georgios Akestoridis, Madhumitha Harishankar, Michael Weber, and Patrick 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)*, pages 77–88, New York, NY, USA, 2020. Association for Computing Machinery.
28. IoTcube. Blackbox-testing zfuzz. https://iotcube.net/userguide/manual/zfuzz, 2021.
29. Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl, 2015.
30. Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. A Novel Fuzzing Method for Zigbee based on Finite State Machine. *International Journal of Distributed Sensor Networks*, 10(1):762891, 2014.

31. Baojiang Cui, Ziyue Wang, Bing Zhao, and Xiaobing Liang. CG-Fuzzing: A Comprehensive Fuzzy Algorithm for ZigBee. *International Journal of Ad Hoc and Ubiquitous Computing*, 23(3-4):203–215, 2016.
32. Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association.
33. Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*, pages 499–513, New York, NY, USA, 2019. Association for Computing Machinery.
34. Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. *Proceedings of the 44th International COnference on Software Engineering (ICSE '22)*, 2022.
35. Boofuzz. Boofuzz Protocol Definition. https://boofuzz.readthedocs.io/en/stable/user/protocol-definition.html, 2020.
36. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
37. IAR Systems. C-SPY Debugging Guide for Amr cores. https://wwwfiles.iar.com/arm/webic/doc/EWARM_DebuggingGuide.ENU.pdf, [2015].
38. Ganesh Devarajan. Unraveling SCADA Protocols: Using Sulley Fuzzer. Defon 15 Hacking Conference, 2007.
39. Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation. In *the 57th ACM/IEEE Design Automation Conference (DAC '20)*, pages 1–6, New York, NY, USA, 2020. ACM/IEEE.
40. Bo Yu, Pengfei Wang, Tai Yue, and Yong 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)*, CCS '19, page 2525–2527, New York, NY, USA, 2019. Association for Computing Machinery.
41. Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security Evaluation of Home-Based Iot Deployments. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, pages 1362–1380, Piscataway, NJ, USA, 2019. IEEE.
42. Zigbee Alliance. Zigbee Cluster Library Specification. https://zigbeealliance.org/wp-content/uploads/2019/12/07-5123-06-zigbee-cluster-library-specification.pdf, Jan 14, 2016.

Appendix A Representative Messages Generation

To generate highly structured test cases and Representative Messages, we developed a message format script based on the ZCL specification [42] to generate test corpus. Figure 6 presents an example of message format script that generates ZCL messages based on the format definition shown in Fig 4. In the script, the message fields are defined according to their data types, such as enumerations (`enum`). For those enumerated fields, we created the fuzzing dictionary with candidate values defined in the specification. For the remaining fields, we concretized them with random values during the fuzzing process. It's worth noting that all the fuzzers utilized the same format script for the initial seed construction, ensuring consistency in the initial test case generation across the different fuzzing tools.

```
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_static(1, name="tranSeq")
6  s_group("commandId", values=<USER_GIVEN_VALUES>)
7  with s_block("payload", dep="commandId", values =
       <USER_GIVEN_VALUES>):
8    ......
```

Fig. 6: Pseudo-code of Message Format Script.

Appendix B Threshold Tuning

In this study, we propose an intelligent Zigbee protocol fuzzing *TaintBFuzz* by inferring the dependency between message fields and path constraints. When the fuzzer reaches a point where no new execution paths are being explored over a certain period of time, TaintBFuzz employs the constraint-field dependency to augment the mutation process and generate diversified seeds.

Table 4: Summary of 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)

To simplify the implementation, we defined a *threshold* as the number of mutation times since the last updated code coverage to represent the timeout. Thus, we performed an empirical experiments to decide the proper value of the threshold. In our experiment, we compared three different threshold values *10, 25, 50* and found that fuzzing performs achieves better result when setting the threshold as 50.

Table 4 presents the comparison results. We categorized all generated test cases based on the field *Command Identifier*. The result indicates 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 use threshold 50 for the TaintBFuzz's mutation when comparing the fuzzing performance with state-of-the-art fuzzers.