LLMIF: Augmented Large Language Model for Fuzzing IoT Devices

Jincheng Wang
The Hong Kong Polytechnic University

Le Yu✉
Nanjing University of Posts and Telecommunications

Xiapu Luo✉
The Hong Kong Polytechnic University

***Abstract*—Despite the efficacy of fuzzing in verifying the implementation correctness of network protocols, existing IoT protocol fuzzing approaches grapple with several limitations, including obfuscated message formats, unresolved message dependencies, and a lack of evaluations on the testing cases. These limitations significantly curtail the capabilities of IoT fuzzers in vulnerability identification. In this work, we show that the protocol specification contains fruitful descriptions of protocol messages, which can be used to overcome the above limitations and guide IoT protocol fuzzing. To automate the specification analysis, we augment the large language model with the specification contents, and drive it to perform two tasks (i.e., protocol information extraction, and device response reasoning). We further design and implement a fuzzing algorithm, LLMIF, which incorporates the LLM into IoT fuzzing. Finally, we select Zigbee as the target protocol and initiate comprehensive evaluations. The evaluation result shows that LLMIF successfully addressed the above limitations. Compared with the existing Zigbee fuzzers, it increases the protocol message coverage and code coverage by 55.2% and 53.9%, respectively. Besides the enhanced coverage, LLMIF unearthed 11 vulnerabilities on real-world Zigbee devices, which include eight previously unknown vulnerabilities. Seven of them are not covered by the existing Zigbee fuzzers.**

1. Introduction

Ensuring the correctness of IoT protocol implementations is important since these protocols guarantee the intended functioning of IoT devices. In critical infrastructures (e.g., healthcare, transportation, and energy), malfunctions or crashes in IoT devices can lead to catastrophic consequences [1]. One error in a protocol implementation can instigate a domino effect of failures, which can be challenging to diagnose and rectify [2]. In particular, the network protocol fuzzing technique has been widely used to identify vulnerabilities in the protocol implementation [3], [4], [5], [6], [7]. It involves generating tailored messages and transmitting them to the target device in an attempt to trigger unexpected behavior. Since the generated message conforms to the protocol format requirement, network protocol fuzzing is more flexible compared with binary fuzzing [8], [9]. It can easily target different protocol implementations without the need to consider the availability of the source codes and the hardware configuration of the target device.

Network protocol fuzzing encompasses several distinct phases, i.e., fuzzing seed generation, seed mutation, test case evaluation, and case enrichment. Unfortunately, the design scope and effectiveness of existing IoT fuzzers [3], [4], [5], [10] suffer from certain limitations, which restrict their design scope and effectiveness. $(L.1)$ *Obfuscated Message Formats.* IoT messages typically exhibit a complex structure, with meticulously constructed headers and payloads. In the absence of knowledge regarding these message formats, two significant issues arise. Firstly, the fuzzer is limited to a relatively small range of message types for seed generation. Secondly, mutation processes may prove ineffective, as they often generate malformed test cases or overlook the mutation of critical bits/bytes, which can potentially expose vulnerabilities. $(L.2)$ *Unresolved Message Dependencies.* IoT protocols showcase an extensive device state space, effectively navigable only through well-orchestrated message sequences. Without resolving these message dependencies, enriching the testing cases and creating complex message sequences becomes a daunting task. $(L.3)$ *Lack of Testing Case Evaluations.* The evaluation of testing cases based on execution feedback (e.g., code coverage) is a standard approach to retain intriguing cases for further examination [11]. Absent appropriate evaluation strategies, the fuzzer continually generates low-quality test cases, resulting in subpar fuzzing performance.

Regrettably, surmounting the aforementioned obstacles to fuzz IoT devices is not a straightforward undertaking. (1) Prior research [12] employed machine learning techniques to infer message formats from plain-text network traces. However, the flexible and vendor-specific authentication schemes employed in IoT protocols, e.g., customized Zigbee link keys [13], render the collection of plain-text traces a challenging endeavor. (2) Owing to the multitude of message formats and device properties, IoT protocols often lack formal definitions of the protocol state machine. This absence complicates the construction of message dependency relationships. Furthermore, existing methods [6], [7] that rely on network trace analysis falter in the face of customized authentication schemes. (3) Driven by a desire to protect their intellectual property and to avoid exposing vulnerabilities, IoT vendors are typically reluctant to share

✉ *The corresponding authors.*

the source code or even binaries of their device firmware. This reluctance impedes the fuzzer's ability to collect valuable feedback, such as code coverage.

To overcome these limitations, we make a critical observation: *the protocol specification provides rich message descriptions that can be harnessed to guide the fuzzing process*. Firstly, given that these descriptions include the introductory details of the message header and payload formats, they can be extracted to enhance message coverage and craft effective mutation operators. Secondly, these descriptions often subtly delineate message dependencies through the interaction of device properties. For instance, the description of the Zigbee Identify message, "This then starts the device's identification procedure," and that of the AddGroupIfIdentifying message, "The message allows the device to add a group on the condition that it is identifying itself," suggest a correlation between these two messages in the context of the device's identifying status. Lastly, these descriptions outline the workflow of message processing and response generation, which can be utilized to evaluate the testing case. Given a transmitted testing case and the subsequent response, one can leverage the descriptions to determine whether the message execution aligns with the specification, leading to a legitimate device state transition, or if the execution contravenes the specification, resulting in an unspecified device state transition. Testing cases that trigger device state transitions will be preserved as new fuzzing seeds for use in subsequent fuzzing rounds.

Leveraging the specification to guide IoT fuzzing necessitates capabilities in text summarization and reasoning. While these tasks are relatively straightforward for humans, they can be laborious and prone to errors. For instance, summarizing the formats for over 140 message types specified in Zigbee's 1,213-page document could consume days of human effort. Motivated by the recent strides made by large language models (LLMs) in various natural language processing tasks, we propose to augment an LLM with specification content and direct it to answer protocol-related questions. The generated responses are then utilized to facilitate various fuzzing phases. We have chosen Zigbee [14] as our target IoT protocol, as it has been implemented in nearly 300 million nodes globally. Specifically, we first apply an LLM augmentation approach, infusing the LLM with the specification content. Then, the augmented LLM is tasked with two duties: *Protocol Information Extraction* and *Device Response Reasoning*. For the task of protocol information extraction, the LLM is asked to extract useful protocol information, such as message formats, intriguing field values, header structure, and message dependencies from the specification. This approach addresses limitations $L.1$ and $L.2$. For the task of device response reasoning, the LLM, given the details of a generated testing case and the device response, uses the message description to determine if the testing case prompts a device state transition. The result is subsequently employed to evaluate the test case, addressing limitation $L.3$.

Finally, we propose a fuzzing algorithm, LLMIF, which incorporates the LLM into Zigbee fuzzing. In each fuzzing round, LLMIF first utilizes the LLM-extracted information (message payload formats, interesting field values, and header structure) to generate seeds and perform mutation. Then it transmits the generated testing cases to the target device, monitors device crashes, and waits for the device response. With the collected response, LLMIF further instructs the augmented LLM to evaluate the quality of the testing cases. For those cases that promote device state transitions, LLMIF enriches them to construct new seeds based on the extracted message dependency relationship. These seeds are prioritized to be used in the later fuzzing rounds.

Our evaluation results demonstrate the effectiveness of LLMIF in terms of code coverage and vulnerability identification. Compared with the baseline, LLMIF improves the message coverage by 55.2% and code coverage by 53.9%. Moreover, LLMIF is highly effective at finding critical vulnerabilities in Zigbee protocol implementations. We tested 11 real-world Zigbee devices and successfully identified eight zero-day vulnerabilities. Seven of them are missed by the existing fuzzers. In summary, our work makes the following contributions.

- We utilize the augmented large language model (LLM) to analyze the protocol specification and overcome the limitations of IoT fuzzing, i.e., unknown message formats, unresolved message dependencies, and lack of testing case evaluations.
- We incorporate the augmented LLM into Zigbee fuzzing and propose a fuzzing algorithm LLMIF. It utilizes the LLM's result to enhance different fuzzing phases including seed generation, mutation, testing case evaluation, and testing case enrichment. We also implement a prototype for fuzzing real-world Zigbee devices [15].
- We conduct experiments that demonstrate LLMIF's effectiveness. Compared with the baselines, LLMIF is more effective in increasing the coverage of the message type and the protocol implementation code. Besides the enhanced coverage, LLMIF discovered 11 vulnerabilities in off-the-shelf Zigbee devices including eight zero-day vulnerabilities, the majority of which are missed by existing fuzzers.

2. Background

In this section, we start by introducing the main concepts in Zigbee and then provide some background on large language models.

2.1. Zigbee Protocol

Zigbee is a communication protocol built to provide low-power, low-cost wireless mesh networking for IoT devices, and it regulates a range of device functionalities, e.g., lighting and locking. To provide seamless communication within the Zigbee ecosystem, Zigbee Alliance provides the concept of "cluster" in the protocol specification [16], which defines a set of common message formats and data structures that

TABLE 1: Format of the "Add Group If Identifying" message

Number of Bytes	2	Variable
Data Type	uint16	string
Field Name	Group ID	Group Name

devices can follow. For example, Zigbee specifies a "Group" cluster (Cluster ID 0x0004), which allows devices to be assigned to one or more groups and supports simultaneous control of several devices. The cluster defines several device properties, e.g., group table, which the device should support to specify the current grouping states. Moreover, the cluster defines six message types with fixed payload formats for communication purposes. For example, Table 1 shows the format of the "Add Group If Identifying" message (Command ID 0x05), which adds the group specified in the message payload if the device currently is in identifying status. Any message that violates the format, e.g., missing fields or invalid field values, will be regarded as malformed commands and filtered by the target device without further processing.

Due to the widely covered device functionalities, Zigbee specifies a large number of message formats. For example, Zigbee specifies 22 standard clusters that cover more than 140 message types. Besides a large number of messages, these messages are also widely intertwined with each other through common device properties, i.e., the prerequisite of a message execution relies on specific settings of device properties, which are updated by other message executions. As a result, they form various message dependencies. For example, the "View Group Membership" message execution depends on the "Add Group" message execution because its execution relies on the "Add Group" message to properly set the entry in the group table.

2.2. Large Language Models

Large language models (LLMs) are a type of machine learning model that can process and generate natural language text. Fundamentally, they are trained on massive amounts of text data to perform statistical language modeling and word prediction. In particular, LLMs achieve amazing performance in text summarization, text reasoning, and contextual conversations. For example, a recent study [17] shows that summaries generated by the LLM are favored by human annotators over the reference summaries in the document. [18] shows that LLMs can perform well in comprehending the given document, e.g., radiology reports and doctor-patient dialogue, and reason it with domain knowledge. Finally, [19] shows that LLMs have the potential to understand the history of conversation and generate responses that are contextually relevant and coherent with the previous conversation turns.

One of the advantages of LLMs is that they can be quickly adapted to a specific task, e.g., mental health classification [20], question and answering [21], and penetration testing [22]. To achieve the goal, prompt engineering is commonly employed [23]. In the prompting paradigm, a pre-trained LLM is provided with a snippet of text as an input and is expected to provide a relevant completion of this input as output. Prompt engineering is intended to provide a set of principles and techniques for designing prompts to squeeze out the best performance from these machine learning models.

While LLMs hold great promise as general task solvers, recent studies [24] show that they suffer from important limitations hindering a broader deployment. The first limitation is that they lack domain-specific knowledge, which makes them insufficient when performing domain-specific tasks. The second limitation is that they usually provide nonfactual but seemingly plausible predictions, often referred to as "hallucinations". As a result, a growing research trend [25] emerged to augment LLMs with domain knowledge for addressing the above limitations, which is known as "Augmented language models" (ALM).

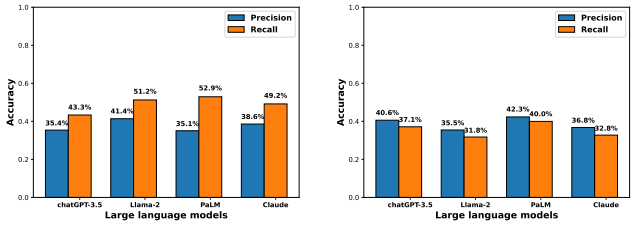
3. LLM Augmentation with Specification

In this section, we discuss our approach to augmenting the LLM for IoT fuzzing. We start by showing that LLM lacks understanding of the IoT protocol, which highlights the necessity of infusing it with domain knowledge, i.e., the specification contents. Then we detail our augmentation approach.

3.1. LLM's Understanding About Zigbee

Pre-trained on vast volumes of internet data, large language models have assimilated knowledge from diverse fields. Their effectiveness has been demonstrated across a variety of tasks, including, notably, question answering. Inspired by recent work [6] which utilizes the LLM's internal knowledge to guide network protocol fuzzing, e.g., RTSP, we are curious about a question: Does the LLM have sufficient understanding of the IoT protocol, e.g., Zigbee, such that they can output useful protocol information to guide the fuzzing process?

We initiate a case study to explore the answer to the above question. Specifically, we select four popular LLMs as the target: openAI's chatGPT [26], Meta's LLama 2 [27], Google's PaLM [28], and Anthropic's Claude [29]. Benefiting from billions of model parameters, these LLMs have been widely used and shown to be effective in various tasks. They are required to address the limitations of existing Zigbee fuzzers and construct the Zigbee message payload format. We select 96 message types from the 22 standard Zigbee clusters to construct the baseline, each of which contains at least one field in their payloads. We specified two sub-tasks to evaluate the LLM's performance. (1) *Message identification*: the LLM should output the correct message name. (2) *Format inference*: The LLM should output the correct field name and field data types for each message. A message format is successfully constructed only if the name is correctly identified and its format is correctly inferred.



(a) cmd identification accuracy (b) Format inference accuracy

Figure 1: Evaluations of general-purpose LLM's understanding for IoT protocols

Figure 1a and Figure 1b show the evaluation result. Among the 96 message types, these LLMs only successfully construct 15 message formats on average, which achieves a low recall of 15.6%. Consequently, if the LLM is directly used for guiding the fuzzing process, the fuzzer will achieve a low message coverage and omit critical bugs lurking in the processing logic of these missing messages. The above result reveals that LLMs do not have enough understanding of the IoT protocol, and it is necessary to augment it with the domain knowledge, i.e., the protocol specification, before performing protocol fuzzing. We also provide a demonstration in our complementary material [15] for reference.

3.2. Augmentation Approach

Common paradigms for LLM augmentation follow the *retrieve-then-read* pipeline [30]. First, due to the LLM's limited context size, a knowledge retriever is needed to retrieve task-related knowledge from external knowledge sources, e.g., Wikipedia pages. Second, the retrieved knowledge is fed into the LLM which adapts it to the downstream task, e.g., question answering. In this work, we aim to retrieve the message descriptions from the specification as domain knowledge and infuse them into the LLM.

Unlike previous works [31], [32] which require a large volume of training data and train a neural retriever, to retrieve the message descriptions, our observation is that *the specification documents are usually organized in well-formed file formats, e.g., PDF, which records fruitful metadata for knowledge indexing and searching.* In particular, the document outline records semantically meaningful titles, e.g., "3.5.2.3 Commands Received", which can help to localize the corresponding sections that contain message descriptions. Given the specification document, we first parse it and build the hierarchy of the outline. Each entry in the hierarchy corresponds to a section, which records the section level, section name, and the covered pages. Then we use regular expressions to match the entries whose section name contains the keyword "commands received". For all matched entries, we further track their children entries, each of which represents a subsection discussing a specific cluster message, e.g., "3.5.2.3.1 Identify Command". As a result, we use the page contents covered by these subsections to build the domain knowledge of the corresponding messages.

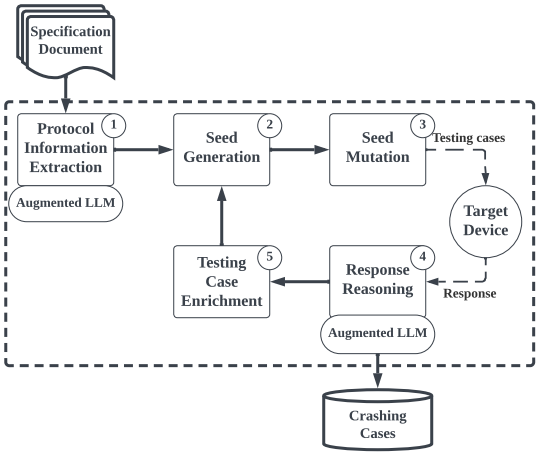


Figure 2: LLMIF workflow

We identified 271 pages and 596,140 characters for 147 cluster messages.

To augment the LLM with the extracted domain knowledge for downstream tasks, e.g., extraction of message payload formats, we further employ the *background-augmented prompting* technique [25], which uses the domain knowledge to construct task-specific prompts. Specifically, given the retrieved message descriptions c and the downstream task instruction q, e.g., "summarize the Add Group message format", we concatenate c and q to build a prompt which drives the LLM to base on the specification contents and construct the message format. Compared with the fine-tuning technique [33], [34], the background-augmented prompting approach avoids the significant tuning cost [35], [36]. Moreover, considering that not all LLMs are open-source and support fine-tuning, the prompting technique is more general and can be employed with any LLMs.

4. LLM-Guided IoT Fuzzing

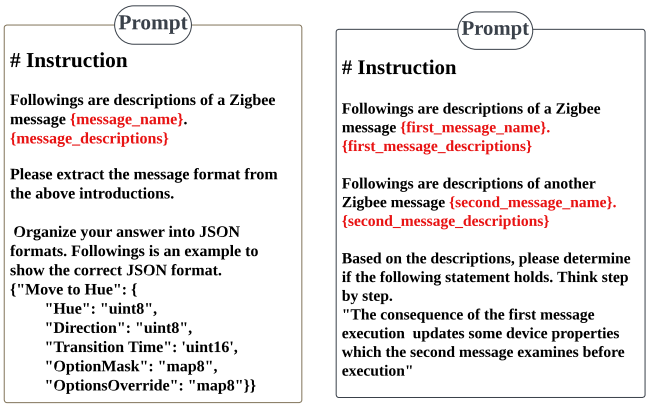
With the augmented LLM, we develop an LLM-guided fuzzing algorithm, LLMIF, to tackle the limitations of existing fuzzers. Figure 2 shows the workflow. The input to LLMIF is the specification document, and the output is the testing cases that crash the target device. Specifically, (1) LLMIF first uses the augmented LLM (Section 3) to analyze the specification and extract four types of protocol information: message payload format, interesting field values, message header structure, and message dependencies. The extraction process is performed only once before the fuzzing process starts, such the extracted information can be directly used in later fuzzing phases without communicating to the LLM. Then LLMIF starts and loops the fuzzing round (Step 2 - 5). (2) In each round, LLMIF first determines the seed. It checks if there are any testing cases enriched in previous rounds. If that is the case, LLMIF picks one and uses it as the seed. Otherwise, LLMIF leverages the extracted header structure, payload format, and interesting field values to generate a message from scratch as the testing case. (3)

After the seed is determined, LLMIF further mutates the seed and generates testing cases. The mutation phase also exploits the extracted information to mutate fields in the header and the payload. The generated testing cases are further transmitted to the target device for execution. (4) LLMIF sets a timeout and waits for the device response. If the timeout is reached while no response is collected, LLMIF repeats the transmission three times. If there are still no responses, the target device is regarded as crashed, and the testing case is recorded. Otherwise, LLMIF utilizes the augmented LLM in real time to analyze the response. Testing cases that promote the device state transition will be recorded. (5) Finally, LLMIF enriched the recorded testing cases by appending new messages that follow the message dependency relationship. LLMIF further prioritizes their usage as seeds in later rounds. In the following section, we will discuss the design of each module in detail.

4.1. Protocol Information Extraction

With the augmented LLM, LLMIF employs the background-augmented prompting technique (Section 3.2) to extract four types of protocol information: Message payload formats, interesting field values, message header structure, and message dependency relationships.

Message payload format. Figure 3a illustrates the prompt template for message format extraction. The template has two slots, "message name" and "message descriptions", which are used to encode the retrieved message descriptions. Note that the message descriptions may span



(a) Message payload format (b) Message dependency

Figure 3: Prompt templates for protocol information extraction

multiple pages and the size may exceed the LLM's context limit. As a result, we propose a summarization approach to generate concise message descriptions for constructing the prompt. Specifically, given a message description that spans from page m to page n ($m \leq n$), we first ask the LLM to summarize information about the message format recorded on each page i. Then we aggregate the summary of each page as the final message description and encode it into

Effect Identifier Field Value	Effect Variant Field Value	Description
0x00	0x00 (default)	Fade to off in 0.8 seconds
	0x01	No fade
	0x02	50% dim down in 0.8 seconds then fade to off in 12 seconds
	0x03 to 0xff	Reserved
0x01	0x00 (default)	20% dim up in 0.5s then fade to off in 1 second
	0x01 to 0xff	Reserved
0x02 to 0xff	0x00 to 0xff	Reserved

Figure 4: Interesting values for "Off with Effect" message

the final prompt. In order to instruct the LLM to generate well-formed answers for further parsing, we employ *few-shot learning* technique [37] and drive the LLM to generate JSON representation, where the key is the message name and the value is the field information (field name: field data type). We constructed 147 model prompts, each of which is for a specific cluster message, and used chatGPT-3.5-turbo as the LLM to construct their formats. Then we evaluate the construction accuracy for the 96 messages whose payloads contain at least one field. The result shows that the augmented LLM successfully constructs the formats for all cluster messages.

Interesting field values. Besides the payload format, we also find that the message descriptions record a large number of interesting field values. In particular, we categorize two types of interesting values from the specification. (1) *Dangerous values*. These values are prohibited by the specification for setting fields. For example, the specification requires that the GroupID field in the "Add Group" message should be within 0x0001 - 0xfff7. Values outside the range, e.g., 0x0000, are invalid and should not be used for setting the field. (2) *Functioning values*. These values are used to trigger the specific device functioning logic. For example, Figure 4 shows that when the EffectVariant field value of the "Off With Effect" message equals 0x02, the target device should perform 50% dim down in 0.8 seconds. Since these interesting values can help to generate initial seeds that explore diverse device functioning and error-handling logic, we use the augmented LLM to collect them. As a result, we constructed 96 prompts, and the augmented LLM successfully collected 421 functioning values for 68 message fields. Moreover, it collects 83 dangerous value intervals which are related to 69 message fields.

Message header structure. The message header contains important fields which are valuable for mutation. Therefore, with the descriptions of the message header, we drive the LLM to draw the structure of the message header. As a result, it successfully extracts seven fields in the header. One important field is the "Disable Default Response" bit field. Upon receiving a message, a Zigbee device will check this bit to determine if it should generate a response. By mutating this bit, the fuzzer can ask the target device to generate responses to the received testing cases, and the response, in turn, can be used as feedback to evaluate the quality of the testing cases (Section 4.4).

Message dependencies. As discussed in Section 2.1,

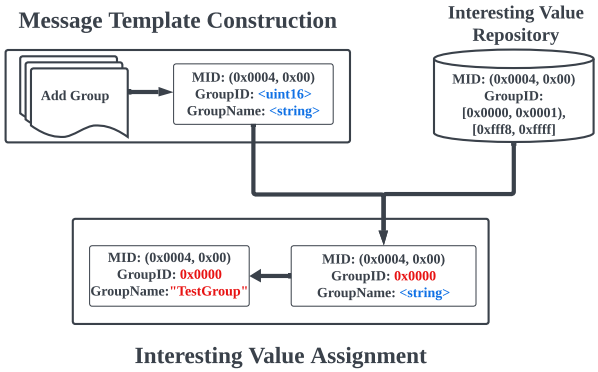


Figure 5: Illustrations of seed generation

Zigbee messages are implicitly correlated with each other through common device properties. That is, messages A and B have a dependency relationship ($A \rightarrow B$) if the execution of A updates some device properties that B examines before its execution. Since the message description usually details how a message interacts with the device property, the augmented LLM should be able to reason if two messages have dependency relationships. Figure 3b shows the model prompt template for message dependency construction. The template takes the descriptions of two messages as input to construct the domain knowledge. Different from the previous tasks which ask the LLM to perform text summarization, the construction of message dependencies requires activating the LLM's reasoning capability. That is, the LLM should first understand the device properties that the two messages respectively interact with, then determine if they work on the same properties. As a result, we employ the chain-of-thought technique [38] to drive the LLM to think step-by-step. With the prompt template, we constructed 21,609 prompts, each of which is for examining a specific message pair. As a result, the augmented LLM successfully constructs 968 message dependencies among the 147 cluster messages.

4.2. Seed Generation

LLMIF further utilizes the extracted message information to facilitate seed generation, which is illustrated in Figure 5. Specifically, for each constructed message format, LLMIF assigns it a unique message identifier:

$$\text{MID} = (\text{clusterID}, \text{cmdID})$$

where clusterID identifies the cluster of the message and cmdID identifies the message in the cluster's message set. Moreover, LLMIF uses the message format to build a set of message templates and maintains the mapping from MID to the template:

$$\{\text{MID} \rightarrow \{(\text{field\_name}_i, \text{field\_type}_i, \text{field\_value}_i)\}$$

where i is the index of fields in the corresponding. The field_value slot in the template marks the regions to be filled with detailed values for generating well-formed messages. Similarly, LLMIF also maintains the mapping from MID to the extracted interesting values:

$$\text{MID} \rightarrow \{(\text{field\_name}_i, \{\text{value\_interval}_j\})\}$$

where i indexes the field and j indexes the recorded intervals of interesting values for the i's field. In each fuzzing round, a random MID is first selected, e.g., (0x0004, 0x00) representing the "Add Group" message from the Groups cluster. Then LLMIF retrieves the corresponding message template and identifies the slot to be filled. For example, the template of the "Add Group" message contains two field slots to be filled, and the value types should be uint16 and string, respectively. To determine the concrete field value, the fuzzer first uses the MID and the field name as the keyword to look for interesting values. If there are any recorded value intervals for the field, LLMIF randomly selects an interval and samples a value as the concrete value to fill the slot. Otherwise, LLMIF follows the data type and randomly generates concrete values. For example, after checking that the GroupID has two interesting value intervals, LLMIF randomly selects the interval [0x0000, 0x0001) and samples a value 0x0000 for the GroupID field. As for the GroupName field which has no recorded interesting values, it randomly generates a string "TestGroup" as the field value.

Besides setting the message payload, LLMIF also leverages the extracted header structure to properly set the message header. We here discuss three important fields. (1) *Manufacturer-specific bit*. This bit is for determining whether the current message is from the Zigbee cluster specification or created by the manufacturer. Since LLMIF mainly focuses on the messages recorded in the specification, the bit is set to zero. (2) *Direction bit*. The bit is used to denote whether the message is a request message or a response to a previously received request. Since in most cases the fuzzer acts as the client to transmit the request to the target device, this bit is set to zero which denotes that the message is a request. (3) *Disable Default Response bit*. As introduced in Section 4.1, this bit is critical for collecting responses from the target device. As a result, LLMIF sets the bit to zero, which asks the target device to always generate a response for every received message.

4.3. Seed Mutation

To mutate the generated seed, LLMIF leverages the extracted protocol information to specify two types of mutation operators: type-aware mutation and header-aware mutation.

Type-aware mutation. LLMIF exploits the knowledge about the data type of each field to perform type-aware mutation. Specifically, we first summarize the extracted data types and categorize them into two classes: Fixed-length type and variant-length type. The former is commonly used for numerical data types, e.g., uint32 and enum16, while the latter is commonly used for string types, e.g., character string and octet string. In particular, the variant-length type usually has additional grammar requirements. For example,

it requires that the first byte should be the length of the string followed by the string contents.

Given a message and its format, the fuzzer traverses the data type of each field and determines the corresponding mutation operator. For the field with the fix-length type, LLMIF uses the *extreme value* operator to mutate them. By setting the numerical values with extreme values, e.g., 0x0000 and 0xffff, it aims to trigger out-of-range vulnerabilities. As for the field with variant-length type, LLMIF utilizes the following two mutation operators. (1) *Byte expansion.* By increasing the string content and the corresponding length byte, LLMIF aims to trigger buffer and heap overflow. (2) *Suspicious length.* By mutating the length byte of the variant-length field, LLMIF aims to create inconsistencies and trigger memory leakage or null pointer de-reference vulnerabilities. Besides the above type-aware mutation operators which tamper with the field value, LLMIF randomly removes some message fields and deliberately creates malformed testing cases. These cases may crash the target device if the payload domain is not properly parsed.

Header-aware mutation. LLMIF also exploits the structure of the message header and enforces a set of header-aware mutation operators. (1) *Crafted command identifier.* LLMIF corrupts the command identifier field with randomly generated values. (2) *Bit flipping.* By flipping the specific bits in the message header, the fuzzer mutates the frame type, the message source, and the message direction, aiming to test the correctness of the header parsing procedure and to trigger unexpected message processing logic. Note that in order to guarantee the response collection, the Disable Default Response field is not flipped.

4.4. Response Reasoning

The generated testing case will be transmitted to the target device for execution. In order to guarantee the same initial state for each testing case execution, LLMIF first transmits a general Zigbee message "Reset to Factory Default" (clusterID=0x0, commandID=0x0) which initializes the device state. After the testing case is transmitted to the target device and a response is collected, LLMIF leverages the augmented LLM to reason the response and evaluate the testing case. The idea is based on two key observations. (1) *The response generated by the target device usually contains a status code, which marks the execution status of the testing case.* For example, the status code 0x00 implies that the last message was successfully executed, while 0x80 implies that the last message was malformed. (2) *The message description details various circumstances under which specific status codes should be generated.* For example, when an Add Group message with normal field values is received, the specification requires that "...the device adds the values of the Group ID and Group Name fields to its Group Table and the status shall be SUCCESS...". As another example, when an Add Group message with dangerous field values is received, the specification requires that "...the device verifies that the GroupID field contains a valid group identifier in the range 0x0001-0xfff7. If the GroupID field contains a group identifier outside this range, the status shall be INVALID_VALUE...".

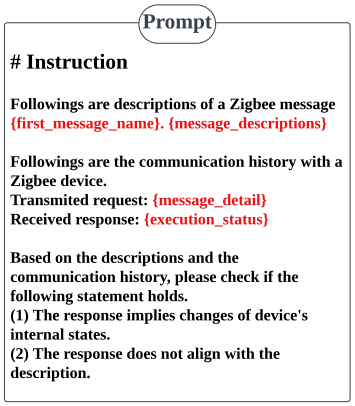


Figure 6: Prompt template for response reasoning

The above observations show that combined with the message description, the response can be used to evaluate the quality of the testing cases. (1) The status code in the response can be used to check if the testing case was successfully executed and changed the device state. For example, if an Add Group message receives a response with a SUCCESS status code, it implies that the device property, in particular, the group table, has been updated and device state transitions happened. As a result, such testing cases deserve to be kept for further investigation. (2) The status code can be used to initiate conformance testing, which detects if the message execution violates the specification and drives the device into dangerous and unspecified states. For example, if an Add Group message with GroupID 0x0000 receives a response with SUCCESS status code, which is supposed to be INVALID_VALUE according to the specification, it implies that the crafted message successfully pollutes the group table. Since these testing cases violate the specification and put the device into unspecified states, they are also valuable for further investigation.

However, considering the large number of status codes, message descriptions, and the large number of testing cases generated in each fuzzing round, it is almost impossible to get humans involved and reason the response of each testing case. To overcome the challenge, LLMIF utilizes the augmented LLM to perform the task. The model prompt template is shown in Figure 6. It takes three pieces of information as inputs: Message description, message details, and message execution status. For each testing case, LLMIF first checks the message in the case and fetches its description to augment the LLM. Since response reasoning requires the detail of the message, e.g., the GroupID values, LLMIF further concatenates the field names and values of the message into a string to construct the message detail. Finally, LLMIF extracts the execution status code from the device response, constructs the prompt, and asks the LLM to determine if the device state transition happens. Either a normal state transition that aligns with the specification or an

abnormal state transition that violates the specification will promote LLMIF to record the corresponding testing case for further investigation.

4.5. Testing Case Enrichment

Once a testing case draws the attention of the LLM and is recorded by LLMIF (Section 4.4), it is enriched to construct new seeds. For each testing case, we define its interacted device properties as the one that is altered by the last message. As a result, when enriching testing cases, we only consider messages that are dependent on the last message, such that the enriched case will examine the interacted device property.

Specifically, given a testing case $s = [m_1, ..., m_n]$ which contains n messages, the task of testing case enrichment is to construct a suitable message m_{n+1} which can be concatenated to the end of s, such that every neighboring message pair in the constructed message sequence $s' = [m_1, ..., m_n, m_{n+1}]$ follows the message dependency relationship.

LLMIF leverages the extracted message dependency relationship to perform the task. Specifically, the dependencies are stored in the form of message pairs:

$$D = \{d : (\text{MID}_{pre}, \text{MID}_{con})\}$$

where the preceding message and the consecutive message represented by the corresponding MID are correlated for the common device property. Given a testing case s, LLMIF first fetches the MID of the last message m_n, and looks for the set of candidate messages M which have correlations with s, i.e.,

$$M = \{\text{MID} | (m_n.\text{MID}, \text{MID}) \in D\}$$

After M is determined, LLMIF randomly selects a MID from M and follows Section 4.2 to initialize the message m_{n+1} whose message identifier equals to MID. Finally, the testing case is enriched with the new message, i.e., $s' = s + [m_{n+1}]$, and s' is saved into the seed corpus and is prioritized to be selected in the next fuzzing round.

4.6. Fuzzing Tool Implementation

Unlike fuzzing traditional network protocols, e.g., SMTP [39], fuzzing real-world Zigbee devices requires the support of specific radio modules (e.g., CC2530). To fuzz the real-world Zigbee device, we design a fuzzing tool that contains two components: *Fuzzing controller* and *stack controller*. The fuzzing controller is responsible for running LLMIF and organizing the fuzzing workflow. In particular, chatGPT-3.5.turbo is used as the large language model. The stack controller, on the other hand, works as a driver to operate a programmable Zigbee radio and provides fundamental support for Zigbee communication, e.g., message transmission and reception.

The fuzzing controller leverages the idea of "building block" to construct the message, which is commonly used in generation-based fuzzing [4], [40]. Specifically, we implement 47 basic data types specified in Zigbee specification, e.g., enum8 and string, and use them as the building block for assembling the message payload. As a result, the extracted protocol information, e.g., the message format and interesting value repositories, can be easily integrated into the assembling process. We further implement a total of seven mutation operators (Section 4.3) to perform mutation. The fuzzing controller is implemented in Python with 3,000 lines of code, and it runs on a Raspberry Pi 4 with Ubuntu 20.04 operating system.

The stack controller aims to create and maintain the Zigbee communication channel with the target device. Considering availability and popularity, we select CC2538 with a fully compliant Zigbee solution Z-Stack [41] as the hardware radio. We develop a driver on top of Z-Stack and transform CC2538 into a Zigbee node. The node forms a fully-controlled Zigbee network which allows the target device to join. On the one hand, it communicates with the fuzzing controller through the universal asynchronous receiver/transmitter (UART) channel, i.e., receiving testing cases and forwarding device responses. On the other hand, it communicates to the target device through the transparent and authenticated Zigbee network, i.e., transmitting the testing case and monitoring device responses. The driver is developed with 1,000 lines of C codes.

5. Evaluation

To evaluate the effectiveness of LLMIF, we seek answers to the following questions.

Q1: Code coverage. How much more code coverage does LLMIF achieve compared to the baseline?

Q2: Ablation. What is the impact of the extracted protocol knowledge on the performance of LLMIF?

Q3: Bug identification. Is LLMIF capable of discovering previously unknown bugs on real-world devices?

In the following sections, we introduce our implementation and our experimental setup. Finally, we discuss the evaluation results for the above questions.

5.1. Experimental Setup

Z-Stack simulation for code coverage evaluation. During our experiment, we found that it is challenging to evaluate code coverage. The main reason is most Zigbee device vendors do not open-source their stack implementations (either source codes or binary firmware), such that we cannot instrument the stack implementation to calculate statement/edge coverage, and only black-box fuzzing can be initiated. The only open-source Zigbee stack we found is the Z-Stack of Texas Instruments [41]. Inspired by [4], we leverage the IAR development toolchain and set up a simulation platform, which supports simulating stack execution, Zigbee message transmission/reception, and most importantly, coverage analysis. (1) We write a stack driver (900 lines of C code) to build a Zigbee end device

application. The stack driver and the stack source codes are used by the simulation tool C-SPY, which allows us to create a simulated end device as the target device. Our driver registers a set of plugins provided by Z-Stack, such that the simulated end device can support 154 cluster messages from the 22 standard clusters, compared with the driver [4] which only supports 22 messages.
(2) We use shared files to simulate the communication channel between the fuzzer and the simulated target device. By writing and reading the shared file, the fuzzer and the simulated device can simulate the over-the-air Zigbee message transmission/reception, respectively.
(3) Given a generated testing case, we use the code coverage analysis and the static analysis tools provided by the IAR toolchain to calculate the statement coverage and edge coverage, which will be detailed in Section 5.2.
(4) By checking the output of the simulation tool, we determine if the testing case triggers exceptions of the simulated device and causes the crash. The call stack for processing the testing case is used for categorizing the crashing type and clustering crashing cases. Moreover, the crashing cases are stored for further verification on real-world Texas Instruments devices (Section 5.4).

Real-world Zigbee devices under test (DUT). Besides fuzzing the simulation device, we also selected 11 off-the-shelf Zigbee devices from various vendors for evaluation, covering well-known brands, such as Philips, Third Reality, Sengled, Aqara, and Tuya. The types of selected devices include smart switches, plugs, lighting, locks, and sensors. These devices are either recommended by Amazon or the well-selling products in supermarkets. Note that some devices have been found with zero-day vulnerabilities and have not been fixed yet. As a result, we anonymize the names of these devices and their models. We promise to de-anonymize the device details once these vulnerabilities are fixed by the vendor and the disclosure period is reached.

Baseline methods. We select four popular Zigbee fuzzing tools as the baseline: BOOFUZZ [42], Z-FUZZER [4], BEEHIVE [5], and chatAFL [6]. They are either widely used in industry or reported in top security conferences. Moreover, the design of these tools covers the existing solutions to important fuzzing tasks, in particular, message format construction and interesting value collection, which makes them suitable for comparison with our method. Specifically, BOOFUZZ is a famous grammar-based fuzzing tool that is customized by [4] to support Zigbee fuzzing. Z-FUZZER is a coverage-guided Zigbee fuzzing tool, and it has been used to identify critical zero-day vulnerabilities in the Z-Stack. BEEHIVE studies the cluster message format, and manually extracts a set of interesting values for enumeration of field values. Finally, CHATAFL uses the LLM to guide the fuzzing process, in particular, message format construction and has been used for fuzzing network protocols (e.g., RTSP). While it is not designed for Zigbee fuzzing, we use its proposed prompt engineering technique and extend it to support Zigbee message format construction and fuzzing.

LLM usage. In our experiment, we use chatGPT-3.5 as the general-purpose LLM. The LLM accepts one hyper-parameter, *temperature* $\in [0, 1]$, which regulates the randomness and creativity. To mitigate the LLM's hallucination problem (i.e., high false positives in the generated answers), the parameter is set to zero. The same setting applies to the baseline method. Since chatAFL is the only LLM-guided fuzzing tool among the baseline methods, it means that LLMIF and chatAFL will use the same general-purpose LLM (chatGPT-3.5 in our case) with the same temperature setting. Because chatAFL purely relies on the LLM's knowledge base to guide IoT protocol fuzzing, its performance will reveal the impact of LLM's insufficient knowledge (Section 3.1) on IoT protocol fuzzing.

5.2. Coverage Analysis

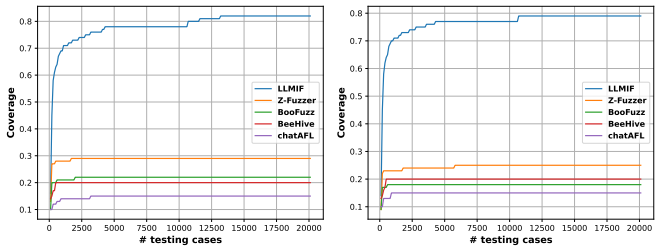
Message coverage analysis. We first evaluate the message coverage when fuzzing with LLMIF, and compare it with the baselines. Specifically, we aim to measure the cluster coverage and message coverage for the 11 devices. To build the ground truth, we first transmit the Zigbee device object (ZDO) commands to the target device and scan its supported clusters and commands. One exception is the device $D1$, which is a CC2538 module loading the Z-Stack. Since Z-Stack is open-source, we compiled a Z-Stack firmware and flashed it into the CC2538 module, such that it supports the 18 standard clusters with 150 message types. Given a fuzzer, to evaluate if it covers a message type, we ask the fuzzer to transmit a message based on its understanding of the message format and check the response status code. If the message is accepted by the target device with the status code SUCCESS, the message type is successfully covered. Otherwise, error codes, e.g., INVALID FORMAT, imply that the message format is incorrect and the message is not covered.

The result is shown in Table 2. Our method successfully covers the 150 message types from the 18 clusters, which achieves 100% cluster and message coverage. Compared with the baselines, our method improves the cluster coverage by 73.1% and improves the message coverage by 55.2%. Specifically, Z-FUZZER, BOOFUZZ, and BEEHIVE rely on human efforts to construct the message format, and they only focus on the foundation cluster, which contains 22 message types in total. For CHATAFL, we initiate 10 rounds for message format construction, and without the guidance of the specification, the LLM only reliably constructs the 15 message formats from the Identify, Groups, and Scenes clusters. Compared with the baseline, we use the augmented LLM to precisely construct all of the message formats.

Code coverage analysis. We further evaluate the code coverage and compare our method with the baseline. Unfortunately, as stated in Section 5.1, most off-the-shelf Zigbee devices do not open source their stack implementations, such that one can only initiate black-box testing and cannot collect code coverage statistics. For example, for the 11 devices from 8 vendors, only the Z-Stack from Texas Instruments is open-sourced. As a result, we use our simulation end device built with Z-Stack (Section 5.1) as the target

TABLE 2: Evaluation results of message coverage and identified vulnerabilities

Device ID	LLMIF			Z-Fuzzer			BooFuzz			BeeHive			chatAFL		
	Cluster	Message	Vul.	Cluster	Message	Vul.	Cluster	Message	Vul.	Cluster	Message	Vul.	Cluster	Message	Vul.
D1	**18(100.0%)**	**147(100.0%)**	**5**	1(5.6%)	15(10.2%)	3	1(5.6%)	15(10.2%)	2	1(5.6%)	22(15.0%)	0	3(16.7%)	15(10.2%)	0
D2	**6(100.0%)**	**55(100.0%)**	**2**	1(16.7%)	15(27.3%)	0	1(16.7%)	15(27.3%)	0	1(16.7%)	22(40.0%)	0	3(50.0%)	15(27.3%)	0
D3	**5(100.0%)**	**39(100.0%)**	0	1(20.0%)	15(38.5%)	0	1(20.0%)	15(38.5%)	0	1(20.0%)	22(56.4%)	0	1(20.0%)	6(15.4%)	0
D4	**9(100.0%)**	**70(100.0%)**	**1**	1(11.1%)	15(21.4%)	0	1(11.1%)	15(21.4%)	0	1(11.1%)	22(31.4%)	0	3(33.3%)	15(21.4%)	0
D5	**6(100.0%)**	**41(100.0%)**	0	1(16.7%)	15(36.6%)	0	1(16.7%)	15(36.6%)	0	1(16.7%)	22(53.7%)	0	3(50.0%)	15(36.6%)	0
D6	**3(100.0%)**	**26(100.0%)**	0	1(33.3%)	15(57.7%)	0	1(33.3%)	15(57.7%)	0	1(33.3%)	22(84.6%)	0	0(0.0%)	0(0.0%)	0
D7	**6(100.0%)**	**41(100.0%)**	**2**	1(16.7%)	15(36.6%)	0	1(16.7%)	15(36.6%)	0	1(16.6%)	22(53.7%)	0	3(50.0%)	15(36.6%)	1
D8	**7(100.0%)**	**51(100.0%)**	0	1(14.3%)	15(29.4%)	0	1(14.3%)	15(29.4%)	0	1(14.3%)	22(43.1%)	0	0(0.0%)	0(0.0%)	0
D9	**6(100.0%)**	**52(100.0%)**	0	1(16.7%)	15(28.9%)	0	1(16.7%)	15(28.9%)	0	1(16.7%)	22(42.3%)	0	0(0.0%)	0(0.0%)	0
D10	**7(100.0%)**	**55(100.0%)**	0	1(14.3%)	15(27.3%)	0	1(14.3%)	15(27.3%)	0	1(14.3%)	22(40.0%)	0	3(42.9%)	15(27.3%)	0
D11	**9(100.0%)**	**67(100.0%)**	**1**	1(11.1%)	15(22.4%)	0	1(11.1%)	15(22.4%)	0	1(11.1%)	22(32.8%)	0	3(33.3%)	15(22.4%)	0
Summary	**100.0%**	**100.0%**	**11**	16.0%	30.6%	3	16.0%	30.6%	2	16.0%	44.8%	0	26.9%	17.9%	1



(a) Edge coverage (b) Statement coverage

Figure 7: Coverage comparison

device for code coverage analysis. Specifically, we use the commonly used metric *statement coverage* (the number of covered lines of codes), and *edge coverage* (the number of covered code branches) for comparison. We first statically analyzed 15 source files that implement the Zigbee cluster functionality to build the ground truth, e.g., "zcl_lighting.c". As a result, a total of 1,665 edges and 3,147 statements are identified. Then for each Zigbee fuzzing tool, we generate 20,000 testing cases and calculate the cumulative rates of statement coverage and edge coverage. The evaluation result is shown in Figure 7a and Figure 7b. Finally, we calculate the coverage improvement by subtracting LLMIF's coverage with the best coverage among baselines. Take statement coverage as an example. Given the total number of 3,147 statements, the best fuzzer (Z-Fuzzer) covers 797 statements while our method covers 2,493. As a result, the improvement is calculated as (2493-797)/3147 = 53.9%. The same applies for the calculation of message coverage.

Compared with the baseline method, LLMIF improves the edge coverage by 52.0% and 53.9% statement coverage, respectively, which shows the out-performance of our method. Generally speaking, the low message coverage results in low code coverage. One exception is BEEHIVE. While it supports more message types than that of Z-FUZZER, it achieves low code coverage due to its insufficient mutation strategies, i.e., only mutating two fields "AttributeID" and "AttributeDataTYpe" with a limited number of interesting values.

Answer to Q1: The experimental result shows that LLMIF outperforms existing Zigbee fuzzers in terms of message coverage and code coverage. Compared with the best baseline fuzzing tool, LLMIF improves the message coverage, edge coverage, and statement coverage by 55.2%, 52.0%, and 53.9%, respectively.

In particular, compared with chatAFL that purely relies on the general-purpose LLM's knowledge base to guide IoT fuzzing, LLMIF improves the message coverage, edge coverage, and statement coverage by 82.1%, 67.4%, and 64.1%. The coverage improvement shows the insufficiency of the general-purpose LLM for fuzzing IoT protocol stacks, and the advantage of our proposed LLM augmentation method and our fuzzing algorithm.

5.3. Ablation Study

LLMIF uses the augmented LLM to extract diverse protocol information for enhancing various fuzzing stages. To evaluate the contribution of the extracted information, we conducted an ablation study. For this purpose, we developed four variants of LLMIF.

- $V1$: LLMIF with only message formats for seed generation.
- $V2$: $V1$ plus the knowledge of interesting values for seed generation.
- $V3$: $V2$ plus the knowledge of the message header for initial seed generation and mutation.
- $V4$: $V3$ plus the knowledge of message dependencies for initial seed generation, mutation, and interesting case enrichment.

Table 3 shows the results in terms of edge and statement coverage. Specifically, we use $V1$ as the baseline and evaluate the improvement of the three LLMIF variants using four metrics: Edge coverage, statement coverage, speed-up (how fast the variant can reach the baseline's edge coverage), and probability (the probability of the variant's coverage outperformance over ten rounds).

The result shows that all of the extracted protocol information plays critical roles in terms of coverage improvement. Specifically, when compared with $V2$, we evaluate the impact of the extracted interesting values. With 8.61%

TABLE 3: Coverage improvement by protocol knowledge

LLMIF variant	Edge coverage	Statement coverage	Speed-up	Probability
V1	1009	1846	1.00	1.00
V2	+8.61%	+7.31%	4.23x	1.00
V3	+25.95%	+22.98%	7.55x	1.00
V4	+35.77%	+35.73%	15.08x	1.00

edge coverage and 7.31% code coverage improvement, we show that testing cases with interesting values effectively explore the protocol stack, especially the error handling logic and the device functioning logic. For example, with the LLM-extracted step mode values 0x01 and 0x03, the fuzzer can generate meaningful Step Saturation messages, which explore the functioning logic to increase and decrease the values of the saturation property, respectively.

By comparing $V2$ with $V3$, we evaluate the impact of the knowledge of message formats and message headers. The result shows that they improve the edge coverage and statement coverage by 17.35% and 15.67%. The knowledge benefits the mutation process in two folds. On the one hand, our mutation operators maintain the message formats of the generated testing cases, and therefore they successfully pass the format checking and avoid being filtered. On the other hand, these operators mutate critical bits and bytes and create a large number of testing cases that are rare in legitimate scenarios, e.g., cluster messages with inverse direction bits.

Finally, we compare $V3$ with $V4$ to evaluate the impact of the message dependencies. As a result, the edge coverage and statement coverage are improved by 9.82% and 12.75%, respectively. The improvement lies in the fact that many code branches condition on device properties, and the generated testing cases can explore these branches by resolving message dependencies and updating device properties before branching. For example, the Get Group Membership message execution will examine the group entry in the group table only if the group size is larger than zero. By resolving the message dependency between the Add Group message and the Get Group Membership message, a testing case with the message sequence [Add Group, Get Group Membership] can effectively activate the branch for entry examination.

Answer to Q2: The extracted protocol information empowers the fuzzing tool for seed generation, mutation, and interesting case enrichment, which increases the code coverage by 8.61%, 17.34%, and 9.82%, respectively.

5.4. Real-world Bugs

We evaluate the effectiveness of vulnerability identification. For each device under testing, we fuzz it for 24 hours and examine the collected crashing cases. The result is shown in Table 2. Specifically, LLMIF successfully reports 11 vulnerabilities on five devices, which include three previously known and eight zero-day vulnerabilities, while the baseline tools only identified one zero-day vulnerability. All of the zero-day vulnerabilities have been reported with five of them being confirmed and three of them under review. The summary of these vulnerabilities is summarized in Table 4, and below we provide their details.

Vulnerability 1, 2, and 3. LLMIF successfully identifies three previously known vulnerabilities (CVE-2020-27890, CVE-2020-27891, CVE-2020-27892) on device $D1$. Specifically, by generating messages with malformed message payloads, the fuzzer triggers the device to incorrectly allocate memory and eventually causes the crash. For the baseline, Z-FUZZER successfully identified all of them, while BOOFUZZ missed CVE-2020-27890. Compared with them, we find different message types which can trigger the vulnerabilities. For example, we show that besides the Read Reporting Configuration Response message, one can also use the Write Attributes Response message to trigger the vulnerability.

Vulnerability 4 and 5. We further discovered two previously unknown vulnerabilities on the device $D1$. Specifically, these two vulnerabilities are triggered when the Disable Default Response bit in the header is set to 0, i.e., the response is required. As a result, the device crashes after it generates the response and prepares to transmit it. Moreover, no baseline fuzzers identified these two vulnerabilities because they neither know the corresponding message format nor the header bit.

Vulnerability 6 and 7. We discovered two vulnerabilities on the device $D2$, which relate to the AddScene and EnhanceAddScene messages. Specifically, both of these two messages' payloads contain a string field "SceneName". With our type-aware mutation operators, in particular, *suspicious length*, LLMIF changed the first byte which denotes the string length to a large value. As a result, when the target device parses the length byte, it triggers byte overflow, and the device crashes. No baseline methods identified these two vulnerabilities because they missed the message format.

Vulnerability 8. We discover a vulnerability on devices $D4$ and $D11$, which is triggered by a message sequence. Specifically, the device first accepts an AddGroup message with a dangerous field value: GroupID=0x0000. This message puts the device in a dangerous state. Then our fuzzer extends the testing case with a GetGroupMembership, which is used for querying the group list with a field named "GroupCount". We find that the GroupCount field can arbitrarily control the length of the returned group list. For example, if groupCount field equals 3, the target device will return 6 bytes representing three GroupIDs. Surprisingly, all the returned GroupIDs are 0x0000. As a result, by setting the GroupCount field to a relatively large number, e.g., 0xA0, the target device tries to return a long group list, and a large number of bytes overflows the transmission buffer, which crashes the device. No baseline fuzzers identified the vulnerability because of the lack of knowledge about dangerous field values and message dependency relationships.

Vulnerability 9. This vulnerability exists on the device $D7$. Specifically, the message payload has two fields: GroupCount (uint8) and GroupList (Array[uint16]). When

TABLE 4: Case studies of discovered vulnerabilities

Vulnerability ID	Device ID	Vulnerable Message Sequences	Cause	Security Issue	Status
1	D1	[DiscoverCommandsResponse]	Field removal mutation	Device crashed	CVE-2020-27892
2	D1	[WriteAttributesResponse], [ReadReportingConfigurationResponse]	Field removal mutation	Device crashed	CVE-2020-27891
3	D1	[WriteAttributesNoResponse]	Field removal mutation	Attribute update failed	CVE-2020-27890
4	D1	[WriteAttributesUndivided]	Header-aware mutation	Device crashed	Under review
5	D1	[PowerProfileRequest], [WriteAttributesResponse]	Header-aware mutation	Device crashed	Under review
6	D2	[AddScene]	Type-aware mutation	Device crashed	Confirmed
7	D2	[EnhanceAddScene]	Type-aware mutation	Device crashed	Confirmed
8	D4, D11	[AddGroup, GetGroupMembership]	Interesting field value, message dependency	Device crashed	Confirmed
9	D7	[GetGroupMembership]	Type-aware mutation	Device crashed	Confirmed
10	D7	[Non-existing command]	Header-aware mutation	Device bricked	Under review

GroupCount is larger than the number of GroupID in the list, the device crashes. By applying the extreme-value mutation operators on the GroupCount field, our fuzzer successfully triggered the vulnerability. CHATAFL also reported this vulnerability.

Vulnerability 10. Finally, we discovered a vulnerability on the device $D7$, which is triggered by a message with a malformed header. Specifically, by mutating the message identifier field in the header with a non-existing message identifier 0x13, the device stopped working. Even worse, it becomes bricked and does not work anymore, even when we push the factory reset button and recycle the power. No baselines report the vulnerability due to the lack of knowledge about the message header.

Answer to Q3: LLMIF is effective in vulnerability identification, which reveals three previously known and eight previously unknown vulnerabilities, respectively. Among the eight previously unknown vulnerabilities, seven of them (87.5%) are not reported by the baseline fuzzers.

6. Discussions

LLM for network protocol fuzzing. chatAFL [6] is the first work which uses the LLM to fuzz network protocols. However, with the same general-purpose LLMs (e.g., chatGPT 3.5), LLMIF outperforms chatAFL in terms of code coverage and bug identification for the following reasons.

(1) *Specification-augmented prompting for driving LLMs.* LLMIF uses the background-augmented prompting method (Section 3.2) which drives the general-purpose LLM to analyze specification contents. Compared with chatAFL that directly drives the general-purpose LLM by the few-shot learning prompting method, LLMIF successfully aids the issue of LLM's insufficient domain knowledge, and achievese precise protocol information extraction.

(2) *Advanced fuzzing algorithms for guiding various fuzzing phases.* LLMIF utilizes the output of the LLM (i.e., extracted protocol information) to guide the phases of seed generation, seed mutation, and testing case enrichment. Moreover, LLMIF leverages the LLM to reason about the device response and guides the phase of testing case prioritization. Compared with chatAFL that uses the LLM to guide only two phases (seed mutation and testing case enrichment), LLMIF achieves a broader (four) phase coverage, and results in a significant improvement for the code coverage (Section 5.3).

LLMIF's generality for other IoT protocols. In this paper, we mainly use Zigbee as the target protocol. However, LLMIF can be extended to fuzz IoT protocols beyond Zigbee. Specifically, users only need to take two steps, i.e., update the input specification and replace the hardware radio. To demonstrate LLMIF's generality, we target the Z-Wave protocol and design a case study. More details can be found in our complementary materials [15].

Update the input specification. LLMIF relies on the protocol specification to augment the LLM and extract critical message information. As a result, users need to provide as inputs the specification that details the messages of the protocol under fuzzing, e.g., Z-Wave Command Classes [43]. With our document slicing and the background-augmented prompting methods (Section 3.2), LLMIF will extract the precise protocol information (e.g., message format and message dependency), and use them to guide the fuzzing round.

Replace the hardware radio. To fuzz the real-world device with a specific IoT protocol, a hardware radio is necessary for transmitting the testing case and receiving the response within a specific wireless channel. Users need to prepare the corresponding hardware (e.g., CC2530 for Zigbee), download the commercial protocol stack (e.g., Z-Stack), and implement the driver that interacts with LLMIF. The implementation of the driver depends on the development environment of the commercial protocol stack, and our implemented driver for Z-Stack provides an example (Section 4.6).

7. Related Work

Format-aware IoT fuzzing. Format-aware fuzzing tools generate messages based on the well-formed message template [3], [4], [5], [10], [42], [44]. Specifically, [42] proposes

a network protocol fuzzing framework BooFuzz. Taking the message format description and the interesting value collection as inputs, the tool automates the testing case generation, monitors the target's status, and records suspicious cases. [4] bases on BooFuzz and designs a fuzzing tool Z-Fuzzer, which aims at fuzzing Zigbee protocol and uses code coverage as feedback to guide the fuzzing process. [5] manually extracts 22 cluster message formats and interesting field values (e.g., attribute ID) from the specification and develops BeeHive, which generates testing cases to enumerate these messages and field values. [10] relies on the message formats recorded in the open-source protocol library to generate testing cases. All of them require significant human efforts to construct and maintain the message formats, which are labor-intensive and error-prone. [3], [44] proposes to use phone apps and APIs that control the device through the vendor platform to construct the message formats. However, they mainly focus on inferring the platform-level message format (e.g., Restful API messages), which only covers a small fraction of the message formats of the underlying protocol that is directly used by the device, e.g., Zigbee.

Compared with the work mentioned above, we propose to utilize the LLM to address the main challenge of format-aware IoT fuzzing. With only the specification document as input, our method automatically constructs the message format with high accuracy while avoiding significant human efforts. Besides the knowledge of message formats, our method also extracts useful protocol information, e.g., interesting values and message dependencies, which benefits the fuzzing process and is omitted by previous works. In particular, our evaluation covers most Zigbee fuzzers [4], [5], [42], and the evaluation result shows that our method outperforms them in terms of code coverage and vulnerability identification. Since some fuzzers [10] are not open-source, we cannot evaluate their performance.

LLM-guided fuzzing. Impressed by the remarkable success of the LLM in various natural language processing tasks, researchers have been exploring the LLM's potential in diverse domains, including in fuzzing [6], [45], [46], [47], [48]. Specifically, [45] proposes CODAMOSA which uses LLMs to automatically generate testing cases for fuzzing Python modules. [46] uses LLMs to generate testing cases for fuzzing deep-learning software libraries. [48] proposes Fuzz4All which takes example codes as inputs and generates slightly different code snippets as testing cases. Instead of asking the LLM to directly generate testing cases, chatFuzz [47] asks the LLM to modify human-written testing cases for mutation purposes. The most related work to our paper is chatAFL [6], which uses the LLM to fuzz network protocols. By exploiting the LLM's understanding of the target protocol, chatAFL asks the LLM to construct message formats and enrich seeds. All of the mentioned LLM-guided fuzzers are built on the assumption that the LLM has sufficient domain knowledge, e.g., Python grammar and protocol details, such that they can be easily adapted to the fuzzing task. Instead, our work initiates a case study for Zigbee fuzzing to show that the assumption does not hold. Moreover, we propose an LLM augmentation approach that feeds the LLM with specification contents, and the evaluation result shows that the augmented LLM can effectively guide various fuzzing phases.

Augmented language model. LLMs by default possess general knowledge across a wide range of topics. However, [24], [25] show that LLMs may not obtain enough knowledge for specific domains. The reasons are various, e.g., knowledge bias exists that more popular or widely-discussed topics, e.g., python programming, maybe over-represented, while very domain-specific topics, e.g., Zigbee protocol, can usually be under-represented. As a result, a series of works are proposed to augment the language model with domain knowledge [31], [32], [49], [50], [51], [52]. Specifically, [31], [32], [49] employ a neural retriever to acquire task-relevant information from a knowledge base, e.g., Wikipedia, and fine-tune the language model. [50], [51], [52], on the other hand, teach the LLMs to call domain tools for answer generation, e.g., writing SQL to query from the database. Our work follows the former paradigm, i.e., retrieving domain knowledge and enhancing the LLM. Different from previous works, we use document analysis to retrieve domain knowledge instead of training neural networks. Moreover, we use prompt engineering to feed the LLM which gets rid of the fine-tuning cost.

8. Conclusion

Fuzzing the IoT protocol(e.g., Zigbee) faces critical challenges including obfuscated message formats, unresolved message dependencies, and lack of testing case evaluation. In this paper, we show that the protocol specification contains fruitful message descriptions, which can be used to address the limitations and guide the fuzzing process. To automate specification analysis, we further augment the large language model with the specification contents and drive it to answer protocol-related questions. Finally, we propose a fuzzing algorithm LLMIF, which incorporates the augmented LLM into Zigbee fuzzing. The evaluation result shows that guided by the LLM, our method increases the message coverage by 55.2% and achieves a 53.9% boost in code coverage. Moreover, LLMIF reveals 11 vulnerabilities on real-world Zigbee devices(including eight zero-day vulnerabilities), while the baseline tools only discovered one zero-day vulnerability.

Acknowledgments

We thank the anonymous (meta) reviewers and shepherds for their valuable comments and instructions. This work is partly supported by HKPolyU Grant (ZVG0), Hong Kong RGC TRS No. T43-513/23-N, Hong Kong ITF project (ITS/359/21FP), National Natural Science Foundation of China (No. 62202406), Natural Science Research Start-up Foundation of Recruiting Talents of Nanjing University of Posts and Telecommunications (Grant No. NY223164).

References

[1] C. Berger, P. Eichhammer, H. P. Reiser, J. Domaschka, F. J. Hauck, and G. Habiger, "A survey on resilience in the iot: Taxonomy, classification, and discussion of resilience mechanisms," *ACM Computing Surveys (CSUR)*, vol. 54, no. 7, pp. 1–39, 2021.

[2] J. Wang, Z. Li, M. Sun, B. Yuan, and J. C. Lui, "Iot anomaly detection via device interaction graph," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2023, pp. 494–507.

[3] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[4] 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*, 2021, pp. 347–358.

[5] 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*, 2022, pp. 2857–2870.

[6] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[7] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[8] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–33, 2022.

[9] C. Zhang, Y. Li, H. Chen, X. Luo, M. Li, A. Q. Nguyen, and Y. Liu, "Biff: Practical binary fuzzing framework for programs of iot and mobile devices," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1161–1165.

[10] 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*, 2023, pp. 205–218.

[11] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

[12] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[13] J. Wang, Z. Li, M. Sun, and J. C. Lui, "Zigbee's network rejoin procedure for iot systems: vulnerabilities and implications," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 292–307.

[14] D. Gislason, *Zigbee wireless networking*. Newnes, 2008.

[15] "LLMIF," https://github.com/wang70880/LLMIF, 2024.

[16] Z. Alliance, "Zigbee Cluster Specification," https://zigbeealliance.org/wp-content/uploads/2021/10/07-5123-08-Zigbee-Cluster-Library.pdf, 2021.

[17] Y. Liu, A. R. Fabbri, P. Liu, D. Radev, and A. Cohan, "On learning to summarize with large language models as references," *arXiv preprint arXiv:2305.14239*, 2023.

[18] D. Van Veen, C. Van Uden, L. Blankemeier, J.-B. Delbrouck, A. Aali, C. Bluethgen, A. Pareek, M. Polacin, W. Collins, N. Ahuja *et al.*, "Clinical text summarization: Adapting large language models can outperform human experts," *arXiv preprint arXiv:2309.07430*, 2023.

[19] K. Mao, Z. Dou, H. Chen, F. Mo, and H. Qian, "Large language models know your contextual search intent: A prompting framework for conversational search," *arXiv preprint arXiv:2303.06573*, 2023.

[20] B. Lamichhane, "Evaluation of chatgpt for nlp-based mental health applications," *arXiv preprint arXiv:2303.15727*, 2023.

[21] Z. Wang, F. Yang, P. Zhao, L. Wang, J. Zhang, M. Garg, Q. Lin, and D. Zhang, "Empower large language model to perform better on industrial domain-specific question answering," *arXiv preprint arXiv:2305.11541*, 2023.

[22] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "Pentestgpt: An llm-empowered automatic penetration testing tool," *arXiv preprint arXiv:2308.06782*, 2023.

[23] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[24] X. ZHAO, J. LU, C. DENG, C. ZHENG, J. WANG, T. CHOWDHURY, L. YUN, H. CUI, Z. XUCHAO, T. ZHAO *et al.*, "Domain specialization as the key to make large language models disruptive: A comprehensive survey," *arXiv preprint arXiv:2305.18703*, 2023.

[25] Z. Luo, C. Xu, P. Zhao, X. Geng, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Augmented large language models with parametric knowledge guiding," *arXiv preprint arXiv:2305.04757*, 2023.

[26] openAI, "ChatGPT," https://openai.com/chatgpt, 2023.

[27] Meta, "LLama 2 - Meta AI," https://ai.meta.com/llama/, 2023.

[28] Google, "AI ACROSS GOOGLE: PaLM 2," https://ai.google/discover/palm2/, 2023.

[29] Anthropic, "Meet Claude," https://www.anthropic.com/product, 2023.

[30] G. Izacard and É. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, 2021, pp. 874–880.

[31] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. B. Van Den Driessche, J.-B. Lespiau, B. Damoc, A. Clark *et al.*, "Improving language models by retrieving from trillions of tokens," in *International conference on machine learning*. PMLR, 2022, pp. 2206–2240.

[32] D. Singh, S. Reddy, W. Hamilton, C. Dyer, and D. Yogatama, "End-to-end training of multi-document reader and retriever for open-domain question answering," *Advances in Neural Information Processing Systems*, vol. 34, pp. 25 968–25 981, 2021.

[33] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," *Advances in neural information processing systems*, vol. 32, 2019.

[34] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 7871–7880.

[35] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2021.

[36] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen *et al.*, "Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models," *arXiv preprint arXiv:2203.06904*, 2022.

[37] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[39] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing."

[40] 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*, 2008, pp. 206–215.

[41] T. Instruments, "A fully compliant ZigBee 3.x solution: Z-Stack," https://www.ti.com/tool/Z-STACK, 2017.

[42] J. Pereyda, "boofuzz documentation," *THIS REFERENCE STILL NEEDS TO BE FIXED*, 2019.

[43] Z.-W. Alliance, "Z-Wave Specifications," https://z-wavealliance.org/development-resources-overview/specification-for-developers/, 2023.

[44] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[45] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

[46] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[47] J. Hu, Q. Zhang, and H. Yin, "Augmenting greybox fuzzing with generative ai," *arXiv preprint arXiv:2306.06782*, 2023.

[48] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv preprint arXiv:2308.04748*, 2023.

[49] Q. Liu, D. Yogatama, and P. Blunsom, "Relational memory-augmented language models," *Transactions of the Association for Computational Linguistics*, vol. 10, pp. 555–572, 2022.

[50] M. Komeili, K. Shuster, and J. Weston, "Internet-augmented dialogue generation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 8460–8478.

[51] Z. Cheng, T. Xie, P. Shi, C. Li, R. Nadkarni, Y. Hu, C. Xiong, D. Radev, M. Ostendorf, L. Zettlemoyer *et al.*, "Binding language models in symbolic languages," in *The Eleventh International Conference on Learning Representations*, 2022.

[52] J. Li, B. Hui, G. Qu, B. Li, J. Yang, B. Li, B. Wang, B. Qin, R. Cao, R. Geng *et al.*, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," *arXiv preprint arXiv:2305.03111*, 2023.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The research work introduces a new IoT fuzzing approach that leverages protocol specifications to create more effective test cases. By using a large language model to analyze specifications, the method improves code and message coverage, leading to the discovery of zero-day vulnerabilities in Zigbee implementations.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

1) LLMIF tackles various stages of fuzzing, leading to more thorough testing and vulnerability discovery. Compared to traditional approaches, LLMIF achieves over 50% improvement in code and message coverage, making it more likely to uncover hidden bugs.
2) In real-world testing on Zigbee devices, LLMIF successfully identified four zero-day vulnerabilities, highlighting its effectiveness in uncovering critical security issues.
3) The impressive results of LLMIF showcase its potential as a powerful tool for securing Zigbee devices. With its ability to achieve wider code coverage and identify critical vulnerabilities, LLMIF paves the way for more robust and secure Zigbee implementations.

A.4. Noteworthy Concerns

1) The paper lacks discussion of LLMIF's novelty compared with chatAFL.
2) The paper lacks discussion of LLMIF's generalization for fuzzing other IoT protocols.
3) The paper misses evaluation setup details (e.g., used LLM version) and evaluation results compared with chatAFL.

Appendix B. Response to the Meta-Review

We would like to express our great gratitude towards the efforts made by the (meta) reviewers and shepherds.

We acknowledge the meta-review and have adhered to the provided instructions. Subsequently, we have implemented the following revisions to enhance our paper.

Clarify the novelty of LLMIF against chatAFL. We added a new section (Section 6) to discuss the novelty of LLMIF. In summary, We outline the novel aspects of LLMIF in comparison to chatAFL as follows. (LLMIF) Specification-augmented prompting v.s. (chatAFL) Simple few-shot learning prompting. (LLMIF) Advanced fuzzing algorithm which uses the LLM to cover four fuzzing phases v.s. (chatAFL) Simple fuzzing algorithm which uses the LLM to cover two fuzzing phases.

Clarify the generalization of LLMIF for fuzzing other IoT protocols. We added a new section (Section 6) to discuss the generality of LLMIF. In summary, we show how to extend LLMIF for fuzzing other IoT protocols with two steps.

- Users need to update the input protocol specification.
- Users need to replace the hardware radio.

The paper misses some evaluation results and setup details. We have updated the corresponding section (Section 5.1) in our paper to add more details about the evaluation. In summary: We updated Section 5.1 to introduce the details of the LLM setup in our evaluation, including the used LLM (chatGPT-3.5) and the parameter setting (temperature = 0). We updated Section 5.2 to compare the code coverage result between LLMIF and chatAFL. Specifically, compared with chatAFL, LLMIF improves the message coverage, edge coverage, and statement coverage by 82.1%, 67.4%, and 64.1%, respectively.