1 Generality of LLMIF

In this section, we show that users can easily extend LLMIF to fuzz other protocols with only two steps. Then we provide an example to show that LLMIF can be easily used to fuzz other IoT protocols (Z-Wave protocol in our case).

1.1 Two Steps for Extending LLMIF to Fuzz Other IoT Protocols

LLMIF is designed for fuzzing general IoT protocols which have explicit specifications. When extending LLMIF for fuzzing a new IoT protocol beyond Zigbee, users only need to take the following two steps: *Update the input specification* and *replace the hardware radio.*

1. Update the input specification. LLMIF relies on the protocol specification to augment the LLM and extract critical message information (e.g., message format, header structure, and message dependency). As a result, users only need to provide the following as the input: (1) The specification that details the messages of the protocol under fuzzing, and (2) several keywords that specify the protocol information to be extracted (e.g., the Zigbee cluster name in which the message information is to be extracted).

With the updated user inputs, LLMIF will take the following operations automatically.

- As described in Section 3.2 of our paper, LLMIF will call the augmentation method to analyze the document and filter out important document pieces (e.g., sub-sections that detail each message under the Groups cluster).
- As described in Section 4.1 of our paper, with our specified prompts, LLMIF is required to further extract the protocol information from the document pieces (e.g., the message format and the message functionality). The output of this phase is well-formed protocol knowledge (e.g., message formats in JSON which records the field names and field data types).
- As described in Section 4.2 to Section 4.5 of our paper, with the extracted protocol information, LLMIF will start the fuzzing loop: It will iteratively perform seed generation, mutation, response reasoning, and testing case enrichment. Since all the protocol message formats have been extracted in the form of separate fields, our building-block approach (Section 4.6) allows LLMIF to construct the message snippet and perform

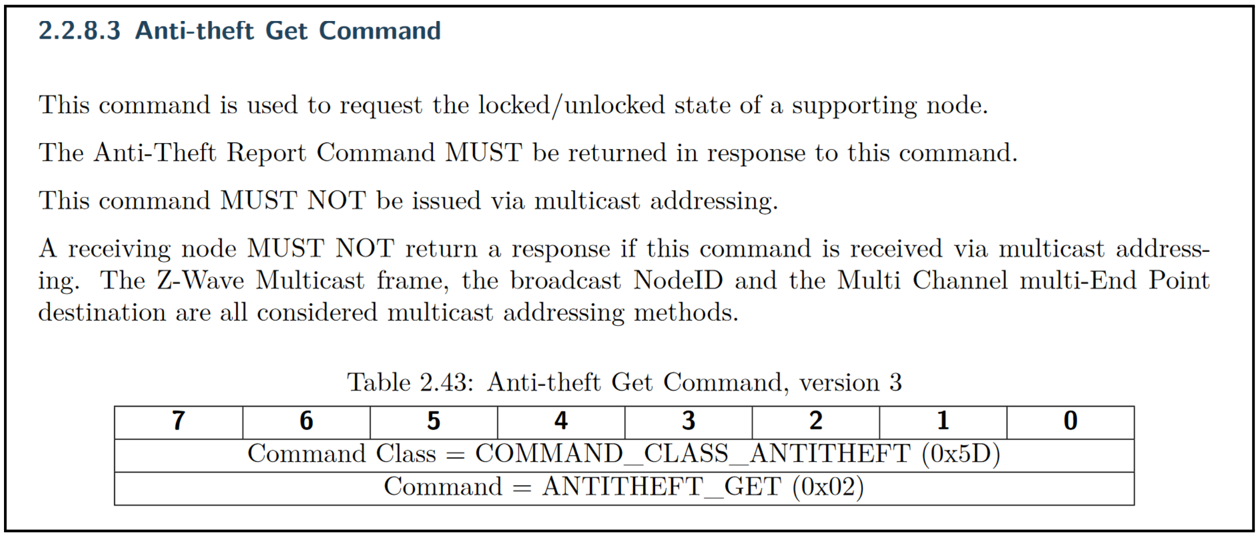
various mutation strategies (type-aware and header-aware mutation operations) in a protocol-agnostic way.

2. Replace the hardware radio. To communicate with the real-world device under testing, a hardware radio is necessary for managing the testing network (e.g., the proprietary Zigbee network), transmitting the testing case, and receiving the response within a specific wireless channel. Given a specific IoT protocol, users need to buy 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 (Section 4.6). 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. Since this is out of our research scope, we will not dive into the details of this part. However, in the future, we will maintain the LLMIF project and provide hardware solutions to different protocols, e.g., the support of Bluetooth and Z-Wave.

1.2 Example of Customizing LLMIF for Fuzzing Z-Wave Protocol

In this section, we illustrate how LLMIF is used for fuzzing IoT protocols beyond Zigbee. In particular, we pick Z-Wave as the target protocol, which is also a leading wireless protocol for IoT devices. Recall that in Section 1.1 we show two steps for extending LLMIF. Due to the time limit, we do not have Z-Wave hardware radio and we cannot complete the fuzzing process for fuzzing real-world Z-Wave devices. As a result, in this section, we mainly show that LLMIF can generate Z-Wave high-quality fuzzing cases by simply updating the input specification, which is a major contribution of our work. Once the hardware radio is ready, the generated fuzzing cases can be directly transmitted to the device under testing for fuzzing purposes.

To generate Z-Wave testing cases, we use the open-source Z-Wave specification as the input to LLMIF. In particular, Z-Wave organizes the messages in the concept of command classes. As a result, we use the name of the command class as the keyword, such that LLMIF can extract the message information inside the command class. We use the Anti-theft Command Class for illustration purposes. With the specification and the command class name, LLMIF first calls the augmentation method to retrieve the related sections discussing the command class and its supported commands. By doing so, it retrieves document slices



2.2.8.3 Anti-theft Get Command

This command is used to request the locked/unlocked state of a supporting node.

The Anti-Theft Report Command MUST be returned in response to this command.

This command MUST NOT be issued via multicast addressing.

A receiving node MUST NOT return a response if this command is received via multicast addressing. The Z-Wave Multicast frame, the broadcast NodeID and the Multi Channel multi-End Point destination are all considered multicast addressing methods.

Table 2.43: Anti-theft Get Command, version 3

7	6	5	4	3	2	1	0
Command Class = COMMAND_CLASS_ANTITHEFT (0x5D)							
Command = ANTITHEFT_GET (0x02)							

Figure 1: Examples of the document slices

introducing the three supported commands: Anti-Theft Set, Anti-theft Get, and Anti-Theft Report. An example of the slice is shown in Figure 1. From the figure, one can check that the specification usually specifies the message format in specific forms (e.g., tables), which is difficult for using traditional methods (e.g., text analysis) to extract. Moreover, the functionality of the command and its dependent device state (locked/unlocked) is specified in the natural language, increasing the complexity for the machine to understand the command and build command dependency relationships.

LLMIF leverages the powerful NLP capabilities of the LLM to absorb the document and extract useful information With the document slice and our prompting method, LLMIF aims to extract the message format, header structure, and message dependency relationships, respectively.

- The example prompt for message format extraction is shown in Listing 1, and the response is shown in Listing 2. One can check that LLMIF successfully drives the augmented LLM to precisely construct the format of the Anti-Theft Set message, which contains seven fields. More specifically, we randomly select 5 command classes containing 13 message types, and the evaluation result shows that LLMIF can precisely construct their formats.

- We also use LLMIF to construct the header structure of Z-Wave messages. As a result, it successfully identifies the two fields contained in the header (1-byte command class ID and 1-byte command ID), which allows to perform header-aware mutation operations.

- Finally, we use LLMIF to construct the message dependency, and the prompt is shown in Listing 3. Our result shows that among the 13 messages, LLMIF reports 8 message dependency relationships which are all confirmed by us. An example response is shown in Listing 4 when testing whether the "Anti-Theft Unlock Set" command $\rightarrow$ "Anti-Theft Unlock State Get" command holds. From the response, one can check that the augmented LLM can accurately understand the functionality of each message, and analyze the dependency relationship, given our specified criteria.

Summary: Given the user-provided Z-Wave specification, LLMIF can precisely extract the message information (e.g., message format, header structure, and message dependency). Later the extracted information can be seamlessly incorporated into the fuzzing process for seed generation, mutation, response reasoning, and testing case enrichment.

2 Detailed Introductions to Preliminary Study (Section 3 of Our Paper)

In this section, we provide a detailed example to illustrate how we conduct the preliminary study (section 3 of our paper) and derive the conclusion that existing general-purpose LLMs do not have sufficient understanding of the IoT protocol, e.g., Zigbee.

2.1 Task Description

We pick one of the most fundamental fuzzing tasks to evaluate whether general-purpose LLMs have a good understanding of the Zigbee protocol, i.e., construction of the Zigbee message format. Given a Zigbee cluster name, the LLM should display all the message names and formats inside the cluster. To better analyze the evaluation result, we split the task into two sub-tasks.

- *Message identification*: the LLM should output the correct message name inside the cluster.

```
The description of a communication protocol message
called "Anti-Theft Set" is as follows.

{documentSlice}

# Instructions
According to the description, please list the field names and field data
types in the payload of the message, which are usually displayed in a
Table or Figure. Strictly follow the description, and do not generate any
fake field names or field types. Note that a message payload may contain
zero fields. To keep the answer as concise as possible, follow
Python list grammar to display your answer. Followings are several
examples for your reference.

# Example Outputs
Example Output 1 (Two fields):
[("Enable": "1 bit"), ("Magic Code", "N bytes")]

Example Output 2 (Two fields):
[("GroupID": "uint8"), ("GroupName", "string")]

Example Output 2 (Empty payloads):
[]
```

Listing 1: Prompt for extracting the format of Anti-Theft Set message

```
[('State', '1 bit'),
 ('Magic Code Length', '7 bits'),
 ('Magic Code', '1 byte'),
 ('Manufacturer ID', '2 bytes'),
 ('Anti-theft Hint Length', '8 bits'),
 ('Anti-theft Hint', '1 byte'),
 ('Z-Wave Alliance locking entity ID', '2 bytes')]
```

Listing 2: Example response for constructing the format of Anti-Theft Set command

```
Followings are descriptions of a communication protocol message called
"{precedingMessageName}".

"{precedingMessageDescription}"

Followings are descriptions of another protocol message called
"{consecutiveMessageName}".

"{consecutiveMessageDescription}"

Please base on the above message description and determine if the
following statement holds: "The consequence of the first message execution
explicitly updates some device properties which the second message
explicitly checks/examines". Think step by step and let me know your
thoughts. Finally, summarize your answer in one sentence: "Statement hold"
or "Statement does not hold".
```

Listing 3: Prompt for constructing the message dependency relationship

```
Based on the descriptions of the two messages, the first message
"Anti-Theft Unlock Set Command" is used to unlock a node and change its
state to unlocked, while the second message "Anti-Theft Unlock State Get
Command" is used to request the locked/unlocked state of the node.

Therefore, the second message explicitly checks/examines the state of the
device, i.e., the lock/unlock state, which is updated by the execution of
the first message.

In conclusion, the statement holds.
```

Listing 4: Example response for constructing the message dependency relationship

- *Format inference*: For the correctly identified message name, the LLM should output the correct field names and field data types.

2.2 Experiment Setup

LLM Selection. In our original study (Section 3.1), we select four popular general-purpose LLMs as the target: openAI's chatGPT 3.5, Meta's LLama 2, Google's PaLM, and Anthropic's Claude. In this complementary study, we only pick chatGPT 3.5 for the demonstration purpose.
Used Prompt. Considering that the state-of-the-art fuzzer (chatAFL) successfully used the general-purpose LLM to infer message formats of small network protocols, in this preliminary study, we follow chatAFL's approach and specify the prompt, which is shown in Figure 5.

Note that we embrace all the techniques used in chatAFL, such that our prompt strictly adheres to the SOTA method for driving LLMs.

- *In-context learning*: Tune the LLM to solve domain-specific tasks (e.g., Zigbee) with its recorded domain knowledge.
- *Few-shot learning*: Enable the LLM to recognize the input prompt syntax and output patterns.
- *Self-consistency check*: To eliminate the hallucination and randomness of the LLM's answer, we ask the LLM to re-answer the same question several times and aggregate the most frequent answer as the final answer.

2.3 Example

Figure 6 is an example of the answer of the general-purpose LLM. Specifically, we ask the LLM to derive the message format of the Poll Control cluster, which consists of four ground-truth messages: (1) Check-in Response, (2) Fast Poll Stop, (3) Set Long Poll Interval, and (4) Set Short Poll Interval.

For the task of command identification, one can check that LLM only identifies two ground-truth message names (i.e., Set Long Poll Interval and Set Short Poll Interval), which only achieves 50.0% recall. At the same time, it also reports six spoofed message names, which results in a low precision of 25.0%.

```
You are a Zigbee expert.

# Instructions
Now you are required to list the command names and formats specified in
the cluster named "{clusterName}". Display your answer in JSON format, and
the template is shown in the Desired Format section. We also provide an
example for your reference, which is shown in the Example Answer section.
To keep the answer concise, only show the resulting JSON.

# Desired Format
{
cluster_name: {
    cmd_name: {
           field1_name: field1_data_type,
           field2_name: field2_data_type,
           ......
           },
.          ......
  },
  ......
}

# Example Answer
{
"Groups": {
    "Add Group": {
        "Group ID": "uint16",
        "Group Name": "string"
    },
    "View Group": {
        "Group ID": "uint16"
    }
}
}
```

Listing 5: Prompt for driving general-purpose LLMs to infer message format

For the task of format inference, we further investigate that if the message format is inferred correctly for the two successfully identified ground-truth messages. As a result, only the Set Short Poll Interval message format is correctly inferred, which results in 50% recall. For the Set Short Poll Interval message format, since its ground-truth format only contains one field (uint32 LongPollInterval), the LLM wrongly generates a spoofed field (uint16 ShortPollInterval), and the resulting format is ruined. As a result, it only achieves 50% precision.

Summary. Among four ground-truth messages in the Poll Control cluster, the general-purpose LLM only successfully constructs one of them, which results in only 25.0% construction accuracy. For the task of command identification, it only achieves 25% precision and 50% recall, which indicates missing and ambiguous understandings about what messages are supported in a Zigbee cluster. For the task of format inference, it only achieves 50% precision and 50% recall, which indicates missing and ambiguous understandings about the detailed message format. Instead, by augmenting the LLM with the specification, our method successfully infers the message format of all four ground-truth messages (100% construction accuracy v.s. 25.0% construction accuracy), which shows the necessity of LLM augmentation and the effectiveness of our approach.

```json
{
"Poll Control": {
"Set Long Poll Interval": {
"LongPollInterval": "uint32",
"ShortPollInterval": "uint16"
},
"Set Short Poll Interval": {
"ShortPollInterval": "uint16"
},
"Set Fast Poll Mode": {

},
"Set Slow Poll Mode": {

},
"Set Long Poll Interval Max": {
"LongPollIntervalMax": "uint32"
},
"Set Short Poll Interval Max": {
"ShortPollIntervalMax": "uint16"
},
"Set Fast Poll Timeout": {
"FastPollTimeout": "uint16"
},
"Set Poll Control Period": {
"PollControlPeriod": "uint16"
}
}
}
```

Listing 6: LLM’s answer for inferring message format in Poll Control cluster