From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices

Xiaoyue Ma
George Mason University

Lannan Luo
George Mason University

Qiang Zeng
George Mason University

Abstract

Matter is an IoT connectivity standard backed by over two hundred companies. Since the release of its specification in October 2022, numerous IoT devices have become Matter-compatible. Identifying bugs and vulnerabilities in Matter devices is thus an emerging important problem. This paper introduces `mGPTFuzz`, the first Matter fuzzer in the literature. Our approach harnesses the extensive and detailed information within the Matter specification to guide the generation of test inputs. However, due to the sheer volume of the Matter specification, surpassing one thousand pages, manually converting human-readable content to machine-readable information is tedious, time-consuming and error-prone. To overcome this challenge, we leverage a large language model to successfully automate the conversion process. `mGPTFuzz` conducts stateful analysis, which generates message sequences to uncover bugs that would be challenging to discover otherwise. The evaluation involves 23 various Matter devices and discovers 147 new bugs, with three CVEs assigned. In comparison, a state-of-the-art IoT fuzzer finds zero bugs from these devices.

1 Introduction

Matter is an open, royalty-free IoT connectivity standard, which is endorsed by over two hundred companies, including Apple, Google, Amazon, and Samsung [6]. It aims to establish a unified standard [66], facilitating interoperability among smart devices of different vendors. With the support of many large technology companies, this unified standard is expected to completely change the IoT ecology [59].

Since the release of the Matter specification in October 2022, Amazon has turned 100 million Echo devices to be Matter-compatible through a software update [4]. Google has updated billions of smart devices to support Matter [58]. Apple TV and HomePod devices extend support for Matter through the installation of iOS 16.1 [28]. Given the popularity, discovering bugs and vulnerabilities in Matter devices is an emerging important problem.

To find IoT bugs, significant efforts have been dedicated to emulating IoT firmware to facilitate greybox or white-box fuzzing [11, 16, 37, 76, 77]. Nevertheless, emulating IoT firmware remains challenging due to the extensive array of custom and proprietary hardware components. Constructing a precise emulator is complex and difficult [51, 69]. Consequently, blackbox fuzzing emerges as an attractive option and has demonstrated noteworthy results [12, 22, 35, 51].

To conduct blackbox fuzzing for Matter, our observation is that the Matter specification contains extensive and detailed information regarding the device behaviors, such as the valid and invalid values for each parameter in a command, the expected effect, and the response message after a command is executed. It is a promising approach to make use of the rich information in a specification for test input generation. To build this approach to Matter fuzzing, however, multiple challenges arise.

Challenge 1 (C1): Sheer volume of specification. The Matter specification describes a unified connection standard that facilitates interoperability of various devices across vendors. It is not surprising that the specification contains thorough details, contained in 1,258 pages. Manually converting the large amount of human-readable content to machine-readable information is tedious, time-consuming and error-prone.

Challenge 2 (C2): Stateful bugs. Many IoT commands only make sense when the device is at a specific state. For example, a command turning off a light has an effect only when the light is on. A bug that can only be triggered when the device is at a certain state is called a *stateful bug* [7]. Prior IoT blackbox fuzzers, such as `IoTFuzzer` [12] and `SNIPUZZ` [22], usually ignore the impact of device states for bug finding. How to obtain the complete state space of IoT devices in the application layer is not studied in prior work.

Challenge 3 (C3): Non-crash bugs. One well-known difficulty in IoT fuzzing is that, in general, one cannot modify the firmware and then install the altered firmware onto the device, as most IoT devices enforce code-signature checking [31]. Thus, it is infeasible to collect the program execution informa-

tion inside a device, such as branch coverage, path conditions, and function return values. While existing fuzzers use side channels, e.g., network connection, to infer whether a test input has triggered a crash bug, how to discover non-crash bugs, such as logic errors [9], is a known challenge.

Challenge 4 (C4): Command coverage. Ideally, a fuzzer should test all the commands of a device. However, manually comprehending the user manual of a device to derive the list of commands tends to be imprecise and unscalable. Prior work, SNIPUZZ [22], collects testing scripts disclosed by IoT vendors, but they are incomplete and very few vendors provide them (detailed in Section 4.3).

Our work has overcome all the challenges and built the first Matter fuzzer, named mGPTFuzz, in the open literature. To address **C1** and **C2**, we employ a large language model (LLM) to transform human-readable content in the specification to machine-readable information. Furthermore, the machine-readable information, including the device states, commands and their relations, is represented in finite-state machines (FSMs), where each node represents a device state and each edge a transition due to command execution. By iterating over the FSMs, our fuzzer performs systematic stateful analysis, effectively identifying stateful bugs.

To handle **C3**, we propose to leverage command semantics. For instance, if the execution of a command is expected to modify specific device attributes as per the specification, our fuzzer queries the corresponding attributes. On the other hand, if a command is meant to be rejected, an error message is expected. Precise extraction of semantic information concerning commands ensures the efficacy of this method.

Finally, to tackle **C4**, we are inspired by prior work, HubFuzzer [35], which makes use of an IoT hub to test ZigBee/Z-Wave devices. Our observation is that there is a special role in Matter, *controller*, which can add (formally, *commission*), manage and control Matter devices. Notably, while the certificate of an ordinary Matter device undergoes stringent verification by the controller, a Matter device does *not* verify the controller equally. Consequently, even an uncertified vendor can create a controller [48]. We thus propose to build our fuzzer within a controller. According to the specification, when a controller adds a device, the device declares all supported commands. Without relying on the device's manual or testing scripts, our fuzzer can extract the supported commands of a device under test from pairing messages.

We implement mGPTFuzz[1] and conduct an extensive evaluation, which involves 23 various Matter devices. It has revealed 147 new bugs (61 zero-day vulnerabilities), including 5 crash bugs and 142 non-crash bugs. Three CVEs have been assigned: CVE-2023-42189, CVE-2023-45955, CVE-2023-45956. In comparison, a state-of-the-art blackbox fuzzer, SNIPUZZ [22], finds zero bugs from these devices.

mGPTFuzz can be used by IoT vendors that cannot afford a security testing team, third-party security analysts, as well as companies and organizations seeking to assess the security of their Matter devices before placing trust in them.

This work makes the following contributions.

- We present the first Matter fuzzer in the literature. It can help IoT vendors, security researchers and numerous companies and organizations identify bugs and vulnerabilities from Matter devices.
- The detailed Matter specification enables a unique research opportunity but also imposes a challenge. We harness LLM to extract information from the sheer volume of specification, showcasing the effectiveness of LLM-assisted fuzzing on a large scale.
- mGPTFuzz is a blackbox IoT fuzzer that is able to uncover non-crash bugs in the application layer. In addition, semantic information is leveraged to conduct systematic stateful fuzzing, unveiling stateful bugs.
- We employ a controller-based fuzzing architecture. This design eliminates the need for reverse engineering any companion apps or collecting API-testing scripts, and can derive a complete list of the supported commands of a device under test.
- We implement mGPTFuzz and test 23 various Matter devices. The evaluation finds 147 new bugs (including 61 zero-day vulnerabilities), comprising 5 crash bugs and 142 non-crash bugs, with three CVEs assigned.

2 Related Work

2.1 Large Language Model Assisted Fuzzing

Large Language Models (LLMs) have demonstrated remarkable performance across a wide range of tasks. Very recently, researchers have started to leverage LLMs for fuzzing. For example, TitanFuzz [17] and FuzzGPT [18] use LLMs to generate code for testing deep learning libraries. Fuzz4All [68] further demonstrates that code of different languages can be generated using LLMs for testing a variety of compilers and libraries. LLM4Fuzz [55] employs LLMs to guide fuzzing of smart contracts. ChatAFL [36] utilizes LLMs to process Request for Comments (RFCs) and generate test inputs. ChatFuzz [26] leverages LLMs to mutate seeds in order to generate format-conforming inputs.

All these works study LLM-assisted greybox fuzzing, assuming that the code under test can be instrumented, while our work is the first that studies LLM-assisted blackbox fuzzing. Without instrumenting the code (i.e., IoT firmware), how to infer the program execution status to guide fuzzing is a unique challenge resolved in this work. In addition, while these works are mostly focused on finding crash bugs, our work also finds non-crash bugs in a principled way. This stems from our approach that extracts meticulous information per command and

[1] https://iot-fuzz.github.io

utilizes it to verify the correctness of each command execution. Finally, this work demonstrates the efficacy of LLM-assisted fuzzing in an important domain—Internet of Things.

2.2 Fuzzing of IoT Firmware

Static analysis of IoT firmware for finding bugs [3, 21, 33, 38, 53, 60, 61, 78, 80] tends to report many false positives. Emulating firmware can support whitebox or greybox fuzzing, which dynamically finds IoT vulnerabilities [11, 16, 20, 34, 37, 52, 69, 77]. Despite extensive efforts [32, 74, 75, 79], precisely emulating IoT firmware remains a challenging task [51, 69, 73]. Worse, many vendors do not provide firmware images publicly and make it difficult to extract unencrypted firmware from devices [12, 22, 51].

Recently, blackbox fuzzing of IoT firmware demonstrates noteworthy results and becomes an attractive approach [12, 22, 35, 51]. For example, `IoTFuzzer` [12] and `DIANE` [51] reverse engineer the companion app of a device under test. For *each* IoT device, they analyze its companion app to locate the code corresponding to IoT functionalities and modify it as strong obfuscation techniques are widely available [8, 71, 72]. However, this requires enormous per-device efforts for analyzing and modifying obfuscated apps [8, 19], and the analysis may be incomplete in identifying IoT functionalities [22, 35]. Furthermore, a Matter device, in general, cannot be controlled by the vendor's companion app.

`SNIPUZZ` [22] first collects API-testing scripts disclosed by IoT vendors and then captures network messages when running these scripts. The network messages are then mutated to conduct fuzzing. It saves the effort for reverse engineering companion apps. However, it has multiple other limitations. First, very few IoT vendors disclose their API-testing scripts (Section 4.3). Second, even when a vendor discloses API-testing scripts, they usually do not cover all the supported commands of a device. Third, `SNIPUZZ` fails if network messages are encrypted, while most IoT devices use encryption for secure communication [12].

`HubFuzzer` [35] uses a hub to issue test messages to IoT devices, and is limited to fuzzing ZigBee and Z-Wave devices. It inspires our controller-based fuzzing architecture.

`BrakTooth` [25] mainly tests the data link and network layers of Bluetooth Classic, while Matter is a standard focused on the application layer. Unlike `BrakTooth`, which infers state machines about Bluetooth Classic from exchanged packets, we extract state machines from the specification (Section 4.1).

In sum, `mGPTFuzz` is the first fuzzer for Matter, an important IoT standard. It does not need to emulate the firmware, reverse engineer companion apps, or collect API-testing scripts.

2.3 Specification-Guided Fuzzing

Specification-guided fuzzing [29, 46, 50] uses manually prepared machine-readable information (or a generator) about a protocol to generate test messages. However, this process is tedious, incomplete and error-prone for non-trivial protocols [47]. We leverage LLMs to obtain the specification information, eliminating the need for excessive manual efforts.

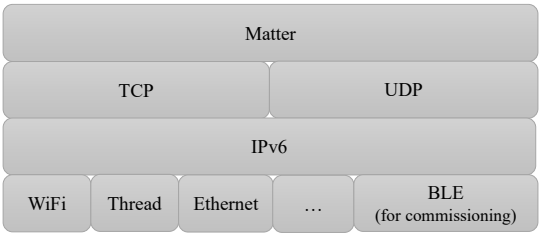


Figure 1: Protocol stack with Matter.

3 Background

3.1 Matter

Figure 1 shows the protocol stack with Matter [65]. Matter is built on top of the IP layer and uses it as a common for communicating with IP-based networks, such as WiFi, Ethernet, and Thread.

A special device, called a *controller*, such as a smart speaker, can add (formally, *commission*) and manage Matter devices. To add a device, the controller (or a trusted *commissioner*, such as the companion app of the controller) first verifies the device's attestation certificate to make sure it is signed by a vetted vendor. Then, an *operational certificate* is signed by the controller and sent to the device. Here, the controller serves as a root certification authority (CA). Matter devices that share the same root CA form a Matter *fabric*, and a controller can control all the Matter devices in its fabric. Matter devices in the same fabric conduct secure P2P communication with each other through their operational certificates.

A Matter device in a fabric is a *node*, which has a number of *endpoints*, each representing a specific functional unit. For example, a smart bulb usually has an endpoint about lighting. An endpoint has one or more *clusters*, where a cluster is a group of related functionalities.

A cluster contains *attributes* and *commands*. (1) An attribute represents a device state. For example, the `On/Off` cluster has an `OnOff` attribute that indicates the on/off state of the device. An attribute has a data type, such as boolean, integer, and string, and may be read-only or read-write. (2) A command represents an action that may be performed by a device.

3.2 Large Language Models

Emerging Large Language Models (LLMs) have demonstrated impressive performance on a variety of tasks. These LLMs are pre-trained on billions of available text. Due to the extensive training data, LLMs can be directly employed for specific downstream tasks without undergoing fine-tuning on

"This command (`KeySetRemove`) SHALL fail with an `INVALID_COMMAND` status code back to the initiator if the `GroupKeySetID` being removed is 0, which is the Key Set associated with the Identity Protection Key (IPK)."

Figure 2: A specification passage ignored by developers.

specialized datasets [10]. This is achieved through prompt engineering [64], wherein a task description, along with a few task demonstrations, is presented to the LLM. Researchers have shown that utilizing the paradigm of directly leveraging Language Models through prompts can already attain state-of-the-art performance on downstream tasks [27, 57, 63, 70].

Since the Matter specification is written in a natural language, LLMs pre-trained on extensive corpora should be capable of processing the specification. We thus leverage an LLM to extract information from the Matter specification. While the approach of `mGPTFuzz` is general for different LLMs, we utilize the well-known LLM: GPT-4 [44].

4 Overview

4.1 Motivation of Using LLM

To extract information from the Matter specification, a straightforward approach is to manually read it carefully and draw all the FSMs. However, we use an LLM for the process because of the following reasons.

- **To save much tedious manual effort.** The specification spans 1,258 pages. While the part describing clusters has 589 pages, the remaining is not useless. For example, it covers data types and their ranges, as well as definitions of the symbols used in the cluster description. Such knowledge is first extracted and then used for subsequent interactions with LLM for cluster-specific queries.
- **To avoid overlooking important information.** Manually extracting information from the specification will likely neglect important information. For example, Matter SDK developers omitted the information, illustrated in Figure 2, that the `GroupKeySetID = 0` should not be deleted (CVE-2023-42189). The many non-crash bugs we found also show that developers omit information.
- **To cope with the quick evolution of the standard.** Since Version 1.0 was released in October 2022, three new versions have been published (V1.1 in May 2023, V1.2 in October 2023, and V1.3 in May 2024). The automation of knowledge-base extraction can accelerate the update of the fuzzer.

Another alternative approach is to extract FSMs from code [23, 25, 56]. However, it is incomplete. For example, the Matter SDK provides a framework for developing an IoT device, but it does not stipulate all the details. Second, it may contain bugs, e.g., due to handling parameter value ranges incorrectly. In sum, the approach does not provide a "standard" that a Matter device should adhere to. We thus choose to extract information from the specification.

Finally, LLM-assisted fuzzing is promising but not mature yet. We encounter multiple questions which, to our knowledge, are not documented in prior LLM-assisted fuzzing work. For example, how to deal with a large text that exceeds the token limit; how to generate a complex FSM; how to provide non-trivial context for subsequent queries. Such experiences can help other research in this direction.

4.2 Threat Model

System Under Attack and Assumptions. We consider a smart environment (such as a warehouse, office, or home) contains various Matter devices. They interact through user-defined automation rules. We assume the devices have vulnerabilities like those discussed in our work.

Attacker Model. We consider two types of attackers. (1) A user in the smart environment, such as a warehouse, can make use of Matter devices under his/her control to launch attacks. (2) An attacker has no authorized control over any Matter devices in the target environment, but can make use of vulnerabilities in commissioning [39, 54] to add a malicious device into the target Matter fabric. The attacker then makes use of the device(s) under his/her control to send exploit commands to vulnerable devices. The goal is to cause a device into a state desired by the attacker, which facilitates subsequent exploitation, such as unauthorized access and burglaries. For example, an attacker can crash a light to cause poor illuminance for surveillance recordings. As another example, given the automation rule "*when temp > 80°F, open the window*", by crashing the thermostat, a non-vulnerable smart window may open. Similar attacks are stressed in the literature [13–15].

Furthermore, without involving attacks, bugs of critical devices, such as heaters, locks, valves, and smoke detectors, can pose hazards to smart homes and their residents [24].

4.3 Limitations of a SOTA IoT Fuzzer

`SNIPUZZ` [22] represents a state of the art approach to black-box IoT fuzzing. Below, we discuss why `SNIPUZZ` does not work well for analyzing Matter devices.

Manually Collecting Testing Programs. `SNIPUZZ` needs to manually collect API-testing scripts for each device under test, while only a few vendors disclose them. For example, among the 23 devices involved in our evaluation, only 6 have their API-testing scripts publicly available.

Low Command Coverage. Even for devices that have API-testing scripts available, these scripts typically only cover a small portion of commands supported by the devices. Figure 3 shows the comparison between the number of commands covered by `SNIPUZZ` and that by our approach.

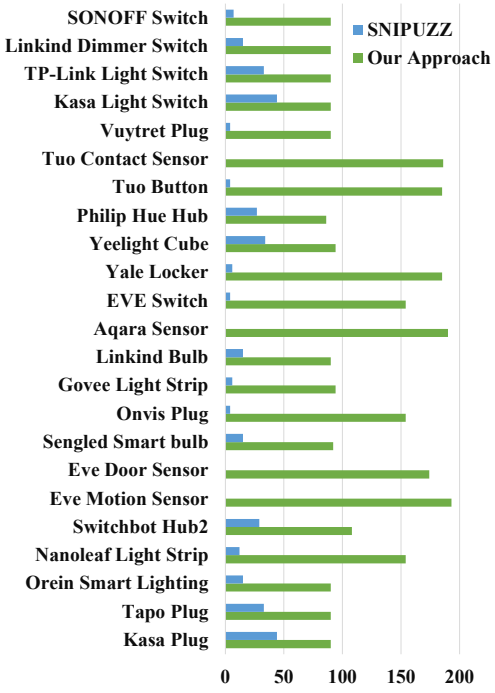


Figure 3: Commands covered by SNIPUZZ vs. mGPTFuzz. Only 6 out of 23 Matter devices (*Kasa Plug*, *Tapo Plug*, *Nanoleaf Light Strip*, *Switchbot Hub2*, *Govee Light Strip*, and *SONOFF Switch*) disclosed their API-testing scripts. For the remaining devices, we enhance SNIPUZZ by considering a device as an abstract one and counting the commands of an abstract device as being covered by SNIPUZZ. For example, a smart switch is abstracted into a *binary switch*, supporting the on() and off() commands.

Neglecting the Rich Information in Specification. SNIPUZZ does not make use of the rich information in specifications for fuzzing, and is unable to detect stateful or non-crash bugs.

Cannot Handle Encrypted Messages. SNIPUZZ mutates test inputs by modifying the collected network messages. The approach fails for encrypted communication used by Matter.

4.4 Goals and Ideas

We aim to build a Matter fuzzer with the following features: (1) No need to collect API-testing scripts or reverse engineer companion apps. (2) High command coverage. (3) Making use of the Specification information to guide fuzzing. (4) Working with encrypted communication protocols that support Matter, such as WiFi, Ethernet and Thread. Below, we present the insights and ideas for constructing these features.

First, as a Matter device can be configured to control another, we initially attempted to build our fuzzer into a custom Matter device. However, the custom device cannot obtain a legitimate attestation certificate signed by a vetted vendor. Our observation is that the certificate of a controller is not checked, and thus a custom controller can be built, integrating our fuzzer. This way, we can use a controller to test a device, without relying on API-testing scripts or companion apps.

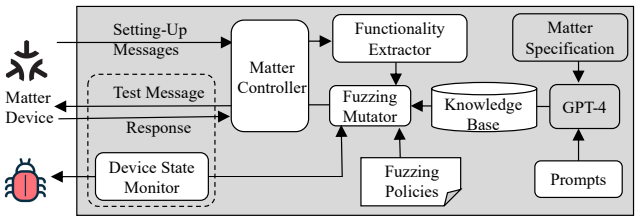


Figure 4: Architecture of mGPTFuzz.

Second, according to the Matter specification, when a device is added by the controller, it announces the device types along with the supported commands and attributes in the setting-up messages. This way, we can obtain a complete list of the supported commands and attributes from the setting-up messages. Hence, high command coverage can be attained.

Third, the Matter specification contains extensive details, regarding the command parameter types, value ranges, the expected response of a command, and the state change due to a command. Given the relatively new Matter implementation and the meticulous specification, it is highly improbable that developers have thoroughly digested the specification and adhered to all the details when writing the code. Thus, it is a promising and valuable idea to check the Matter implementation against the specification. Given the lengthy specification, we leverage a pre-trained large language model to convert the human-readable content to machine-readable information, which guides the fuzzing.

Fourth, we do not mutate network messages for fuzzing, but modify the code of a controller to generate test messages in plaintext, which is then encrypted and sent to the device. Furthermore, we configure a Thread border router in the computer that runs our custom controller. This way, the controller can test Thread devices, as well as WiFi and Ethernet devices.

4.5 System Architecture

Figure 4 shows the architecture of mGPTFuzz. It contains the following main components. (1) A custom **Matter Controller** commissions Matter devices, sends test messages to them, and receives responses. (2) When a Matter device is commissioned, it generates a sequence of setting-up messages. From these messages, the **Functionality Extractor** component learns the functionalities of the device, such as the supported commands and attributes (Section 5.1). (3) An LLM is leveraged, through prompt engineering, to convert the Matter specification to a **Knowledge Base** (Section 5.2). (4) According to our rich **Fuzzing Policies** (Section 5.3), the **Fuzzing Mutator** generates test messages (Section 5.4). (5) The **Device State Monitor** monitors the IoT device to capture bugs and vulnerabilities, and the results are used to further guide the fuzzing (Section 5.5).

5 Design of mGPTFuzz

5.1 Learning Functionality of Matter Devices

There are two parts in the Matter specification: *Matter Core Specification* [2] and *Matter Application Cluster Specification* [1]. The former provides information about the foundational clusters (such as the group key management and network diagnose cluster) for establishing and maintaining communications. The latter provides information about the application clusters, detailing how devices interact via specific application data and commands.

A *cluster* represents a group of related functionalities and has a unique 2-byte cluster identifier (CID). For instance, in the Matter Core Specification, the *Access Control* cluster (with the CID = 0x001F) sets the rules for managing the access control list of a device. As another example, in the Matter Application Cluster Specification, the *Level Control* cluster (with the CID = 0x0008) allows for the regulation of a device's physical quantity level, such as the brightness of a bulb or the extension length of a blind. A list of all available clusters can be found in the two Matter specification.

Extracting Supported Clusters of Devices. A sequence of setting-up messages are generated when a Matter device connects a controller, which contains rich information about the device, including the device id, manufacture code, and supported clusters. Based on the reported information and according to the two Matter specification, we can learn the functionalities supported by the device, and determine (1) which commands can control this device, and (2) which attributes are supported in the device. As an example, from the setting-up messages of the *Kasa Plug* device, we learn that the device contains two endpoints, where endpoint 0 includes 10 clusters, and endpoint 1 includes 5 clusters.

5.2 Learning Knowledge Base via LLM

The Matter specification provides a comprehensive description about commands and attributes, including the data type of each argument for every command, the value range of each argument, as well as how these commands and attributes mutually influence each another. To support fuzz testing, we need to extract critical information from the specification and convert the large amount of human-readable content into machine-readable information.

We employ an LLM to convert human-readable content from the specification into machine-readable information. We first extract information related to commands and attributes from the specification using the LLM (Section 5.2.1), and then query the LLM to represent the information as FSMs (Section 5.2.2). In an FSM, each node represents a device state and each edge represents a transition triggered by a command. The entire process of generating the knowledge base takes roughly 15 minutes for all clusters. By iterating over the FSMs, our fuzzer performs systematical fuzzing to detect bugs in devices.

5.2.1 Information Extraction

We design prompt engineering to extract information from the Matter specification. However, it is known that LLMs can be creative and may make up information in their responses. Worse, given the same prompts, LLMs may produce different outputs across interactions. Thus, a challenge arises: *How to extract accurate and stable information via LLMs?* To overcome this challenge, we employ three methods.

First, the temperature in an LLM is a parameter that controls the randomness of the LLM's output [43]. A higher temperature results in more creative and imaginative text, while a lower one results in more factual and stable text. We aim to obtain factual information extracted from the Matter specification; thus, `temperature=0` is employed. This setting ensures that the LLM strictly adheres to the factual nature of the source material for extracting knowledge, providing *stable* and consistent information across different queries.

Second, we employ *In-context few-shot learning* [57] to ensure the information extracted by LLMs is *accurate* and follow the specified output format. In-context few-shot learning is an effective strategy for improving the model accuracy by augmenting the context with a small number of examples that illustrate desired inputs and outputs. This approach enriches the context for LLMs, enabling them to better understand the syntax of the prompt, recognize output patterns, and accurately extract information. By employing this technique, we guide the LLM with examples to accurately extract useful information in the desired format.

Third, we employ *self-consistency checks* [62] to refine and validate the generated responses, ensuring reliability of the results. Even with the methods above employed, the model may still output answers that contain some stochastic information, although such instances are rare. We engage multiple conversations with the LLM and consider the majority of consistent answers as the final results.

Due to the token limit of GPT-4, we cannot feed the whole specification to it. We notice that each cluster corresponds to one chapter in the specification. We thus segment the cluster description part of the specification into multiple pieces, each for one cluster. However, prior to its extension of the token limit in November 2023, one long cluster, `DoorLock`, spanning 67 pages, exceeds the token limit. Thus, we further segment the content of the `DoorLock` cluster and query the information from each segment one by one. Afterwards, we concatenate the responses.[2]

Prompts. There are two types of datatypes: (1) base datatypes, such as `uint`, `int`, and `bool`, and (2) derived datatypes, which are derived from the base datatypes. The base datatypes are

[2]Since November 2023, the token limit has increased to 128,000 tokens, which corresponds to around 96,000 words or 192 single-spaced pages.

Prompt for Base Datatype

Please provide responses to the question, and it is imperative that your responses be strictly based on the text provided below.

[Base Datatype Text].

The **query** is as follows:
List the data types in the text and their value ranges in JSON format.

Figure 5: Prompt for querying base datatypes.

shared across all clusters; thus, we only need to make one query for all the 26 base datatypes, rather than a query for each cluster. Figure 5 shows the prompt that queries the base datatypes and their corresponding value ranges.

Figure 6 shows the prompt template for querying information per cluster, consisting of the **Cluster Text**, **Queries**, and an **Example Output**. There are totally 67 clusters. Each cluster is queried separately, and the prompt generation is *automated* by assembling the prompt template using scripts. Given a cluster, the Cluster Text is converted from its chapter in the specification. Specifically, we use the *Optical Character Recognition* tool [49] to convert the PDF-formatted specification into text.

Derived datatypes only appear in certain clusters. Thus, for each cluster, we query the derived datatype and the corresponding value range (Query 1 in Figure 6). Given a cluster, we also need to know its commands and attributes, the data types and value ranges for each command's argument and each attribute, as well as their IDs (Queries 2-5 in Figure 6).

Responses. Figure 7 shows an example response for the `OnOff` cluster, which includes five pieces of information (corresponding to the five queries in the prompt). Specifically, there is one derived datatype, six commands, and five attributes for this cluster. For each command, its arguments, the corresponding datatype, and the command ID are extracted. For each attribute, its datatypes and ID are also extracted. In particular, `StartUpOnOffEnum` is a derived datatype in the `OnOff` cluster, serving as the datatype for an argument of the `StartUpOnOff` command.

5.2.2 FSM Generation

An FSM is a tuple $(Q, \Sigma, \Delta, \delta)$, where Q denotes a finite set of states, Σ represents the initial state, Δ represents the destination state, and δ stands for the commands that can map Σ to Δ. Given a cluster, we make a query for each command, and generate an FSM specific to that command. After that, all FSMs are combined to form a comprehensive FSM representing the entire cluster.

Figure 8 shows the prompt template designed to query the useful information of a command (specified using the command ID) to generate FSMs. In this process, we employ *in-context few-shot learning* as illustrated in the **Shot 1** section of Figure 8. Moreover, we provide the datatype knowledge about the base datatypes (extracted by the prompt for base datatypes in Figure 5) and the derived datatypes related to the cluster (extracted by Query 1 in Figure 6). This enables the LLM to accurately understand the data range in the second step of the Chain-of-Thought process. In addition, we leverage *Chain-of-Thought prompting* [63] to ensure the precision of information extracted by LLMs (see the **Chain-of-Thought** part in Figure 8). Chain-of-Thought prompting involves structuring prompts to guide LLMs through a series of logical steps,

Prompt Template for Extracting Cluster Information

Please provide responses to the questions in the specified order and format.

[Cluster Text].

Queries are as follows:
1. Derived Datatypes and Corresponding Value Ranges
2. List of Commands
3. List of Attributes
4. List of Command IDs
5. List of Attribute IDs

Here is an Example Output in JSON format.

[Example Output].

Figure 6: Prompt template for querying information of a cluster. It is simplified for the sake of presentation. Appendix A provides a verbatim example of assembled prompts.

Model Response

"OnOff": {
1. "Derived DataTypes and Value Ranges": {
"StartUpOnOffEnum": {"values":[0,1,2]}
},
2. "Commands": {
"Off":[], "On":[], "Toggle":[],
"OffWithEffect":["EffectIdentifier":"uint8", "EffectVariant":"uint8"],
"OnWithRecallGlobalScene":[],
"OnWithTimedOff":["OnOffControl":"map8","OnTime":"uint16",
"OffWaitTime":"uint16"]
},
3. "Attributes": {
"OnOff":"bool", "GlobalSceneControl":"bool",
"OnTime":"uint16", "OffWaitTime":"uint16",
"StartUpOnOff":"StartUpOnOffEnum"
},
4. "Command IDs": ["0x00", "0x01", "0x02", "0x40", "0x41", "0x42"]
5. "Attribute IDs": ["0x0000", "0x4000", "0x4001", "0x4002", "0x4003"],
}

Figure 7: Model response for the `OnOff` cluster.

Prompt Template for Generating an FSM

You will be assigned the role of a **Software Testing Assistant** and will receive a portion of the protocol specification text related to the cluster **[Name]**. Your primary task is to prepare the **Finite State Machine (FSM)** test cases for software black-box testing. It is imperative that your responses be strictly based on the text provided. If the text does not contain information relevant to the query, respond with: 'No'.

Chain-of-Thought:

1. Extract the initial and destination states as mentioned in the "Effect" section in the provided cluster text.
2. Extract the value range of each command's argument to transfer from the initial state to the destination state. Please refer to the content marked as "Datatype Knowledge" regarding datatypes and their value ranges.
3. Extract the invalid value of each command's argument and error messages, if specified in the cluster text.
4. Extract unaccepted values of each state.

[Cluster Text]

[Datatype Knowledge]

According to the text, extract the FSM information for the command with the ID = **[CommandID]**.

Shot 1:
[Place an example of generating an FSM here.]

Desired Output Format:
[Here is the desired output format for the FSM.]

Figure 8: Prompt template for generating an FSM.

similar to a human thought process, to arrive at the desired output. This technique proves especially effective in complex situations when the straightforward question-to-answer format may not produce comprehensive results. Moreover, we leverage *self-consistency checks* [62] to enhance the reliability of the response.

We observed that even when employing methods like setting the temperature=0 to prevent randomness and creativity, there are still rare situations that 1) the LLM may produce a response whose format does not align with **Shot 1**, and 2) the LLM fabricates information when the provided **Cluster Text** does not contain information relevant to a query. We attribute the first issue to the complexity of the output. To ensure the LLM response follows the desired format, we add the **Desired Output Format** section at the end of the prompt. To address the second issue, we emphasize not making up information, as shown in the last sentence of the first paragraph in Figure 8.

Through these methods of designing the prompt, we are able to effectively obtain FSM information. After generating FSMs for each command within a cluster, we merge all FSMs into a comprehensive FSM representing this cluster. There are 52 FSMs generated, with a total of 521 states and 522 transitions. Note that 15 clusters do not have any command and the involved attributes are read-only (e.g., the `BooleanState` cluster has only one read-only attribute and zero commands). For the 52 FSMs, the number of states is in the range of $[1, 46]$, and the number of edges is in the range of $[1, 50]$. The most complex FSM is for the `DoorLock` cluster, which contains 46 states and 50 edges.

Example. Figure 9 shows part of the generated FSM, achieved by merging multiple FSMs corresponding to all commands within the `LevelControl` cluster. This example contains 10 states and 13 edges. Each edge encompasses detailed information on the state transition process, including the command name and the possible value and data type for each argument.

Verifying the Quality of FSMs. Besides self-consistency checks with multiple queries, we also manually validate the quality of the FSMs. We first *randomly* sample 100 out of the 522 transitions. Three authors spent a total of 9 hours manually and independently checking the accuracy of the information described in the 100 transitions. We confirm that all the information is accurate. We then pick a cluster, `LevelControl`, and check whether all the transitions described in the specification are covered by the FSM, and result is positive. The checking demonstrates that the LLM is able to accurately extract the FSM information.

5.3 Fuzzing Policies

Our fuzzer iterates over the FSMs to generate test inputs. The following policies are used.

Policy 1: For each FSM edge, we (a) change the argument values to the values specified by edge; (b) if the valid argument value is a range, provide extreme values (such as min and max of the valid range); (c) provide random valid values excluding the extreme values. Moreover, for each command, we (d) change the length of a string-type argument trying to trigger buffer overflows; (e) provide empty values to strings to trigger uninitialized read or null pointer deference; and (f) provide `NULL` or only one element to arrays, sets, or bags to cause null pointer deference or out-of-bounds access.

Policy 2: *Changing Argument Types*. Given an argument supposedly with the data type t, we change its type to a randomly selected one t'. For example, for an argument with the *String* type, we change its type to the *integer* type by replacing a string value with an integer value, to check whether the device can handle the special "string".

Policy 3: *Changing the Number of Arguments*. For a command requiring n arguments, we provide $n+1$, $n-1$, or 0 arguments.

Policy 4: *Trying Unsupported Clusters and Commands*. Be-

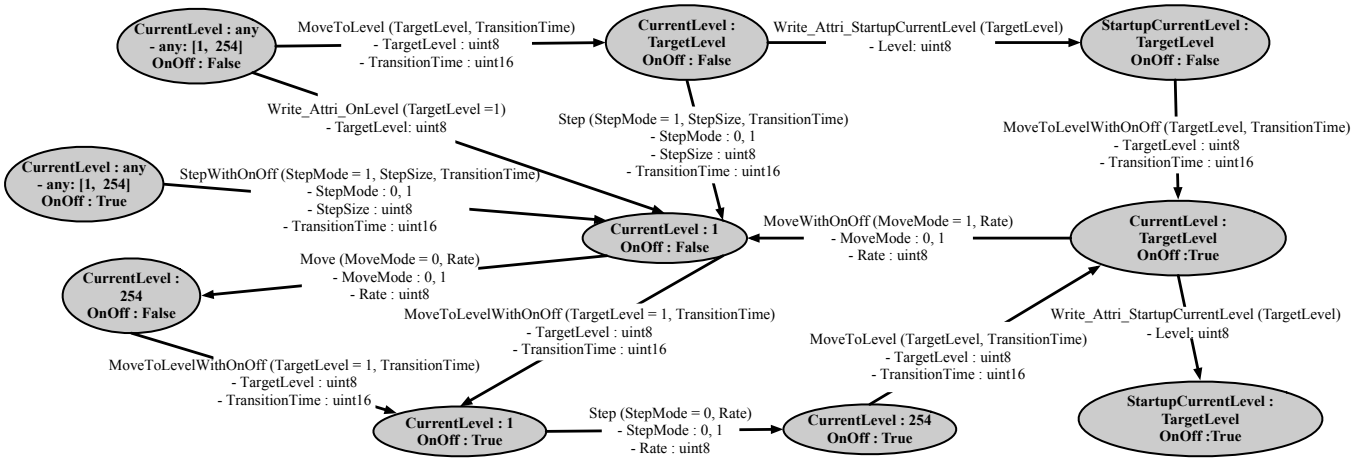


Figure 9: The generated FSM for the `LevelControl` cluster.

sides the supported clusters, we also randomly select a few *unsupported clusters*. For each command in a selected unsupported cluster, we generate test messages following the command definition. Through this, we check whether *unexpected commands* can cause the device to crash.

5.4 Constructing Test Messages

To build a test message, given a command, a straightforward way is to invoke the API in the controller that invokes the command. However, such APIs contain various input sanitization. Consequently, invalid test messages cannot be constructed.

To resolve this issue, our solution is to locate the procedure that packs messages, which we call the *(message) packing procedure*. It is invoked by each API to generate messages to be sent to IoT devices. We then remove the input sanitization in the packing procedures. Note unlike prior work [12,51] that removes sanitization in the companion app of *each* device, our sanitization elimination is a one-time effort.

There are two types of commands.

- ***Ordinary commands***. Each cluster contains zero or more ordinary commands. The packing procedure, *InteractionModelCommands::SendCommand*, from the controller `chip-tool` [48], is used for generating such commands.
- ***Write-Attribute*** can modify the specified cluster attribute. The packing procedure, *InteractionModelCommands::WriteAttribute*, from `chip-tool` [48], is used for generating such commands.

5.5 Device State Monitor

From the perspective of `mGPTFuzz`, it is trivial to detect a device crash, as a crash causes a disconnection (and a timeout exception for the next test message). To detect a non-crash bug, for each test message, if the response message and the destination state, in terms of the involved attribute values, do not adhere to the transition described in the FSMs, a non-crash bug is captured. Specifically, given a valid test message (i.e., valid command/attribute ID and argument values), the controller expects a response message `SUCCESS` from the device under test and the destination state is also checked by querying the attribute describing the state. Given an invalid one, it expects an error message, such as `INVALID_COMMAND`.

Given a bug, if the symptom can only be reproduced when the device is at certain states, it is a *stateful* bug; otherwise, a *non-stateful* one.

6 Evaluation

This section presents the implementation and the evaluation results. Section 6.1 describes the implementation details. Section 6.2 presents the experimental setup. Section 6.3 summarizes the bug-finding results. Section 6.4 presents the results of detecting crash bugs and Section 6.5 non-crash bugs. We compare `mGPTFuzz` with a state-of-the-art work in Section 6.6. Finally, Section 6.7 discusses the efficiency.

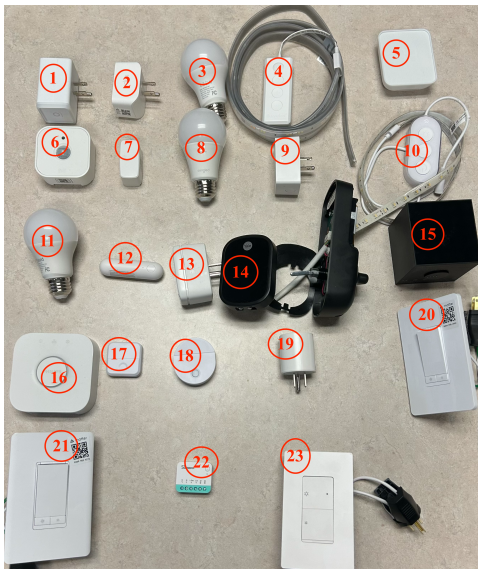
6.1 Implementation

We implement a prototype of `mGPTFuzz`. We utilize an open-source tool, `chip-tool` [48], provided by the Matter Consortium, to build our custom controller. We remove the input sanitization in its message packing procedures, such that our test inputs are not rejected due to sanitization [51].

The controller is able to commission Matter-over-WiFi and Matter-over-Ethernet devices using the `chip-tool code-wifi pairing` script. To support the Thread radio communication capabilities, we insert an *nRF52840 Micro Dev Kit USB Dongle* (priced at $21.99 on Amazon [5]) to our desktop. Moreover, we install the `ot-br-posix` library, which turns our desktop into an OpenThread Border Router (OTBR) [45]. Subsequently, our custom controller

ID	Device Type	Vendor	Model	Firmware Version	Protocol
1	Plug	Kasa	KP125MP4	1.0	Matter over WiFi
2	Plug	Tapo	P125M	1.0.7	Matter over WiFi
3	Bulb	Orein	OS0100811267	3.01.26	Matter over WiFi
4	Lightstrip	Nanoleaf	NF080K03-2LS	v3.5.10	Matter over Thread
5	Hub	SwitchBot	W3202100	v1.0-0.8	Matter over WiFi
6	Motion Sensor	Eve	20EBY9901	2.11	Matter over Thread
7	Door Sensor	Eve Door	20EBN9901	2.1.1	Matter over Thread
8	Bulb	Sengled	W41-N15A	v22	Matter over WiFi
9	Plug	Onvis	S4	1.1	Matter over Thread
10	LED Strip	Govee	H61E1	v3.00.42	Matter over WiFi
11	Smart Bulb	Linkind	LS0101811266	3.01.26	Matter over WiFi
12	Water Sensor	Aqara	DW-S02E	1.0	Matter over Thread
13	Switch	Eve	20EBU4101	3.2.1	Matter over Thread
14	Locker	Yale	W41-N15A	1.0	Matter over Thread
15	Cube Smart Lamp	Yeelight	YLFWD-0009	v1.12.69	Matter over WiFi
16	Hub	Philip Hue	453761	v1.59.195909703	Matter over Ethernet
17	Button	Tuo	TSB3194	1.0	Matter over Thread
18	Contact sensor	Tuo	TCS-07505	1.0	Matter over Thread
19	Wifi plug	Vuytret	YX-WS02B	v1.0.5	Matter over WiFi
20	Light Switch	TP-Link	KS225	1.0	Matter over WiFi
21	Light Switch	Tapo	Tapo S505	1.0	Matter over WiFi
22	Dimmer Switch	Linkind	B0C74J9FCN	1.0	Matter over WiFi
23	Smart Switch	SONOFF	MINIR4M	1.0	Matter over WiFi

(a) Device details



(b) Photo of devices

Figure 10: IoT devices used in our experiments.

is able to commission Matter-over-Thread devices using the `chip-tool code-thread pairing` script.

For LLM, we use GPT-4-Turbo [44]. The temperature in an LLM is a parameter that controls the randomness of the LLM's output [42]. A higher temperature results in more creative and imaginative text, while a lower one results in more accurate and factual text. We aim to obtain precise and factual information extracted from the Matter protocol specification; thus, `temperature=0` is employed.

To fuzz-test a device, the only manual effort is to pair it with `mGPTFuzz`. Note that developing `mGPTFuzz`, including prompt engineering, is a one-time effort.

6.2 Experimental Setup

Matter Devices Under Test. We acquire 23 popular consumer Matter IoT devices from both online and offline markets, covering various brands, such as Philip Hue, Yeelight, and Yale. The types of the Matter devices include smart switches, plugs, lighting, lockers, sensors, and hubs. These devices are either recommended by Amazon or the best-selling products in supermarkets. Their details are illustrated in Figure 10.

Testing Environment. Our `mGPTFuzz` runs on a Ubuntu 20.04 PC with 4.9 GHz Intel® Core(TM) i7 CPU and 32 GB RAM. We configure the Matter devices in a fully controlled network to avoid the interference of irrelevant traffic.

Baseline Method. Blackbox fuzzing of IoT firmware demonstrates noteworthy results. There are a variety IoT blackbox fuzzers, such as `IoTFuzzer` [12], `Diane` [51], `HubFuzer` [35], `FIoT` [78] and `SNIPUZZ` [22]. (1) We excluded IoTFuzzer and Diane for comparison because they send test inputs from companion apps, while a Matter device cannot be controlled through the device's companion app. (2) We excluded HubFuzzer, as it only tests ZigBee and ZWave devices. (3) We also excluded fuzzers that are not open source. We thus picked SNIPUZZ for comparison. Another reason we chose SNIPUZZ is because the evaluation of SNIPUZZ shows that it outperforms prior work, such as `NEMESYS` [30], `BooFuzz` [29] and `DooNA` [67].

6.3 Bug Discovery Results

We divide bugs into two categories: (1) *crash bugs*, which result in device crashes, and (2) *non-crash bugs*, which cause incorrect behaviors but do not crash devices. From the 23 Matter devices, we discover 147 bugs, including 5 crash bugs and 142 non-crash bugs. Among the 147 bugs, there are 10 stateful bugs, where 4 are stateful crash bugs and 6 stateful non-crash bugs. The other 137 bugs can be triggered regardless of the current device state. The detailed results are outlined in Table 1. Among the 147 bugs, 61 bugs lead to a denial of service, i.e., the devices crash (CVE-2023-45955 & CVE-2023-45956), or do not respond until they are re-paired with the controller (CVE-2023-42189). Given the DoS nature, we classify the 61 bugs as vulnerabilities.

6.4 Crash Bugs

The 5 identified crash bugs are distributed as follows:

- One crash bugs exist in the device *Nanoleaf Lighting NF080K03-2LS* (with the device ID = 4), and it has been assigned CVE-2023-45955.
- Four crash bugs exist in *Govee Lighting H61E1* (with the device ID = 10), which are *stateful bugs*, requiring

Table 1: Summary of bugs detected by mGPTFuzz. (1) *UT* stands for *Unexpected Transition*, meaning the device transits to an unexpected state. (2) *DoS* means *Denial of Service*. Note that all the bugs are missed by the baseline tool SNIPUZZ. The fuzzing time is mainly determined by the number of commands and attributes supported by the device.

Device ID	Crash Bugs # of Bugs	Crash Bugs Impact	Non-crash Bugs # of Bugs	Non-crash Bugs Impact	Fuzzing Time
1	0	-	8	UT, DoS	1.28 h
2	0	-	8	UT, DoS	1.24 h
3	0	-	10	UT, DoS	1.45 h
4	1	DoS	9	UT, DoS	3.00 h
5	0	-	3	UT, DoS	2.01 h
6	0	-	4	UT, DoS	1.68 h
7	0	-	4	UT, DoS	4.67 h
8	0	-	7	UT, DoS	4.00 h
9	0	-	9	UT, DoS	3.70 h
10	4	DoS	12	UT, DoS	4.20 h
11	0	-	10	UT, DoS	4.55 h
12	0	-	4	UT, DoS	1.67 h
13	0	-	9	UT, DoS	1.03 h
14	0	-	4	UT, DoS	3.09 h
15	0	-	9	UT, DoS	3.67 h
16	0	-	3	UT, DoS	1.37 h
17	0	-	2	DoS	4.50 h
18	0	-	2	DoS	2.93 h
19	0	-	6	UT, DoS	3.92 h
20	0	-	7	UT, DoS	3.40 h
21	0	-	6	UT, DoS	3.60 h
22	0	-	2	DoS	3.07 h
23	0	-	4	UT, DoS	3.70 h

the device to be set to a particular state to be triggered. These bugs have been assigned CVE-2023-45956.

Note that to save CVE resources, given multiple bugs of a device that are related to a group of similar commands or exploit messages, only one CVE is requested.

Below we discuss these discovered crash bugs. The details of these bugs are summarized in Table 2. A *hidden API* means that the command or attribute is not covered in the vendor's API-testing scripts or described in its website.

6.4.1 Crash Bug in *Nanoleaf Lighting Device*

This bug is related to a *hidden* write-attribute command, `Write_Attribute_Binding`, within the `Binding` cluster, which is used to establish a persistent relationship between an endpoint and local/remote endpoints. This command accepts one argument with the data type `List` in the following two formats: `List[node-id, endpoint-id, cluster-id]` or `List[group-id]`.

Triggering Bug. In the fourth column of Table 2, for the sake of presentation simplicity, only the message payload is displayed. The payload of a message should follow a specific JSON format. For example, given a payload, `{"0" : 1}`, the integer of 0 within the quotation marks indicates the index of the argument, and the value of 1 after the colon represents the value of the corresponding argument. If there is more than one argument, there will be more than one set of quotation marks (with each enclosed value indicating the index of each argument) as well as the corresponding value.

To trigger the bug, mGPTFuzz constructs a command message, where the list has only one element with value = 0 (following the *fuzzing policy 1* in Section 5.3), the generated payload is `["0" : {"0" : 0}]`.

Observation. The device is supposed to reject the aforementioned invalid input. However, it accepts the message. We observe that the light initially exhibited a flickering behavior, followed by a crash.

6.4.2 *Stateful* Crash Bugs in *Govee Lighting Device*

The four bugs are related to four *hidden* commands, `Move_up (uint8)`, `Move_down (uint8)`, `Move_up_OnOff (uint8)`, and `Move_down_OnOff (uint8)`, which can increase or decrease the brightness of the device with or without the *OnOff* effect at a certain *rate*. Each command accepts one argument with the data type `uint8`, which specifies the *rate* value.

Triggering Bugs. If the device is at the required initial state and then an invalid value of 0 is provided as the command argument (following the *fuzzing policy 1* in Section 5.3), the test message makes the device crash. Taking `Move_up_OnOff (unit8)` as an example (the last row in Table 2), the normal payload is `{"0" : 20}`, where the value of 20 (without quotes) denotes the rate value. If the brightness level (`CurrentLevel`) is the lowest (i.e., 1) and the rate in the command is an invalid value of 0, the generated test message causes the device to crash. Note 254 represents the highest brightness value in Matter, and 1 the lowest.

It is worth noting that a stateful bug can be triggered only if the device is first set to a certain initial state. Otherwise, an identical exploit input cannot trigger the bug. This exemplifies the importance of stateful fuzzing.

6.5 Non-Crash Bugs

From the 23 Matter devices, mGPTFuzz finds 142 non-crash bugs, 6 of which are *stateful* non-crash bugs. Detecting non-crash bugs presents a greater challenge compared to crash bugs, as network connection state, which can be employed as clues for crash bug detection, is not useful for detecting non-crash bugs.

We find **two types** of non-crash bugs: N1) bugs where the device should reject the corresponding exploit messages but accepts and processes them; N2) bugs where the device should accept the corresponding exploit messages but mistakenly rejects them. The distribution of the two types of non-crash bugs across the tested devices is shown in Table 3.

Below we present some cases of non-crashed bugs. The details of these cases are summarized in Table 4. Specifically, Section 6.5.1 and Section 6.5.2 discuss some cases of non-

Table 2: Details of discovered crash bugs.

Device ID	Hidden API?	Command	Normal Message ⟶ Exploit (only *payload* is shown)	Required Initial State	Observation	CVE
4	✓	`Write_Attribute_Binding(List[uint16])`	["0": {"0" : 20}] ⟶ ["0":{"0" : 0}] ***Policy 1:*** Provide an invalid value 0 to the element of the argument	Any state	Device Crashed	CVE-2023-45955
10	✓	`Move_down(uint8)`	{"0": 10} ⟶ {"0": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentLevel = 254	Device Crashed	CVE-2023-45956
10	✓	`Move_up(uint8)`	{"0": 20} ⟶ {"0": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentLevel = 1	Device Crashed	CVE-2023-45956
10	✓	`Move_down_OnOff(uint8)`	{"0": 20} ⟶ {"0": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentLevel = 254	Device Crashed	CVE-2023-45956
10	✓	`Move_up_OnOff(uint8)`	{"0": 20} ⟶ {"0": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentLevel = 1	Device Crashed	CVE-2023-45956

Table 3: Summary of non-crashed bugs. There are two types of non-crashed bugs. *Type N1* refers to bugs where the device should reject the test messages but instead accepts them. *Type N2* refers to bugs where the device should accept the test messages but mistakenly rejects them.

Device ID	Type *N1* # of Bugs	Type *N2* # of Bugs	Device ID	Type *N1* # of Bugs	Type *N2* # of Bugs
1	8	-	13	9	-
2	8	-	14	4	-
3	9	1	15	9	-
4	9	-	16	3	-
5	3	-	17	2	-
6	4	-	18	2	-
7	4	-	19	6	-
8	7	-	20	7	-
9	9	-	21	6	-
10	12	-	22	2	-
11	9	1	23	4	-
12	4	-			

crash bugs of Type N1 (Section 6.5.2 focuses on *stateful* non-crash bugs of Type N1), and Section 6.5.3 Type N2.

6.5.1 Non-Crash Bugs of Type N1 in *All* Matter Devices

Two non-crash bugs affect all the Matter devices, as shown in Table 2 (with the device ID labeled as *All*). These bugs have been assigned CVE-2023-42189. They are not stateful bugs, so can be triggered in any device states. Both bugs are related to the *hidden* command, `KeySetRemove (uint16)` (see Figure 2). This command is used to remove a key set from an entire stack storing all keys, where the index for removal is determined by the argument of the command. This command accepts one argument with the data type `uint16` and its valid value range is [1, 65534].

Triggering Bugs. There are two ways to exploit the *hidden* command `KeySetRemove()` to trigger the non-crash bugs, as shown in Table 2 (with the device ID labeled as *All*).

(1) If an invalid value `{"0" : 0}` is generated for the argument (following the *fuzzing policy 1* in Section 5.3), no matter what the current device state is, the message causes the device to become unresponsive and out of service.

(2) If no arguments are provided to the command (following the *fuzzing policy 3* in Section 5.3), the generated test message causes a device to become unresponsive to any subsequent request, regardless of the current device state.

Analysis. Since the non-crash bugs are present across all the devices, we suspect that they are associated with the Matter SDK. We thus report the bugs to the Matter SDK developer, and they confirm the bugs and have fixed them promptly (in `Matter v1.1`), as documented in [40]. Below, we analyze the root cause of the bug and its patch.

When a Matter controller sends the `KeySetRemove` command to a Matter device, the Matter device invokes the `KeySetRemoveCallBack` function to manage the command and process its payload. The payload specifies the index of the key set that should be removed. When the index is 0, the key set is associated with the *Identity Protection Key* (IPK). On the other hand, if the index is not specified (i.e., no argument is provided to the `KeySetRemove` command), the `KeySetRemoveCallBack` function automatically assigns the removal index to 0. The IPK serves as a crucial public key utilized by both the Matter device and Matter controller. It undergoes verification throughout the entire communication between the Matter device and controller to guarantee the integrity of the communication. If the verification of IPK fails, any further service request of the device is denied, resulting in a Denial of Service (DoS). Therefore, the IPK should not be removed in order to maintain the security and functionality of the Matter system.

However, the vulnerable `KeySetRemoveCallBack` function, when provided with a removal index of 0, incorrectly removes the IPK, and sends a `SUCCESS` response to the controller (or `mGPTFuzz` in our work). As the IPK is removed, the encrypted communication cannot be decrypted and verified, rendering the Matter device unresponsive to any subsequent requests. To rectify this issue, the patched `KeySetRemoveCallBack` function incorporates this checking: it first verifies whether the removal index is 0 or not

before executing the key removal action. If the index is 0, it replies with `INVALID_COMMAND` as a status code.

6.5.2 *Stateful* Non-Crash Bugs of Type N1

We find six Type N1 stateful non-crash bugs in the device *Govee Lighting H61E1* (with the device ID = 10). These non-crash bugs are related to three *hidden* commands, `MoveHue`, `MoveSaturation`, `EnhancedMoveHue`. Each command accepts two arguments, where the first argument is of the data type `enum` and takes a value $\in [0,1,2,3]$, and the second one is of the data type `uint8`. Taking `MoveHue` as an example, the first argument is `MoveMode`, which determines the direction of the hue change. Specifically. When `MoveMode` equals `0`, it indicates stop direction (i.e., no hue change); when `MoveMode` equals `1`, the device should increase its hue. When `MoveMode` equals `3`, the device should decrease its hue. The second argument `Rate` specifies the rate of movement per second.

Triggering Bugs. The Matter specification explicitly states that (1) a message, where the first argument, `MoveMode`, is set to 1 (increase) or 3 (decrease), and the second argument `Rate` equals 0, is considered as an *invalid* message, and (2) if this invalid message is sent to the device, the device should reject the message and respond with `INVALID_COMMAND`.

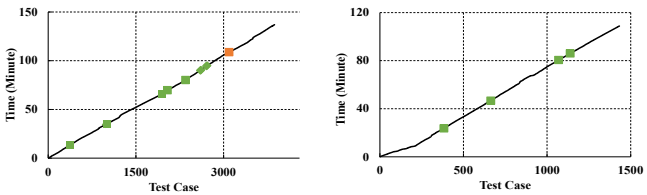
Our tool `mGPTFuzz` successfully extracts this critical information from the specification, and finds two ways to trigger the non-crash bugs for each command, as shown in Table 4 (corresponding to the rows where the device ID = 10). Taking `MoveHue` as an example, if the first parameter is assigned a value of 1 or 3, the second parameter is set to a value of 0, and at the same time, the current hue value is set to the maximum (i.e., 254), the device accepts the *invalid* test message. To trigger other non-crash bugs, the details of the test messages are also outlined in Table 4.

Observations. According to the Matter specification, the aforementioned test messages are *invalid*, and the *expected behavior* of the device is to reject them and reply with `INVALID_COMMAND`. But our observations reveal that upon receiving these test messages, the *actual behavior* of the device is to accept and process them, resulting in an alteration of the light color and the hue value changed to 0.

6.5.3 Non-Crash Bugs of Type N2

We find one Type N2 non-crash bug in each of the two devices, *Orein Bulb OS0100811267* (with the device ID = 3) and *Linkind Bulb LS0101811266* (with the device ID = 11). This bug is related to the *hidden* command `MoveColor(int16, int16)`, which is used to modify the `ColorMode` attribute on a device, prompting it to transition colors continuously at the specified rates. It accepts two arguments of the data type `int16`, which specify the rates of color changes per second.

Triggering Bug and Observations. When both the first and second arguments are assigned a value of 0, the resulting test message is *valid* and should be accepted by the device. However, the devices reject the message.



(a) Nanoleaf Lightstrip NF080K03-2LS (ID = 4) (b) Eve Motion Sensor 20EBY9901 (ID = 6)

Figure 11: Efficiency results, where a *red dot* denotes a crash bug and a *green dot* denotes a non-crash bug.

6.6 Comparison with Baseline Method

We consider `SNIPUZZ` as the baseline, which represents a state of the art in blackbox fuzzing of IoT devices [22]. For fair comparison, we have extended the capabilities of `SNIPUZZ`. (1) `SNIPUZZ` is integrated into our custom controller, so plaintext messages are presented to `SNIPUZZ`. Consequently, it is able to test Matter devices, which always use encrypted communication. (2) Hidden commands are provided to it. The details are described below.

`SNIPUZZ` is designed to detect crash bugs, and is not capable of detecting non-crash bugs. We thus compare the performance of crash bugs detection between `SNIPUZZ` and our tool `mGPTFuzz`. The original version of `SNIPUZZ` cannot fuzz-test Matter devices. We thus enhance `SNIPUZZ` by integrating it into our custom controller, so plaintext messages are presented to `SNIPUZZ`. We use the enhanced `SNIPUZZ` to test all the 23 Matter devices. However, after 24 hours of fuzz testing on each device, no bugs are found by `SNIPUZZ`.

`SNIPUZZ` requires the API-testing programs of IoT devices to collect seed messages and can only test the commands covered by the API-testing programs. As a result, it cannot detect the crash bugs triggered by the *hidden* commands, which include all the 5 crash bugs detected by `mGPTFuzz`.

We then proceed to investigate whether `SNIPUZZ` could detect these bugs if the corresponding *hidden* commands were provided to it. Specifically, for each discovered crash bug, we provide `SNIPUZZ` with a message associated with the hidden command that involves this bug. We then use the snippet determination algorithm of `SNIPUZZ` to partition these messages. The results show `SNIPUZZ` is unable to accurately determine the snippets for any of them. E.g., the message that can trigger a bug related to the command `Move_up` (discussed in Section 6.4.2) should be partitioned into 14 snippets, but it is inaccurately partitioned into 6 snippets after a 2-hour analysis.

We further investigate the snippet determination algorithm of `SNIPUZZ`, and have the following findings. The Matter protocol requires the payload of a Matter message to follow a JSON format. However, as `SNIPUZZ` removes the bytes in a message one by one to generate probe messages, this results in probe messages not following the JSON format.

Table 4: Some of the discovered non-crash bugs.

Device ID	Hidden API?	Command	Normal Message ⟶ Test (only *payload* is shown)	Required Device Initial State	Expected Behavior	Actual Behavior
All	✓	`KeySetRemove(uint16)` (**CVE-2023-42189**)	{"0": 20} ⟶ {"0": 0} ***Policy 1:*** Provide an invalid value 0 to the argument {"0": 20} ⟶ {} ***Policy 3:*** provide no argument	Any state	Device should reject message & respond `INVALID_COMMAND`	Device was Out of Service
10	✓	`MoveHue(enum, uint8)`	{"0": 1, "1": 2} ⟶ {"0": 1, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument {"0": 3, "1": 5} ⟶ {"0": 3, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentHue = 254	Device should reject message & respond `INVALID_COMMAND`	Device state was changed to CurrentHue = 0
10	✓	`MoveSaturation(enum, uint8)`	{"0": 1, "1": 2} ⟶ {"0": 1, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument {"0": 3, "1": 5} ⟶ {"0": 3, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	CurrentSaturation = 254	Device should reject message & respond `INVALID_COMMAND`	Device state was changed to CurrentSaturation = 0
10	✓	`EnhancedMoveHue(enum, uint8)`	{"0": 1, "1": 2} ⟶ {"0": 1, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument {"0": 3, "1": 5} ⟶ {"0": 3, "1": 0} ***Policy 1:*** Provide an invalid value 0 to the argument	EnhancedCurrentHue = 254	Device should reject message & respond `INVALID_COMMAND`	Device state was changed to EnhancedCurrentHue = 0
3, 11	✓	`MoveColor(int16, int16)`	{"0": 3, "1": 2} ⟶ {"0": 0, "1": 0} ***Policy 1:*** Provide a value 0 to both arguments	Any State	Device should accept message	Device rejected message

6.7 Efficiency

In the last column of Table 1, we present the total time spent by `mGPTFuzz` on testing each device. The longest testing time is approximately 5 hours for the device with ID = 7.

We use two devices as examples to illustrate the fuzzing efficiency in term of bugs discovered over time (Y-axis) and over the number of test messages (X-axis). As shown in Figure 11. `mGPTFuzz` can discover crash bugs and non-crash bugs efficiently. For the device *Nanoleaf Lightstrip NF080K03-2LS* shown in Figure 11(a), all bugs are found within 110 minutes and ≤3100 test message, and the first bug is found within 10 minutes. For the device *Eve Motion Sensor 20EBY9901* shown in Figure 11(b), all the four bugs are found within 90 minutes and ≤1200 test messages.

7 Discussion

Compared to prior work [12, 22, 51], `mGPTFuzz` is limited to fuzzing Matter devices. However, given the importance of Matter, it is worth the dedicated effort. Furthermore, the approach of LLM-assisted blackbox fuzzing can be generalized to other scenarios where the specification is available, such as Zigbee, Thread and Bluetooth.

Ethical Considerations and Proactive Harm Prevention. We have contacted all the vendors regarding the bugs and vulnerabilities of their products. We have reported the vulnerability (CVE-2023-42189) to the Matter SDK developer, since it impacts all the Matter devices. It has been fixed in Matter V1.1. After contacting them, we waited at least 90 days before reporting the vulnerabilities for CVE assignment.

Matter is released under the Apache 2.0 license, permitting various uses. The specification is publicly accessible on the official website [1, 2]. More importantly, according to the instructions of ChatGPT [41], ChatGPT does not use content from its business offerings such as ChatGPT Team or ChatGPT Enterprise to train its models. We utilized ChatGPT Team throughout the study. Therefore, our approach using ChatGPT does not cause an ethical issue.

8 Conclusion

As an industry-wide IoT standard, Matter is expected to completely change the ecology of smart devices. Thus, fuzzing of Matter devices is an emerging important problem. We present the first Matter fuzzer in the literature. A large language model is leveraged to transform the human-readable specification, over one thousand pages, to machine-readable information in the form of finite state machines (FSMs). Guided by the FSMs, our blackbox fuzzing is able to find stateful bugs and non-crash bugs, as well as crash bugs. We have built a prototype of `mGPTFuzz` and conducted an extensive evaluation involving 23 Matter devices. It finds 147 new bugs, including 61 zero-day vulnerabilities with three CVEs assigned.

Acknowledgements

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-2304720, CNS-2310322, CNS-2309550, and CNS-2309477. It was also supported in part by the Commonwealth Cyber Initiative (CCI). The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] Matter 1.0 application cluster specification, 2022. https://csa-iot.org/wp-content/uploads/2022/11/22-27350-001_Matter-1.0-Application-Cluster-Specification.pdf.

[2] Matter 1.0 core specification, 2022. https://csa-iot.org/wp-content/uploads/2022/11/22-27349-001_Matter-1.0-Core-Specification.pdf.

[3] Zafeer Ahmed, Ibrahim Nadir, Haroon Mahmood, Ali Hammad Akbar, and Ghalib Asadullah Shah. Identifying mirai-exploitable vulnerabilities in IoT firmware through static analysis. In *Proc. IEEE International Conference on Cyber Warfare and Security*, 2020.

[4] Amazon Developer. Alexa and matter, 2024. https://developer.amazon.com/en-US/alexa/matter.

[5] Amazon.com. GeeekPi nRF52840 Micro Dev Dongle. https://www.amazon.com/GeeekPi-nRF52840-Micro-Dev-Dongle/dp/B07MJ12XLG/ref=sr_1_2?crid=OQHJSOLRCRHI&keywords=nRF52840+Dongle&qid=1678331320&sprefix=nrf52840+dongle%2Caps%2C74&sr=8-2.

[6] Arrow Electronics. Matter solves IoT and smart home challenges, 2023. https://www.arrow.com/en/research-and-events/articles/matter-solves-iot-and-smart-home-challenges.

[7] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful REST API fuzzing. In *Proc. IEEE International Conference on Software Engineering (ICSE)*, 2019.

[8] Lina Berzinskas. Obfuscating Android Apps: Do you know your choices for protection?, 2020. https://proandroiddev.com/obfuscation-is-important-do-you-know-your-options-30b3ef396dfe.

[9] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *USENIX Security Symposium (USENIX Security)*, 2020.

[10] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv:2303.12712*, 2023.

[11] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[12] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through App-based fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[13] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Delay wreaks havoc on your smart home: Delay-based automation interference attacks. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2022.

[14] Haotian Chi, Qiang Zeng, and Xiaojiang Du. Detecting and handling IoT interaction threats in multi-platform multi-control-channel smart homes. In *USENIX Security Symposium (USENIX Security)*, 2023.

[15] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Jiaping Yu. Cross-App interference threats in smart homes: Categorization, detection and handling. In *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

[16] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proc. ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2016.

[17] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[18] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT. *arXiv preprint arXiv:2304.02014*, 2023.

[19] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *Proc. Security and Privacy in Communication Networks (SecureComm)*, 2018.

[20] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. AflIoT: Fuzzing on Linux-based IoT device with binary-level instrumentation. *Computers & Security*, 122:102889, 2022.

[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[22] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2021.

[23] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *USENIX Security Symposium (USENIX Security)*, 2020.

[24] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. HAWatcher: Semantics-aware anomaly detection for appified smart homes. In *USENIX Security Symposium (USENIX Security)*, 2021.

[25] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. *BrakTooth*: Causing havoc on bluetooth link manager via directed fuzzing. In *USENIX Security Symposium (USENIX Security)*, 2022.

[26] Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative AI. *arXiv preprint arXiv:2306.06782*, 2023.

[27] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. LLM-Adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*, 2023.

[28] Apple Inc. Matter support in ios 16, 2023. https://developer.apple.com/apple-home/matter/.

[29] Joshua Pereyda. Boofuzz: Network protocol fuzzing for humans, 2017. https://github.com/jtpereyda/boofuzz.

[30] Stephan Kleber, Henning Kopp, and Frank Kargl. NEMESYS: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In *USENIX Workshop on Offensive Technologies*, 2018.

[31] Platon Kotzias, Srdjan Matic, Richard Rivera, and Juan Caballero. Certified PUP: abuse in authenticode code signing. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[32] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proc. IEEE International Conference on Software Engineering (ICSE)*, 2022.

[33] Lannan Luo and Qiang Zeng. Solminer: Mining distinct solutions in programs. In *Proc. IEEE International Conference on Software Engineering (ICSE)*, 2016.

[34] Lannan Luo, Qiang Zeng, Bokai Yang, Fei Zuo, and Junzhe Wang. Westworld: Fuzzing-assisted remote dynamic symbolic execution of smart Apps on IoT cloud platforms. In *Annual Computer Security Applications Conference (ACSAC)*, 2021.

[35] Xiaoyue Ma, Qiang Zeng, Haotian Chi, and Lannan Luo. No more companion Apps hacking but one dongle: Hub-based blackbox fuzzing of IoT firmware. In *Proc. ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2023.

[36] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, 2024.

[37] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2021.

[38] Ibrahim Nadir, Zafeer Ahmad, Haroon Mahmood, Ghalib Asadullah Shah, Farrukh Shahzad, Muhammad Umair, Hassam Khan, and Usman Gulzar. An auditing framework for vulnerability analysis of IoT system. In *Proc. IEEE European Symposium on Security and Privacy Workshops*, 2019.

[39] National Institute of Standards and Technology (NIST). CVE-2022-25836, 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-25836.

[40] National Institute of Standards and Technology (NIST). CVE-2023-42189, 2023. https://nvd.nist.gov/vuln/detail/CVE-2023-42189.

[41] OpenAI. What is ChatGPT, 2022. https://help.openai.com/en/articles/6783457-what-is-chatgpt.

[42] OpenAI. API for authentication, 2023. https://platform.openai.com/docs/api-reference/authentication.

[43] OpenAI. Createtranscription: Temperature parameter, 2023. https://platform.openai.com/docs/api-reference/audio/createTranscription#audio-createtranscription-temperature.

[44] OpenAI. Gpt-4 and GPT-4 turbo documentation, 2023. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.

[45] OpenThread. ot-br-posix, 2024. https://github.com/openthread/ot-br-posix.

[46] PeachTech. Peach fuzzer community. https://peachtech.gitlab.io/peach-fuzzer-community/.

[47] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A greybox fuzzer for network protocols. In *Proc. IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[48] Project CHIP. Connected home over IP, 2024. https://github.com/project-chip/connectedhomeip.

[49] PyPI. pytesseract: Python-tesseract is an optical character recognition (OCR) tool for python, 2023. https://pypi.org/project/pytesseract/.

[50] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. NSFuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–26, 2023.

[51] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. DIANE: Identifying fuzzing triggers in Apps to generate under-constrained inputs for IoT devices. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2021.

[52] Mengfei Ren, Xiaolei Ren, Huadong Feng, Jiang Ming, and Yu Lei. Z-fuzzer: device-agnostic fuzzing of zigbee protocol implementation. In *Proc. ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021.

[53] Vinay Sachidananda, Suhas Bhairav, and Yuval Elovici. OVER: Overhauling vulnerability detection for IoT through an adaptable and automated static analysis framework. In *Proc. ACM Symposium on Applied Computing (SAC)*, 2020.

[54] Schutzwerk GmbH. Security considerations for matter developers, 2023. https://www.schutzwerk.com/en/blog/matter-security-considerations/.

[55] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. LLM4Fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108*, 2024.

[56] Zhan Shu and Guanhua Yan. Iotinfer: Automated blackbox fuzz testing of IoT network protocols guided by finite state machine inference. *IEEE Internet of Things Journal*, 9(22):22737–22751, 2022.

[57] Simeng Sun, Yang Liu, Dan Iter, Chenguang Zhu, and Mohit Iyyer. How does in-context learning help prompt tuning? *arXiv preprint arXiv:2302.11521*, 2023.

[58] The Verge. Nest thermostat gains Matter support, works with Apple home, 2023. https://www.theverge.com/2023/4/18/23687751/nest-thermostat-matter-support-apple-home.

[59] Jennifer Pattison Tuohy. Matter's plan to save the smart home, 2021. https://www.theverge.com/22787729/matter-smart-home-standard-apple-amazon-google.

[60] Junzhe Wang and Lannan Luo. Privacy leakage analysis for colluding smart apps. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2022.

[61] Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. Can a deep learning model for one architecture be used for others? retargeted-architecture binary code analysis. In *USENIX Security Symposium (USENIX Security)*, 2023.

[62] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[64] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[65] Wiki. Matter (standard), 2022. https://en.wikipedia.org/wiki/Matter_(standard).

[66] Wired. What is matter?, 2023. https://www.wired.com/story/what-is-matter/.

[67] wireghoul. Doona, 2019. https://github.com/wireghoul/doona.

[68] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proc. IEEE International Conference on Software Engineering (ICSE)*, 2024.

[69] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[70] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proc. ACM Conference on Human Factors in Computing Systems (CHI)*, 2023.

[71] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. Resilient decentralized android application repackaging detection using logic bombs. In *Proc. ACM International Symposium on Code Generation and Optimization (CGO)*, 2018.

[72] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs (tdsc). *IEEE Transactions on Dependable and Secure Computing*, 18(6):2582–2600, 2021.

[73] Yu Zhang, Nanyu Zhong, Wei You, Yanyan Zou, Kunpeng Jian, Jiahuan Xu, Jian Sun, Baoxu Liu, and Wei Huo. NDFuzz: A non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity*, 5(1):1–21, 2022.

[74] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *USENIX Security Symposium (USENIX Security)*, 2019.

[75] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[76] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *USENIX Security Symposium (USENIX Security)*, 2021.

[77] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2022.

[78] Lipeng Zhu, Xiaotong Fu, Yao Yao, Yuqing Zhang, and He Wang. FIoT: Detecting the memory corruption in lightweight IoT device firmware. In *Proc. International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2019.

[79] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[80] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

A Example of an Assembled Prompt

Figure 12 shows the prompt for querying the information of the `Group` cluster.

Prompt:

You will be provided with a section of the protocol specification text about the `Groups` Cluster. Please provide responses to the questions in the specified order and format as outlined in the text provided. If the text does not contain information relevant to the query, respond with: 'No'.

`Groups` Cluster text:
The Groups cluster manages, per endpoint, the content of the node-wide Group Table that is part of the underlying interaction layer. ... The GroupID field is set to the GroupID field of the received RemoveGroup command.

Please respond to the questions based on the provided text of the `Groups` cluster with the required format. Queries are as follows:
1. From the "Data Types" section, extract all derived datatypes and their corresponding value ranges, especially the datatypes with the suffixes "Enum" and "Struct". Ensure the return format is a Dictionary similar to the format of the Example Output.
2. From the "Commands" section, extract all commands, and for each command, extract the datatype and value range for each of its arguments. Ensure the return format is a Dictionary similar to the format of the Example Output.
3. From the "Attributes" section, extract all attributes, and for each attribute, extract its datatype and value range. Ensure the return format is a Dictionary similar to the format of the Example Output.
4. Return a Python list that includes all command IDs.
5. Return a Python list that includes all attribute IDs.

Here is an Example Output in JSON format.

[Example Output].

Figure 12: Assembled prompt for querying the information of the `Groups` cluster.