Supplementary Materials for Submission

Jincheng Wang, Le Yu, and Xiapu Luo

TABLE I: Format of the ”Add Group If Identifying” message

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

I. Background

In this section, we introduce the main concepts in Zigbee and provide some background on large language models.

A. Zigbee Protocol

Zigbee is a communication protocol built to provide low-power, low-cost wireless mesh networking for IoT devices, and it regulates a range of device functionalities, e.g., lighting and locking. To provide seamless communication within the Zigbee ecosystem, Zigbee Alliance proposed the concept of “cluster” in the protocol specification [1], which defines a set of common message formats and data structures that devices can follow. For example, Zigbee specifies a “Group” cluster (Cluster ID 0x0004), which allows devices to be assigned to one or more groups and supports simultaneous control of several devices. The cluster defines several device properties (e.g., group table), which the device should support to specify the current grouping states. Moreover, the cluster defines six message types with fixed payload formats for communication. For example, Table I shows the format of the “Add Group If Identifying” message (Command ID 0x05), which adds the group specified in the message payload if the device currently is in identifying status. Any message that violates the format (e.g., missing fields or invalid field values), will be regarded as malformed commands and filtered by the target device without further processing.

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

Jincheng Wang and Xiapu Luo are from The Hong Kong Polytechnic University. Le yu is from Nanjing University of Posts and Telecommunications.

B. Large Language Models

Large language models (LLMs) are a type of machine learning model that can process and generate natural language text. Fundamentally, they are trained on massive amounts of text data to perform statistical language modeling and word prediction. In particular, LLMs achieve amazing performance in text summarization, text reasoning, and contextual conversations. For example, a recent study [2] shows that summaries generated by the LLM are favored by human annotators over the reference summaries in the document. Van et al. show that LLMs perform well in comprehending the given document (e.g., radiology reports and doctor-patient dialogue), and reason it with domain knowledge [3]. Finally, Mao et al. find that LLMs have the potential to understand the history of conversation and generate responses that are contextually relevant and coherent with the previous conversation turns [4].

One advantage of LLMs is that they can be quickly adapted to a specific task (e.g., mental health classification [5], question and answering [6], and penetration testing [7]). To achieve the goal, prompt engineering is commonly employed [8]. In the prompting paradigm, a pre-trained LLM is provided with a snippet of text as an input and is expected to provide a relevant completion of this input as output. Prompt engineering is intended to provide a set of principles and techniques for designing prompts to squeeze out the best performance from these machine learning models.

The multi-agent framework represents a burgeoning paradigm in the field of LLMs. This innovative framework leverages the collaborative efforts of multiple language models, referred to as “agents”, to tackle complex tasks more effectively [9], [10]. Each agent within the framework is specialized to perform distinct functions, such as tool usage, scheduling, or providing contextual understanding. The advantages of adopting a multi-agent framework include enhanced context management within LLMs, dynamic problem-solving capabilities, and an overall increase in efficiency and effectiveness through specialization [9], [11]. This framework has been successfully applied across various applications, from automated code generation [12] to security-oriented tasks such as penetration testing [7] and incident management [13]. Despite its growing popularity and evident potential, the application of multi-agent frameworks in the context of fuzzing remains under-explored. This gap in research and application provides the primary motivation for our study, aiming to harness the collaborative power of multiple agents to enhance the efficiency and effectiveness of fuzzing methodologies.

II. Experimental Setup

Z-Stack simulation for code coverage evaluation. During our experiment, we found that it is challenging to evaluate code coverage. The main reason is most Zigbee device vendors do not open-source their stack implementations (either source codes or binary firmware), such that we cannot instrument the stack implementation to calculate statement/edge coverage, and only black-box fuzzing can be initiated. The only open-source Zigbee stack we found is the Z-Stack of Texas Instruments [14]. Inspired by [15], we leverage the IAR development toolchain and set up a simulation platform, which supports simulating stack execution, Zigbee message transmission/reception, and most importantly, coverage analysis. We write a stack driver (900 lines of C code) to build a Zigbee end device application. The stack driver and the stack source codes are used by the simulation tool C-SPY, which allows us to create a simulated end device as the target device. Our driver registers a set of plugins provided by Z-Stack, such that the simulated end device can support 154 cluster messages from the 22 standard clusters, compared with the driver [15] which only supports 22 messages.

Real-world Zigbee devices under test (DUT). Besides fuzzing the simulation device, we also selected 11 off-the-shelf Zigbee devices from various vendors for evaluation, covering well-known brands, such as Texas Instruments, Philips, Third Reality, Sengled, Aqara, and Tuya. The types of selected devices include smart switches, plugs, lighting, locks, and sensors. These devices are either recommended by Amazon or the well-selling products in supermarkets. Note that some devices have been found with zero-day vulnerabilities and have not been fixed yet. As a result, we anonymize the names of these devices and their models. For the remaining devices, the list of their names and models are recorded in our complementary material for reference [16].

Baseline methods. We select four popular Zigbee fuzzing tools as the baseline: BOOFUZZ [17], Z-FUZZER [15], BEEHIVE [18], and LLMIF [19]. They are either widely used in industry or reported in top security conferences. Moreover, the design of these tools covers the existing solutions to important fuzzing tasks, in particular, message format construction and interesting value collection, which makes them suitable for comparison with our method. Specifically, BOOFUZZ is a famous grammar-based fuzzing tool that is customized by [15] to support Zigbee fuzzing. Z-FUZZER is a coverage-guided Zigbee fuzzing tool, and it has been used to identify critical zero-day vulnerabilities in the Z-Stack. BEEHIVE studies the cluster message format, and manually extracts a set of interesting values for enumeration of field values. Finally, LLMIF uses the naive RAG method to analyze the specification document and extracts incomplete information (e.g., command formats without command identifiers). Moreover, it randomly samples message sequences to schedule the fuzzing process. Note that to generate testing cases with correct message formats, for information cannot be extracted by the RAG method (e.g., the command identifier), LLMIF directly calls the LLM to generate it.

LLM usage. In our experiment, we use GPT-4o as the general-purpose LLM. The LLM accepts one hyper-parameter, *temperature* $\in [0, 1]$, which regulates the randomness and creativity. To mitigate the LLM's hallucination problem (i.e., high false positives in the generated answers), the parameter is set to zero. The same setting applies to the baseline method. Since LLMIF is the only LLM-guided fuzzing tool among the baseline methods, it means that MAIF and LLMIF will use the same general-purpose LLM (GPT-4o in our case) with the same temperature setting.

ID	Device Type	Vendor	Model
1	Light Bulb	Philips Hue	LWB014
2	Signal Intermediator	Tuya	Tuya
3	Light Bulb	Sengled	E11-N1EAW
4	Plug	Aqara	KTBL12LM
5	Motion Sensor	TREATLIFE	TREATLIFE
6	Plug	SmartThings	GP-WOU019
7	thermostat	THIRDREALITY	B0BF9W3WMK
8	Switch	Tuya	Tuya
9	Locker	Yale	YRD226
10	Lihgt Strip	Aqara	LGYCDD01LM
11	Texas general device	Texas Instruments	CC2538
12	Texas lighting bulb	Texas Instruments	CC2538
13	Texas closure lock	Texas Instruments	CC2538
14	Texas security alarm	Texas Instruments	CC2538

TABLE II: Zigbee devices under test

III. Device List Under Testing

The details of devices under testing are shown in Table II.

IV. MAIF Implementation

Unlike fuzzing traditional network protocols, e.g., SMTP [20], fuzzing real-world Zigbee devices requires the support of specific radio modules (e.g., CC2530). To fuzz the real-world Zigbee device, we design a fuzzing tool that contains two components: *Fuzzing controller* and *stack controller*. The fuzzing controller is responsible for running the execution agent and organizing the fuzzing workflow. In particular, chatGPT-4o is used as the large language model. The stack controller, on the other hand, works as a driver to operate a programmable Zigbee radio and provides fundamental support for Zigbee communication, e.g., message transmission and reception.

The fuzzing controller leverages the idea of "building block" to construct the message, which is commonly used in generation-based fuzzing [15], [21]. Specifically, we implement 47 basic data types specified in Zigbee specification, e.g., enum8 and string, and use them as the building block for assembling the message payload. As a result, the extracted protocol information, e.g., the message format and interesting value repositories, can be easily integrated into the assembling process. We further implement a total of seven mutation operators (Section V.B) to perform mutation. The fuzzing controller is implemented in Python with 3,000 lines of code, and it runs on a Raspberry Pi 4 with Ubuntu 20.04 operating system.

The stack controller aims to create and maintain the Zigbee communication channel with the target device. Considering availability and popularity, we select CC2538 with a fully compliant Zigbee solution Z-Stack [14] as the hardware radio. We develop a driver on top of Z-Stack and transform CC2538 into a Zigbee node. The node forms a fully-controlled Zigbee network which allows the target device to join. On the one hand, it communicates with the fuzzing controller through the universal asynchronous receiver/transmitter (UART) channel, i.e., receiving testing cases and forwarding device responses. On the other hand, it communicates to the target device through the transparent and authenticated Zigbee network, i.e., transmitting the testing case and monitoring device responses. The driver is developed with 1,000 lines of C codes.

V. Discussions on MAIF Generality

In this paper, we mainly use Zigbee as the target protocol. However, MAIF can be extended to fuzz IoT protocols beyond Zigbee. Specifically, users only need to take two steps, i.e., update the input specification and replace the hardware radio.

Update the input specification. MAIF relies on the protocol specification to extract critical message information. As a result, users need to provide as inputs the specification that details the messages of the protocol under fuzzing, e.g., Z-Wave Command Classes [22]. With the specification analysis agent, MAIF will extract the precise protocol information (e.g., message format and message dependency), and use them to guide the fuzzing round.

Replace the hardware radio. To fuzz the real-world device with a specific IoT protocol, a hardware radio is necessary for transmitting the testing case and receiving the response within a specific wireless channel. Users need to prepare the corresponding hardware (e.g., CC2530 for Zigbee), download the commercial protocol stack (e.g., Z-Stack), and implement the driver that interacts with MAIF. 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.

VI. Related Work

Format-aware IoT fuzzing. Format-aware fuzzing tools generate messages based on the well-formed message template [15], [17], [18], [23]–[25]. Specifically, [17] proposes a network protocol fuzzing framework BooFuzz. Taking the message format description and the interesting value collection as inputs, the tool automates the testing case generation, monitors the target's status, and records suspicious cases. [15] bases on BooFuzz and designs a fuzzing tool Z-Fuzzer, which aims at fuzzing Zigbee protocol and uses code coverage as feedback to guide the fuzzing process. [18] manually extracts 22 cluster message formats and interesting field values (e.g., attribute ID) from the specification and develops BeeHive, which generates testing cases to enumerate these messages and field values. [24] relies on the message formats recorded in the open-source protocol library to generate testing cases. All of them require significant human efforts to construct and maintain the message formats, which are labor-intensive and error-prone. [23], [25] proposes to use phone apps and APIs that control the device through the vendor platform to construct the message formats. However, they mainly focus on inferring the platform-level message format (e.g., Restful API messages), which only covers a small fraction of the message formats of the underlying protocol that is directly used by the device, e.g., Zigbee.

Compared with the work mentioned above, we propose to utilize the LLM to address the main challenge of format-aware IoT fuzzing. With only the specification document as input, our method automatically constructs the message format with high accuracy while avoiding significant human efforts. Besides the knowledge of message formats, our method also extracts useful protocol information, e.g., interesting values and message dependencies, which benefits the fuzzing process and is omitted by previous works. In particular, our evaluation covers most Zigbee fuzzers [15], [17], [18], and the evaluation result shows that our method outperforms them in terms of code coverage and vulnerability identification. Since some fuzzers [24] are not open-source, we cannot evaluate their performance.

References

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

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

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

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

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

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

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

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

[9] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From llms to llm-based agents for software engineering: A survey of current, challenges and future," *arXiv preprint arXiv:2408.02479*, 2024.

[10] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[11] Y. Li, H. Wen, W. Wang, X. Li, Y. Yuan, G. Liu, J. Liu, W. Xu, X. Wang, Y. Sun *et al.*, "Personal llm agents: Insights and survey about the capability, efficiency and security," *arXiv preprint arXiv:2401.05459*, 2024.

[12] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, "Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing," *arXiv preprint arXiv:2409.10737*, 2024.

[13] K. An, F. Yang, L. Li, Z. Ren, H. Huang, L. Wang, P. Zhao, Y. Kang, H. Ding, Q. Lin *et al.*, "Nissist: An incident mitigation copilot based on troubleshooting guides," *arXiv preprint arXiv:2402.17531*, 2024.

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

[15] M. Ren, X. Ren, H. Feng, J. Ming, and Y. Lei, "Z-fuzzer: device-agnostic fuzzing of zigbee protocol implementation," in *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2021, pp. 347–358.

[16] "MAIF," https://github.com/wang70880/MAIF, 2024.

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

[18] X. Wang and S. Hao, "Don't kick over the beehive: Attacks and security analysis on zigbee," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2857–2870.

[19] J. Wang, L. Yu, and X. Luo, "Llmif: Augmented large language model for fuzzing iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 196–196.

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

[21] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008, pp. 206–215.

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

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

[24] X. Ma, Q. Zeng, H. Chi, and L. Luo, "No more companion apps hacking but one dongle: Hub-based blackbox fuzzing of iot firmware," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, 2023, pp. 205–218.

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