Towards Improving Fuzzer Efficiency for the MQTT Protocol

Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista
Department of Computer Science
University of São Paulo
São Paulo, Brazil
Email: {luisgar, batista}@ime.usp.br

2021 IEEE Symposium on Computers and Communications (ISCC) | 978-1-6654-2744-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCC53001.2021.9631520

***Abstract*—MQTT's security has been a major concern because of its weak protocol implementations. Over the last few years, several fuzzing frameworks have been proposed to mitigate this issue. However, these frameworks lack sufficient knowledge of MQTT's specifications, requiring a considerable amount of network packets to cover all of its features and functionality. In this paper, we explain how to improve the efficiency of fuzzing frameworks for MQTT by using a grammar based on its specifications. Although defining a grammar is time-consuming and complex, these drawbacks are overshadowed by its benefits, such as deep state exploration and efficiency. Our improvements are implemented in MQTTGRAM, a new grammar-based fuzzer for MQTT. Due to these improvements, MQTTGRAM offers higher code coverage with significantly fewer packets than existing MQTT fuzzers. For instance, MQTTGRAM exchanges up to 9x fewer packets than its counterparts without reducing the line coverage.**

***Index Terms*—MQTT, Internet of Things, Security, Software Vulnerability Testing, Fuzzing, Grammar**

I. Introduction

It is estimated that the Internet of Things (IoT) will expand to 50 billion devices in the coming decade [1]. As a result, IoT will be characterized by a high degree of heterogeneity in terms of devices and network protocols, even more so than the traditional Internet [2].

Several protocols have been developed specifically for IoT, such as the Advanced Message Queuing Protocol (AMQP), the Constrained Application Protocol (CoAP), the Data Distribution Service (DDS), the Extensible Messaging and Presence Protocol (XMPP), and the Message Queuing Telemetry Transport (MQTT). Among IoT protocols, MQTT is considered the best, offering lightweight messaging and low-bandwidth consumption [3]. MQTT allows users to take advantage of publish-subscribe (pub/sub) messaging in low-powered devices [4], [5]. Pub/sub messaging has several advantages over traditional client-server approaches. First, subscribers receive messages regardless of their connectivity. For example, if a subscriber is offline, messages can be queued for delivery after regaining connectivity. Second, pub/sub messaging offers asynchronous communication, meaning data is transmitted at irregular intervals. This means that publishers and subscribers can send and receive messages quickly, without being synchronized by an external clock.

Despite being suitable for several IoT scenarios, MQTT's security has been a major concern because of its weak protocol implementations [6]–[9]. Over the last few years, several fuzzing frameworks have been proposed to mitigate this issue [10]–[15]. However, they lack sufficient knowledge of MQTT's specifications, requiring a considerable amount of network packets to cover all of its features and functionality, thus being inefficient.

This paper presents an approach to improve the efficiency of fuzzing frameworks for MQTT by using a grammar based on its specifications. Although defining a grammar is time-consuming and complex, these drawbacks are overshadowed by its benefits such as deep state exploration and efficiency. We've developed a tool called `MQTTGRAM`, which is a fuzzer for MQTT based on this approach. `MQTTGRAM` offers higher code coverage with significantly fewer packets than existing MQTT fuzzers. For instance, `MQTTGRAM` exchanges up to 9x fewer packets than its counterparts without reducing the line coverage.

The research presented in this paper involves four efforts: (1) development of a grammar; (2) proposal of a packet generator that uses the MQTT grammar to produce syntactically valid inputs; (3) proposal of a grammar-based fuzzer, capable of injecting inputs into the broker; and (4) a comparison of our grammar-based approach with other popular fuzzers.

The remainder of this paper is organized as follows. Section II explains the MQTT protocol and fuzz testing. Section III presents an overview of existing fuzzers for MQTT. Section IV explains our approach to improve fuzzer efficiency for the MQTT protocol. Section V presents the performance evaluation of `MQTTGRAM`. Finally, Section VI marks the conclusions of this paper.

II. Background

A. MQTT

MQTT is a publish-subscribe messaging protocol developed by IBM in 1999. MQTT has two main components: MQTT clients and MQTT brokers. An MQTT client can be either a *publisher* or a *subscriber*. A publisher sends messages to subscribers, however messages are not sent directly. The MQTT broker acts as an intermediary, receiving messages from publishers and sending them to interested subscribers. Messages are sent using control packets, and are organized in topics.

978-1-6654-2744-9/21/$31.00 ©2021 IEEE

MQTT has fourteen types of control packets, which can be divided into three fields: (1) the fixed header; (2) the variable header; (3) and the payload. The fixed header is required for every MQTT control packet, while the variable header and payload are optional.

Several research papers compare MQTT with other protocols in different environments. For instance, Joshi et al. [16] evaluated the performance of MQTT, HTTP, and CoAP in smart home environments. MQTT outperformed HTTP and CoAP, having lower latency and energy consumption.

MQTT and HTTP are the most popular protocols for IoT [17]. Regardless of HTTP's popularity, MQTT's low-latency, energy consumption, and reliability make it the de facto protocol for IoT environments [18].

B. Fuzz Testing

Fuzz testing is an automated technique that injects random or invalid inputs into a system [19], [20]. The system behavior is later analyzed for possible vulnerabilities. Fuzzing is one of the most popular dynamic analysis techniques for several reasons. First, it has higher scalability and accuracy than other automated testing techniques [21]. Second, it has proven to have considerable impact for detecting vulnerabilities in IoT [8] and MQTT [11], [14]. Third, it can mitigate implementation issues and zero-day attacks. These benefits are reasons why fuzz testing is considered the most promising method for discovering vulnerabilities in IoT [22].

Fuzzing frameworks, also known as *fuzzers*, can be classified based on their program knowledge or input generation.

A *naive fuzzer* generates random strings without considering input formats or program specifications. Although naive fuzzers are fairly simple to implement, they have low code coverage because of their lack of program knowledge.

Over the last few years, parsers have become more robust, often rejecting invalid inputs. Thus, current fuzzers require semi or syntactically valid inputs to avoid immediate rejection by the System Under Test (SUT). These types of fuzzers are commonly referred to as *smart fuzzers*. *Smart fuzzers* possess program knowledge, which allows them to traverse through deeper code paths. Smart fuzzers offer higher code coverage than naive fuzzers, but lack simplicity and flexibility.

In terms of input generation, a *mutation-based fuzzer* generates test cases by modifying legal/valid inputs. These types of fuzzers gain program knowledge through samples of valid inputs. Examples of mutation-based techniques are flipping bits or deleting characters. Over time, mutated inputs may drastically differ from the original, requiring more manipulations and thus slowing down performance. Another drawback is that mutation-based fuzzers require considerable amount of test cases to traverse through deeper code paths [22]. However, mutation-based fuzzers offer simplicity and flexibility compared to other types of fuzzers.

A *generation-based fuzzer* generates inputs from scratch. These type of fuzzers gain program knowledge through grammars or input specifications. The process of defining a grammar for a given target system is time-consuming and complex, however generation-based fuzzers offer significant advantages over their counterparts, such as higher code coverage and deeper state exploration.

Although fuzzing covers a wide range of domains, it shows more promise for network protocols [23]. *Network-based fuzzers* can play two roles: a client, interacting directly with the SUT; or a man-in-the-middle, interacting between a client and the SUT.

Fig. 1 illustrates an example of a network-based fuzzer, specifically an MQTT subscriber, interacting directly with the broker. The interaction shown is based on a real-world vulnerability (CVE-2019-11779) triggered by sending a crafted subscribe packet to the MQTT broker, which can lead to a stack overflow.

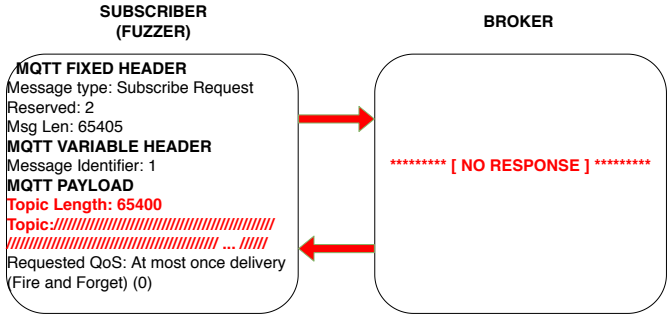


Fig. 1: Fuzzer sending a crafted subscribe packet and causing a stack overflow (CVE-2019-11779)

III. Related Work

It is important for fuzzers to: (1) possess knowledge of MQTT's specifications; (2) cover all functionalities; and (3) guarantee acceptance by the broker. However, current fuzzing frameworks have limitations in all three categories.

Existing naive fuzzers, such as Scapy's `fuzz()` function [15], lack program knowledge, meaning fuzzed packets may be immediately rejected by the broker, and thus deep protocol states are difficult to reach. Although injecting abnormal packets is effective, it is important to adhere to protocol specifications.

Existing mutation-based fuzzers [10]–[12], gain program knowledge through samples or templates. Although mutation-based fuzzing speeds up test generation, it also lacks knowledge of protocol specifications, meaning it may occasionally: (1) craft syntactically invalid inputs; (2) require considerable amount of test cases to reach deep protocol states; (3) cover limited functionality of MQTT; and (4) require a deep understanding of MQTT's specifications to configure the fuzzer effectively.

Existing generation-based fuzzers [13], [14], have limited knowledge of MQTT's specifications, and thus have difficulties to cover all of its features. For example, analyzing the packet generator proposed by Casteur et al. [13] reveals that if the User Name Flag of a *CONNECT* packet is set to 1, then the Password Flag is also set to 1. However, according to the

MQTT 3.1.1 specification, passwords are optional if the User Name Flag is set to 1.

Grammar-based approaches, such as the one proposed in this paper, offer several advantages over existing MQTT fuzzers: (1) higher level of automation for generating packets; (2) higher flexibility for integrating into fuzzing frameworks; (3) better understanding of the protocol and its control packet structure; (4) higher flexibility for developers to modify for their own needs, (5) guaranteed acceptance by the SUT; (6) higher feature coverage of the protocol; and (7) systematic and efficient test generation.

IV. An Approach to Improve the Efficiency of Fuzzing Frameworks for MQTT

We propose a generation-based fuzzing approach that uses a grammar to generate network packets from scratch. Algorithm 1 presents the pseudocode of our approach. The algorithm is explained in the following subsections.

Algorithm 1: Grammar-based fuzzing approach for MQTT

Input: Protocol State S

while *stopping_criteria == false* **do**
- $t \leftarrow select\_packet\_type(S)$;
- $m \leftarrow generate\_packet(t)$;
- $r \leftarrow feed\_input(m)$;
- $S \leftarrow evaluate\_response(r)$;
- $L \leftarrow monitor\_program(S)$

Output: L

A. Selecting the packet type

Network packets must follow a particular order to reach deep protocol states. For example, a *DISCONNECT* packet must be sent only if the MQTT client is in a connected state. The interactions must adhere to formal specifications of the protocol, transitioning to different states in the correct order. Protocol states are determined by control packets. For example, if an MQTT client receives a *CONACK* packet from the broker, then the client is connected.

Our approach suggests generating packets based on the current broker state. For example, at the beginning of the fuzzing campaign, a fuzzer based on our approach must be in a disconnected state because it has not established a connection with the broker. Thus, it performs a TCP handshake and sends a *CONNECT* packet to the broker.

B. Generating the packets

We propose a packet generator to produce syntactically valid inputs. The packet generator consists of two main components: a grammar and a state engine. Protocol fuzzers need to possess knowledge of both the input structure and states, maintaining execution consistency throughout the fuzzing campaign [24]. It is not enough to simply gain program knowledge through a grammar. The fuzzer needs to be state-aware [25], and generate packets based on the type of packets received from the broker. Moreover, the fuzzer needs to generate the necessary packets to reach deep protocol states.

The goal of the state engine in the packet generator is to handle messages exchanged between the fuzzer and the broker during the fuzzing campaign. The state engine performs mainly three tasks: (1) sends packets produced from the MQTT grammar; (2) handles TCP sequence and acknowledgment numbers; and (3) analyzes incoming messages from the broker.

The proposed grammar is based on the MQTT 3.1.1 standard [26], which is among the most popular versions of MQTT. Our MQTT grammar specifies the input format of the fourteen control packet types. Each packet type has expansion rules considering the MQTT 3.1.1 standard.

Algorithm 2 presents the pseudocode of the packet generator. The packet generator randomly chooses an expansion rule for each nonterminal. The process of choosing a nonterminal and applying an expansion rule is adapted from [27] considering the MQTT standard.

The MQTT 3.1.1 standard states that several packet fields must be represented as UTF-8 encoded strings, such as usernames, passwords, and topic names. Each UTF-8 encoded string is prefixed with a two-byte length field. This is required in order to identify multiple UTF8-encoded strings correctly. Thus, when there are no further expansions, the algorithm calculates the length of each UTF-8 encoded string and remaining length.

Algorithm 2: Packet generator

Input: Packet Type t, Grammar g

while *nonterminals from $t > 0$* **do**
- $n \leftarrow choose\_nonterminal(t, g)$;
- $e \leftarrow choose\_expansion\_rule(n, g)$;
- $s \leftarrow apply\_expansion(e, n)$;

for *field_lengths in s* **do**
- $calculate(field\_lengths)$;

$p \leftarrow convert\_string\_to\_bytes(s)$;

$m \leftarrow calculate\_remaining\_length(p)$;

Output: m

Fig. 2 presents an example of our grammar-based approach when generating a *PUBLISH* packet. Nonterminal symbols are represented with rectangles, and terminal symbols with ellipses. It is worth noting that the grammar is recursive, meaning nonterminal symbols such as `topic-name` can be expanded an infinite number of times, yielding complex inputs.

C. Feeding the packets

In our approach, the fuzzer interacts directly with the broker rather than acting as a man-in-the-middle. This approach provides more control over packets exchanged during the fuzzing campaign. Rather than requiring initial test cases, the fuzzer generates its own network packets to communicate with the broker successfully.

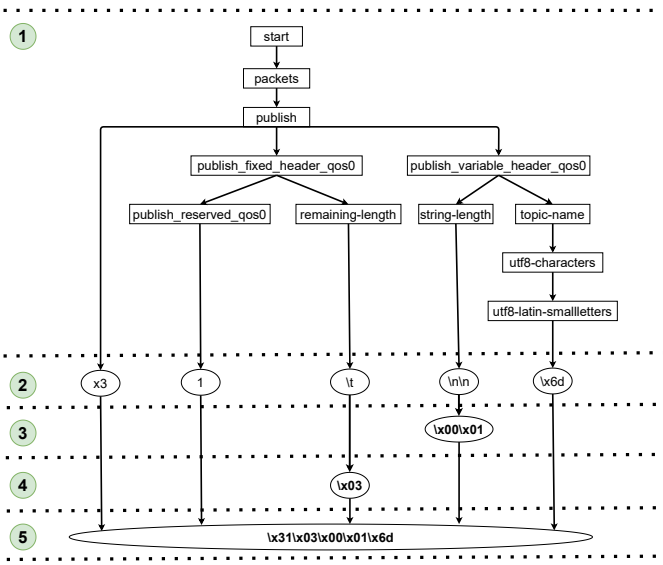


Fig. 2: Generating a PUBLISH packet

D. Evaluating the response

We propose sending mainly five types of packets to the broker: *PUBLISH*, *SUBSCRIBE*, *UNSUBSCRIBE*, *PINGREQ*, and *DISCONNECT*. For every packet sent, the last response from the broker is evaluated based on the TCP flag and the broker state. If the packet's TCP flag is set to Finish (FIN) or Reset (RST), the fuzzer must reconnect to the broker automatically; otherwise the fuzzer will analyze the type of MQTT packet, and respond accordingly. For example, if the broker sends a *PUBLISH* packet, the fuzzer must respond by sending a *PUBACK* packet or *PUBREC* packet when the QoS level is set to 1 and 2 respectively (MQTT 3.1.1 supports three QoS levels [26]).

E. Monitoring the program

The fuzzing process must be monitored constantly to ensure that the fuzzer is executing correctly. Logs are used to store information about the fuzzing campaign. A stopping criteria, such as time or number of exchanged packets, must be specified in each fuzzing campaign.

V. Performance Evaluation

The goal of the experiments is to determine, for the first time, the efficiency of grammar-based approaches for MQTT. We've developed and evaluated a network-based fuzzer called `MQTTGRAM` that incorporates the grammar-based approach presented in Section IV. The experiments are executed until a stopping criterion has been satisfied. For this paper, we consider two stopping criterion:

- **Execution Time:** The experiments were performed for 3 minutes and 30 minutes;
- **Number of Packets:** Since input coverage needs to be ensured when developing grammar-based approaches [28], we've conducted experiments until the fuzzer and the MQTT broker exchange 500 and 8000 packets.

A. Experiment Setup

Each experiment was repeated 100 times. Fig. 3 presents the experiment setup. The operating system in both the Host Machine and Virtual Machine was Ubuntu 16.04.6 LTS x86_64. The specifications of these machines are: Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz, 16 GB RAM (Host Machine); 1 GB RAM (Virtual Machine).

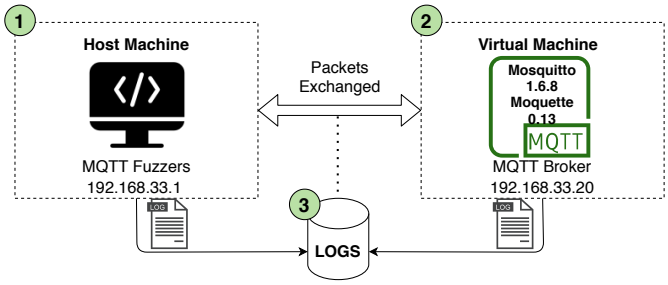
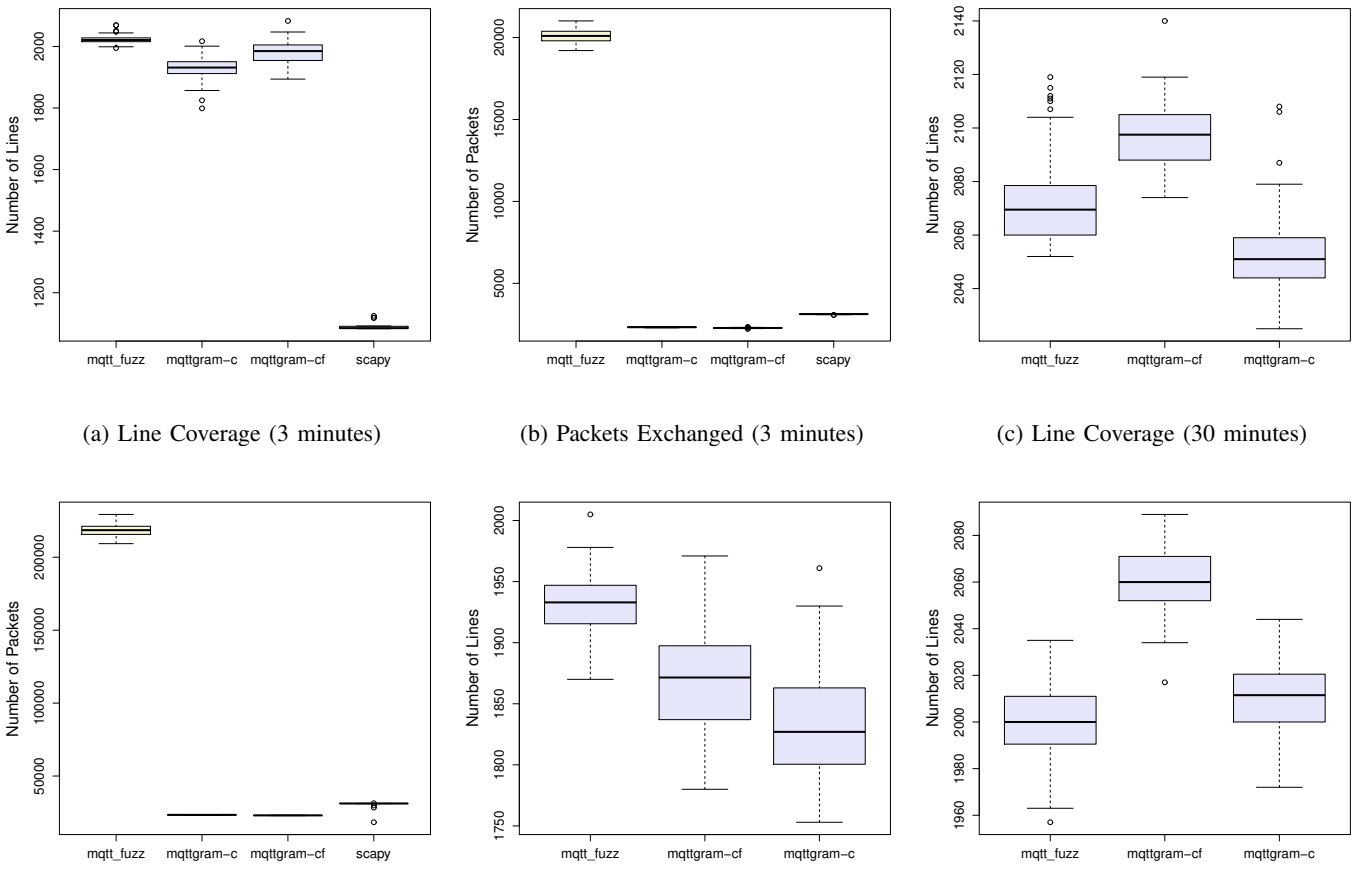


Fig. 3: Experiment Setup

1) Host Machine: The MQTT fuzzers run in the host machine. We compare `MQTTGRAM` with a naive fuzzer (Scapy's protocol fuzzer [15]), a mutation-based fuzzer (FSecure's `mqtt_fuzz` [10]), and a generation-based fuzzer (Sochor et al. [14]). Scapy's protocol fuzzer and FSecure's `mqtt_fuzz` are among the most popular naive and mutation-based fuzzers, respectively, and thus were chosen for the experiments. The fuzzer by Sochor et al. was chosen for being recent. In addition to these two fuzzers, two variants of `MQTTGRAM` were evaluated. The first variant, `mqttgram-c`, sends MQTT packets to the broker in correct order, and packets sent to the broker have the same probability of being generated. The second variant, `mqttgram-cf`, sends MQTT packets in correct order, and occasionally out of order. Moreover, *PUBLISH*, *SUBSCRIBE*, and *UNSUBSCRIBE* packets have a 25% chance of being generated; *PINGREQ* has a 15% chance, and *DISCONNECT* has a 10% chance. Considering these slight modifications, `mqttgram-c` covers only correct functionalities, while `mqttgram-cf` covers both correct and failed functionalities. Moreover, `mqttgram-cf`'s configuration of sending packets out-of-order and preferring packets with topic fields is based on existing MQTT vulnerabilities.

2) Virtual Machine: The virtual machine executes the MQTT brokers. Since Mosquitto and Moquette are among the most popular MQTT brokers [11], [14], we've chosen them as the target systems. Specifically, Mosquitto 1.6.8 and Moquette 0.13 were selected because they provide support for code coverage analysis.

3) Logs: Two types of logs are generated at the end of the fuzzing campaign. The first type stores the following broker-related information: average CPU usage, average memory usage, and line coverage. Line coverage measures the number of statements covered during the fuzzing campaign. The second type of log is a PCAP file that contains the packets exchanged during the fuzzing campaign. Because of space limitations, we will present a comparison between the fuzzers in terms of line coverage and packets exchanged.



(a) Line Coverage (3 minutes)

(b) Packets Exchanged (3 minutes)

(c) Line Coverage (30 minutes)

(d) Packets Exchanged (30 minutes)

(e) Line Coverage (500 packets)

(f) Line Coverage (8000 packets)

Fig. 4: Experiment results (Mosquitto)

B. Results

1) Mosquitto: Fig. 4a presents the line coverage reached by all fuzzers for 3 minutes. Scapy's protocol fuzzer has the lowest code coverage, reaching at most 1,125 lines and at the very least 1,083 lines. `mqttgram-c` manages to have higher code coverage than Scapy's fuzzer, reaching at most 2,017 lines and at least 1,799. Thus, `mqttgram-c` performs at most 59.91% better because it has knowledge of MQTT's specifications. FSecure's fuzzer `mqtt_fuzz` manages to reach at most 2,069, performing 83.91% and 2.58% better than Scapy's fuzzer and `mqttgram-c` respectively. `mqttgram-cf` fared slightly better, achieving a coverage increase of at most 0.68% and 85.15% compared to `mqtt_fuzz` and Scapy's fuzzer respectively.

`mqtt_fuzz` manages to have better line coverage overall because it exchanges a considerable amount of packets with the broker, as shown in Fig. 4b. If a fuzzer exchanges less packets with the broker, then it means it is more efficient. On average, `mqtt_fuzz` and the broker exchange 20,111 packets, approximately 544.79%, 765.36%, and 783.61% more than Scapy's fuzzer, `mqttgram-c`, and `mqttgram-cf` respectively. Since `mqtt_fuzz` sends more packets, there is a higher probability that more code paths are traversed during a short amount of time (3 minutes). However, `mqttgram-cf` is the most efficient fuzzer, reaching 2,083 lines and exchanging only 2,258 packets. `mqttgram-cf` has higher code coverage because it considers correct and failed functionalities of MQTT. Moreover, `mqttgram-cf` triggers more functionalities of MQTT because it has knowledge of its specifications. Scapy's fuzzer manages to send more packets than `mqttgram-c` and `mqttgram-cf` during the same time. This is because Scapy's fuzzer generates random packets without considering the MQTT standard. For simplicity purposes, the following graphs do not present the results of Scapy's fuzzer since it achieves low performance in terms of line coverage.

Although `mqtt_fuzz` has the highest line coverage overall for 3 minutes, a different outcome occurs for 30 minutes (Fig. 4c). Among all MQTT fuzzers, `mqttgram-cf` has the highest line coverage, executing 2,140 lines at most, which is approximately up to 90%, 1.51%, and 0.99% more than Scapy's fuzzer, `mqttgram-c`, and `mqtt_fuzz` respectively.

TABLE I: Line Coverage of Moquette reached by MQTT Fuzzers in 30 minutes (Results by Sochor et al. were copied directly from [14])

Source Code (Directory)	Fuzzed packets not based on existing vulnerabilities				Fuzzed packets based on existing vulnerabilities	
	Scapy	mqtt_fuzz	Sochor et al. 2020	mqttgram-c	Sochor et al. 2020	mqttgram-cf
broker	21%	64.2%	57.8%	**66.34%**	58%	**67.57%**
broker.security	12%	13.02%	11.3%	**15%**	12.8%	**15%**
broker.subscriptions	8%	**73.86%**	59.0%	73.52%	63.9%	**73.92%**
persistence	3%	9%	**9.1%**	9%	**9.1%**	9%
interception	11%	28.04%	**32.3%**	30%	**32.3%**	30%
broker.config	**51%**	**51%**	49.3%	**51%**	49.3%	**51%**
broker.metrics	44%	71.78%	**77.5%**	73%	**77.5%**	73%
Logging	43%	43%	**62.5%**	43%	**62.5%**	43%
Total	19.21%	55.24%	50.0%	**56.76%**	51.4%	**57.24%**

It is worth noting that on average `mqttgram-cf` and the broker exchange the lowest number of packets (Fig. 4d), approximately 22,995; thus confirming the efficiency and effectiveness of our grammar-based approach. Over time, both variants of `MQTTGRAM` have higher code coverage because they possess knowledge of MQTT's packet format, thus triggering all possible behaviors. The remaining fuzzers have an average number of packets exchanged with the broker as follows: `mqtt_fuzz`: 218,557; `mqttgram-c`: 23,364; and Scapy: 30,993.

Fig. 4e presents the line coverage reached by the MQTT fuzzers when exchanging exactly 500 packets with the broker. `mqtt_fuzz` has the highest line coverage, reaching 2,005. `mqttgram-c`, `mqttgram-cf`, and Scapy reach at most 1,961; 1,971; and 1,118, respectively. Although variants of `MQTTGRAM` have lower coverage than their counterparts, exchanging 8000 packets has a different outcome (Fig. 4f). On average, `mqttgram-c` and `mqttgram-cf` reach 2,009 and 2,060 lines respectively. The results of both variants are higher than their counterparts, confirming that they cover more functionality of MQTT quickly and efficiently.

2) Moquette: Table I presents the line coverage reached by all MQTT fuzzers during 30 minutes. In addition to all of the fuzzers from previous experiments, we also compare `MQTTGRAM` with the fuzzer proposed by Sochor et al. [14]. Similar to our approach, Sochor et al. also developed two variants of their fuzzer. The first variant generates packets that are not based on existing vulnerabilities of MQTT, while the second variant generates packets based on a list of twenty-five cyberattacks. Since Sochor et al. measure code coverage for each source code directory of Moquette, we conducted our experiments in the same manner.

It is worth noting that the fuzzer proposed by Sochor et al. is not publicly available. Thus, we were unable to perform experiments with their fuzzer, and had to rely on results from their paper for comparison purposes. Moreover, results by Sochor et al. are based on a single thirty-minute fuzzing campaign, while the results of the other fuzzers in Table I are based on the average of a hundred trials.

Table I highlights in bold the highest code coverage for each Moquette directory and input generation. Both variants of `MQTTGRAM` have the highest code coverage in the `broker.security` directory. Although `mqttgram-c` is geared towards correct functionality of MQTT, it manages to outperform the vulnerability-oriented fuzzer by Sochor et al. Moreover, although Sochor et al. generated fuzzed packets based on twenty-five cyberattacks, `mqttgram-cf` outperforms their approach by considering only two patterns of existing MQTT vulnerabilities. This confirms that grammar-based approaches for MQTT are not only efficient, but have better chances of triggering more flaws in MQTT.

It is worth noting that both variants of `MQTTGRAM` have the overall highest code coverage. These variants exchange 7x - 9x fewer packets with the broker than `mqtt_fuzz`. We are unaware of how many packets were exchanged in the experiments conducted by Sochor et al., since they do not specify that information in their paper.

Similar to the experiments with Mosquitto, `mqtt_fuzz` has the highest code coverage in short-lived fuzzing executions.

3) Takeaway from experiments: A summary of the results is as follows.

- `MQTTGRAM` has the highest coverage increase (5.81% - 6.43% on Mosquitto and 6.93% - 13.76% on Moquette) from 3 to 30 minutes;
- `mqttgram-cf` reaches the highest code coverage in 3 minutes and 30 minutes; and
- `MQTTGRAM` exchanges 7x - 9x fewer packets than `mqtt_fuzz`.

There are three takeaways from the experiments. First, mutation-based approaches, such as `mqtt_fuzz`, are more suitable for short-lived fuzzing executions. Second, MQTT fuzzers must possess knowledge of the protocol standard to test brokers effectively over longer sessions. Third, `mqttgram-cf` demonstrated the possibility of using different grammar-based approaches to achieve better results.

It is worth noting that, while building our MQTT grammar, we've noticed that a popular MQTT library was lacking functionality or features described in the MQTT 3.1.1 standard. We've improved that library as an additional contribution of this research [29].

VI. Conclusions

Over the last few years, researchers have found fundamental problems that expose MQTT to cyberattacks. Testing MQTT with effective mechanisms such as fuzzing can mitigate these issues. However, existing fuzzing frameworks lack sufficient knowledge of MQTT's specifications, requiring a considerable amount of network packets to cover all of its features or functionalities, thus being inefficient. Several research papers have expressed interest in grammar-based approaches, however defining a grammar is usually considered time-consuming and complex. This paper aims to address limitations of existing MQTT fuzzers by presenting a grammar-based approach that aims for efficiency and high code coverage. Based on our experiments, the approach implemented in `MQTTGRAM`, is considerably more efficient and has higher code coverage than existing MQTT fuzzers. We expect this paper to motivate developers to build more efficient fuzzing frameworks for MQTT and IoT protocols in general. Future works will focus on using the grammar to generate packets based on more existing vulnerabilities.

Acknowledgment

We would like to thank Andreas Zeller from CISPA, and members of the ISSTA 2020 Doctoral Symposium Program Committee for their suggestions to improve our research. We would also like to thank CAPES for funding this research. This paper would not have been possible without their support. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9. It is also part of FAPESP proc. 18/22979-2 and FAPESP proc. 18/23098-0.

References

[1] J. Ploennigs, J. Cohn, and A. Stanford-Clark, "The Future of IoT," *IEEE Internet of Things Magazine*, vol. 1, no. 1, pp. 28–33, 2018.

[2] F. Liljedahl, "Exploring the Possibilities of Robustness Testing CoAP Implementations Using Evolutionary Fuzzing," Master's dissertation, KTH Royal Institute of Technology, 2019.

[3] M. B. Yassein, M. Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi, "Internet of Things: Survey and Open Issues of MQTT Protocol," in *Proceedings of the ICEMIS*, 2017, pp. 1–6.

[4] K. Karamitsios and T. Orphanoudakis, "Efficient IoT Data Aggregation for Connected Health Applications," in *Proceedings of the IEEE ISCC*, 2017, pp. 1182–1185.

[5] D. R. C. Silva, G. M. B. Oliveira, I. Silva, P. Ferrari, and E. Sisinni, "Latency Evaluation for MQTT and WebSocket Protocols: an Industry 4.0 Perspective," in *Proceedings of the IEEE ISCC*, 2018, pp. 01 233–01 238.

[6] P. Anantharaman, M. Locasto, G. F. Ciocarlie, and U. Lindqvist, "Building Hardened Internet-of-Things Clients with Language-Theoretic Security," in *Proceedings of the IEEE Security and Privacy Workshops*, 2017, pp. 120–126.

[7] P. Bellavista, L. Foschini, N. Ghiselli, and A. Reale, "MQTT-based Middleware for Container Support in Fog Computing Environments," in *Proceedings of the IEEE ISCC*, 2019, pp. 1–7.

[8] 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 *Proceedings of the NDSS*, 2018.

[9] L. G. Araujo Rodriguez and D. Macêdo Batista, "Program-Aware Fuzzing for MQTT Applications," in *Proceedings of the ACM ISSTA*, 2020, p. 582–586.

[10] F-Secure Corporation, "A Simple Fuzzer for the MQTT Protocol," https://github.com/F-Secure/mqtt_fuzz, 2015, [Online; accessed 16-September-2019].

[11] S. Hernández Ramos, M. T. Villalba, and R. Lacuesta, "MQTT Security: A Novel Fuzzing Approach," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–11, 2018.

[12] Eclipse Foundation, "Eclipse IoT-Testware," https://iottestware.readthedocs.io/en/development/smart_fuzzer.html, 2018, [Online; accessed 17-October-2019].

[13] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, and R. Zitouni, "Fuzzing Attacks for Vulnerability Discovery within MQTT Protocol," in *Proceedings of the IWCMC*, 2020, pp. 420–425.

[14] H. Sochor, F. Ferrarotti, and R. Ramler, "Automated Security Test Generation for MQTT Using Attack Patterns," in *Proceedings of the ARES*, 2020.

[15] Philippe Biondi and the Scapy Community, "Usage – Scapy 2.4.5 documentation – Fuzzing," https://scapy.readthedocs.io/en/latest/usage.html#fuzzing, 2021, [Online; accessed 06-June-2021].

[16] J. Joshi, V. Rajapriya, S. R. Rahul, P. Kumar, S. Polepally, R. Samineni, and D. G. K. Tej, "Performance Enhancement and IoT Based Monitoring for Smart Home," in *Proceedings of the ICOIN*, 2017, pp. 468–473.

[17] Eclipse Foundation, "IoT Developer Survey," https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf, 2019, [Online; accessed 15-June-2019].

[18] S. Katsikeas, K. Fysarakis, A. Miaoudakis, A. Van Bemten, I. Askoxylakis, I. Papaefstathiou, and A. Plemenos, "Lightweight Secure Industrial IoT Communications via the MQ Telemetry Transport Protocol," in *Proceedings of the IEEE ISCC*, 2017, pp. 1193–1200.

[19] A. Mehta, N. Hantehzadeh, V. K. Gurbani, T. K. Ho, and F. Sander, "On Using Multiple Classifier Systems for Session Initiation Protocol (SIP) Anomaly Detection," in *Proceedings of the IEEE ICC*, 2012, pp. 1101–1106.

[20] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The Art, Science, and Engineering of Fuzzing: A Survey," *IEEE Transactions on Software Engineering*, 2019.

[21] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a Survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[22] J.-Z. Luo, C. Shan, J. Cai, and Y. Liu, "IoT Application-Layer Protocol Vulnerability Detection using Reverse Engineering," *Symmetry*, vol. 10, no. 11, p. 561, 2018.

[23] T. L. Munea, H. Lim, and T. Shon, "Network Protocol Fuzz Testing for Information Systems and Applications: a Survey and Taxonomy," *Multimedia Tools and Applications*, vol. 75, no. 22, pp. 14 745–14 757, 2016.

[24] Y. Chen, T. lan, and G. Venkataramani, "Exploring Effective Fuzzing Strategies to Analyze Communication Protocols," in *Proceedings of the ACM FEAST*, 2019, p. 17–23.

[25] T. Kitagawa, M. Hanaoka, and K. Kono, "AspFuzz: A State-Aware Protocol Fuzzer based on Application-Layer Protocols," in *Proceedings of the IEEE ISCC*, 2010, pp. 202–208.

[26] A. Banks, R. Gupta, and N. O. L. (editors), "MQTT Version 3.1.1 Plus Errata 01," http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, 2015, [Online; accessed 06-June-2021].

[27] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Fuzzing with Grammars," in *The Fuzzing Book*. Saarland University, 2019.

[28] N. Havrikov and A. Zeller, "Systematically Covering Input Structure," in *Proceedings of the IEEE ASE*, 2019, pp. 189–199.

[29] L. G. Araujo Rodriguez, "MQTTSubscribe now supports multiple topic subscriptions in the payload," https://github.com/secdev/scapy/pull/2759, 2020, [Online; accessed 06-June-2021].