Resource-Intensive Fuzzing for MQTT Brokers: State of the Art, Performance Evaluation, and Open Issues

Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista, *Member, IEEE*

***Abstract*—Pub/sub messaging is a promising design pattern that relies on a centralized component called the broker, which routes messages to different devices. Due to its prominent role, the broker must be capable of providing high quality services under heavy workload. In this letter, we evaluate the resource consumption of state-of-the-art fuzzing frameworks, thereby understanding the degree to which brokers are tested before deployment. Our results attest that only one framework shows the most promise for memory-intensive testing, outperforming its counterparts by more than 20%. We conclude this letter by highlighting shortcomings that prevent existing frameworks from consuming considerable system resources.**

***Index Terms*—MQTT, fuzzing, testing, publish-subscribe, smart cities, IoT.**

I. Introduction

THE INTERNET of Things (commonly abbreviated as IoT) is an adaptive, complex, and global network of physical devices such as sensors and actuators, which interact with one another through standard communication protocols. The evolution of IoT technologies and protocols has led to the emergence of *Smart Cities*, which are expected to play a pivotal role in the global economy [1]. The main goal of a smart city is to improve the lives of its citizens through the use of IoT technologies. Centralized pub/sub architectures have been adopted for several smart city applications, including transportation systems, traffic control, and healthcare services. In a centralized pub/sub architecture, data transmission and storage fall under the responsibility of a single entity called the *broker* [2], which forwards messages from publishers to subscribers.

A major disadvantage of centralized architectures, when compared to their distributed counterparts, is that the broker most often becomes a performance bottleneck. In the worst case scenario, a performance bottleneck leads to a Denial of Service (DoS), which has negative effects, especially in large-scale environments such as smart cities. For example, the lack of transport services is detrimental to citizens *and* city revenue; the absence of traffic control systems may increase the number of car accidents; and unavailable healthcare systems can delay urgent medical procedures or appointments.

DoS attacks can also be performed *intentionally* to either *flood* or *crash* the target system. Both of these outcomes usually occur by elevating the CPU and memory usage of the broker to such a degree that its functionality is affected. According to the MITRE Corporation [3], uncontrolled resource consumption has a high probability of leading to an exploit. In fact, resource-related vulnerabilities rank within the top 25 most dangerous software weaknesses [3]. Furthermore, resource exhaustion is one of the most common types of DoS attacks in IoT according to the literature [4].

The robustness of the broker therefore plays a *key* role in the success and reliability of smart city applications, which must be capable of providing high quality services under heavy workload. A DoS can be mitigated or avoided altogether if the robustness of the broker is tested *thoroughly* before deployment, thereby guaranteeing its reliability in real-world environments [5]. The robustness can be tested through several techniques, with the most common being *fuzz testing* [6], which involves sending random messages to a target system and then monitoring its behavior for potential weaknesses such as performance bottlenecks.

The MITRE Corporation lists *fuzzing* as one of the three main detection methods for resource-exhaustion problems [3]. However, in order to guarantee that the broker is robust and reliable enough against performance bottlenecks or issues in smart cities, a fuzz testing tool (or *fuzzer*) should perform CPU- and memory-intensive tests [7]. Despite the wide range of broker fuzzers at our disposal, there is a lack of information in the literature regarding their capabilities to perform resource-exhaustion attacks during the test run.

In this letter, we evaluate *how* existing fuzzing strategies impact the CPU and memory usage of the broker during testing. Although there have been several publications about the broker's performance while under heavy workload [8], none have evaluated the capabilities of existing fuzzing tools for *stress-testing* purposes. To the best of our knowledge, we are the *first* to investigate about the fuzzers' capabilities in regards to resource exhaustion. This letter also highlights questionable design choices that prevent existing fuzzers from consuming more system resources.

We focus on broker implementations and fuzzers of the Message Queuing Telemetry Transport (MQTT) protocol, which is the most widely-used IoT and pub/sub protocol for smart city applications [9]. Research studies have

Manuscript received 12 November 2022; revised 1 March 2023; accepted 27 March 2023. Date of publication 31 March 2023; date of current version 6 June 2023. This work was supported in part by CNPq under Grant 381249/2022-0; in part by the INCT of the Future Internet for Smart Cities funded by CNPq under Grant 465446/2014-0; in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brazil (CAPES)—Finance Code 001; and in part by FAPESP under Grant 14/50937-1, Grant 15/24485-9, and Grant 18/23098-0. The associate editor coordinating the review of this article and approving it for publication was K. T. Kim. *(Corresponding author: Luis Gustavo Araujo Rodriguez.)*

The authors are with the Department of Computer Science, University of São Paulo, São Paulo 05508-090, Brazil (e-mail: luisgar@ime.usp.br; batista@ime.usp.br).

Digital Object Identifier 10.1109/LNET.2023.3263556

demonstrated that IoT protocols such as MQTT are extremely vulnerable to resource-exhaustion attacks [10]. Since 2016, *at least* ten vulnerabilities related to memory management have been triggered in MQTT brokers. Examples of memory-management vulnerabilities are CVE-2016-10523, CVE-2017-2894, CVE-2017-7651, CVE-2018-17614, CVE-2018-19417, CVE-2018-19587, CVE-2018-5879, CVE-2018-8531, and CVE-2019-12951. All of the aforementioned vulnerabilities were triggered by sending malicious or malformed packets to the broker, which is a common practice among cyberattackers.

This letter is organized as follows. Section II explains the evolution of MQTT broker fuzzing over time. Section III presents the evaluation criteria for MQTT fuzzers across the literature. Section IV describes our experiment setup and the performance achieved by MQTT fuzzers in terms of the broker's CPU and memory usage. Section V discusses the experiment results. Finally, Section VI marks the conclusions of this letter, and proposes future work.

II. Evolution of MQTT Broker Fuzzing

MQTT fuzzing dates back to more than a decade ago [11], when IoT and smart cities began to gain prominence. Scapy's `fuzz()` function [12], `mqtt_fuzz` [13], and `Defensics` [14] were among the first MQTT fuzzers to emerge during this period, each having its own distinct input generation mechanism. `mqtt_fuzz` is classified as a *mutation-based* fuzzer, meaning it introduces small changes to existing inputs. `Defensics` and Scapy's `fuzz()` function are *generation-based* fuzzers, meaning inputs are generated from scratch. The main difference between the generation-based fuzzers lies in their level of knowledge of MQTT. For example, Scapy's `fuzz()` function lacks a *complete* understanding of the message syntax. Despite using simple testing strategies, the first three fuzzers helped set the foundation for subsequent techniques.

For the next few years, research on MQTT fuzzing lacked considerably. It was only until the release of the `Polymorph` framework [15] that MQTT fuzzing became renown as a necessary research field to further improve the robustness and reliability of brokers in smart cities. The `Polymorph` framework aims to mitigate the limitations of its predecessors by using a proxy-based approach. The success and popularity of the `Polymorph` framework led to the emergence of seven MQTT fuzzers from 2020 to 2022.

`Multifuzz` [16] and `AFLNet-MQTT` [17] use genetic algorithms to generate coverage-increasing test cases. `CyberExploit` [18] generates attack scenarios to detect vulnerabilities in MQTT brokers. Sochor et al. 2020 [19] integrate attack patterns based on known vulnerabilities into their fuzzer to outperform previous approaches. The automata learning algorithm by Aichernig et al. 2021 [20] automatically understands the different states of the broker. `MQTTGRAM` and its variants [21] use a grammar to achieve high code and state coverage. `FUME` [22] carefully chooses between mutation- and generation-guided fuzzing to achieve optimal results.

It is worth noting that most of the aforementioned broker fuzzers do not receive any feedback whatsoever about the quality of the test cases. The sole exceptions are `Multifuzz` and `AFLNet-MQTT`, which receive only coverage-related feedback during testing.

III. Evaluation Criteria for Existing MQTT Fuzzers

Despite their differences, broker fuzzers have proven to be successful for their intended purpose. The success or effectiveness of each fuzzing technique is measured differently across research studies. However, resource-related metrics are barely used in the literature. To the best of our knowledge, only Hernández Ramos et al. [15] evaluated their fuzzer by considering the CPU consumption of the broker.

A *resource-guided* fuzzer receives and analyzes resource-related feedback about the system under test while fuzzing. Although none of the aforementioned broker fuzzers are designed specifically for resource-intensive purposes, it is important to evaluate their stress-testing capabilities because of three reasons. First, open-source resource-guided fuzzers designed specifically for MQTT brokers are nonexistent. Developers have two choices: (1) develop their own resource-guided fuzzer from scratch; or (2) use and extend an existing framework to perform stress testing. Developers have thus far preferred the latter, as evidenced by the research studies conducted by Fehrenbach 2018 [23] and Morelli et al. 2021 [24], which propose to use or extend an open-source MQTT fuzzer to perform more complex resource-exhaustion attacks. Second, the tools proposed by Fehrenbach 2018 [23] and Morelli et al. 2021 [24] are proprietary, meaning developers have no choice but to select one of the state-of-the art fuzzing frameworks evaluated in our experiments. As such, it is important for developers to be aware of the capabilities and shortcomings of each open-source fuzzer in order to select the most appropriate for testing purposes. Third, although there exists open-source resource-guided fuzzers such as `MEMLock` [25] or `ResFuz` [26], they are incompatible with network-based target systems such as brokers, meaning developers are more likely to use and extend the open-source fuzzers evaluated in our experiments.

IV. Performance Evaluation

Fig. 1 presents the architecture of our testbed, whose main purpose is to monitor CPU and memory usage of the broker while fuzzing.

For this letter, we evaluated six open source MQTT fuzzers [12], [13], [18], [20], [21], [22]. We chose `Mosquitto 1.6.8` (released on 11/28/2019) as the target broker because of its considerable popularity in the literature [27]. Mosquitto and the MQTT fuzzers were evaluated using default settings. The CPU and memory usage of Mosquitto are monitored throughout the fuzzing campaigns. The more CPU- and memory-intensive inputs are sent to the broker, the higher the probability of finding resource-exhaustion bugs. Fuzzing campaigns are repeated 100 times in order to calculate the average and standard deviation for

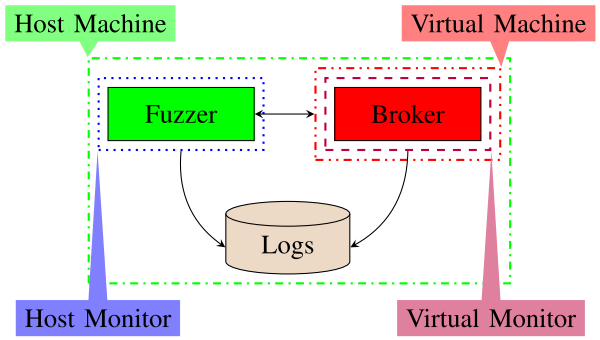

Fig. 1. Testbed (Both the host and virtual machines have an Intel i7-2700K CPU @ 3.50GHz, and run the Ubuntu 16.04.6 LTS x86_64 operating system. The host machine has 16 GB of RAM, whereas its virtual counterpart has only 1 GB.).

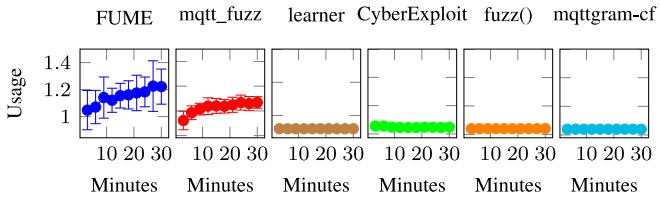

Fig. 2. Average CPU Usage of Mosquitto during 30 minutes.

each stopping criterion. We conducted two separate experiments, each satisfying a different stopping criterion. For the first experiment, the fuzzers are executed until a certain time has elapsed (30 minutes). For the second experiment, the fuzzers are executed until a specific number of packets has been exchanged with the broker (8000). Resource usage is measured after every three minutes and after 500, 1000, 2000, 4000, 6000, and 8000 packets. We refer to fuzzers either by their designated name (if available) or main characteristic. For simplicity purposes, we will refer to the fuzzer by Aichernig et al. 2021 [20] as `learner` in the remainder of this letter due to lacking a formal name.

Fig. 2 presents the CPU usage of Mosquitto when fuzzing for 30 minutes. It is worth noting that the figures in this section present the results of only one variant of `MQTTGRAM` (`mqttgram-cf`) because it performs nearly identical to `mqttgram-c`.

Among all MQTT fuzzers, `FUME` consumes the most CPU resources on Mosquitto, followed by `mqtt_fuzz` and `learner` respectively. The CPU usage by `FUME` ranges from 1.05% to 1.23%, and peaks at 2.40%. `mqtt_fuzz` consumes at most 1.50% of the CPU, and normal usage ranges from 0.92% to 1.07%. `learner` uses up to 0.40% of the CPU, and average consumption is fairly consistent throughout 30 minutes, remaining mostly at 0.30%. Peak CPU usage when testing with `fuzz()` and `CyberExploit` is slightly lower than its counterparts, reaching at most 0.30% and 0.20% respectively. Average CPU usage by both fuzzers is mostly consistent throughout 30 minutes, ranging from 0.10% to 0.14%. Finally, both variants of `MQTTGRAM` perform mostly the same, with `mqttgram-c` and `mqttgram-cf` using at most 0.10% and 0.20% respectively. Average CPU usage by

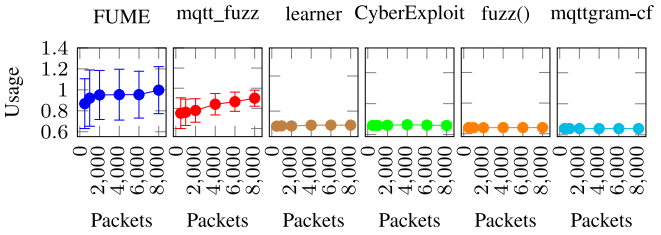

Fig. 3. Average CPU Usage of Mosquitto when exchanging 8000 packets.

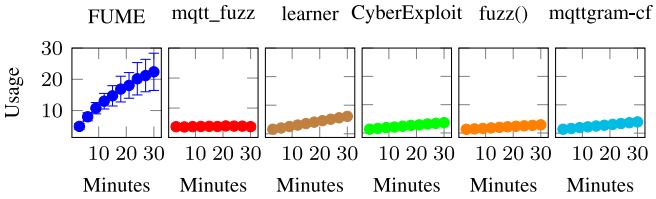

Fig. 4. Average Memory Usage of Mosquitto during 30 minutes.

both grammar-based approaches remains at 0.10% throughout most of the fuzzing campaign.

Based on our time-based experiments, all MQTT fuzzers are incapable of creating CPU-intensive scenarios that test the broker's performance under heavy workload. The results of the packet-based experiments, shown in Fig. 3, further confirm this limitation.

Fig. 4 presents the memory usage of Mosquitto when fuzzing for 30 minutes.

`FUME` peaks Mosquitto's memory usage at 43.50%, outperforming all other state-of-the-art fuzzing techniques *combined*. The memory usage by `learner` peaks at 6.10%, followed by `mqtt_fuzz` at 5.80%, and `mqttgram-cf` at 4.90%, respectively. The memory usage by `FUME` ranges from 4.93% to 22.39%, increasing at a higher rate than those of its counterparts. The percentage increase may be directly linked to the network traffic during testing. For example, `FUME` exchanges 544,082 packets with Mosquitto in 30 minutes, outperforming all other fuzzers by a wide margin. In fact, `mqtt_fuzz` generates over 50% less network traffic (218,557) than `FUME` in the same time frame. Similarly, `mqtt_fuzz` exchanges over 50% more packets with the broker than `learner` (75,855). Despite outperforming `learner`, `mqtt_fuzz` consumes only from 3.71% to 4.04% of memory on average. In contrast, `learner` increases Mosquitto's memory usage from 1.59% to 6.08%. According to our results, a broker's memory consumption seems to increase at a higher rate when testing MQTT 5.0, as is the case of `FUME` and `learner`. The latter fuzzer consumes less memory at the beginning of the fuzzing campaign because its interactions with the broker are mostly short-lived, consisting solely of connection and disconnection requests. However, `learner` slightly covers more functionality over time by performing publish and subscribe requests, increasing Mosquitto's memory usage as a result.

Despite covering more functionality than `learner`, `mqttgram-cf` increases Mosquitto's memory usage only from 1.39% to 3.97% on average. Both `mqttgram-c` and `CyberExploit` consume at most 3.70% of Mosquitto's

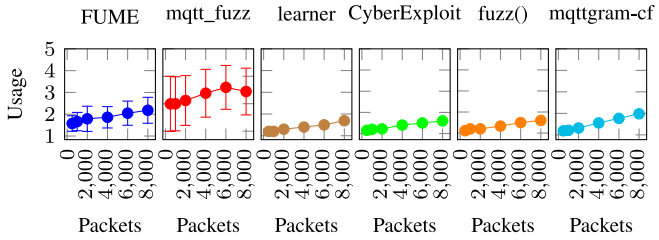

Fig. 5. Average Memory Usage of Mosquitto when exchanging 8000 packets.

memory, and their average usage is nearly identical. The former's memory usage ranges from 1.32% to 3.57%, whereas the latter's from 1.33% to 3.65%. Among all fuzzers for MQTT brokers, `fuzz()` underperformed the most in terms of Mosquitto's memory usage, consuming at most 2.90%. Average memory usage by `fuzz()` ranges from 1.30% to 2.89% throughout 30 minutes. Despite covering less functionality than most of its counterparts, `learner` consumes more memory on Mosquitto than most fuzzers at the end of 30 minutes, except for `FUME`. More specifically, `learner` outperforms `mqtt_fuzz`, despite the latter exchanging considerably more packets with the broker than the former. In fact, all fuzzers compatible with MQTT 5.0 have the largest percent increase from 3 to 30 minutes. For example, `mqtt_fuzz` increases memory usage by up to 0.33% from 3 to 30 minutes, whereas `learner` increases memory usage by up to 4.50% in the same time frame.

Similarly, `fuzz()` uses the least amount of memory when exchanging up to 8000 packets with Mosquitto, as shown in Fig. 5.

In fact, fuzzers that cover the least amount of functionality seem to consume fewer memory resources than their counterparts. For example, `fuzz()` and `CyberExploit` perform roughly the same, consuming at most 1.60% of Mosquitto's memory. The average memory usage by both fuzzers ranges from 1.11% to 1.60%. Despite `learner` ranking second-best in terms of memory usage at the end of 30 minutes, it ranks as the third lowest when exchanging up to 8000 packets, consuming at most 1.70% of memory. Average memory usage by `learner` ranges from 1.20% to 1.69%. `mqttgram-c` and `mqttgram-cf` perform slightly better than the aforementioned approaches, using at most 2.10% and 2.20% of memory respectively. More specifically, `mqttgram-cf` performs better than `mqttgram-c` overall, consuming from 1.20% to 2.13% of memory on average. Memory usage by `mqttgram-c` ranges from 1.19% to 1.98%. Among all fuzzers, `mqttgram-cf` has the largest percent increase (0.93%) when exchanging up to 8000 packets with Mosquitto, followed by `mqttgram-c` (0.78%). However, it is worth noting that `learner` outperforms both `mqttgram-cf` and `mqttgram-c` at the end of 30 minutes because it exchanges more packets with the broker than the aforementioned approaches, hence the importance of evaluating fuzzers based on the number of packets. Among all fuzzers, `mqtt_fuzz` has the highest average memory usage, which ranges from 2.99% to 3.60%. The minimum value (2.99%) is higher than all standout performances of nearly every fuzzer except for `FUME`, which consumes at most 4.80% of memory resources, whereas `mqtt_fuzz` uses at most 4.60%. Although `FUME` has the highest standout performance (4.80%) among all fuzzers, its average memory usage ranges from 1.57% to 2.18%, meaning it is slightly outperformed by `mqtt_fuzz`.

V. Discussion

Although fuzzers underperformed in terms of the brokers' CPU consumption, their results provide insight into two ineffective strategies to avoid for future work. First, sending a considerable amount of packets, as is the case of `mqtt_fuzz` and `FUME`, does not increase the CPU usage considerably. Second, performing publish requests with long topic names, as is the case of `FUME` and `CyberExploit`, also consumes few CPU resources on Mosquitto. Future work should be directed towards improving these strategies. For example, running multiple instances of fuzzers simultaneously could potentially increase the CPU usage of the broker. Future research could analyze the effectiveness of fuzzers when running multiple instances, exploring strategies that increase CPU resources and hinder the performance of the broker.

There are also several opportunities for future work considering the lessons learned from Mosquitto's memory usage while fuzzing. First, performing multiple requests simultaneously and successively seems to be an effective strategy to consume more of the broker's memory. In this letter, we refer to *performing multiple requests simultaneously* as sending multiple requests in a single packet to the broker, whereas *performing multiple requests successively* refers to sending multiple packets one after the other without waiting for a response from the broker. Future work should attempt to develop fuzzers that apply both approaches to increase the broker's workload.

Second, storing significant amount of user data on the broker should be performed in order to increase memory usage during testing. This task can be accomplished by disabling clean sessions *and* enabling retained messages when performing connection and publish requests respectively. Storing user sessions and topic messages can increase the broker's memory usage because multiple messages will have to be published by the broker when a client resumes communications or subscribes to a given topic. Furthermore, topic filters with wildcard characters, such as # or +, should be used to receive multiple publish messages from the broker.

Third, MQTT 5.0 sessions are more memory intensive than their older counterparts. This is evidenced by the following three observations. First, `learner` outperforms `CyberExploit` in terms of memory usage despite generating shorter topic names for both publish and subscribe requests. Second, `learner` exchanges fewer packets with the broker than `mqtt_fuzz`, but still manages to use more memory. Third, in contrast to `mqtt_fuzz`, `learner` stores neither user- nor subscription-related information in the broker. The reason for the outperformance of `learner` stems from the fact that MQTT 5.0 is a substantial upgrade over its older siblings, supporting a wide variety of features such as user

properties or subscription options. Thus, most MQTT 5.0 packets possess more fields, meaning the broker must store additional subscription-related information. For example, SUBSCRIBE packets in MQTT 5.0 contain more fields for configuration options, such as scheduled message transmissions, all of which are stored on the broker. User sessions are therefore much more memory consuming, which explains why 5.x compatible fuzzers have the largest percent increase over time.

Our memory consumption analysis of MQTT fuzzers also reveals different performance rankings for each type of stopping criterion. For example, `FUME` consumed the most memory at the end of 30 minutes, followed by `learner` and `mqtt_fuzz` respectively. However, when exchanging the same number of packets with Mosquitto, `mqtt_fuzz` ranks as the most memory-consuming fuzzer on average, whereas `FUME` and `learner` fell to second and third place respectively. The different performance rankings may indicate that certain strategies are more effective than others in terms of memory consumption. In fact, when fuzzers generate the same amount of network traffic (8000 packets), their limitations in terms of memory consumption become more noticeable. Considering the research findings presented in this letter, we strongly recommend developers to incorporate resource-exhaustion techniques into their MQTT fuzzers.

VI. Conclusion

MQTT brokers are currently being deployed to large-scale scenarios such as smart cities. In that regard, the broker should be robust enough to handle requests from its citizens. Several fuzzers are made available to developers for testing-purposes. However, developers are unaware of the fuzzers' shortcomings in terms of resource-exhaustive testing. This letter presented a performance evaluation of most open-source fuzzers in the literature. Our results confirm that current fuzzers are incapable of intensively testing both the broker's CPU and memory. Among existing fuzzers, `FUME` shows the most promise for resource-intensive testing, peaking the memory usage of the broker at 43.5%. The main and common drawback of existing fuzzers is their lack of semantic test cases for the broker's pub/sub functionality. Developers can use the guidelines presented in this letter to either improve existing fuzzers or incorporate them into their own testing tool. For future work, we plan to develop a simple traffic generator in order to provide baseline results against which to compare the performance of existing and subsequent fuzzers. We also plan to develop a resource-intensive fuzzer considering the lessons learned from this letter. For future work, we will also conduct experiments with other broker implementations such as Moquette, HiveMQ, VerneMQ, and RabbitMQ.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.

[2] F. T. Johnsen, L. Landmark, M. Hauge, E. Larsen, and Ø. Kure, "Publish/subscribe versus a content-based approach for information dissemination," in *Proc. IEEE MILCOM*, 2018, pp. 1–9.

[3] "CWE-400 uncontrolled resource consumption." MITRE. 2006. Accessed: Sep. 27, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/400.html

[4] H. Mrabet, S. Belguith, A. Alhomoud, and A. Jemai, "A survey of IoT security based on a layered architecture of sensing and data analysis," *Sensors*, vol. 20, no. 13, p. 3625, 2020.

[5] D. L. de Oliveira, A. F. D. S. Veloso, J. V. V. Sobral, R. A. L. Rabêlo, J. J. P. C. Rodrigues, and P. Solic, "Performance evaluation of MQTT brokers in the Internet of Things for smart cities," in *Proc. 4th SpliTech*, 2019, pp. 1–6.

[6] V. J. Manè et al., "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[7] C. Săndescu, O. Grigorescu, R. Rughiniş, R. Deaconescu, and M. Calin, "Why IoT security is failing. The need of a test driven security approach," in *Proc. 17th RoEduNet*, 2018, pp. 1–6.

[8] B. Mishra, B. Mishra, and A. Kertesz, "Stress-testing MQTT brokers: A comparative analysis of performance measurements," *Energies*, vol. 14, no. 18, p. 5817, 2021.

[9] "The eclipse foundation releases 2022 IoT & edge developer survey results." Eclipse Foundation. 2022. Accessed: Oct. 12, 2022. [Online]. Available: https://newsroom.eclipse.org/news/announcements/eclipse-foundation-releases-2022-iot-edge-developer-survey-results%C2%A0

[10] J. Y. Kim, R. Holz, W. Hu, and S. Jha, "Automated analysis of secure Internet of Things protocols," in *Proc. ACSAC*, 2017, pp. 238–249.

[11] L. G. A. Rodriguez and D. M. Batista, "Program-aware fuzzing for MQTT applications," in *Proc. 29th ACM ISSTA*, 2020, pp. 582–586.

[12] P. Biondi and the Scapy Community. "Usage—Scapy 2.4.5 documentation—Fuzzing." 2021. Accessed: Sep. 29, 2021. [Online]. Available: https://scapy.readthedocs.io/en/latest/usage.html#fuzzing

[13] "A simple fuzzer for the MQTT protocol." F-Secure Corporation. 2015. Accessed: Sep. 16, 2019. [Online]. Available: https://github.com/F-Secure/mqtt_fuzz

[14] "Defensics fuzz testing." Synopsis. 2021. Accessed: Sep. 29, 2021. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html

[15] S. H. Ramos, M. T. Villalba, and R. Lacuesta, "MQTT security: A novel fuzzing approach," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–11, Feb. 2018.

[16] Y. Zeng et al., "MultiFuzz: A coverage-based multiparty-protocol fuzzer for IoT publish/subscribe protocols," *Sensors*, vol. 20, no. 18, p. 5194, 2020.

[17] S. S. Galiveeti and P. Malae. "MQTT fuzzing using AFLNET." 2020. Accessed: Mar. 13, 2022. [Online]. Available: https://github.com/SuhithaG/MQTT-fuzzing-using-AFLNET

[18] G. Casteur et al., "Fuzzing attacks for vulnerability discovery within MQTT protocol," in *Proc. IWCMC*, 2020, pp. 420–425.

[19] H. Sochor, F. Ferrarotti, and R. Ramler, "Automated security test generation for MQTT using attack patterns," in *Proc. ARES*, 2020, pp. 1–9.

[20] B. K. Aichernig, E. Muškardin, and A. Pferscher, "Learning-based fuzzing of IoT message brokers," in *Proc. 14th IEEE ICST*, 2021, pp. 47–58.

[21] L. G. A. Rodriguez and D. M. Batista, "Towards improving fuzzer efficiency for the MQTT protocol," in *Proc. IEEE ISCC*, 2021, pp. 1–7.

[22] B. Pearson, Y. Zhang, C. Zou, and X. Fu, "FUME: Fuzzing message queuing telemetry transport brokers," in *Proc. IEEE INFOCOM*, 2022, pp. 1699–1708.

[23] P. Fehrenbach. "Messaging queues in the IoT under pressure." 2017. Accessed: Jan. 11, 2023. [Online]. Available: https://blog.it-securityguard.com/wp-content/uploads/2017/10/IOT_Mosquitto_Pfehrenbach.pdf

[24] U. Morelli, I. Vaccari, S. Ranise, and E. Cambiaso, "DoS attacks in available MQTT implementations: Investigating the impact on brokers and devices, and supported anti-DoS protections," in *Proc. ARES*, 2021, pp. 1–9.

[25] C. Wen et al., "MemLock: Memory usage guided fuzzing," in *Proc. 42nd ACM/IEEE ICSE*, 2020, pp. 765–777.

[26] L. Chen, R. Huang, D. Luo, C. Ma, D. Wei, and J. Wang, "Estimating worst-case resource usage by resource-usage-aware fuzzing," in *Proc. FASE*, 2022, pp. 92–101.

[27] M. Bender, E. Kirdan, M.-O. Pahl, and G. Carle, "Open-source MQTT evaluation," in *Proc. 18th IEEE CCNC*, 2021, pp. 1–4.