Mechanisms to improve fuzz testing for message brokers

Luis Gustavo Araujo Rodriguez

THESIS PRESENTED TO THE
INSTITUTE OF MATHEMATICS AND STATISTICS
OF THE UNIVERSITY OF SÃO PAULO
IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF SCIENCE

Program: Computer Science
Advisor: Prof. Dr. Daniel Macêdo Batista

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and by CNPq (grant 381249/2022-0).

São Paulo
September, 2023
Mechanisms to improve fuzz testing for message brokers

Luis Gustavo Araujo Rodriguez

This version of the thesis includes the corrections and modifications suggested by the Examining Committee during the defense of the original version of the work, which took place on September 1, 2023.

A copy of the original version is available at the Institute of Mathematics and Statistics of the University of São Paulo.

Examining Committee:

Prof. Dr. Daniel Macêdo Batista (advisor) – IME-USP
Profª. Drª. Michelle Silva Wangham – Univali
Prof. Dr. Alfredo Goldman vel Lejbman – IME-USP
Prof. Dr. Fabio Moreira Costa – UFG
Prof. Dr. Lucas Carvalho Cordeiro – University of Manchester
The content of this work is published under the CC BY 4.0 license

(Creative Commons Attribution 4.0 International License)
This PhD thesis is dedicated to my dear and beloved mother, Dilma Nereyda Rodriguez Saravia. I love you mom.
Acknowledgments

La grandeza de una persona no está en su superficie, sino en las Honduras de su alma y mente.

First and foremost, I thank God for giving me the opportunity to pursue and complete the doctoral program. I can not thank you enough for so many blessings I have received throughout my life. Thank you for making my dream come true, and for giving me the experience of a lifetime at the University of São Paulo. I also want to thank God for my wonderful family and friends.

I would like to thank my family, whose support was one of the driving forces behind my hard work and dedication. Thank you to my dear and beloved mother, Dilma Nereyda Rodriguez Saravia, for her hard work and huge sacrifice in order to give me a good education. Thank you for loving me despite my imperfections, and for always putting my happiness, future, dreams, ambitions, and well-being first rather than your own. I can not thank you enough for everything you have done for me, and I promise that no matter what happens I will continue making you proud. This thesis is dedicated to you because I know that I would not have been able to even apply to a doctoral program without your unconditional and everlasting love. I am so incredibly blessed to have you as my mother, and I hope that my accomplishments brought joy and happiness into your life. Thank you to my sister, Nereyda Maria Araujo Rodriguez, for the support during the graduate studies. Thank you to Katie Thompson, a family member, true friend, and incredible mentor who has supported me since my childhood. Thank you Katie for believing in me and caring about my future. I am incredibly honored and grateful to have you always by my side. Thank you to my aunts and uncles for looking out for me. Thank you to my aunt Miriam Rodriguez for her generosity. Thank you to my aunt Josefa Rodriguez for her compassion. Thank you to my aunt Alba Rodriguez for always lending a helping hand. Thank you to my uncle Carlos Rodriguez for teaching me the importance of education for both personal and professional growth. Thank you to my aunt Norma Saravia for taking care of me when I was young. Thank you to my uncles Luis Rodriguez and Rafael Soto for the conversations.
Thank you to my cousins Yami Soto, Paulina Rivera, Samuelito Rivera, Samuel Rivera, Larissa Soto, Nelson Rodriguez, and Fanny Rodriguez for being by my side through tough times. Thank you to Martha Andino for being such a kind-hearted person to my family. Thank you to Vilma Bonano for being so supportive of my decision to pursue a doctoral degree. Thank you to Ricardo Cervantes for teaching me the importance of aspirations and goals in life. Thank you to Jenny Giron, Mariela Giron, and Mirna Cano for being extremely generous whenever I needed assistance. Thank you to Norma Rivera for always having my back.

In addition to my family, there are so many people that played a key role in my journey towards finishing the doctoral degree. I want to thank my wonderful friends that I had the pleasure of meeting at the University of São Paulo. Thank you to Thamillys Marques, the first friend I made at the university, for providing the necessary strength to overcome all the challenges faced during the first years of the doctoral program. Thank you to Carlos Enrique Paucar Farfan for the comforting words to handle stressful situations. Thank you to Erik Miguel de Elias for always taking the time off to have stress-reducing conversations over a cup a coffee. Thank you to Fatemeh Mosaiyebzadeh for always reassuring me that I was going to earn the doctoral degree. Thank you to Felipe Caetano Silva and Julio Kenji Ueda for reducing my anxiety with humor, laughter, video games, and subway sandwiches. Thank you to Antônio Kaique Barroso Fernandes for always showing friendliness and generosity towards me. Thank you to Guilherme Vieira dos Santos for being so kind and supportive during the writing process of this thesis. Thank you to Igor Moreira Félix and his family for their trust and company during the COVID-19 pandemic. Thank you to Bernardo Martins, Gabriel Morete, Lucas Stankus, Marcelo Schmitt, Matheus Tavares, Pedro Siqueira, Pedro Arraes, Renato Cordeiro Ferreira, Thiago Lima Oliveira, and Ygor Requenha Romano for their friendliness towards me. Thank you to Priscila Lima for helping me to overcome self-doubt.

I especially wanted to thank the following friends who have made such a profound and positive impact on my life. I will forever cherish the moments we shared at the University of São Paulo. In particular, I will always remember how we persevered and supported one another in order to earn our graduate degrees. Thank you to Danilo Pereira Escudero for giving me hope and strength in moments of despair. Thank you to Douglas Chagas da Silva for always helping me whenever I was struggling in life. Thank you to Luiz Felipe Fronchetti Dias for always supporting me, despite the distance. Thank you to Luiz Henrique Neves Rodrigues for being a friend I could rely on anytime and anywhere. Thank you to Mairieli Santos Wessel for always wanting the best for me. Thank you to Samuel Plaça de Paula for always being by my side on this long and arduous journey; and for being an encouraging, a dependable, a humorous, and most importantly, a true friend.
am truly happy for your constant presence in my life. Thank you to Thatiane de Oliveira Rosa, my loyal companion on this journey since the first day of the doctoral program. Thank you for lifting me up everytime I felt life was dragging me down. We have endured many hardships together, and I am incredibly lucky and blessed to have you by my side helping me to overcome every obstacle in my life. Thank you for the real and wonderful friendship you have given me.

Thank you to my friends at the Regional Center of Environmental Documentation and Interpretation (CREDIA), which is an institute that taught me so much about the importance of discipline and hard work. Thank you to Gaby Montoya for always making me laugh and feel calm despite all the hardships and challenges faced over the last few years. Thank you to Miriam Elisa Lopez Rosales for being a major support system throughout the doctoral studies; for always believing in my capabilities, and encouraging me to face my fears and to never give up on my dreams. I would not have gotten to where I am today without your guidance and friendship.

I also would like to thank my dear friends from Colombia that have supported me ever since we met in Maringá. Thank you to David Fernando Posso Suarez for always providing me with words of encouragement whenever I needed them the most. Thank you to Yessica Acosta Urian for the everlasting support, continuous presence, and real friendship provided during stressful times.

I’d also like to thank my friends from the Catholic University of Honduras for being so supportive and understanding since pursuing my graduate studies. A special thanks to Cindy Montoya, Diana Calix, Debby Martinez, Hector Pallares, Jeny Aguilera, Karen Rodriguez, Karen Pinto, Kenia Gutierrez, Mariela Raudales, Rony Fiallos, Sory Streber, and Tiffany Coca.

I’d also like to express gratitude to my friends from La Ceiba Bilingual School. Thus, thank you to Andrea Galindo, Carlos Fermin Arellano Oliva, Cinthia Klarissa Corea Santos, Jonathan Villafranca, and Rodrigo Sabillon for their continuous presence and support in my life.

This thesis would not have been possible without the unconditional support and valuable guidance from Prof. Dr. Daniel Macêdo Batista, my advisor, who I have had the pleasure of collaborating with for five years. Thank you for the lessons learned, patience, and most importantly for the belief that I could undertake this research project. It was a great honor conducting research with you, and I hope we can continue working together in the future.

I’d also like to thank my professors at the Institute of Mathematics and Statistics who
have imparted so much knowledge and wisdom in order to better hone my skills as a researcher. Thank you to Prof. Dr. Fabio Kon for supporting me financially, and caring about my future. Thank you to Prof. Dr. Jose Coelho de Pina Junior for the engaging and insightful conversations during our lunch breaks. Thank you to Profa. Dra. Kelly Rosa Braghetto for being so kind and supportive to me. Thank you to Prof. Dr. Leônidas de Oliveira Brandão for the positive messages during the COVID-19 pandemic.

Thank you to my examining committee, Profa. Dra. Michelle Wangham, Prof. Dr. Alfredo Goldman, Prof. Dr. Fabio Costa, and Prof. Dr. Lucas Cordeiro for their valuable and insightful suggestions to improve the doctoral thesis.

I also would like to thank employees at IME-USP for their support and kindness. Thank you to the security personnel who always kept me company, especially during the difficult times of the COVID-19 pandemic. In that regard, thank you to Alice Ferreira, Daniel Jeferson, Denilson Ribeiro, Fatima Silverio, Gabriel Flores, Gabriela Borges, Guilherme Almeida, José Geraldo, Jozined Ferreira, Kaire Gustavo, Ktreson Silva, Rosineide Dias, Leandro Maia, Luan Carlos de Jesus, Lucimar Ribeiro, Manuel Izau, Marcone da Silva, Milton Rafael, Thiago Wesley, Valyuira Fernandes, and Wendell Augusto.

I also want to deeply thank the library staff of the Institute of Mathematics and Statistics for their constant support, kindness, and friendship during my last few months as a PhD candidate. Thank you to Maria das Graças Silva de Lima for always taking the time to listen to my worries, and providing the necessary words to ease my fears. Thank you to Ricardo Vila Real Tanaka for being so kind and willing to help me focus on the doctoral studies. Thank you to Daniel Feltrin Amaral for always providing assistance whenever I needed it.

I would like to thank the staff of the graduate program in computer science for their crucial assistance throughout the doctoral studies, especially in the months leading up to the defense. Thank you to Katia Kiesshau for the patience, advice, and kindness whenever I had inquiries about the PhD program. Thank you to Luiza Ribeiro Camilo da Silva for also always willing to help me in times of need. Thank you to Ana Carla de Souza, Cristiane de Fátima Braulino, and Edina Arouca for always caring about my well-being during the doctoral studies.

I could not leave out the professors and employees from the State University of Maringá who encouraged me to send my application to the University of São Paulo. Thank you to Prof. Dr. Anderson Faustino da Silva, my Master’s advisor, for inspiring me with words of wisdom and teaching me the importance of self-improvement, perseverance, and critical thinking. The lessons learned from you played a big role in preparing me for the doctoral
studies. Thank you to Prof. Dr. Elvio João Leonardo and Prof Dr. Nardênio Almeida Martins for their unconditional support and trust in me. Thank you to Inês Laccort for the friendship, encouragement, and support during the difficult times of my graduate studies. Thank you to Elenir Voi Xavier, my portuguese teacher and friend, for always believing in me. Thank you to Denise Dalcol for the friendship we have maintained after earning our Master’s degrees.

Thank you to CAPES and CNPq (grant 381249/2022-0) for the financial support, which was crucial for the completion of this thesis.

Finally, thank you to everyone who has supported me on this incredible journey. I am truly honored and blessed to have known every single one of you, and to have represented my home country Honduras for the last couple of years.

Os protocolos de publicação-assinatura têm desempenhado um papel fundamental no sucesso da Internet das Coisas. À medida que a Internet das Coisas se expande para novos usuários e ambientes, a necessidade de testar melhor os protocolos de publicação-assinatura torna-se ainda maior. No entanto, a área de testes para protocolos publicação-assinatura é pouco explorada, com poucos estudos que examinam estratégias eficazes para aumentar a confiabilidade e a robustez de brokers de mensagens contra pacotes malformados. Considerando que diversas falhas descobertas nos brokers são por causa de pacotes malformados, testes baseados em fuzzing surgiram como uma das técnicas mais importantes para mitigar esse problema. No entanto, o fuzzing enfrenta muitos desafios quando aplicado aos protocolos de publicação-assinatura, que se distinguem de outros protocolos por sua funcionalidade de publicação de mensagens e arquitetura orientada a eventos. Isso levanta a questão sobre se os desenvolvedores e as ferramentas baseadas em fuzzing (ou fuzzers) estão considerando os atributos exclusivos do padrão publicação-assinatura na hora de realizar os testes. O objetivo desta tese de doutorado é apresentar estratégias eficazes de fuzzing para os protocolos de publicação-assinatura, com o objetivo de contribuir para o desenvolvimento de aplicações mais robustas na Internet das Coisas e Cidades Inteligentes. De acordo com as pesquisas preliminares, há uma falta de abordagens sistemáticas baseadas em fuzzing na literatura para testar os protocolos de publicação-assinatura. Além disso, MQTT se destaca como o protocolo de publicação-assinatura mais popular para o qual os desenvolvedores propuseram técnicas de fuzzing na literatura. Por tanto, MQTT oferece uma oportunidade de entender os requisitos e estratégias para testar efetivamente um protocolo de publicação-assinatura. Esta pesquisa de doutorado foi dividida em três fases. Na primeira fase, foi analisado se uma abordagem de fuzzing baseada em gramática pode ser aplicada a um protocolo de publicação-assinatura, entendendo assim os desafios e requisitos necessários. Assim, foi proposta uma metodologia e arquitetura para desenvolver um fuzzer baseado em gramática para testar um protocolo de publicação-assinatura. O resultado final é um fuzzer chamado MQTTGRAM, que foi então comparado com duas outras abordagens de fuzzing, e superou ambas, apesar de realizar menos testes. Na segunda fase, foi desenvolvida uma taxonomia que classifica todas as técnicas de fuzzing existentes para MQTT, das quais seis foram avaliadas em condições equivalentes para determinar se os desenvolvedores estão considerando os atributos exclusivos do padrão publicação-assinatura na hora de realizar os testes. Além disso, os fuzzers para MQTT foram avaliados em termos de testes de estresse. Na terceira fase, MQTTGRAM foi aprimorado, incorporando três elementos essenciais para testar os protocolos de publicação-assinatura: comunicação bidirecional; conhecimento de tópicos; e suporte a múltiplas versões. Esta pesquisa de doutorado fornece três contribuições principais: (1) o desenvolvimento e aprimoramento de uma abordagem de fuzzing baseada em gramática para um protocolo de publicação-assinatura; (2) taxonomia e avaliação de desempenho de fuzzers para MQTT em condições equivalentes; e (3) identificação de deficiências dos fuzzers para trabalhos futuros.

Abstract


Publish/subscribe (or pub/sub) protocols have played a key role in the success of the Internet of Things. As the Internet of Things expands to new users and environments, the need to better test pub/sub protocols becomes even more pressing. However, pub/sub protocol testing is an under-explored field, with few research studies examining effective strategies to increase the reliability and robustness of message brokers against malformed packets. Considering that several bugs discovered in message brokers are related to malformed packets, fuzz testing (or fuzzing) has emerged as one of the most promising, necessary, and ideal techniques to mitigate this issue. However, fuzzing faces many challenges when applied to pub/sub protocols, which distinguish themselves from other network-based systems by their message-publishing features and event-driven architecture. This poses the question as to whether developers and existing fuzz testing tools (or fuzzers) consider the unique attributes of the pub/sub messaging pattern. The objective of this PhD thesis is to study and develop effective fuzzing strategies for pub/sub protocols, aiming at contributing to the development of more robust applications in IoT and Smart Cities. According to the research findings, there is a lack of systematic approaches in the literature to fuzz-test pub/sub protocols. Furthermore, MQTT stands out as the most popular pub/sub protocol for which developers have proposed fuzzing techniques in the literature. However, as MQTT is the most widely-used pub/sub protocol, it provides an opportunity to understand the requirements and strategies to effectively fuzz a pub/sub protocol. This PhD research was divided into three phases. In the first phase, it was analyzed whether a systematic testing approach such as grammar-based fuzzing can be applied to a pub/sub protocol such as MQTT, thereby understanding the challenges and necessary requirements. A grammar-based methodology and architecture was therefore conceived and proposed for a pub/sub protocol. The end result is a fuzzer called **MQTTGRAM**, which was then compared with two other fuzzing approaches and outperformed both of them, despite exchanging up to 9x fewer packets. In the second phase, a taxonomy was developed that classifies all existing fuzzing techniques for MQTT, six of which were evaluated under equivalent conditions in order to determine whether developers are considering the unique attributes of the pub/sub design pattern. Furthermore, the fuzzers were evaluated in terms of their resource usage or stress-testing capabilities. In the third phase, **MQTTGRAM** was improved by incorporating three essential elements for pub/sub protocol fuzzing, which are lacking across all fuzzing techniques proposed for MQTT: two-way communication; topic awareness; and version support. Overall, this PhD research provides three main contributions: (1) the development and refinement of a grammar-based fuzzing approach for a pub/sub protocol; (2) taxonomy and performance evaluation of MQTT fuzzers under equivalent conditions; and (3) identification of shortcomings for future work.

**Keywords:** Publish-subscribe. Fuzzing. Testing. Network Protocols. Message Broker. MQTT. IoT.
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Access Control Lists</td>
</tr>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol</td>
</tr>
<tr>
<td>DDS</td>
<td>Data Distribution Service</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>DTLS</td>
<td>Datagram Transport Layer Security</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IME</td>
<td>Institute of Mathematics and Statistics</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>ITS</td>
<td>Intelligent Transportation Systems</td>
</tr>
<tr>
<td>M2M</td>
<td>Machine-to-Machine</td>
</tr>
<tr>
<td>MQTT</td>
<td>Message Queuing Telemetry Transport</td>
</tr>
<tr>
<td>NVD</td>
<td>National Vulnerability Database</td>
</tr>
<tr>
<td>OASIS</td>
<td>Organization for the Advancement of Structured Information Standards</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>Pub/Sub</td>
<td>Publish/Subscribe</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>USP</td>
<td>University of São Paulo</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Components of a Pub/Sub Network</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Overview of Doctoral Studies</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Testbed Developed and Used for Doctoral Studies</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Fuzz Testing/Fuzzing</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Architectural Messaging Patterns</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Traditional Network Protocol Fuzzing (1 Client)</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Pub/Sub Protocol Fuzzing (2 Clients)</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Network-Based Fuzzer Sending a Malicious Subscribe Packet to the Broker and Causing a Stack Overflow (CVE-2019-11779)</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Causes of Vulnerabilities in MQTT (As of April 19th, 2023)</td>
<td>18</td>
</tr>
<tr>
<td>2.7</td>
<td>Vulnerability Disclosure Delays (As of April 19th, 2023)</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Outdated Brokers Accessible (As of April 19th, 2023)</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture of MQTTGRAM</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Simple Example of the MQTT 3.1.1 Grammar in Backus-Naur Form</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Derivation Tree for a PUBLISH Packet in Hexadecimal Notation</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Results of 3-Minute Test Runs</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Results of 30-Minute Test Runs</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Results of 500- and 8000-Packet Test Runs</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Last Will and Testament Messages</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Retained Messages</td>
<td>41</td>
</tr>
<tr>
<td>4.3</td>
<td>Persistent Session</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Taxonomy of Fuzzing Techniques for MQTT</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>Statement Coverage (Mosquitto 1.6.8)</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Statement Coverage (Moquette 0.13)</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Refined Architecture for MQTTGRAM-R</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Simple Example of the MQTT 5.0 Grammar in Backus-Naur Form</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Timeline of Mosquitto Version Releases (As of April 19th, 2023) ..... 20
2.2 Open Source MQTT Fuzzers ..... 21
2.3 Related Work ..... 21
3.1 Statement Coverage of Moquette Achieved by MQTT Fuzzers in 30 Minutes
(Results by Sochor et al. (2020b) were copied directly from their research paper due to their fuzzer being proprietary).

4.1 Statement Coverage of Moquette (30 Minutes) (Cells highlighted in blue indicate the highest coverage percentage for a particular directory.)

4.2 Packet Generation - Connect Flags
✔: Correct usage of a flag value | ✗: Incorrect usage of a flag value | ☐: No usage of a flag value

4.3 Packet Generation - Publish Flags
✔: Correct usage of a flag value | ✗: Incorrect usage of a flag value | ☐: No usage of a flag value

4.4 Feature Coverage of MQTT Fuzzers (30 Minutes)
✔: Tested successfully | ✗: Tested unsuccessfully | ✗*: Tested successfully if clients > 1

5.1 Classification of Fuzzers for Pub/Sub Brokers

B.1 Comparison of IoT Protocols

C.1 MQTT Control Packets

C.2 Fixed Header Format of the PUBLISH Packet

D.1 Input Coverage of Moquette (8000 packets)
Contents

1 Introduction .......................................................... 1
  1.1 Objectives of Doctoral Studies ................................. 3
  1.2 Phases and Contributions of Doctoral Studies ............... 4
    1.2.1 Dissemination of Research Findings ....................... 6
    1.2.2 Development of Open Source Fuzzers and Testbed ........ 8
    1.2.3 Contributions to Open Source Software ................... 10
    1.2.4 Contribution to Research Paper Unrelated to Doctoral Studies 10
  1.3 Thesis Outline .................................................. 10

2 State of the Art in Pub/Sub Protocol Fuzzing .................. 11
  2.1 Background ..................................................... 11
  2.2 Challenges ..................................................... 13
  2.3 Literature Review .............................................. 15
  2.4 MQTT Fuzzing ................................................... 16
    2.4.1 Related Work ............................................... 19

3 A Grammar-Based Fuzzing Technique for a Pub/Sub Protocol ... 25
  3.1 Architecture .................................................... 26
  3.2 Algorithms ...................................................... 28
    3.2.1 Selecting the Packet Type ................................ 28
    3.2.2 Generating the Packets .................................... 28
    3.2.3 Feeding the Packets ........................................ 30
    3.2.4 Evaluating the Response ................................... 31
    3.2.5 Monitoring the Program .................................... 31
  3.3 Performance Evaluation ......................................... 31
    3.3.1 Mosquitto ................................................... 32
    3.3.2 Takeaway from Experiments ................................. 35
  3.4 Threats to Validity .............................................. 36
  3.5 Concluding Remarks ............................................. 36
4 Taxonomy and Coverage Evaluation of Fuzzing Techniques for the MQTT Protocol
  4.1 Design Considerations for MQTT Fuzzers .................................. 39
  4.2 Taxonomy ................................................................................. 42
  4.3 Performance Evaluation .............................................................. 43
    4.3.1 Statement Coverage .......................................................... 45
    4.3.2 Feature Coverage .............................................................. 53
  4.4 Concluding Remarks .................................................................. 56

5 Refinement of a Grammar-Based Fuzzing Technique for a Pub/Sub Protocol
  5.1 Refinements .............................................................................. 58
  5.2 Performance Evaluation .............................................................. 61
    5.2.1 handle_connect.c .............................................................. 62
    5.2.2 handle_publish.c .............................................................. 63
    5.2.3 handle_subscribe.c ........................................................... 65
    5.2.4 subs.c .............................................................................. 66
  5.3 Concluding Remarks .................................................................. 69

6 Stress Test Evaluation of Fuzzing Techniques for MQTT Brokers ............ 71
  6.1 State of the Art ......................................................................... 72
  6.2 Performance Evaluation .............................................................. 73
    6.2.1 Mosquitto ........................................................................ 73
    6.2.2 Moquette .......................................................................... 76
  6.3 Discussion .................................................................................. 77
  6.4 Concluding Remarks .................................................................. 79

7 Conclusions and Future Work ................................................................ 81
  7.1 Future Work ............................................................................. 82
    7.1.1 Improvements to Existing Fuzzing Techniques ...................... 82
    7.1.2 Development of Built-In Fault Detection Mechanisms .......... 84
    7.1.3 Development of a Resource-Intensive Fuzzer for MQTT Brokers 85
    7.1.4 Development of Fuzzers Based on Hybrid and Machine Learning Approaches ......................................................... 85
    7.1.5 Performance Evaluation Over Longer Test Runs .................. 85
    7.1.6 Maintenance of MQTT Fuzzers ......................................... 85
    7.1.7 Expand the Research Scope to Include Other Pub/Sub Protocols . 86


Appendixes

A Usage of MQTT in IoT Applications 87
A.1 Home Automation Systems 87
A.2 Health Monitoring Systems 87
A.3 Intelligent Transportation Systems 88
A.4 Electricity Metering Systems 88

B Comparison Between MQTT and Other IoT Protocols 89
B.1 HTTP 89
B.2 XMPP 89
B.3 AMQP 90
B.4 DDS 90
B.5 CoAP 90
B.6 Summary 90

C MQTT Packets 93
C.1 Structure 93
C.2 Transmission 94

D Input Coverage of Fuzzing Techniques for the MQTT Protocol 99
D.1 Mosquitto 99
D.2 Moquette 101

E Packet Exchange of Fuzzing Techniques for the MQTT Protocol 103

References 107
Chapter 1

Introduction

For decades, Smart Cities have been envisioned as technologically advanced urban areas that offer automated, efficient, and immersive services to their citizens (Zanella et al., 2014). Smart Cities began merely as a proof of concept, but have recently taken off after the rise of the Internet of Things (IoT). The objective of a Smart City is to improve the lives of its citizens through the use of information and communication technologies. However, the heterogeneous and ubiquitous environments of a Smart City demand a flexible messaging model capable of withstanding the multitude of requests from its citizens. In that regard, the traditional client-server model is unsuitable for Smart Cities because the server provides information only upon receiving client requests. This could result in the server receiving far too many simultaneous requests than it can handle, possibly leading to a city-wide denial of service.

The publish/subscribe (pub/sub) messaging model mitigates this issue to a certain degree by allowing the server to provide information automatically without the need for a client request. In order to accomplish this task, clients subscribe to a topic of their choice, receiving new and updated messages from the server automatically at any given moment. The pub/sub messaging model is therefore integral to the realization of Smart Cities, whose success relies heavily on automated services. In contrast to their traditional client-server counterparts, pub/sub networks usually have three main components: (1) subscriber, (2) publisher, and (3) broker. The latter component is the most important to the success of the network, and it is responsible for receiving messages from publishers and sending them to interested subscribers.

For example, Figure 1.1 presents an IoT scenario in which temperature sensors and light sensors act as publishers. Both types of sensors inform their status (ON) to a ventilation and illumination system, which act as interested subscribers. For clarity purposes, publishers and their respective subscribers are highlighted in the same colors. The broker acts as an intermediary between the publishers and subscribers, receiving messages from the former component and sending them to the latter.

Although successful, the pub/sub messaging model introduces several challenges that have yet to be studied or addressed. Most notably, adapting client-server applications, frameworks, and testing techniques to the pub/sub model requires further research efforts. In fact, pub/sub protocol testing is a field that is extremely important, yet somehow unex-
explored and neglected to a certain extent in the literature. This research gap is especially worrisome when considering that the majority of bugs found in message or pub/sub brokers are related to malformed packets (Araujo Rodriguez and Macêdo Batista, 2020), most of which are officially disclosed several days after their discovery (Araujo Rodriguez, Selvatici Trazzi, et al., 2018). In addition to vulnerability disclosure delays, deploying and applying patches to pub/sub brokers in real-world environments is usually complex due to either their location or lack of auto-update functionality (Husnain et al., 2022).

The root cause of these problems stems from the heavy emphasis placed on functionality rather than security in IoT (Săndescu et al., 2018), meaning that pub/sub protocols tend to be insecure by design and unreliable in large-scale areas such as Smart Cities. In fact, insecure design and injection are two of the most common causes of vulnerabilities globally (OWASP, 2021). Insecure design refers to weak protocol implementations with design flaws, whereas injection refers to untrusted inputs. Specifically, weak protocol implementations are considered to be the most serious security threat in IoT (Munea, Lim, et al., 2016; Munea, Luk Kim, et al., 2017; Makhshari and Mesbah, 2021; Husnain et al., 2022), potentially exposing smart applications to catastrophic cyberattacks.

All of the aforementioned issues can be mitigated or avoided altogether with effective testing mechanisms for pub/sub brokers. In that regard, one of the most ideal and necessary testing techniques for pub/sub protocols is fuzzing (Luo et al., 2018; Praveen et al., 2023), which consists in generating and sending random inputs to a message broker. The output of the broker is then analyzed for potential weaknesses. Fuzzing emerged in the early 1990s as an effective testing strategy for UNIX utilities (Miller et al., 1990). The success of fuzzing led to its application across several target systems, including network protocols. Fast forward to the present, and pub/sub protocol fuzzing is an under-explored field, with few research studies examining effective strategies to increase the reliability and robustness of message brokers against random or malformed packets.
1.1 Objectives of Doctoral Studies

Considering the important role of message brokers and their susceptibility to malformed packets, the objective of this PhD research is to study and develop effective fuzzing strategies for pub/sub protocols, aiming to contribute to the development of more robust applications in IoT and Smart Cities. This PhD research focuses specifically on fuzzers for broker-side implementations of pub/sub protocols because of their primary and critical role in the network. In order to satisfy the objective of this doctoral research, a preliminary study was first conducted to examine the advancements and trends of pub/sub protocol fuzzing since its inception. The research findings of the preliminary study revealed two important findings. First, at the time of writing, the Message Queuing Telemetry Transport (or MQTT) stands out as the only pub/sub protocol for which developers have proposed several different fuzzing techniques in the literature. Second, there is a lack of systematic approaches in the literature to fuzz-test pub/sub protocols (Araujo Rodriguez and Macêdo Batista, 2020). Considering the preliminary studies, the research questions formulated for the doctoral studies are as follows:

**RQ1:** How can a grammar-based fuzzer for a pub/sub protocol such as MQTT be developed?

**Research Gap:** As their name suggests, grammar-based fuzzers generate test cases by using a grammar, which describes the syntax and structure of an input (or packet in the case of network protocols). Despite being renowned as one of the most effective testing strategies for network protocols (Godefroid et al., 2017), grammar-based fuzzing has been avoided in favor of other alternatives throughout the years due to its complicated nature (Hernández Ramos et al., 2018; Sochor et al., 2020b). The situation is further aggravated by the fact that there are no sources explaining how to adapt grammar-based fuzzing to pub/sub protocols. Thus, the goal of *RQ1* is to propose and explain a grammar-based methodology and architecture for a pub/sub protocol, which can be used as a reference by developers and researchers in future studies.

**RQ2:** What fuzzing techniques have been proposed for MQTT over the last few years?

**Research Gap:** Research on MQTT fuzzing has grown considerably, with several different techniques found across the literature, more so than for any other IoT or pub/sub protocol. Thus, the goal of *RQ2* is to develop a taxonomy based on fuzzing techniques proposed for MQTT across all research studies. The taxonomy can be used by researchers to either gain a better understanding of advancements in the field for future studies, or benchmark each technique for the most optimal decision-making.

**RQ3:** How effective are fuzzing frameworks for MQTT in terms of their testing and pub/sub capabilities?

**Research Gap:** The current state of the art lacks a comprehensive performance comparison between MQTT fuzzers. The problem stems from the fact that few research papers on MQTT fuzzing consider traditional testing metrics (Zeng et al., 2020; Sochor et al., 2020b; Araujo Rodriguez and Macêdo Batista, 2021). Moreover, fuzzing performance is measured differently across research studies, either by calculating the number of paths
(path coverage) or statements (statement coverage) executed in the source code, thereby hindering a comparison. The situation is further aggravated when considering that there are no standardized methods or metrics in the literature to evaluate fuzzers for pub/sub protocols such as MQTT. This research gap also raises the question of whether developers are considering the unique characteristics of the pub/sub design pattern when building their own fuzzers. The goal of RQ3 is to examine open-source MQTT fuzzers in terms of their testing and pub/sub capabilities, identifying shortcomings and missteps by developers for the aforementioned design pattern.

*RQ4:* How effective is an MQTT fuzzer when considering three essential elements for pub/sub fuzzing: two-way communication capabilities; topic awareness; and multiversion support?

**Research Gap:** Regardless of their technique, MQTT fuzzers are essentially pub/sub test suites, whose overall goal should be to cover functionalities pertaining to the unique attributes of the design pattern. In that regard, developing an effective and high-coverage fuzzer for MQTT or any other pub/sub protocol requires a keen focus on message publication, which is not only the most defining, distinguishing, and important feature of the broker, but also the most computationally intensive. However, the research findings indicate that all fuzzers have shortcomings in regards to pub/sub protocol testing. The goal of RQ4 is to improve an existing MQTT fuzzer by incorporating three essential elements for successful pub/sub fuzzing: (1) two-way communication capabilities; (2) topic awareness; and (3) multiversion support.

*RQ5:* How effective are existing fuzzing strategies at impacting the CPU and memory usage of the broker during testing?

**Research Gap:** At the time of writing, none of the open-source MQTT fuzzers are capable of receiving resource-related feedback about the broker during testing. Developers have no choice but to select and extend one of the open-source fuzzers available for testing purposes. It is therefore important for developers to understand the capabilities of each option available. The goal of RQ5 is to survey, benchmark, and evaluate existing state-of-the-art fuzz testing tools for MQTT brokers to determine their capabilities in detecting CPU- and memory-consumption bugs. In order to move the state of the art forward, shortcomings and guidelines will be provided for developers to build better resource-heavy fuzzers for MQTT brokers in Smart Cities.

It is worth noting that all of the aforementioned research questions arose sequentially throughout the doctoral studies, rather than all at once. In other words, the findings of an experiment led to another research question, and so forth.

### 1.2 Phases and Contributions of Doctoral Studies

Based on the five research questions, the doctoral studies consisted of three phrases. Figure 1.2 presents an overview of the research activities performed for each phase throughout the doctoral studies.
Figure 1.2: Overview of Doctoral Studies
The technical contributions of this research are highlighted in yellow, whereas the scientific contributions are highlighted in green. A total of six papers (indicated by blue circles) were written to disseminate the research findings, four of which have been published in peer-reviewed conferences and journals at the time of writing. The remaining two papers are currently under review for publication.

1.2.1 Dissemination of Research Findings


The objective of this paper was to evaluate vulnerability disclosure delays from the National Vulnerability Database (NVD) in order to state its efficiency. Among several findings, it was observed that the majority of vulnerabilities are disclosed within 1-7 days after their discovery. Based on these results, the paper provided recommendations for those who currently rely on NVD, such as IoT manufacturers and developers. This paper won an honorable mention for the Best Paper Award.


Due to vulnerability disclosure delays, P1 confirmed the necessity for prevention and proactive approaches such as test-driven mechanisms in the field of IoT. In order to narrow down the scope of the PhD research, the objective of this paper was two-fold: (1) identify the test-driven approach that is most suited for IoT message brokers based on MQTT standards; and (2) analyze existing MQTT-based testing frameworks. This paper revealed that test-driven approaches for MQTT should be based on malformed or random packets, which are the most common causes of vulnerabilities in message brokers. At the time of publication, the paper also unveiled that testing frameworks were based on blackbox fuzzing, meaning that vulnerabilities are difficult and time-consuming to find. The paper therefore presented and proposed the overall design of a new greybox fuzzer for testing MQTT applications. Overall, one of, if not the most important finding of this paper was the lack of systematic and modern approaches in the literature to fuzz-test pub/sub protocols. The paper therefore presented a research proposal to study and develop effective fuzzing strategies for pub/sub protocols, aiming to contribute to the development of more robust applications in IoT and Smart Cities. The research proposal and findings of this paper were presented at a Doctoral Symposium, which offered PhD students the opportunity to receive feedback from field specialists. The main feedback received from field specialists was to develop a new grammar-based fuzzing technique for a pub/sub protocol.

A new grammar-based methodology and architecture was developed based on the feedback received from the Doctoral Symposium Program Committee. The end result is a fuzzer called **MQTTGRAM**, which was then compared with two other fuzzing approaches and outperformed both of them, despite exchanging up to 9x fewer packets with the broker. The research findings were presented in this paper, which at the time of its publication, was the first to evaluate the effectiveness of a grammar-based fuzzer for a pub/sub protocol (*Araujo Rodriguez and Macêdo Batista, 2021*).


This paper evaluates the resource consumption of state-of-the-art fuzzing frameworks, thereby understanding the degree to which brokers are tested before deployment. The research findings attest that only one framework shows the most promise for memory-intensive testing, outperforming its counterparts by more than 20%. This paper also highlights shortcomings that prevent existing frameworks from consuming considerable system resources. This paper was the first to investigate about the fuzzers’ resource-exhaustion capabilities.


After developing a grammar-based approach for a pub/sub protocol, a question was raised as to whether developers were designing their fuzzers considering the unique attributes and features of pub/sub messaging. This uncertainty stems from the fact that there is currently a lack of research regarding the strengths and weaknesses of each fuzzing technique for MQTT. This situation also hinders a developer’s understanding of the pros and cons of each technique in order to select the most appropriate for testing purposes. In order to mitigate these issues, this paper analyzes the existing fuzzing techniques in depth and under equivalent conditions, which will not only clarify and reveal their effectiveness to developers, but also identify future directions. Overall, this paper provides three main contributions. First, it presents a literature review and taxonomical classification of fuzzing techniques for MQTT, followed by an analysis and performance benchmarks of six popular frameworks. Finally, this paper presents open issues and proposes improvements for existing MQTT fuzzers. The research findings attest that most fuzzing frameworks for MQTT are incapable of testing all the pub/sub functionalities and features defined in the standard, thus providing research opportunities to mitigate their limitations and apply their techniques effectively on other pub/sub protocols.

P6 Luis Gustavo Araujo Rodriguez and Daniel Macêdo Batista. **Improving Pub/Sub**

After carefully reviewing the literature, the research findings presented in P5 indicate that all fuzzers lack three essential elements for pub/sub fuzzing: (1) Two-way communication capabilities; (2) Topic awareness; and (3) Multiversion support. This paper therefore proposes a new fuzzer that incorporates all three elements within a single architecture. The proposed fuzzer manages to outperform all of its state-of-the-art competitors in terms of standardized testing metrics such as code coverage, executing up to 12x more statements in pub/sub-related files.

1.2.2 Development of Open Source Fuzzers and Testbed

Fuzzers

The source code of the original and refined version of MQTTGRAM, as well as the grammars, are freely available under the General Public License (Version 2)\(^1\)\(^2\), allowing developers to better test message brokers, and facilitating further research studies.

Testbed

The testbed built and used for the doctoral studies is also publicly available under the same license \(^3\), providing developers the opportunity to evaluate their own fuzzers for testing purposes. Figure 1.3 presents the architecture of the testbed, whose main purpose is to monitor the broker while fuzzing.

![Testbed Architecture](image)

**Figure 1.3:** Testbed Developed and Used for Doctoral Studies

The repository for the testbed contains several automation scripts including a Vagrantfile, which is a configuration file that creates the exact same test environment used for the doctoral studies. More specifically, the Vagrantfile configures the broker according to the architecture presented in Figure 1.3. The Vagrantfile installs the necessary dependencies for the broker to communicate with the fuzzer over a private network. As a result,

---

\(^1\) https://github.com/luisgar1990/MQTTGRAM. Accessed on May 17th, 2023


the automation scripts and Vagrantfile play a key role in improving and increasing the reproducibility of the research findings from the doctoral studies.

**Experimental Setup for Doctoral Studies**

The experiments for the doctoral studies were performed on an Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz with 16GB of RAM running the Ubuntu 16.04.6 LTS operating system. The MQTT fuzzers ran natively on the system hardware, whereas the MQTT brokers were hosted on a virtual machine. The native and virtual machine have the same hardware specifications except for the RAM, which is 1 GB on the latter. The MQTT fuzzers and brokers were executed using default settings. The brokers interact with each MQTT fuzzer individually. The MQTT brokers selected as the target systems for the doctoral studies were *Mosquitto 1.6.8* and *Moquette 0.13*, both of which rank among the most popular in the literature ([Hernández Ramos et al., 2018; Palmieri et al., 2019; Casteur et al., 2020; Sochor et al., 2020b; Zeng et al., 2020; Aichernig et al., 2021; Di Paolo et al., 2021; Araujo Rodriguez and Macêdo Batista, 2021]).

Test runs were repeated 100 times in order to calculate the average and standard deviation for each stopping criterion. The fuzzers test the brokers until a certain time has elapsed or a specific number of packets has been exchanged between both parties. After each test run, information regarding a fuzzer’s performance is calculated and recorded in three different types of logs.

The *broker log* contains broker-related information such as the average CPU and memory usage, as well as the number of statements or lines executed in its source code during a test run. The latter metric is known as *statement coverage*, and it has proven to be more reliable than other metrics at predicting the quality of a test suite ([Gopinath et al., 2014]). Statement coverage is calculated using gcov and Cobertura, which are coverage tools designed for programs written in C (Mosquitto) and Java (Moquette) respectively. A variant of statement coverage called *input coverage* was also used throughout the doctoral studies to measure a fuzzer’s performance. Input coverage refers to the statement coverage achieved by a fuzzer after exchanging a specific number of packets with the broker.

In addition to statement and input coverage, fuzzers were evaluated based on their *feature coverage*, which refers to pub/sub functionalities or behaviors that were covered during testing. A tool called tshark was used to capture and record the network packets exchanged between the MQTT fuzzers and the broker in the *packet log* during the test run. The packet logs were then manually inspected to determine which pub/sub features were tested by the MQTT fuzzers.

The *fuzzer log* contains information regarding crashes. For each test run, the Mosquitto broker is compiled with AddressSanitizer, which is a debugging tool that detects potential memory corruption vulnerabilities even if a crash is untriggered. The fuzzing process is also monitored constantly during the test runs in case either Mosquitto or Moquette crashes.

Further details on the stopping criteria, metrics, MQTT brokers, and fuzzers chosen for each experiment are explained in later chapters of the thesis.
1.2.3 Contributions to Open Source Software

While developing the grammar-based fuzzer for MQTT, a popular Python library called Scapy was lacking functionality or features described in the MQTT 3.1.1 standard. Improvements were made to that library as an additional contribution of this research (Araujo Rodriguez, 2020). Scapy was further improved afterwards to support MQTT 5.0, which will play a major role in several research fields including Fuzzing. This contribution will allow developers to analyze, parse, craft, and modify MQTT 5.0 packets for their own research and needs. Although the pull request is still under review at the time of writing (Araujo Rodriguez, 2021), the changes requested by the maintainers are currently being addressed.

1.2.4 Contribution to Research Paper Unrelated to Doctoral Studies

During the doctoral studies, an opportunity emerged to collaborate on a research paper related to MQTT and intrusion detection systems:


Suggestions were provided throughout the writing process to further improve the research paper.

1.3 Thesis Outline

This PhD thesis is organized as follows. Chapter 2 first explains about architectural messaging patterns and fuzzing. These explanations are then followed by a literature review on pub/sub protocol fuzzing, as well as MQTT fuzzers that are available to developers for testing purposes. Chapter 3 presents the new grammar-based fuzzing technique for a pub/sub protocol such as MQTT. Chapter 4 presents a taxonomy and performance evaluation of pub/sub fuzzers, which at the time of writing are mostly for MQTT. Chapter 5 presents a refined version of the grammar-based approach that incorporates three essential elements for pub/sub fuzzing. Chapter 6 presents and discusses the performance achieved by MQTT fuzzers in terms of the broker’s CPU and memory. Chapter 7 marks the conclusions of this PhD thesis, and discusses future work.
Chapter 2

State of the Art in Pub/Sub Protocol Fuzzing

The aim of this chapter is to first explain the concepts of fuzzing and pub/sub protocol messaging. These explanations are then followed by an overview of the challenges and advancements regarding pub/sub protocol fuzzing. The final sections of this chapter focus on MQTT, which at the time of writing is the only pub/sub protocol for which developers have proposed several different fuzzing techniques in the literature.

2.1 Background

A vulnerability is a flaw or weakness that can be exploited because of design, implementation or configuration mistakes (Antunes et al., 2010). Testing software applications with effective mechanisms is necessary to identify and mitigate design issues or flaws before deployment, thereby increasing their reliability (J. Chen et al., 2018; Liljedahl, 2019). The tests can be performed either manually or automatically. However, finding bugs manually is complex in large-scale systems. Automated methods have therefore become the norm for bug finding. Automated program analysis is classified into two categories: static analysis and dynamic analysis.

Static analysis involves examining the source code of the program at compile-time to discover vulnerabilities (J. Chen et al., 2018). In other words, static analysis searches for vulnerabilities without executing the program. Dynamic analysis, however, involves monitoring the program at run-time to detect potential vulnerabilities.

Automated test analysis may provide misleading claims regarding the existence or absence of bugs in a program. The former claim is known as a false positive, whereas the latter is referred to as a false negative. Static analysis may provide several false positives, requiring user intervention to identify genuine vulnerabilities. Furthermore, static analysis focuses only on limited properties of the program at compile-time, thereby hindering its detection capabilities. However, static analysis has higher statement coverage and detection speeds than its dynamic counterpart, which compensates for its low performance by having few false positives (Zaddach et al., 2014).
One of the most common and popular types of dynamic analysis is Fuzz Testing (or fuzzing) (J. Li et al., 2018), which is an automated technique that consists in generating and sending random or unexpected inputs to a target system. The system behavior is then constantly monitored for any errors (Manès et al., 2019). The term fuzzing was introduced by Miller et al. (1990), which developed a tool called fuzz that generated and fed invalid inputs to eighty-eight UNIX utilities. Program behavior along with their corresponding inputs were stored in a log file for further analysis. The tool discovered several vulnerabilities, which were undetected by traditional testing practices.

Since then, fuzzing has become one of the most popular dynamic analysis techniques for several reasons. First, it has higher scalability and accuracy than other automated testing techniques (J. Li et al., 2018). Second, it has proven to have considerable impact for detecting vulnerabilities in IoT (J. Chen et al., 2018). Third, it can mitigate implementation issues and zero-day attacks. These benefits are reasons why fuzz testing is considered the primary bug-finding technique for most software applications (Munea, Lim, et al., 2016; Munea, Luk Kim, et al., 2017; Luo et al., 2018; Boehme et al., 2021; Vinzenz and Oka, 2021).

Figure 2.1 illustrates a simple interaction between a fuzzer and a software application, which will read and parse random inputs for the entire test run.

![Figure 2.1: Fuzz Testing/Fuzzing](image)

There are three potential outcomes of the software application while interacting with a fuzzer. The first, and probably the most devastating, outcome is a crash immediately after it reads a random input. The second outcome is an immediate rejection of a random input. The third and probably the most desired outcome by software testers is the recognition and acceptance of a random input. The latter outcome is the most promising as it will allow testers to monitor how random inputs affect the functionality of the software application. In order to produce promising outcomes, a fuzzer needs to interact intelligently with the target system in a fully-automated manner.

Target systems can range from simple file-processing programs to complex network protocols. Developers must tailor their fuzzers to the specific characteristics and architectures of a target system in order to ensure a successful test run. Network protocols, in particular, may vary considerably in terms of their architecture. For example, Figure 2.2 presents three different types of messaging patterns.

One-way communication (Figure 2.2a) refers to unidirectional messaging, meaning that packets flow in a single direction. In a two-way communication (Figure 2.2b), packets flow from a sender to a receiver, and vice versa. A two-way communication, or bidirectional messaging, is commonly used for distributed systems, in which its components are spread across multiple devices.

The pub/sub messaging model (Figure 2.2c) is another form of two-way communication that, despite also using the request-response pattern, has certain features that distinguishes
itself from its traditional client-server counterpart. Most notably, pub/sub communications are less isolated, meaning a client A is capable of sending messages to a client B, and vice versa. Clients interact with one another through the server (or broker), which redirects messages considering certain attributes such as topics of interest. The role of the broker in pub/sub communications is two-fold: (1) handle requests and (2) route messages to interested subscribers.

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.

Due to pub/sub messaging being a promising design pattern, several pub/sub protocols have been developed for low-powered devices in the IoT. Examples of pub/sub protocols include, but are not limited to, the Advanced Message Queuing Protocol (AMQP), the Data Distribution Service (DDS), the Extensible Messaging and Presence Protocol (XMPP), and MQTT. Among the aforementioned pub/sub protocols, MQTT is the most popular in IoT environments (See Appendix B for a comparison between MQTT and other IoT protocols).

### 2.2 Challenges

Traditional network protocol fuzzing is difficult (Pham et al., 2020), and follows the premise that the server interacts with only one type of client (or requester). Figure 2.3
presents the traditional approach for network protocol fuzzing, which involves the fuzzer exchanging messages with the server as if it were a normal client. The server perceives no difference whatsoever between a fuzzer and a legitimate client.

For most one- and two-way communication protocols, the traditional approach will suffice to cover all of their functionalities during testing. For pub/sub messaging systems, however, the traditional approach is unsuitable because it is designed with single-client protocols in mind. In fact, several research studies have raised awareness about the shortcomings of traditional protocol fuzzers for pub/sub brokers (Sneha Suhitha Galiveeti and Pranitha Malae, 2020; Zeng et al., 2020).

This concern is due in part to the literature lacking information about design considerations and challenges when developing a pub/sub protocol fuzzer. Most survey papers explain the common challenges of network protocol fuzzing, neglecting those associated with the unique attributes of the pub/sub design pattern. For example, Zhu et al. (2022) provide a roadmap for developers to grasp a better understanding of fuzz testing and its use across multiple domains such as network protocols. The authors mainly focus on file-transfer and cryptographic protocols, whereas pub/sub protocols were left out to limit the scope of their study. Their roadmap therefore provides insufficient information to developers interested in fuzzing pub/sub protocols.

Several other research papers focus specifically on two challenges regarding network protocol fuzzing (Munea, Lim, et al., 2016; Manès et al., 2019; Zhao, 2020; Liang et al., 2018; J. Li et al., 2018; Yurong Chen et al., 2019; Boehme et al., 2021): (1) highly-structured inputs and (2) state traversal. In terms of the former challenge, a network-based fuzzer must generate inputs that satisfy the syntax and structural requirements established by the protocol standard. Traditional fuzzers, such as American Fuzzy Lop (AFL) (Michael Zalewski, 2013), have difficulty meeting this criteria (Y. Li et al., 2021), forcing developers to resort to more specialized approaches for pub/sub protocols. In terms of the latter challenge, the success of state traversal depends on a fuzzer’s capability to generate highly-structured inputs, which must be sent in the correct order to guide the test run to a specific state. Although these are two of the most common challenges faced by developers when fuzzing network protocols, research studies fail to provide information regarding a third challenge: version support. Protocol implementations support several versions of a standard. For example, Mosquitto is a broker-side implementation that supports versions 3.1, 3.1.1, and 5.0 of the MQTT protocol. A fuzzer must therefore have an understanding of the input structure and state traversal for each version in order to increase its effectiveness. The effectiveness of protocol fuzzers therefore depends on their success to overcome all three of the aforementioned challenges (highly-structured inputs; state traversal; and version support).

Pub/sub protocols further complicate the testing process by locking its core features behind message subscriptions. This means that a pub/sub fuzzer must at the very least be
capable of creating test scenarios based on message publishing features. This entails that fuzzers act as both a publisher and a subscriber for each test scenario in order to trigger the broker into publishing messages. A pub/sub fuzzer therefore distinguishes itself from its counterparts by its dual-role testing strategy and heavy emphasis on topic generation, both of which must be performed systematically because of two reasons. First, a pub/sub fuzzer can publish messages only after it creates a subscription topic. Second, a pub/sub fuzzer needs to be able to generate topics that match across publication and subscription requests, otherwise the broker will not route messages to interested subscribers. Figure 2.4 presents an example in which a fuzzer fails to create a test scenario that involves both a publisher and a subscriber.

![Figure 2.4: Pub/Sub Protocol Fuzzing (2 Clients)](image)

In such a case, a fuzzer, in the role of a publisher, generates and sends random inputs to a broker, which never delivers messages to the subscribers. As a result, message publications are completely absent during testing, becoming a major hindrance towards effective pub/sub protocol testing. However, it is worth noting that the focus should not be strictly tied to publication and subscription requests during testing because specific pub/sub features may also depend on connection parameters established between the fuzzer and the broker.

### 2.3 Literature Review

Despite its importance for IoT brokers, pub/sub protocol fuzzing is still in its early stages of adoption. In fact, research on pub/sub protocol fuzzing has progressed slowly compared to other fields. For example, file fuzzing has advanced and improved drastically over the past decade, employing sophisticated algorithms based on the principles of natural selection and genetics to carefully select coverage-increasing test cases. This technique is commonly referred to as greybox fuzzing. AFL, in particular, is one of the most popular and successful greybox fuzzers for file-processing programs (Pham et al., 2020). The success of AFL led to the emergence of several extensions that add support for different types of programs such as compilers and, more recently, network protocols.

The main issue of AFL and most of its extensions is their limited support for the pub/sub design pattern (Zeng et al., 2020). This in turn has led to a lack of consideration for pub/sub target systems in popular benchmark suites such as ProFuzzBench (Natella and Pham, 2021), which at the time of writing supports only the client-server model. Over time, these issues have hindered the progression of pub/sub protocol fuzzing in two key aspects. From a technological point of view, the fuzzers, while advanced, are incapable of testing core features of the pub/sub design pattern such as message publication. From a developer’s point of view, there is a lack of information in the literature regarding effective strategies to properly fuzz a pub/sub protocol. As a result, developers have applied ineffective testing strategies for pub/sub protocols across several research studies. For example, Sneha Suhitha Galiveeti and Pranitha Malae (2020) create two test
scenarios in which the broker interacts with a publisher and a subscriber separately rather than collectively. Aljaafari et al. (2020) create test scenarios that involve only a publisher. Both of these research studies lack scenarios involving both a publisher and a subscriber, meaning message publishing features are most likely neglected during testing. In order to mitigate this issue, Zeng et al. (2020) propose MultiFuzz, which is an extension based on AFL that is designed to create test scenarios that involve publishing MQTT messages to interested subscribers. This makes MQTT the first and only pub/sub protocol supported by AFL.

Effort has been put into supporting MQTT because of its importance in the realization of Smart Cities. The popularity and adoption of MQTT for IoT applications has been unprecedented. In fact, MQTT has had a huge impact in real-world environments such as cloud computing (See Appendix A for further details). For example, AWS IoT 1, Google Cloud IoT Core 2, and Azure IoT Hub 3 connect IoT devices to cloud computing services via communication protocols such as MQTT.

Developers seem to prefer MQTT over other pub/sub protocols because of its simplicity and lightweight nature. For example, MQTT offers three levels of Quality of Service (QoS), whereas other pub/sub protocols such as the Data Distribution Service (DDS) offer twenty-three (see Appendix B, Table B.1 for further details). These and several other differences hinder the possibilities of developing a single generic fuzzer that is capable of testing every pub/sub protocol effectively. Developers are well aware of this difficulty, building fuzzers geared towards a specific pub/sub protocol. The most notable evidence of this trend has been the sheer number of fuzzers available specifically for MQTT. However, quantity does not equal quality, and in the case of MQTT, there is a lack of information regarding the testing capabilities of existing fuzzers for pub/sub functionalities. For developers, the main question is whether fuzzers are built and designed with the unique attributes of the pub/sub design pattern in mind (Zeng et al., 2020).

2.4 MQTT Fuzzing

Despite these concerns, research on pub/sub or MQTT fuzzing lacked considerably until studies found faulty or weak protocol implementations in real-world environments (Anantharaman et al., 2017; Alghamdi et al., 2018; J. Chen et al., 2018; Maggi et al., 2018; Palmieri et al., 2019). A manual study was later conducted to determine the root cause of vulnerabilities (Araujo Rodriguez and Macêdo Batista, 2020). The research study revealed that most MQTT-vulnerabilities are exploited because of malformed packets, which could allow information disclosure, remote code execution, and denial of service, thereby hindering confidentiality, integrity, and availability, respectively. In fact, a stack overflow vulnerability (CVE-2019-11779) 4 was discovered in 2019 by sending a crafted subscribe packet to the broker, as shown in Figure 2.5.

---

### 2.4 MQTT FUZZING

**Figure 2.5:** Network-Based Fuzzer Sending a Malicious Subscribe Packet to the Broker and Causing a Stack Overflow (CVE-2019-11779).

Since then, the situation remains the same. As of April 19th 2023, searching for keywords related to MQTT brokers (MQTT, mosquitto, mosca, Paho, ActiveMQ) in the NVD database yields one-hundred and twenty vulnerabilities, ninety-four of which are triggered by malformed packets. Among the remaining vulnerabilities, seven are related to cryptographic issues, whereas nineteen are triggered by improper authentication or configuration. As of April 19th 2023, malformed packets account for 78.33% of all MQTT-related vulnerabilities, and have consistently been prominent for the last few years, as shown in Figure 2.6.

For seven consecutive years, malformed packets account for more than 60% of vulnerabilities discovered annually, representing a major threat to MQTT implementations in real-world environments. More specifically, malformed packets accounted for 75%, 82.4%, 61.0%, 76.9%, 100%, 90.9%, and 77.8% of vulnerabilities discovered in 2017, 2018, 2019, 2020, 2021, 2022, and 2023 respectively. It is expected that this trend will continue in the future (Anantharaman et al., 2017), which is especially worrisome when considering that vulnerabilities are disclosed several days after their discovery (Araujo Rodriguez, Selvatici Trazzi, et al., 2018). Figure 2.7 presents disclosure delays of vulnerabilities related and unrelated to malformed packets.

The horizontal line within each rectangle indicates the median, which is 8 and 7 days for the former and latter type of vulnerability respectively. The upper quartiles denote the median of the larger disclosure delays, which are more common for vulnerabilities related to malformed packets. It is worth noting that although all types of vulnerabilities had similar disclosure delays on average, vulnerabilities related to malformed packets had twelve disclosures delayed for over 100 days, whereas their counterparts had only one. Moreover, there were exactly nineteen vulnerabilities related to malformed packets that were disclosed after seventy days, whereas their counterparts had only four cases. Based on these results, vulnerabilities triggered by malformed packets had higher disclosure delays than their counterparts.
However, regardless of disclosure delays, patching vulnerabilities found in brokers deployed to real-world environments is complex because of their limited hardware resources or location (Săndescu et al., 2018; Thantharate et al., 2019; Eceiza et al., 2021; Newman and Al-Nemrat, 2021; Husnain et al., 2022). As a result, it is common for brokers to use outdated versions of MQTT implementations (Araujo Rodriguez and Macêdo Batista,
2.4 | MQTT FUZZING

For example, searching for mqtt on the Shodan search engine yields 52,411 MQTT brokers accessible from the Internet, with Mosquitto ranking as the most popular. Figure 2.8 presents the top eleven versions of Mosquitto accessible from the Internet as of April 19th, 2023.

Figure 2.8: Outdated Brokers Accessible (As of April 19th, 2023)

Assuming that version numbers are incremented whenever a patch is applied, the results presented in Figure 2.8 confirm the lack of updates applied to MQTT brokers. This situation is especially worrisome when considering that it is common for multiple versions of Mosquitto to be released on the same day (Table 2.1), further reducing the likelihood of IoT devices being up to date.

2.4.1 Related Work

Over the last few years, several fuzzing frameworks have been proposed to mitigate these issues. In fact, MQTT is the only pub/sub protocol for which there have been a multitude of fuzzing techniques proposed in the literature. As of April 19th 2023, searching for mqtt fuzzers using Google Scholar yields several publications about fuzzing techniques for MQTT. These research papers have been published in a wide variety of sources, ranging from conference workshops to peer-reviewed journals. More specifically, publications about MQTT fuzzers were mainly found in 6 conferences and 2 journals. The 6 conferences are as follows: ACM International Conference on Availability, Reliability, and Security; IEEE Conference on Software Testing, Verification, and Validation; IEEE International Symposium on Computers and Communications; IEEE International Wireless Communications and Mobile Computing; IEEE World Congress on Services; and Italian Conference on Cybersecurity. The 2 journals are Sensors; and Wireless Communications and Mobile Computing.

5 https://www.shodan.io/. Accessed on April 19th, 2023
There are also remote-hosting services such as GitHub that provide access to MQTT fuzzers based on different techniques than those proposed in the literature. As of April 19th, 2023, searching for mqtt fuzzers on GitHub yields over 10 fuzzing frameworks specifically designed for MQTT. For this research, the search results were narrowed by considering high-ranking MQTT fuzzers on GitHub that were either developed by corporations or published in peer-reviewed sources. High-ranking repositories on GitHub are determined by the number of stars. Three MQTT fuzzers were found that meet this criteria: Scapy’s fuzz() function (Philippe Biondi and the Scapy Community, 2023), mqtt_fuzz (F-Secure Corporation, 2015) and IoT-Testware (Eclipse Foundation, 2018). Table 2.2 presents URLs for every open source MQTT fuzzer found on Google Scholar and GitHub. The column Published highlights the fuzzers that were peer-reviewed and accepted for a journal or conference publication.

All of the fuzzers shown in Table 2.2 use different strategies for testing purposes. Thus, Table 2.3 further classifies each MQTT fuzzer into two categories: (1) understanding the
target broker; and (2) test case generation technique.

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>GitHub Repository</th>
<th>Year</th>
<th>Last Updated</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqtt_fuzz</td>
<td><a href="https://github.com/F-Secure/mqtt_fuzz">https://github.com/F-Secure/mqtt_fuzz</a></td>
<td>2015</td>
<td>March 21, 2022</td>
<td></td>
</tr>
<tr>
<td>Polymorph</td>
<td><a href="https://github.com/shramos/polymorph">https://github.com/shramos/polymorph</a></td>
<td>2018</td>
<td>April 11, 2022</td>
<td>✔</td>
</tr>
<tr>
<td>IoT-Testware</td>
<td><a href="https://github.com/eclipse/iottestware.fuzzing">https://github.com/eclipse/iottestware.fuzzing</a></td>
<td>2018</td>
<td>October 15, 2019</td>
<td></td>
</tr>
<tr>
<td>MQTTSA</td>
<td><a href="https://github.com/stfbk/mqttsa">https://github.com/stfbk/mqttsa</a></td>
<td>2019</td>
<td>March 9, 2022</td>
<td>✔</td>
</tr>
<tr>
<td>CyberExploit</td>
<td><a href="https://github.com/CyberExploitProject/CyberExploit">https://github.com/CyberExploitProject/CyberExploit</a></td>
<td>2020</td>
<td>August 9, 2020</td>
<td>✔</td>
</tr>
<tr>
<td>MultiFuzz</td>
<td><a href="https://github.com/bduosoftsec/MultiFuzz">https://github.com/bduosoftsec/MultiFuzz</a></td>
<td>2020</td>
<td>September 3, 2021</td>
<td>✔</td>
</tr>
<tr>
<td>FUME</td>
<td><a href="https://github.com/PBearson/FUME-Fuzzing-MQTT-Brokers">https://github.com/PBearson/FUME-Fuzzing-MQTT-Brokers</a></td>
<td>2022</td>
<td>March 6, 2023</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Open Source MQTT Fuzzers

<table>
<thead>
<tr>
<th>Existing Frameworks</th>
<th>Understanding Target Broker</th>
<th>Test Case Generation Technique</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuzz()</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>mqtt_fuzz</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Polymorph</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>IoT-Testware</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>MQTTSA</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>MultiFuzz</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>AFLNet-MQTT</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Anantharaman et al (2017)</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>CyberExploit</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Sochor et al (2020a)</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Di Paolo et al (2021)</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Aichernig et al (2021)</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>Defensics</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
<tr>
<td>FUME</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 2.3: Related Work

The first category refers to a fuzzer’s awareness of the internals of a broker. In that regard, a blackbox fuzzer is completely unaware of the internals of the broker. No program analysis is used to generate the test cases. A greybox fuzzer is partially aware of the internals of the broker. New test cases are generated based on feedback received during testing, such as the number of lines or statements executed in the source code (statement coverage). A whitebox fuzzer is completely aware of the internals of the broker. In terms of the second category, a naive fuzzer generates test cases without considering the formal specifications of the protocol. Mutation-based fuzzers introduce small changes to existing test cases, whereas generation-based fuzzers generate test cases from scratch. A hybrid fuzzer combines aspects of both mutation- and generation-based fuzzers to generate the test cases. For example, it generates test cases from scratch; mutates them; generates new test cases, and so forth. The following subsections are organized based on the second category, explaining each MQTT fuzzer in depth.

Naive

Recent works have used Scapy to generate malformed network packets (Melo and Geus, 2017; Eclipse Foundation, 2018; Hernández Ramos et al., 2018). Network packets can be crafted from scratch and fuzzed using Scapy’s `fuzz()` function

---

classified as a *naive fuzzer* because it lacks knowledge of protocol specifications. Scapy’s `fuzz()` function generates packets that are rejected by the broker almost immediately, and thus has difficulty reaching deep protocol states.

**Mutation**

**F-Secure Corporation (2015)** proposed *mqtt-fuzz*, a mutation-based fuzzer whose goal is to be user friendly for testing purposes. *mqtt-fuzz* is currently one of the most popular MQTT fuzzers (*Hernández Ramos et al., 2018; Palmieri et al., 2019*), renowned recently for finding a flaw in a message broker (*Kwon et al., 2021*). However, in its current form, *mqtt_fuzz* is unaware of certain features and syntax rules defined in the standard, meaning it may (1) occasionally craft syntactically invalid inputs; (2) require considerable amount of test cases to reach deep protocol states; and (3) cover limited functionality of MQTT.

**Hernández Ramos et al. (2018)** proposed Polymorph, a template-based fuzzer that aims to reduce user effort. The architecture of Polymorph consists of three modules. The *sniffer module* captures network packets exchanged between the client and the broker. The *template module* generates templates based on network packets captured by the sniffer module. The templates enable the *fuzzer module* to perform three tasks: (1) mutate user-selected fields; (2) recalculate field attributes to maintain packet consistency; and (3) send mutated packets to the broker. Despite having low CPU consumption, Polymorph lacks syntactic knowledge about MQTT messages. Moreover, the test effectiveness of Polymorph depends on packets captured by the sniffer module, which may not understand all of MQTT’s functionality.

Similar to *Hernández Ramos et al. (2018)*, **Eclipse Foundation (2018)** proposed IoT-Testware, a fuzzer that is incapable of generating MQTT messages on its own, requiring valid templates to gain program knowledge. Moreover, the fuzzer’s mutation strategy is based on rules predefined by the user, which requires a deep understanding of MQTT to configure the tests effectively.

**Palmieri et al. (2019)** proposed MQTTSA, a tool that detects security misconfigurations in MQTT brokers, and then suggests mitigation strategies to users. MQTTSA consists of several penetration-testing mechanisms, including a data tampering module that mutates and sends packets to MQTT brokers. However, a recent study by **Araujo Rodriguez and Macêdo Batista (2020)** showed that MQTTSA has low statement coverage during testing.

**Zeng et al. (2020)** proposed MultiFuzz, a coverage-based fuzzer specifically designed for multi-party protocols. The main feature of MultiFuzz is its ability to support multiple connections during the fuzzing campaign, enabling it to act as multiple clients. In contrast to other coverage-guided approaches such as AFL and AFLNet (*Pham et al., 2020*), MultiFuzz can act as both a publisher and subscriber, thus being capable of generating test cases that cover the main functionalities of the pub/sub design pattern. MultiFuzz has higher *path coverage* than generic fuzzers such as AFL, MOPT, and AFLNet. For this thesis, *path coverage* is defined as the number of paths executed in the source code. However, at the time of writing, its coverage performance against fuzzers designed specifically for MQTT is unknown.
Sneha Suhitha Galiveeti and Pranitha Malae (2020) proposed AFLNet-MQTT, a coverage-guided approach based on AFLNet for MQTT. The architecture of AFLNet-MQTT is mostly the same as AFLNet, with the only two differences being the request and response sequence parser, which are slightly modified to support MQTT. AFLNet-MQTT offers both a stateless and stateful mode of execution. Sneha Suhitha Galiveeti and Pranitha Malae (2020) evaluate AFLNet-MQTT using three metrics: number of hangs, number of crashes, and path coverage. Despite testing Mosquitto for approximately 20 hours, AFLNet-MQTT was incapable of triggering hangs or crashes in stateless mode (Sneha Suhitha Galiveeti and Pranitha Malae, 2020). However, AFLNet-MQTT manages to produce better results in stateful mode, triggering 29 unique hangs within 18 hours. Regardless of the execution mode, AFLNet-MQTT underperforms considerably in terms of path coverage, achieving at most 3.75%.

Generation

Anantharaman et al. (2017) developed MQTT parsers that recognize valid and invalid messages, thereby avoiding potential security vulnerabilities. A parser and input language were developed for specific MQTT messages. The message is processed only if it is recognized by the parser, otherwise it is discarded. The authors developed a generation-based fuzzer to test their parsers. The main limitations of this research are as follows. First, their fuzzer only sends MQTT messages to the broker in correct order, meaning incorrect functionality is neglected during testing. Second, more focus is given to the parsers than their fuzzer’s input generation capabilities.

Casteur et al. (2020) demonstrated the effectiveness of Docker containers7 to fuzz MQTT. For their fuzzer, the authors opted to define packet fields manually or randomly, rather than systematically using a grammar. As a result, their fuzzer does not cover all of MQTT’s functionality during testing. For example, analyzing their fuzzer’s packet generator8 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 standard, passwords are optional if the User Name Flag is set to 1.

Sochor et al. (2020a) proposed a fuzz testing architecture for MQTT. The architecture mainly consists of a test case generator and a test adapter. The test cases are generated using Randoop, which is an online random test case generator. The test adapter is responsible for the following tasks. First, it handles messages exchanged between the client and the broker. Second, it sends packets based on existing vulnerabilities to the MQTT broker. Although the architecture proved to be effective, the test case generator is not designed with MQTT’s complex packet syntax or structure in mind, increasing the likelihood of it crafting invalid messages more frequently.

Di Paolo et al. (2021) performed fuzz testing on five broker implementations and three client implementations of MQTT. The authors developed a fuzzing framework to detect undefined behaviors in MQTT implementations. The fuzzing framework receives a JSON

---

file as input, which specifies the message sequence for each test scenario. The behavior of
the target system was monitored when handling long message topics, incorrect packet
sequences/fields, and delayed transmissions. Despite detecting potential weaknesses in
MQTT implementations, the authors failed to provide information regarding their fuzzer’s
statement or path coverage.

Aichernig et al. (2021) introduced a tool that uses a technique based on automata
learning to automatically infer the message syntax and sequence of a network protocol.
The learned model is then used to generate test cases for the fuzzing campaign. The authors
performed a case study on five MQTT implementations to validate their learning-based
technique. According to the authors, the learning-based fuzzing approach found inconsis-
tencies in five MQTT implementations, proving to be effective. The fuzzing framework
supports only version 5.0 of the MQTT protocol, which is considerably less popular than
version 3.1.1 (Di Paolo et al., 2021).

Hybrid

Synopsis (2021) developed Defensics, which is a hybrid fuzzer for multiple domains,
including interfaces, files, and network protocols. Defensics allows users to: (1) customize
pre-built test cases, or (2) write custom-made test cases using a Software Development Kit
(SDK). Defensics is updated regularly to support new protocol specifications. However, at
the time of writing, Defensics lacks support for the latest version (5.0) of MQTT. Further
details of Defensics’ test generation capabilities are unknown due to its proprietary
nature.

Pearson et al. (2022) proposed FUME, a hybrid fuzzer that uses Markov modeling and
finite state machines to make several choices throughout the fuzzing campaign, such as
selecting between mutation- and generation-based approaches.
Chapter 3

A Grammar-Based Fuzzing Technique for a Pub/Sub Protocol

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 (Philippe Biondi and the Scapy Community, 2023), lack program knowledge, meaning that fuzzed packets may be immediately rejected by the broker, and thus deep protocol states are difficult to reach during testing. Although injecting abnormal packets is effective, it is important to adhere to protocol specifications.

Existing mutation-based fuzzers (F-Secure Corporation, 2015; Hernández Ramos et al., 2018; Eclipse Foundation, 2018), 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 (Casteur et al., 2020; Sochor et al., 2020b), have limited knowledge of MQTT’s specifications, and thus have difficulties to cover all of its features.

All of the aforementioned problems stem from the lack of a systematic and effective input generation scheme, which is a current research challenge for several target systems (C. Chen et al., 2018) including pub/sub protocols. Among several schemes, grammar-based fuzzing is considered to be the most effective for applications with complex input structures (Godefroid et al., 2017). Grammar-based fuzzers, as their name suggests, gain knowledge from grammars, which are among the most reliable sources for knowledge acquisition.
Thus far, research on grammar-based fuzzing for MQTT or any other pub/sub protocol is currently non-existent and unexplored. Developers have opted for alternatives to grammar-based approaches throughout the years due to their complicated and time-consuming nature (Hernández Ramos et al., 2018). For example, Aichernig et al. (2021) propose an automata-learning technique to infer the message syntax of MQTT without the need for a grammar. Hernández Ramos et al. (2018), Zeng et al. (2020), Casteur et al. (2020), Di Paolo et al. (2021), and Pearson et al. (2022) provide their MQTT fuzzers with templates for them to gain an understanding of the message syntax, thereby avoiding grammars altogether.

Despite their complexity, grammars offer several advantages over other knowledge acquisition sources: (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 broker; (6) higher feature coverage of the protocol; and (7) systematic and efficient test generation.

Grammar-based approaches for MQTT are not only necessary, but also demanded across several research studies (Hernández Ramos et al., 2018; Sochor et al., 2020b). Although developers and researchers have expressed interest, the literature lacks information about how to apply grammar-based fuzzing to pub/sub protocols. Thus, the main question worth answering is:

\textbf{RQ1: How can a grammar-based fuzzer for a pub/sub protocol such as MQTT be developed?}

The following sections propose a new methodology and an architecture in order to develop a grammar-based fuzzer for a pub/sub protocol such as MQTT.

\section{Architecture}

The grammar-based approach is incorporated into a new network-based fuzzer called \texttt{MQTTGRAM}. The architecture of \texttt{MQTTGRAM} consists of two major components, which are the Response Engine and the Packet Generator, as shown in Figure 3.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.1.png}
\caption{Architecture of \texttt{MQTTGRAM}}
\end{figure}
The response engine and packet generator work collectively to fuzz a pub/sub broker. The response engine is mainly responsible for supporting a two-way communication between the fuzzer and the broker. The response engine consists of two subcomponents. The first subcomponent, TCP State, handles the TCP sequence and acknowledgment numbers for each packet sent to the broker. The second subcomponent, MQTT State, handles and analyzes messages received from the broker in order to identify which state of the protocol has been reached at a particular moment during testing. This information is then provided to the packet generator, which generates a response message to guide the fuzzing process to the next state considering the sequence defined by the standard (See Appendix C, Section C.2 for further details). The messages are in hexadecimal notation, which is often used in networking to represent packet formats. The packet generator of MQTTGRAM generates hexadecimal strings from a grammar considering the format defined by the MQTT 3.1.1 standard (Andrew Banks and Rahul Gupta, 2014). The hexadecimal string is then sent to the broker as an MQTT packet. A simple example of the MQTT 3.1.1 grammar and its expansion rules is shown below:

\[
\begin{align*}
\langle \text{start} \rangle & ::= \langle \text{packets} \rangle \\
\langle \text{packets} \rangle & ::= \langle \text{publish} \rangle \\
\langle \text{publish} \rangle & ::= \langle \text{fixed-header} \rangle \langle \text{variable-header} \rangle \\
& \quad | \langle \text{fixed-header} \rangle \langle \text{variable-header} \rangle \langle \text{payload} \rangle \\
\langle \text{fixed-header} \rangle & ::= \text{‘Hexadecimal string of fixed header’} \\
\langle \text{variable-header} \rangle & ::= \text{‘Hexadecimal string of variable header’} \\
\langle \text{payload} \rangle & ::= \text{‘Hexadecimal string of payload’}
\end{align*}
\]

**Figure 3.2: Simple Example of the MQTT 3.1.1 Grammar in Backus-Naur Form**

Nonterminal symbols are on the left and enclosed between angle brackets (<>). Substitution options or expansion rules for the nonterminal symbol are on the right. Expansion rules describe how to replace nonterminals with terminal symbols, which can not be substituted or expanded any further. The end result is a string that consists entirely of terminal symbols, which in this case are hexadecimal digits representing an MQTT packet. The proposed grammar has a nonterminal symbol <packets> that describes expansion rules for each MQTT packet. The selection of the expansion rule depends on the packet required by the response engine to reach the next protocol state. For example, if the response engine requires a publish packet, then the nonterminal symbol chosen to be expanded is <publish>. The algorithm then randomly selects a production rule to generate a publish packet either with or without a payload. It is worth noting that network packets are usually divided into three fields: (1) the fixed header; (2) the variable header; and (3) the payload (See Appendix C, Section C.1 for further details). Following this premise, the grammar developed for MQTTGRAM has production rules to generate each field accordingly. The end result of the proposed grammar is the concatenation of hexadecimal strings for the fixed header, variable header, and payload, which together make up an MQTT 3.1.1 message.
3.2 Algorithms

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

**Algorithm 1:** Grammar-Based Fuzzing Approach for MQTT

**Input:** Protocol State \( S \)

1. while \( \text{stopping}_\text{criterion} == \text{false} \) do
2. \( t \leftarrow \text{select}_\text{packet}_\text{type}(S) \);
3. \( m \leftarrow \text{generate}_\text{packet}(t) \);
4. \( r \leftarrow \text{feed}_\text{input}(m) \);
5. \( S \leftarrow \text{evaluate}_\text{response}(r) \);
6. \( L \leftarrow \text{monitor}_\text{program}(S) \)

**Output:** \( L \)

3.2.1 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 (See Appendix C, Table C.1 for further details). For example, if an MQTT client receives a CONNACK packet from the broker, then the client is connected.

MQTTGRAM uses an approach that selects and generates packets based on the current broker state. For example, at the beginning of the test run, a fuzzer based on the proposed approach is in a disconnected state because it has not established a connection with the broker. Thus, it performs a Transmission Control Protocol (TCP) handshake and sends a CONNECT packet to the broker. If MQTTGRAM receives a CONNACK packet from the broker, then it has successfully transitioned to a connected state.

3.2.2 Generating the Packets

Protocol fuzzers need to possess knowledge of both the input structure and states, maintaining execution consistency throughout the fuzzing campaign (Yurong Chen et al., 2019). It is not enough to simply gain program knowledge through a grammar. The fuzzer needs to be state-aware (Kitagawa et al., 2010), 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.

**Derivation of Hexadecimal Strings**

Algorithm 2 presents the pseudocode for the derivation of hexadecimal strings from an MQTT grammar. The process of choosing a nonterminal and applying an expansion rule is adapted from (Zeller et al., 2020) considering the MQTT standard.
Algorithm 2: Derivation of Hexadecimal String from an MQTT Grammar

Input: Packet Type \( t \), Grammar \( g \)

1. while nonterminals from \( t > 0 \) do 
2. \[ n \leftarrow \text{choose_nonterminal}(t, g); \]
3. \[ e \leftarrow \text{choose_expansion_rule}(n, g); \]
4. \[ s \leftarrow \text{apply_expansion}(e, n); \]
5. for field_lengths in \( s \) do
6. \[ \text{calculate(field_lengths)}; \]
7. \[ p \leftarrow \text{convert_string_to_bytes}(s); \]
8. \[ m \leftarrow \text{calculate_remaining_length}(p); \]

Output: \( m \)

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 UTF-8 encoded strings correctly. Thus, when there are no further expansions, Algorithm 2 calculates the length of each UTF-8 encoded string and remaining length (lines 5-8).

Figure 3.3 presents an example of Algorithm 2 when generating a PUBLISH packet.

The steps performed by Algorithm 2 to generate the hexadecimal string that represents a PUBLISH packet, shown in Figure 3.3, are as follows.

Step 1: The derivation process begins with a start symbol \(<\text{start}>\), which is expanded
into the nonterminal symbol <packets>. The symbol <packets> has fourteen expansion alternatives, each one being a type of MQTT packet. In this case, the nonterminal symbol <publish> is chosen to be expanded. The symbol <publish> has four expansion rules. However, Figure 3.3 presents only one of these expansions rules (\x3<publish_fixed_header_qos0><publish_variable_header_qos0>) for simplicity purposes. The expansion rule shown in Figure 3.3 is for PUBLISH packets whose QoS level is set to 0, and the payload is absent. The expansion rule is divided into three fields in Figure 3.3 to better illustrate the methodology. The first field \x3 is a terminal symbol that consists of three characters: \x mean that digits are in hexadecimal; and 3 represents the value of a PUBLISH packet according to the MQTT 3.1.1 standard. The second field <publish_fixed_header_qos0> is expanded into two nonterminal symbols: <publish_reserved_qos0> and <remaining_length>. The symbol <publish_reserved_qos0> is substituted for 1, which signifies that the Retain Flag in the packet is set to 1. The symbol <remaining_length> is substituted for \t, which is a temporary symbol that will be replaced with the remaining length in Step 4. The third field <publish_variable_header_qos0> is also expanded into two nonterminal symbols: <string-length> and <topic-name>. Similar to <remaining_length>, the symbol <string-length> is substituted for \n\n, which will be replaced with the actual length of <topic-name> in Step 3. The symbol <topic-name> is expanded into the nonterminal symbol <utf8-characters>, which currently has four expansion alternatives: <utf8-numbers>, <utf8-latin-capitalletters>, <utf8-latin-smallletters>, <utf8-symbols>. In this case, Algorithm 2 randomly chooses <utf8-latin-smallletters>, which is substituted for \x6d, representing the letter m. It is worth noting that the grammar is recursive, meaning nonterminal symbols such as <utf8-latin-smallletters> or <topic-name> can be expanded an infinite number of times, yielding complex inputs.

**Step 2:** When all nonterminals have been expanded, a hexadecimal string is produced.

**Step 3:** Using regular expressions, all UTF-8 encoded strings are retrieved by searching for \n\n, and their length is calculated.

**Step 4:** The remaining length of the packet is calculated, hence the importance of calculating the length of all UTF-8 encoded strings in Step 3.

**Step 5:** When all necessary field lengths are calculated, the derived output is a byte string that represents an MQTT packet.

### 3.2.3 Feeding the Packets

MQTTGRAM interacts directly with the broker rather than acting as a man-in-the-middle. This approach provides more control over the packets that are generated during the test run. Rather than requiring initial test cases, MQTTGRAM generates its own network packets to communicate with the broker successfully.
3.3.4 Evaluating the Response

MQTTGRAM sends 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), MQTTGRAM reconnects to the broker automatically; otherwise MQTTGRAM analyzes the type of MQTT packet, and responds accordingly. For example, if the broker sends a PUBLISH packet, MQTTGRAM responds by sending a PUBACK packet or PUBREC packet when the QoS level is set to 1 and 2 respectively.

3.3.5 Monitoring the Program

The fuzzing process is monitored constantly for any crashes or errors until a stopping criterion has been satisfied. Logs are used to store information about the test run.

3.3 Performance Evaluation

The performance of the proposed grammar-based approach is compared with that of a naive fuzzer (fuzz()), a mutation-based fuzzer (mqtt_fuzz), and a generation-based fuzzer (Sochor et al., 2020b). fuzz() and 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. (2020b) was chosen for its promising vulnerability-oriented approach, however it is proprietary, meaning it could not be evaluated on the testbed developed for the doctoral studies. As a result, the grammar-based approach was evaluated in the same manner as the fuzzer by Sochor et al. (2020b) for comparison purposes.

Two variants of MQTTGRAM were developed in order to evaluate how probabilistic settings for control packet types affect the performance of the proposed grammar-based approach. 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, each PUBLISH, SUBSCRIBE, and UNSUBSCRIBE packet has 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, whereas 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.

The effectiveness of the proposed grammar-based approach is evaluated using two metrics: (1) statement coverage and (2) input coverage, the latter of which is considered indispensable when developing grammar-based approaches (Havrikov and Zeller, 2019). Mosquitto and Moquette were chosen as the target systems because they rank among the most popular MQTT brokers (Hernández Ramos et al., 2018; Sochor et al., 2020b). Two separate experiments were conducted, each satisfying a different stopping criterion. For the first experiment, the fuzzers perform 3- and 30-minute test runs. For the second experiment, the fuzzers test the brokers until 500 and 8000 packets have been exchanged between both parties. The testbed shown in Figure 1.3, and explained in Section 1.2.2 was
used for the performance evaluation. It is worth noting that Sochor et al. (2020b) chose Moquette as the target system to evaluate their fuzzer, hence its absence in figures based on Mosquitto.

### 3.3.1 Mosquitto

Figure 3.4a presents the statement coverage achieved by the fuzzers in 3 minutes. `fuzz()` has the lowest statement coverage, executing at most 1125 statements and at the very least 1083. `mqttgram-c` manages to have higher statement coverage than `fuzz()`, executing at most 2017 statements and at least 1799. Thus, `mqttgram-c` performs at most 59.91% better because it has knowledge of MQTT’s specifications. `mqtt_fuzz` manages to execute at most 2069 statements, performing 83.91% and 2.58% better than `fuzz()` 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 `fuzz()` respectively.

![Figure 3.4: Results of 3-Minute Test Runs](image)

`mqtt_fuzz` manages to have better statement coverage overall because it exchanges a considerable amount of packets with the broker, as shown in Figure 3.4b. On average, `mqtt_fuzz` and Mosquitto exchange 20111 packets, approximately 544.79%, 765.36%, and 783.61% more than `fuzz()`, `mqttgram-c`, and `mqttgram-cf` respectively. Since `mqtt_fuzz` sends more packets to Mosquitto, there is a higher probability that more code paths are traversed during a short amount of time (3 minutes). However, `mqttgram-cf` is more effective than its counterpart, executing 2083 statements while exchanging only 2,258 packets. `mqttgram-cf` has higher statement 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. `fuzz()` manages to send more packets to Mosquitto than `mqttgram-c` and `mqttgram-cf` within the same time frame. This is because `fuzz()` generates random packets without considering the MQTT standard.
For simplicity purposes, the following figures do not present the results of `fuzz()` because it achieves low performance in terms of statement coverage.

Although `mqtt_fuzz` has the highest statement coverage overall in 3 minutes, a different outcome occurs in 30 minutes (Figure 3.5a). Among all MQTT fuzzers, `mqttgram-cf` has the highest statement coverage, executing 2140 statements at most, which is approximately up to 90%, 1.51%, and 0.99% more than `fuzz()`, `mqttgram-c`, and `mqtt_fuzz` respectively. It is worth noting that on average `mqttgram-cf` and Mosquitto exchange the lowest number of packets (Figure 3.5b), approximately 22995; thus confirming the effectiveness of the proposed grammar-based approach.

![Statement Coverage (30 minutes)](image)

![Packet Exchange (30 minutes)](image)

**Figure 3.5: Results of 30-Minute Test Runs**

Over time, both variants of MQTTGRAM have higher statement 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`: 218557; `mqttgram-c`: 23364; and `fuzz()`: 30993.

Figure 3.6a presents the input coverage achieved by the MQTT fuzzers when exchanging exactly 500 packets with the broker. `mqtt_fuzz` has the highest coverage performance, executing 2005 statements, `mqttgram-c`, `mqttgram-cf`, and `fuzz()` execute at most 1961, 1971, and 1118 statements respectively. Although variants of MQTTGRAM have lower coverage than their counterparts, exchanging 8000 packets with Mosquitto has a different outcome (Figure 3.6b). On average, `mqttgram-c` and `mqttgram-cf` execute 2009 and 2060 statements respectively. The results of both variants are higher than their counterparts, confirming that they test more functionality of MQTT effectively.

### Moquette

This section presents a performance comparison between MQTTGRAM and the fuzzer proposed by Sochor et al., 2020b, which also developed two variants of their approach.
The first variant generates packets that are not based on existing vulnerabilities of MQTT, whereas the second variant generates packets based on a list of twenty-five cyberattacks. `mqttgram-cf` is classified as being "vulnerability-oriented" in Table 3.1 because it is designed to send packets with topic fields much more frequently. This design choice is based on the fact that several implementation flaws in MQTT have been uncovered with topic-based packets (Di Paolo et al., 2021).

Since Sochor et al. (2020b) measure statement coverage for each source code directory of Moquette, the performance of MQTTGRAM is evaluated in the same manner to enable a comparison between the two fuzzers. Table 3.1 presents the statement coverage achieved by the two variants of MQTTGRAM and the fuzzer proposed by Sochor et al. (2020b) during 30 minutes.

<table>
<thead>
<tr>
<th>Source Code (Directory)</th>
<th>Fuzzed packets not based on existing vulnerabilities</th>
<th>Fuzzed packets based on existing vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sochor et al. 2020</td>
<td>mqttgram-c</td>
</tr>
<tr>
<td>broker</td>
<td>57.8%</td>
<td>66.34%</td>
</tr>
<tr>
<td>broker.security</td>
<td>11.3%</td>
<td>15%</td>
</tr>
<tr>
<td>broker.subscriptions</td>
<td>59.0%</td>
<td>73.52%</td>
</tr>
<tr>
<td>broker.config</td>
<td>49.3%</td>
<td>51%</td>
</tr>
<tr>
<td>broker.metrics</td>
<td>77.3%</td>
<td>73%</td>
</tr>
<tr>
<td>persistence</td>
<td>9.1%</td>
<td>9%</td>
</tr>
<tr>
<td>interception</td>
<td>32.3%</td>
<td>30%</td>
</tr>
<tr>
<td>Logging</td>
<td>62.5%</td>
<td>43%</td>
</tr>
<tr>
<td>Total</td>
<td>50.0%</td>
<td>56.76%</td>
</tr>
</tbody>
</table>

Table 3.1: Statement Coverage of Moquette Achieved by MQTT Fuzzers in 30 Minutes (Results by Sochor et al. (2020b) were copied directly from their research paper due to their fuzzer being proprietary)

It is worth noting that the fuzzer proposed by Sochor et al. (2020b) is not publicly available. Thus, the performance of MQTTGRAM is compared with results shown in their research paper (Sochor et al., 2020b). Moreover, results by Sochor et al. (2020b) are based
on a single thirty-minute fuzzing campaign, whereas the results of MQTTGRAM in Table 3.1 are based on the average of a hundred test runs.

Table 3.1 highlights in bold the highest statement coverage for each directory of Moquette. Both variants of MQTTGRAM have the highest statement 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. (2020b). Moreover, although Sochor et al. (2020b) 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.

3.3.2 Takeaway from Experiments

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 no crashes were triggered during the test runs.

After developing and evaluating MQTTGRAM, the answer to RQ1 is as follows:

RQ1: How can a grammar-based fuzzer for a pub/sub protocol such as MQTT be developed?

Network protocols support different types of packets, each usually consisting of a fixed header, a variable header, and a payload. For the grammar, production rules should be defined in accordance with the structure of a packet. For example, each type of packet should have its own designated nonterminal symbol (<publish>), which then should be substituted with three nonterminal symbols, each representing a layer of the packet (<fixed-header><variable-header><payload>). A production rule has to be created for each optional field. For example, suppose the payload of a packet is optional. Thus, two production rules will need to be created for each scenario: (<fixed-header><variable-header><payload>) and (<fixed-header><variable-header>). For variable-length fields, recursive production rules are needed to generate complex values. Among variable-length fields, subscription topics are one of, if not, the most important to test pub/sub protocols. A compromise has to be made for production rules to generate unexpected, but semantically-meaningful topics that activate and test message publishing features. A strategy to overcome this challenge is to limit the character set in order to increase the chances of generating the same topics across publication and subscription requests. However, the derivation process by itself is insufficient to test message publishing features because the fuzzer has to interact correctly with the broker in order to receive subscription messages. Thus, a response engine and a packet generator should operate collectively during a test run.
3.4 Threats to Validity

**Hardware:** Sochor *et al.* (2020b) failed to provide any information whatsoever regarding their testbed’s hardware configuration, which may or may not differ from that of Figure 1.3. In terms of the former case, different hardware configurations may provide an unfair performance comparison between the fuzzer proposed by Sochor *et al.* (2020b) and its competitors. This threat to the validity of the experiments is minimized, however, by using the same stopping criterion (30 minutes), metric (statement coverage), and target broker (Moquette 0.13) chosen by Sochor *et al.* (2020b) for their research study.

**Metrics:** Due to the absence of crashes, the effectiveness of the fuzzers was measured and determined using statement coverage, which is a metric that indicates how likely an implementation flaw will be discovered during a test run. However, statement coverage provides no guarantees that a fuzzer will actually uncover an implementation flaw in a target broker.

**Time:** In this research study, the fuzzers test the brokers for up to 30 minutes, which is the same stopping criterion used by Sochor *et al.* (2020b) for their experiments. The reason for using the same stopping criterion is to provide a more fair and consistent comparison between results generated by Sochor *et al.* (2020b) and those generated by the testbed presented in Figure 1.3. However, the coverage performance of the fuzzers after 30 minutes is unknown, meaning that there is no way to guarantee that the number of statements will either be the same or improve over longer periods of time.

**Repetitions:** Experiment results generated by the testbed shown in Figure 1.3 are based on one hundred test runs, whereas those provided by Sochor *et al.* (2020b) are based on only one. The different number of test runs between both research studies is a potential threat to the validity of the performance analysis and comparison presented in Table 3.1.

3.5 Concluding Remarks

Existing fuzzing frameworks lack knowledge of MQTT’s specifications, hindering their testing capabilities. As a result, several research papers have expressed interest in grammar-based approaches, however the literature lacks information on how to apply them to pub/sub protocols such as MQTT. This research aimed to address both of these limitations by presenting a grammar-based approach that aims for high code and state coverage. Based on our experiments, the grammar-based approach implemented in MQTTGram has higher statement coverage than existing MQTT fuzzers. Thus, an important contribution of this research is the confirmation that grammar-based approaches for pub/sub protocols are not only viable, but also effective and promising for testing purposes. Furthermore, the design considerations, architecture, methodology, and even challenges presented for this research study can help developers to build grammar-based fuzzers for other pub/sub protocols.
Chapter 4

Taxonomy and Coverage Evaluation of Fuzzing Techniques for the MQTT Protocol

While developing the new grammar-based fuzzing technique for MQTT, four additional research gaps were identified in the literature regarding pub/sub protocol fuzzing:

Gap 1: Most research studies focus on the fuzzers’ bug-finding capabilities rather than their coverage performance Although bug-finding is an important metric during testing, it fails to provide sufficient feedback on how thoroughly the tests are performed by the fuzzers. Several test coverage metrics are unused or neglected in most research studies, such as statement coverage. For example, despite being among the most popular and accessible fuzzers, mqtt_fuzz was evaluated in terms of statement coverage only until recently (Araujo Rodriguez and Macêdo Batista, 2021). The literature also currently lacks information regarding the fuzzers’ shortcomings in terms of feature and functionality testing. This poses the question as to whether developers and existing fuzzers, regardless of their testing technique, are considering the unique attributes of the pub/sub messaging pattern. The issue is further aggravated by proprietary fuzzers such as Defensics, which despite being popular, remains largely unstudied from a scientific perspective, with limited information regarding its testing or coverage capabilities. As a result, unknown testing capabilities currently hinder further advancements in the field of MQTT and pub/sub protocol fuzzing.

Gap 2: The lack of baseline results for comparison purposes is clearly evident across the literature, hindering possible opportunities for future work. For example, Sneha Suhitha Galiveeti and Pranitha Malae (2020) noted the lack of baseline results in the literature, having no choice but to avoid performance comparisons for their study. Zeng et al. (2020) compared their fuzzer with others unrelated to MQTT due to the lack of baseline results. For this thesis, it was necessary to first create baseline results of a naive- and mutation-based approach to demonstrate the advantages of MQTTGRAM, as shown in Chapter 3.
Gap 3: Fuzzer evaluation is conducted inconsistently across the literature. **Aichernig et al. (2021)** and **Di Paolo et al. (2021)** measure fuzzing effectiveness by the number of security inconsistencies found across several brokers, neglecting coverage metrics altogether. **Casteur et al. (2020)** use a scoring system to rank the effectiveness of each scenario generated by their fuzzer. At the beginning of the fuzzing campaign, scenarios are assigned an initial score, which is later incremented for each inconsistency found during testing. Inconsistent behavior is determined by several factors such as slow broker response times and error logs. The more inconsistencies found during testing, the higher the rank of each scenario. Top-ranked scenarios are therefore considered more promising and effective at testing MQTT implementations. **Hernández Ramos et al. (2018)** evaluate their proxy fuzzer by considering the following criteria: the time to process and mutate packets; the number of implementation flaws found during testing; and the CPU consumption of the host machine. **Palmieri et al. (2019)** present the number and types of messages intercepted by their fuzzer, as well as its threat detection and pub/sub capabilities. As a future work, **Palmieri et al. (2019)** propose a more in-depth analysis of the effectiveness of their framework in providing feedback to developers about the current broker configuration. **Anantharaman et al. (2017)** fail to provide any sort of explanation regarding their fuzzers’ performance during testing. In contrast, **Sochor et al. (2020b)** deeply explain the issues identified across multiple brokers, as well as the number of unhandled exceptions and test failures throughout the experiments. **Sochor et al. (2020b)** also present their fuzzers’ statement coverage. For this thesis, statement coverage was chosen for evaluation purposes, whereas **Zeng et al. (2020)** preferred path coverage, thereby hindering a performance comparison between the two research studies. At the time of writing, only the latter three research studies use traditional testing metrics to evaluate their proposals. The reason behind this issue is due to the lack of standardized methods or metrics in the literature to evaluate fuzzers for pub/sub protocols such as MQTT.

**Gap 4: Lack of a performance comparison between MQTT fuzzers.** This in turn raises uncertainty about the improvement of subsequent fuzzers compared to their predecessors. There is also a lack of awareness regarding the benefits and limitations of each fuzzing technique for MQTT. This situation stems from the fact that no research study has benchmarked existing MQTT fuzzers across a wide variety of testing and pub/sub metrics.

Analyzing and evaluating multiple fuzzing techniques under the same conditions provides opportunities to: (1) establish a necessary baseline for future research studies; (2) raise awareness about their testing and pub/sub capabilities; and (3) offer constructive feedback to their developers. This chapter therefore aims to answer the following research questions:

**RQ2:** What fuzzing techniques have been proposed for MQTT over the last few years?

**RQ3:** How effective are fuzzing frameworks for MQTT in terms of their testing and pub/sub capabilities?

For **RQ2**, a literature review is conducted in order to develop a taxonomy based on fuzzing techniques proposed specifically for MQTT brokers. The taxonomy presents a total
of sixteen fuzzing frameworks found across the literature, more so than for any other IoT protocol. Table 2.3 classified MQTT fuzzers into four main categories based on their test case generation technique. The proposed taxonomy goes one step further, and classifies the MQTT fuzzers based on their distinguishing or innovative feature for testing purposes. The taxonomy therefore provides a more detailed overview of the current state of the art in terms of fuzzing techniques for MQTT.

In order to answer RQ3, at least one MQTT fuzzer from each category in the taxonomy is evaluated using three different types of criteria based on testing, pub/sub, and MQTT-specific metrics respectively. The first criterion (testing) is based on statement coverage, which is the most traditional, yet the most reliable metric to predict the quality of test suites (Gopinath et al., 2014). The second criterion (pub/sub) is based on pub/sub behavior, which is a metric that is used to evaluate whether the MQTT fuzzers are testing message publication, which is the process in which the broker sends messages to interested subscribers. Message publication is the most important and representative feature of the pub/sub design pattern, meaning every pub/sub fuzzer should at least be capable of testing it regardless of whether they are designed for MQTT or not. The second criterion is based on a generic metric for pub/sub protocols, whereas the third criterion is based on features specific to MQTT. For the third criterion, the MQTT fuzzers are evaluated in terms of their testing capabilities for core features of MQTT: (1) QoS Levels; (2) Last Will and Testament; (3) Retained Messages; and (4) Persistent Sessions. The results of the second and third criteria are presented collectively as Feature Coverage in Section 4.3. The research findings attest that most fuzzing frameworks are incapable of testing message publications and the four core features of MQTT defined in the standard.

Overall, this chapter provides three main research contributions: (1) a taxonomical classification of fuzzing techniques for MQTT; (2) a performance comparison and discussion of fuzzing techniques proposed by several researchers for MQTT; and (3) a baseline for future studies to compare with. It is worth noting that MQTT is the only pub/sub protocol for which developers have a proposed a wide variety of fuzzing techniques in the literature at the time of writing. Thus, this chapter classifies and evaluates most, if not all, advancements in the field of pub/sub protocol fuzzing as well, despite focusing specifically on MQTT. The lessons learned from this research study can therefore help lay the foundation for effective pub/sub protocol fuzzing in general.

This chapter begins by explaining aspects that need to be considered by developers when designing MQTT fuzzers regardless of their testing strategy. These explanations are then followed by a taxonomical classification of techniques used by developers to fuzz MQTT and pub/sub protocols in general. The final section of this chapter presents a performance evaluation of each technique presented in the taxonomy. The design considerations presented in the earlier sections of this chapter will play a key role in the evaluation.

4.1 Design Considerations for MQTT Fuzzers

The most difficult, and possibly the most important, types of bugs to find in pub/sub brokers are those related to their message publishing features. These types of bugs are not only a major threat to the security of the IoT application, but also to its QoS. From
a security standpoint, a bug can *intentionally* be exploited by a publisher to either crash the broker or an end device acting as a subscriber. Another potential outcome is the publisher uncovering a software weakness *unintentionally*, which can then be found and exploited by subscribers to perform unauthorized actions. The latter outcome is actually based on a real-world vulnerability found in an MQTT broker (CVE-2018-12546)\(^1\). From a user standpoint, publication-related bugs may lead to low-quality experiences when subscribers receive messages based on their topics of interest. In order to reduce these risks, features based on the unique characteristics of the pub/sub design pattern need to be tested thoroughly.

The main challenge of fuzzing MQTT implementations is that most of its pub/sub features can only be tested if the fuzzer performs a specific type of connection, subscription, and publication request. Certain features of MQTT also depend on user interactions from previous sessions. The fuzzer must therefore reconnect using an existing client ID to activate certain features at some point during testing. The complexity is further increased with the release of MQTT 5.0 (Andrew Banks and Ed Briggs and Ken Borgendale and Rahul Gupta, 2019), which introduces 27 user properties, each with its own specific syntax and format. As a result, an MQTT fuzzer has to be designed in such a way that it satisfies the necessary requirements to activate and test these features despite its random nature. Regardless of the version of MQTT, every fuzzer should *at least* strive to activate and test the following four pub/sub features defined in the standard.

**QoS Levels**

The delivery of messages to interested subscribers is guaranteed by specifying a QoS level:

- Messages set to *QoS level 0* are delivered to the subscriber *at most once*. The publisher does not receive an acknowledgment from the broker.

- Messages set to *QoS level 1* are delivered to the subscriber *at least once* until receiving an acknowledgment from the broker.

- Messages set to *QoS level 2* are delivered to the subscriber *exactly once* using a four-way handshake.

**Last Will and Testament**

The Last Will and Testament is a feature whereby subscribers are notified if a publisher disconnects from the network unexpectedly. The broker detects an unexpected disconnection if the publisher does not respond within a given time period. MQTT’s Last Will and Testament feature can be considerably useful to inform the status of an IoT device if a network failure or loss of battery power occurs. Since environment settings can occasionally hinder device connectivity, this feature can help keep track of devices and their status. Figure 4.1 presents an example of a device that unexpectedly disconnects from the broker, which will then notify subscribers of the outcome.

Publishers can specify their last will messages when connecting to the broker. The CONNECT packet has a Will Flag to activate this feature. If activated, publishers will need to specify the Last Will Topic and its corresponding message. Clients subscribed to this topic will be notified in the event of an unexpected disconnection.

Retained Messages

Retained Messages are messages delivered immediately to newly-subscribed MQTT clients, as shown in Figure 4.2.

The main benefit of this feature is that it offers subscribers the ability to receive information automatically, rather than manually. Publishers specify these messages by setting the Retain Flag to True. Only one retained message is allowed per topic. However, retained messages can be updated when desired. If updated, messages are pushed automatically to subscribers.
**Persistent Session**

A Persistent Session is a feature where a subscriber’s information is stored in the broker, as shown in Figure 4.3.

1. Reconnect to broker

2. Sending messages from previous session

**Figure 4.3: Persistent Session**

If the subscriber disconnects from the broker, information such as subscriptions will be retained, meaning resubscribing is unnecessary in later sessions. Persistent sessions provide several benefits for IoT applications. First, persistent sessions can reduce workload for users. For example, resubscribing can be unmanageable if users are subscribed to a considerable number of topics. Second, persistent sessions can reduce inefficient network usage and overhead for constrained devices (Giambona et al., 2018). Third, QoS 1 and 2 messages are queued until clients reconnect to the network. This can be useful if a device disconnects temporarily to preserve battery power.

### 4.2 Taxonomy

Figure 4.4 presents the proposed taxonomy of MQTT fuzzers, all of which are classified based on their core testing strategy.
The taxonomy uncovers that a wide variety of fuzzing-inspired approaches have been proposed for MQTT, ranging from simple techniques based on templates to more sophisticated algorithms based on Markov modeling and automata learning. In fact, more techniques have been proposed for MQTT than for any other protocol in the last few years. The MQTT fuzzers shown in the taxonomy are classified into four main categories: (1) Naive; (2) Mutation; (3) Generation; and (4) Hybrid. The four categories are based on the fuzzers’ input generation capabilities. Mutation-based fuzzers are further classified by their type of interaction with the broker, being either direct or indirect, the latter of which is referred to as proxy or man-in-the-middle fuzzing. Generation-based fuzzers are usually interacting with the broker directly, and thus were classified by their packet-crafting technique.

The fuzzers shown in the taxonomy are further classified by the version(s) of MQTT supported for testing purposes. Dashed rectangles indicate fuzzers that support only version 3.1.1 of MQTT, whereas dotted rectangles indicate fuzzers that support only version 5.0. Fuzzers that support versions 3.1 and 3.1.1 are indicated by densely dashed rectangles, whereas fuzzers that support all three versions of MQTT (3.1, 3.1.1, and 5.0) are indicated by a rectangle with a dash-and-dot pattern.

After developing the taxonomy of MQTT fuzzers, the answer to RQ2 is as follows:

**RQ2: What fuzzing techniques have been proposed for MQTT over the last few years?**

The proposed taxonomy presents sixteen MQTT fuzzers, most of which are classified as either a mutation- or generation-based approach. Mutation-based fuzzers are split into direct- and man-in-the-middle-based approaches, the former of which is more prevalent in the literature at the time of writing. Two main mutation-based approaches currently exist for MQTT: coverage-guided and basic mutational fuzzing. Generation-based fuzzers are more varied, consisting of four types: attack, learning, grammar, and scenario; the latter being the most popular option. FUME is the only fuzzier that supports all versions of the MQTT standard; whereas two fuzzers are specifically built for v5.0; one is designed for both v3.1 and v3.1.1; and the remaining twelve are only for v3.1.1.

### 4.3 Performance Evaluation

Now that the taxonomical classification has been developed, the main question left unanswered is whether the techniques found across the literature were designed considering the unique attributes of the pub/sub design pattern. Thus, the goal of this section is to determine the testing and pub/sub capabilities of each technique, thereby answering RQ3. The MQTT fuzzers, brokers, and metrics chosen for the performance evaluation are explained in the following sections.
MQTT Fuzzers

Figure 4.4 highlights in yellow the MQTT fuzzers that were evaluated on the testbed. At least one fuzzing technique from each category in the taxonomy is evaluated to provide a more in-depth comparison. In order to evaluate the scenario-based fuzzer developed by Casteur et al. (2020), its source code was modified for it to interact with brokers in virtual machines rather than in Docker containers. As a result, the following modifications were applied to the source code:

- **The TCP/IP layer is now managed for every packet** because the original framework used Scapy supplied sockets to interact with Docker containers;

- **The fuzzer now disconnects from the broker at the end of each test scenario.** The original framework does not perform disconnection requests because the test environment consisted of several brokers in Docker containers, each handling different test scenarios. In contrast, the testbed developed for this research study (Figure 1.3) consists of a single broker, meaning disconnection requests are necessary to successfully transition between different scenarios.

There are several fuzzers that could not be evaluated on the testbed due to being either proprietary (Anantharaman et al., 2017; Sochor et al., 2020b; Synopsis, 2021), partially automated (Hernández Ramos et al., 2018; Eclipse Foundation, 2018; Palmieri et al., 2019; Zeng et al., 2020; Di Paolo et al., 2021), or unfinished (Gotkowski and Cordeiro, 2022). Despite these hindrances, two workarounds were performed in order to evaluate their effectiveness in some manner. First, the source code of less automated fuzzers (highlighted in red in Figure 4.4) was analyzed in order to slightly estimate their effectiveness. Second, if the source code is unavailable, then a fuzzer’s performance from other research studies (highlighted in orange in Figure 4.4) is compared with those of its counterparts on the testbed. For example, the attack-based fuzzer by Sochor et al. (2020b) could not be evaluated on the testbed due to it being proprietary. In order to mitigate this limitation, the MQTT fuzzers were evaluated using the same metrics, tools, and broker chosen by Sochor et al. (2020b) for their experiments. This allows for a comparison between results from their paper (Sochor et al., 2020b) and performances achieved by its counterparts on the testbed.

MQTT Brokers

Mosquitto and Moquette were selected as the target systems for the experiments. In particular, Mosquitto was chosen because it has proven to be the implementation that is most representative of the standard, offering support for most, if not all, features and versions of MQTT (Mladenov et al., 2017; Tappler et al., 2017; Aichernig et al., 2021). Mosquitto’s adherence to the standard therefore makes it a suitable target system to determine whether fuzzers are testing all features and functionalities of MQTT thoroughly.

Version 1.6.8 of Mosquitto was specifically chosen because of three reasons. First, Mosquitto 1.6.8 is still widely used despite being released in late 2019. Second, Mosquitto 1.6.8 is the most widely used and preferred version to evaluate different fuzzing techniques for MQTT (Sochor et al., 2020b; Aichernig et al., 2021; Araujo Rodriguez and Macêdo Batista, 2021). Third, certain fuzzers were designed specifically with version 1.6.8 in mind.
For example, Aichernig et al. (2021) chose version 1.6.8 of Mosquitto specifically in their case study to generate the necessary test cases and evaluate the performance of their fuzzer.

In addition to Mosquitto, Moquette 0.13 was chosen as the second target system because it is the same version used by Sochor et al. (2020b) to evaluate their proprietary fuzzer based on attack patterns. Using the same version therefore enables a performance comparison between open-source fuzzers and their attack-based counterpart. It is worth noting that Mosquitto supports versions 3.1, 3.1.1, and 5.0 of MQTT, whereas Moquette supports the former two. It was therefore impractical to test Moquette with the learning-based approach proposed by Aichernig et al. (2021) because it was specifically designed for MQTT 5.0.

Performance Metrics

The test runs are monitored constantly in case the broker crashes. When no bugs are found during testing, statement coverage is regarded as the ideal metric to measure the effectiveness of fuzzing frameworks (Boehme et al., 2021). This is because the higher the number of statements executed in the source code, the more likely the fuzzers will find broker errors. Thus, statement coverage was chosen as one of the performance metrics to measure the effectiveness of the tests performed by the fuzzers during 30 minutes. It is worth noting that it was impractical to monitor the statement coverage of the fuzzers in real-time due to their blackbox nature. In order to provide a more in-depth analysis of their performance during testing, statement coverage is measured after every 3 minutes (3, 6, 9, 12, 15, 18, 21, 24, 27, 30). Fuzzing campaigns are repeated 100 times in order to calculate the average and standard deviation for each increment. In addition to statement coverage, the fuzzers are evaluated based on their Feature Coverage, which refers to the pub/sub functionalities or behaviors that were covered during testing.

4.3.1 Statement Coverage

Testing MQTT implementations in depth is only achievable with high-coverage fuzzers. Figures 4.5 and 4.6 are each divided into several subfigures, all of which display the statement coverage achieved by MQTT fuzzers throughout 30 minutes. In order to display the results as clearly as possible, fuzzers with similar performance were arranged in the same row and in ascending order. As a result, the subfigures in each row have different y-axis labels. For simplicity purposes, each fuzzer is referred to by their name shown in the title of the subfigures. The learning-based approach by Aichernig et al. (2021) is referred to as learner from this point forward due to lacking a formal name. In the following sections, the name MQTTGRAM refers to both mqttgram-c and mqttgram-cf. Each fuzzing technique is assigned its own color.

Mosquitto

Among all MQTT fuzzers, FUME achieves the highest statement coverage in 30 minutes, executing at most 2311 statements. mqttgram-cf, mqtt_fuzz, and mqttgram-c execute 2140, 2120, and 2108 statements respectively, slightly less than FUME. learner and CyberExploit execute at most 1818 and 1396 statements respectively, underperforming
considerably compared to their counterparts. Unsurprisingly, `fuzz()` executes at most 1126 statements, ranking as the worst-performing fuzzer. The following paragraphs explain the statement coverage of each fuzzer more in depth. The explanations are in order from the worst- to best-performing fuzzer.

**fuzz():** `fuzz()` executes on average between 1088 and 1091 statements in 30 minutes. `fuzz()` underperforms considerably because its test runs mostly consist of failed connection attempts, stemming from the fact that it generates packets in a completely random manner. Protocol states were largely unexplored, so much so that the `fuzz()` executed 0% of the statements in files unrelated to connection requests. In terms of connection files, the fuzzer executes approximately 6% of the statements.

**CyberExploit:** In contrast to `fuzz()`, CyberExploit connects successfully with the broker. However, CyberExploit ranks as the second-worst fuzzer based on statement coverage according to the research findings. Its coverage improves by up to 0.29% from 3 to 30 minutes, meaning it executes the same statements for most of the test run. The coverage performance of CyberExploit remains mostly static, executing on average between 1345
and 1383 statements in 30 minutes. Its underwhelming performance can be attributed to the following reasons. First, CyberExploit assigns the same values to most fields except for topics in `SUBSCRIBE` and `PUBLISH` packets. CyberExploit generates random topics of different lengths, ranging from 2 to 400 characters. Although random topic generation has been a successful technique for bug detection in MQTT implementations (Sochor et al., 2020b), it is largely ineffective at reaching high statement coverage because fuzzers will rarely subscribe to `valid` topics. In order to effectively test a pub/sub protocol such as MQTT, topics should not only be syntactically valid, but also semantically meaningful. Crafting valid and meaningful topics will ensure that the core functionality of pub/sub protocols, message publication, is covered during testing.

Second, CyberExploit generates scenarios based on short-lived interactions with the broker. Scenarios are performed in a similar manner, usually beginning with a connection request to the broker, followed by either a subscription or publish request. The test scenarios consist of a small set of packets, meaning connection requests are performed 3x more frequently than its publish or subscribe counterparts.

Third, CyberExploit is incapable of generating test cases for several functionalities of MQTT. For example, several MQTT flags for optional features, such as username and password authentication, are disabled during the test run. Mandatory packet flags also lack variety, with several of them set to the default or same value in most test cases. For example, CyberExploit generates `CONNECT` packets with the same client ID, repeating the exact test case for 30 minutes. The root cause of this issue lies in the packet generator developed for CyberExploit, which is unaware of several features defined in the standard.

Fourth, CyberExploit is also incapable of generating both `UNSUBSCRIBE` and `PING` packets. MQTT fuzzers should be capable of generating all control packet types, especially those related to the core functionality of the pub/sub design pattern.

**learner:** Despite outperforming CyberExploit, learner has similar drawbacks that hinders it from executing more statements. First, learner is incapable of generating three control packet types: `PING`, `AUTH`, and `PUBLISH` (QoS 2). Second, similar to CyberExploit, learner connects to the broker in the same manner throughout the entire fuzzing campaign, reusing a predefined username for authentication purposes. Publish and subscription requests by learner are always performed using QoS 1. It is also worth noting that learner generates publication and subscription messages with the same topic at the beginning of the fuzzing campaign. Reusing the same topic triggers the broker into publishing messages to subscribers. Third, learner lacks support for all the user properties of MQTT 5.0, except `Session Expiry Interval` and `Topic Alias Maximum`, both of which were used within 30 minutes. Future fuzzing frameworks should support all the user properties of MQTT 5.0, covering different behaviors or features. Fourth, on several occasions, learner disconnects immediately after the broker accepts a connection request. As learner gains a better understanding of MQTT over time, immediate disconnections become less frequent. Similar to CyberExploit, learner requests up to 3x more connections than message publications or subscriptions. The connection requests are performed in the exact manner for 30 minutes, meaning several authentication mechanisms are left untested.
learner outperforms CyberExploit for several reasons. First, learner supports only MQTT 5.0, meaning it executes both version-specific and independent statements. Second, CyberExploit is incapable of triggering the broker into publishing messages, which is yet another important behavior left untested. Third, learner connects to the broker with a username and client ID, whereas CyberExploit uses only the latter.

Despite outperforming CyberExploit, learner rarely executes more statements over time. The largest increase in statement coverage (0.94%) occurs between 3 and 6 minutes. After approximately 4 minutes and 16 seconds, learner changes its testing strategy to one that is based solely on random topic generation. As a result, statement coverage rarely increases after 6 minutes. In that regard, learner, with its stale statement coverage, performs similar to CyberExploit when testing MQTT implementations. It is worth noting that CyberExploit occasionally sends packets out of order, whereas learner always follows the correct sequence.

**mqtt_fuzz:** mqtt_fuzz offers a substantial improvement over its scenario-, learning-, and naive-based counterparts, executing at most 2120 statements. On average, mqtt_fuzz executes between 2022 and 2073 statements in 30 minutes. mqtt_fuzz outperforms CyberExploit and learner because of the following reasons. First, mqtt_fuzz is capable of generating every type of control packet. Second, mqtt_fuzz performs connection requests using multiple client IDs and MQTT flags, the latter of which are all related to core features such as QoS, Last Will and Testament, and Persistent Sessions. Third, mqtt_fuzz occasionally uses nonrandom and predefined topics to trigger the broker into publishing messages. Fourth, since mqtt_fuzz is a mutation-based approach, test cases are generated from existing MQTT packets rather than from scratch. As a result, mqtt_fuzz exchanges a considerable amount of packets with the broker compared to its counterparts.

Despite offering a substantial improvement over its counterparts, mqtt_fuzz has several shortcomings that remain unaddressed. Most notably, mqtt_fuzz is incapable of generating valid CONNECT packets with usernames, passwords, and will retain fields. Another shortcoming is the constant reuse of test cases, meaning more time is spent on executing the same, rather than new, statements. At the time of writing, mqtt_fuzz also lacks support for MQTT 5.0.

**MQTTGGRAM:** The grammar-based approaches presented in Chapter 3, mqttgram-c and mqttgram-cf, have a similar performance to mqtt_fuzz. On average, mqttgram-c executes from 1928 to 2052 statements in 30 minutes. mqttgram-cf outperforms mqttgram-c, executing from 1981 to 2096 statements on average. In terms of both grammar-based approaches, the standard deviation of their coverage results decreases as time progresses. The opposite occurs for mqtt_fuzz, which outperforms mqttgram-c mainly because it occasionally sends packets out of order to the broker, thereby testing both correct and incorrect functionalities of MQTT. The superior performance by mqtt_fuzz over mqttgram-c highlights the importance of testing both types of functionalities.

**FUME:** On average, FUME executes from 2138 to 2242 statements in 30 minutes. FUME executes more statements than its counterparts because it supports all major versions of MQTT, being the only framework to do so at the time of writing. Despite outperforming
its counterparts, FUME also suffers from shortcomings with regard to pub/sub protocol testing. Most notably, there are two issues preventing it from effectively testing message publication. First, FUME generates invalid, rather than valid publish messages for most of the test run. As a result, the broker rejects most publish requests immediately, preventing FUME from reaching certain states or publishing messages to subscription topics. Second, for each test scenario, FUME connects to the broker, sends a small set of packets in both random and rapid succession, and then disconnects before receiving any acknowledgment or response whatsoever. This process is repeated until a stopping criterion has been satisfied. Each test scenario therefore consists of short interactions with the broker. As a result, the fuzzing campaigns consist mostly of connection rather than pub/sub-related requests, which should be the focus when testing network protocols such as MQTT.

**Moquette**

As stated previously, it was impractical to perform experiments with learner and FUME because Moquette lacks support for version 5.0 of MQTT at the time of writing. The results in Figure 4.6 follow mostly the same pattern as in Figure 4.5.

![Figure 4.6: Statement Coverage (Moquette 0.13)](image)

mqttgram-cf executes at most 1613 statements in 30 minutes, outperforming all of its counterparts. In contrast to results obtained in the previous study with Mosquitto, mqttgram-c executes more statements in Moquette than mqtt_fuzz. mqttgram-c and mqtt_fuzz execute at most 1596 and 1566 statements respectively. CyberExploit and fuzz() underperform considerably compared to their counterparts, executing at most 995 and 571 statements respectively. Similar to their performance in Mosquitto, both fuzzers rarely execute more statements over time.

All of the aforementioned MQTT fuzzers were further evaluated using the same metrics.
as Sochor et al. (2020b) in order to compare them with their attack-based counterpart, which is proprietary. Before going any further, it is important to explain three aspects of the experiments performed by Sochor et al. (2020b) to evaluate their fuzzer:

1. Sochor et al. (2020b) evaluated their fuzzer using Moquette instead of Mosquitto.

2. Sochor et al. (2020b) did not perform a single repetition of their experiments, meaning that their fuzzers’ coverage performance is based on a result from a single test run, lacking sufficient evidence. In contrast, coverage performance by its counterparts are based on the average of 100 test runs.

3. Sochor et al. (2020b) developed two variants of their proprietary fuzzer, one of which is based on existing vulnerabilities. The explanations regarding the coverage performance by both of these variants may therefore be inaccurate due to their proprietary nature. For simplicity purposes, the variant based on existing vulnerabilities is referred to as vulnerability-oriented, whereas its counterpart is referred to as vulnerability-unoriented.

Table 4.1 presents the results of the attack-based approach by Sochor et al. (2020b) and its counterparts. Cells highlighted in blue indicate the highest coverage percentage for a particular directory. Table 4.1 classifies fuzzers considering both types of variants developed by Sochor et al. (2020b) in order to enable a more fair comparison. The following paragraphs discuss the coverage performances for each directory in depth.

<table>
<thead>
<tr>
<th>Source Code (Directory)</th>
<th>Fuzzed packets not based on existing vulnerabilities</th>
<th>Fuzzed packets based on existing vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fuzz()</td>
<td>CyberExploit</td>
</tr>
<tr>
<td>broker</td>
<td>21.00%</td>
<td>39.42%</td>
</tr>
<tr>
<td>broker.security</td>
<td>12.00%</td>
<td>12.00%</td>
</tr>
<tr>
<td>broker.subscriptions</td>
<td>8.00%</td>
<td>25.56%</td>
</tr>
<tr>
<td>broker.config</td>
<td>51.00%</td>
<td>51.00%</td>
</tr>
<tr>
<td>broker.metrics</td>
<td>44.00%</td>
<td>67.81%</td>
</tr>
<tr>
<td>persistence</td>
<td>3.00%</td>
<td>3.00%</td>
</tr>
<tr>
<td>interception</td>
<td>31.00%</td>
<td>28.84%</td>
</tr>
<tr>
<td>Logging</td>
<td>43.00%</td>
<td>43.00%</td>
</tr>
<tr>
<td>Total</td>
<td>19.21%</td>
<td>33.83%</td>
</tr>
</tbody>
</table>

**Table 4.1: Statement Coverage of Moquette (30 Minutes) (Cells highlighted in blue indicate the highest coverage percentage for a particular directory.)**

**broker:** The broker directory contains files for client authorizations, session storage, queued/retained messages, among others. Coverage performance depends primarily upon a fuzzer’s understanding of the states and packets defined in the standard. In that regard, both variants of MQTTGRAM outperform their counterparts within their respective categories. More specifically, mqttgram-c, mqttgram-cf, and mqtt_fuzz outperform the fuzzers developed by Sochor et al. (2020b), executing 66.34%, 67.57%, and 64.2% of the statements respectively. Unsurprisingly, CyberExploit and fuzz() execute only 39.42% and 21% of the statements respectively. Similar to mqtt_fuzz, their underwhelming performance is also due to the lack of test cases for certain functionalities of MQTT. The vulnerability-unoriented approach by Sochor et al. (2020b) executes 57.8% of the statements, whereas its vulnerability-oriented counterpart executes 58%. Although the latter fuzzer narrowly outperforms the former, both seem to be incapable of testing several features of MQTT compared to their grammar- and mutation-based counterparts.
The results in the broker directory provide a general overview of the fuzzers’ coverage performance during testing. However, a more in-depth analysis is required in order to gain a better understanding of the fuzzers’ shortcomings and coverage capabilities. Thus, the following paragraphs explain the fuzzers’ coverage performance in specific subdirectories of the broker directory.

**broker.security:** Files in the broker.security directory are associated with authentication mechanisms, most notably Access Control Lists (ACL). ACL are used by Moquette to assign pub/sub permissions to users, therefore acting as network-based filters for topics. Moquette uses ACL to block interactions with unknown publishers or subscribers, improving network security as a result. Coverage performance in security-related files is relatively low for all fuzzers. Both variants of MQTTGRAM execute approximately 15% of the statements, the result of which is the highest among all fuzzers. mqtt_fuzz and the vulnerability-oriented approach by Sochor et al. (2020b) execute 13.02% and 12.8% of the statements respectively. MQTTGRAM outperforms its counterparts due to its use of usernames and passwords when connecting to the broker. In contrast, both fuzz() and CyberExploit are incapable of generating syntactically valid packets with usernames and passwords, executing only 12% of the statements. However, despite of this hindrance, they outperform the vulnerability-unoriented approach by Sochor et al. (2020b). The subpar performance of all fuzzers is caused by their disuse of ACL during testing.

**broker.subscriptions:** Files in the broker.subscriptions directory are necessary for subscription-related tasks, such as topic matching and message delivery, which play a key role in enabling pub/sub communication between devices. When testing pub/sub protocols such as MQTT, fuzzers should strive to execute as many statements as possible in subscription-related files, such as those located in this directory. This outcome is not only achieved by triggering the broker into publishing messages, but also by testing exceptional pub/sub behaviors. For example, in addition to triggering the broker into publishing messages, mqtt_fuzz tests exceptional behavior by sending packets out of order, and using incorrect levels of QoS. This allows mqtt_fuzz to slightly outperform mqttgram-c, which triggers only normal pub/sub behaviors. Similar to mqtt_fuzz, mqttgram-cf also tests exceptional behavior by sending packets out of order. However, despite performing less subscription requests than mqtt_fuzz, mqttgram-cf achieves the highest statement coverage at 73.92%. mqttgram-cf outperforms mqtt_fuzz because it generates topics that contain wildcard characters (# or +) in a manner that is prohibited by the formal specification of MQTT. CyberExploit underperforms considerably, executing approximately 23.56% of the statements. Its low coverage performance is because of its incapability to trigger the broker into publishing messages. The interaction between CyberExploit and the broker is therefore more similar to the traditional client-server approach than pub/sub, meaning that the core functionality of MQTT is neglected during testing. Unsurprisingly, fuzz() executes the fewest statements in 30 minutes. fuzz() is unable to connect to the broker successfully, hence the lack of topic subscriptions during testing.

**broker.config:** The broker.config directory contains configuration files that allow users to modify a wide variety of settings, such as port numbers. All fuzzers evaluated
on the testbed have the same coverage performance in configuration-related files (51%) because Moquette was executed using default settings.

**broker.metrics:** The `broker.metrics` directory contains the necessary files to collect statistical information such as the number of clients connected to the broker. Both fuzzers by Sochor et al. (2020b) outperform their counterparts, executing 77.5% of the statements. As stated previously, it cannot be determined if their superior performance is due to Sochor et al. (2020b) modifying Moquette’s source code or triggering an exceptional crash that generates more statistical information about the broker session. In terms of its counterparts, coverage performance seems to depend primarily on the fuzzers’ knowledge of MQTT. Both variants of MQTTGRAM execute 73% of the statements, followed by `mqtt_fuzz`, `CyberExploit`, and `fuzz()` with 71.78%, 67.81%, and 44% respectively.

**persistence:** The `persistence` directory contains the necessary files to enable both persistent sessions and storage. When enabling the former, user-related information, such as subscriptions and queued messages, are stored in memory by default. However, Moquette deletes the information on shutdown, requiring users to create new sessions from scratch after restart. In order to mitigate this issue, Moquette offers an option called persistent storage, in which the information is stored in a database and persists regardless if the broker is offline or online. All fuzzers achieve subpar performance when executing statements in persistence-related files because Moquette was executed using default settings, meaning persistent storage is disabled during testing. Results indicate that their performance depends on the level of understanding of MQTT. Both fuzzers developed by Sochor et al. (2020b) achieve the highest coverage performance, executing 9.1% of the statements. These results hint that Sochor et al. (2020b) probably tested Moquette using default settings.

**interception:** Interceptors can be extended to monitor Moquette’s incoming messages and notify about specific events. The source files of these interceptors can be found in the `interception` directory, which contains a subdirectory named `messages`. The former directory contains files associated with the Interception Handler, whereas the latter subdirectory contains the necessary files to notify about specific message events, such as normal or abrupt disconnections. Moquette disables packet interception by default, meaning files in the `messages` directory are neglected throughout the experiments. As a result, the coverage tool used for the experiments (Cobertura) reports 0% of statement coverage for all files in the `messages` directory. However, despite the lack of packet interception, statements from some files in the `interception` directory are executed, as shown in Table 4.1. It cannot be determined which statements are executed because Cobertura generates coverage reports with limited information. However, it should be highlighted that files associated with the Interception Handler import functions from broker-related files, such as those used for subscription, connection, and publish requests. Since no messages are intercepted during the fuzzing campaign, coverage performances in the interception directory are most likely associated with functions imported from broker-related files, or more specifically with the functionalities and features of MQTT covered during testing. This may be a valid observation because coverage performances are ranked in decreasing order from most to least knowledgeable fuzzer. Both variants of MQTTGRAM outperform all of their counterparts, executing 30% of the statements, followed
by \texttt{mqtt\_fuzz}, \texttt{CyberExploit}, and \texttt{fuzz()} with 28.04\%, 25.65\%, and 11\% respectively. It cannot be determined if the superior performance of the proprietary fuzzer proposed by \textit{Sochor et al. (2020b)} is due to it intercepting packets during testing. If that were the case, files in the \texttt{messages} directory would have been executed, thereby outperforming its counterparts. Another explanation for the fuzzer’s superior performance may have to do with the possibility of Moquette’s source code being extended or modified by the researchers to better intercept messages.

\textbf{Logging:} The Logging directory contains the necessary files for Moquette to perform log-related tasks. Default logging activities are performed by Moquette on the testbed, meaning coverage performance is the same across all fuzzing-inspired approaches, which achieve 43\%. Both of the fuzzers proposed by \textit{Sochor et al. (2020b)} outperform their counterparts, executing 62.5\% of the statements. The superior performance of the fuzzers proposed by \textit{Sochor et al. (2020b)} may be due to either: (1) a possible alteration of the source code to generate additional or more detailed logs about Moquette’s behavior during testing; or (2) exceptional crashes. However, the exact reason cannot be determined due to the lack of information provided by \textit{Sochor et al. (2020b)} about logging procedures for their experiments.

\textbf{Summary:} The last row of Table 4.1 presents the fuzzers’ average performance considering the statement coverage of each subdirectory. Both \texttt{mqtt\_gram\_c} and \texttt{mqtt\_gram\_cf} execute 56.76\% and 57.24\% of the statements respectively, outperforming all of their counterparts. \texttt{mqtt\_fuzz} executes 55.24\% of the statements, followed by the vulnerability-oriented approach by \textit{Sochor et al. (2020b)} with 51.4\%, and its unoriented counterpart with 50.0\%. The low coverage performance of the latter two fuzzers may be due to their incapability of triggering important functionalities during testing. \texttt{CyberExploit} and \texttt{fuzz()} are the worst-performing fuzzers, executing 33.83\% and 19.21\% of the statements respectively. Among all directories, \texttt{broker}, \texttt{broker\_security}, and \texttt{broker\_subscriptions} are the most important because their files are directly linked to MQTT’s pub/sub functionalities. The grammar-based approaches presented in Chapter 3 test the contents of these directories more in depth, despite exchanging the fewest packets with the broker.

\subsection*{4.3.2 Feature Coverage}

Thus far, only the statement coverage of each fuzzer has been discussed. A testing metric that also plays a crucial role in measuring the effectiveness of fuzzing frameworks is \textit{feature coverage}. As its name suggests, feature coverage refers to the pub/sub features or functionalities tested by each fuzzer. However, before explaining about feature coverage, it is important to discuss which MQTT flags are enabled or disabled by each fuzzer throughout the test run. MQTT offers several flags for a wide variety of purposes, such as basic authentication, guaranteed message delivery, and persistent sessions, among others. Flags are directly tied to the functionality of a network protocol, meaning a fuzzer’s packet generator should be designed with them in mind. Most flags are disabled by default, meaning the fuzzer must enable them by generating syntactically valid packets, otherwise the broker will close the connection immediately. Fuzzers should also attempt to explore several flag combinations in order to test several features simultaneously.
Tables 4.2 and 4.3 present a detailed summary of the flags used by MQTT fuzzers when performing connection and publish requests respectively. The first column presents the MQTT fuzzer, whereas the remaining columns indicate whether a specific flag value is used during the test run. Each flag value is represented by a circle in Tables 4.2 and 4.3. It is worth mentioning that most flags in network protocols are represented by a single bit, and can be set to a value of either 0 or 1. There are exceptions, such as the QoS flag, which can be set to a third possible value of 2. The first, second, and third circle (if any) therefore represent the value of 0, 1, and 2 respectively. The type of circle shown for each value indicates either correct ✅, incorrect ❌, or no usage ❌. For example, ✅ is correct usage of a flag value | ❌: Incorrect usage of a flag value | ❌: No usage of a flag value

Table 4.2: Packet Generation - Connect Flags

<table>
<thead>
<tr>
<th>MQTT Fuzzer</th>
<th>User Name Flag</th>
<th>Password Flag</th>
<th>Will Retain</th>
<th>Will QoS</th>
<th>Will Flag</th>
<th>Clean Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqtt_fuzz</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>CyberExploit</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Di Paolo et al. (2021)</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>learner</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>MQTTGRAM</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>FUME</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

Table 4.3: Packet Generation - Publish Flags

<table>
<thead>
<tr>
<th>MQTT Fuzzer</th>
<th>DUP flag</th>
<th>QoS Level</th>
<th>Retain flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqtt_fuzz</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
</tr>
<tr>
<td>CyberExploit</td>
<td>✅ 1</td>
<td>✅ 1 2</td>
<td>✅ 1</td>
</tr>
<tr>
<td>Di Paolo et al. (2021)</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
<td>✅ 1</td>
</tr>
<tr>
<td>learner</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
<td>✅ 1</td>
</tr>
<tr>
<td>MQTTGRAM</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
</tr>
<tr>
<td>FUME</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
<td>✅ 1 2</td>
</tr>
</tbody>
</table>

Table 4.4 presents the features tested by each fuzzer within 30 minutes. The feature coverage of each fuzzer is determined by examining the packet logs. It is worth noting that Tables 4.2 and 4.3 are directly related to Table 4.4 because certain flags have to be enabled to test specific features. Each column name in Table 4.4 is based on a core pub/sub feature of MQTT. The symbol ✔ indicates if a feature is tested by an MQTT fuzzer whereas the symbol ✗ indicates the opposite. The symbol ✗* indicates that the fuzzer would have been capable of testing a feature if other clients had been connected to the broker during testing. The following paragraphs explain the results for each feature.
4.3 | PERFORMANCE EVALUATION

<table>
<thead>
<tr>
<th>MQTT Fuzzer</th>
<th>Pub/Sub Behavior</th>
<th>All QoS Levels</th>
<th>Last Will and Testament</th>
<th>Retained Messages</th>
<th>Persistent Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqtt_fuzz</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CyberExploit</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Di Paolo et al. (2021)</td>
<td>×</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>learner</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>MQTTGRAM</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>FUME</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 4.4: Feature Coverage of MQTT Fuzzers (30 Minutes)

✔: Tested successfully | ×: Tested unsuccessfully | ✔*: Tested successfully if clients > 1

Pub/Sub Behavior: In order to trigger the broker into publishing messages, a fuzzer must perform a subscription and publish request with the same topic. In that regard, CyberExploit and the fuzzer proposed by Di Paolo et al. (2021) fail to meet this requirement, generating long random topics that are never used more than once. It is worth noting that, in most cases, both mqtt_fuzz and learner are capable of triggering the broker into publishing messages only because of their use of predefined topics.

All QoS Levels: QoS-based functionality is fully tested by all MQTT fuzzers except CyberExploit and learner, which test only level 0 and 1 respectively.

Last Will and Testament: The will flag enables immediate notifications for when an MQTT client disconnects from the network. The will flag is never enabled by CyberExploit, learner, and the fuzzer proposed by Di Paolo et al. (2021) during the test run. As such, these three fuzzers are incapable of testing the Last Will and Testament feature of MQTT. Among all the core functionalities of MQTT, the Last and Will Testament is the most difficult to test thoroughly because it requires multiple clients connected to the broker. In that regard, none of the fuzzers manage to fully test this feature due to the lack of additional clients during testing. However, mqtt_fuzz, FUME, and both variants of MQTTGRAM would have been capable of fully testing this feature if other clients had also been connected to the broker. This is because of two reasons. First, all three fuzzers are capable of triggering the broker into publishing messages to subscribers. Second, the fuzzers subscribe to will topics occasionally during testing. The symbol ✔* in Table 4.4 therefore indicates that these fuzzers would have been capable of fully testing the Last Will and Testament feature if other clients had been connected to the broker.

Retained Messages: As shown in Tables 4.2 and 4.3, learner and the fuzzer proposed by Di Paolo et al. (2021) disable retain flags during testing. CyberExploit attempts, but ultimately fails, to generate valid packets that enable retain flags. As a result, learner, CyberExploit, and the fuzzer proposed by Di Paolo et al. (2021) are all unable to test the functionality of retained messages.

Persistent Session: CyberExploit, learner, and the fuzzer proposed by Di Paolo et al. (2021) enable the clean session flag for the entire test run, meaning user information is never stored in the broker.

After analyzing the performance of all the fuzzers evaluated on the testbed, the answer to RQ3 is as follows:
**RQ3:** How effective are fuzzing frameworks for MQTT in terms of their testing and pub/sub capabilities?

FUME executes the most number of statements in MQTT implementations because it supports multiple versions of the standard, thereby being the most effective for testing purposes. In terms of pub/sub capabilities, most fuzzers proposed thus far support a single version of MQTT, meaning that message publishing features introduced in other releases are left untested. Existing learning-based techniques are incapable of understanding and testing the core functionalities of MQTT. Scenario-based approaches are mainly geared towards random topic generation, neglecting pub/sub functionality altogether during testing. Proxy fuzzers apply random mutations to packet fields selected by the user, unaware of necessary requirements for testing message publications. The Markov modeling approach incorporated into FUME supports multiple versions of MQTT, but lacks awareness of valid message sequences and state transitions. In fact, FUME struggles to craft valid packets, meaning that it tests pub/sub functionality on rare occasions. MQTTGRAM and mqtt_fuzz are more consistent in terms of testing message publications. However, mqtt_fuzz manages to test message publishing features only because of its use of predefined topics, meaning that its test cases lack variety. Overall, MQTTGRAM manages to generate more varied, but semantically-meaningful topics that activate and test message publishing features of the broker. Future research should address the aforementioned issues, combine techniques, and cover more pub/sub functionality during testing, the latter of which is neglected to some extent across all fuzzers proposed for MQTT thus far. The shortcomings presented in this chapter can raise awareness and motivate developers to better test pub/sub protocols such as MQTT in future work.

**4.4 Concluding Remarks**

MQTT Fuzzing has been evolving considerably in the last few years, quickly gaining prominence due to its necessity in the IoT. However, several issues have arisen that hinder its progression and future research studies. Most notably, the lack of baseline data and standardized pub/sub metrics in the literature has led to limited information regarding the strengths, weaknesses, and even improvements of each fuzzing technique over its predecessors. To mitigate these issues, a literature review was first conducted in order to create a taxonomical classification of all fuzzing techniques proposed for MQTT, providing developers a clear and more detailed overview of the current state of the art. Afterwards, each technique was benchmarked across a wide variety of testing metrics under equivalent conditions, further highlighting their benefits and shortcomings to developers for future research opportunities. As a result, developers can now compare their findings with the baseline results presented in this study to further improve their testing strategy. The findings presented in this chapter are further complemented by the results shown in Appendixes D and E, which delve deep into the fuzzers’ input coverage and packet exchange capabilities respectively.
Chapter 5

Refinement of a Grammar-Based Fuzzing Technique for a Pub/Sub Protocol

The research findings presented in Chapter 4 indicate that all open-source fuzzers have shortcomings in regards to pub/sub protocol testing. Upon further investigation, it is found that developers neglect three essential elements for successful pub/sub fuzzing: (1) two-way communication capabilities; (2) topic awareness; and (3) multiversion support. The following paragraphs explain each element in more detail.

Two-way communication: According to the taxonomy shown in Figure 4.4, there are two main approaches to fuzzing a pub/sub broker. Both approaches differ in terms of the fuzzer’s role during testing, which can either be a man in the middle or a normal client. In terms of the former role, the fuzzer intercepts and modifies messages in transit between the broker and the clients. In terms of the latter role, the fuzzer exchanges messages directly with the broker as if it were a normal client. Regardless of the role, the test architecture should support a request-response (or two-way) communication pattern between the broker and a client, otherwise the fuzzer will fail to trigger message publications. Four fuzzers fail to meet this requirement. fuzz() lacks a complete understanding of MQTT, thereby preventing it from sending valid requests and responses to the broker. FUME first sends a connection request, followed by several other messages in random order. All of the messages generated by FUME are sent within a single packet, after which it proceeds to end the connection. This means that FUME sends only one packet for each connection. Similarly, the fuzzer proposed by Di Paolo et al. (2021) sends messages without acknowledging to any request or response by the broker. AFLNet-MQTT is incapable of behaving as a publisher or subscriber because its architecture lacks a complete understanding of two-way communication with MQTT.

Multiversion: Most protocol implementations support multiple versions of the standard. Subsequent versions of the standard either add or remove features for message publications. For this research study, a fuzzer is classified as multiversion if it supports at least two versions of a protocol standard. At the time of writing, FUME is the only
fuzzer in the literature that supports all three major versions of MQTT (3.1, 3.1.1, and 5.0). AFLNet-MQTT, fuzz(), and Polymorph have flexible architectures that enable them to test any implementation regardless of the version of MQTT. In contrast, the remaining fuzzers shown in the taxonomy (Figure 4.4) are designed for a specific version of MQTT.

**Topic Awareness:** Topic generation should be a central focus of pub/sub fuzzers. In fact, it is the element or requirement that sets it apart from fuzzers for other types of target systems. As stated previously, a pub/sub fuzzer must generate topics that strike a balance between semantically-meaningful and error-prone values. For this research study, a fuzzer is classified as *topic aware* if it is capable of generating random, but semantically-meaningful topics that trigger the broker into publishing messages. Five fuzzers are currently incapable of testing message publications because of different reasons. For example, fuzz() is unaware of the MQTT message format, whereas AFLNet-MQTT is incapable of acting as a genuine MQTT client. Polymorph mutates messages in transit between the broker and MQTT clients, lacking the necessary capabilities to generate semantically-meaningful topics. CyberExploit generates invalid topics throughout the entire test run, but is able to respond and acknowledge requests from the broker. The fuzzer proposed by Di Paolo et al. (2021) has the opposite problem, generating valid topics without acknowledging messages from the broker.

According to the research findings, all MQTT fuzzers were designed with either 1 or 2 elements in mind. However, none of the MQTT fuzzers available for testing purposes encompass all three elements. Thus, the research hypothesis for this study is:

*When considering the three elements, effectiveness is higher than that of existing MQTT fuzzers.*

The research question that will be used to validate the research hypothesis is as follows:

**RQ4:** How effective is an MQTT fuzzer when considering three essential elements for pub/sub fuzzing: two-way communication capabilities; topic awareness; and multiversion support?

In order to answer **RQ4**, the architecture and the algorithm of the original MQTTGRAM were modified to meet all three of these criteria. The modifications and improvements ultimately resulted in a new grammar-based fuzzer called mqttgram-r. Table 5.1 compares mqttgram-r with its counterparts.

### 5.1 Refinements

As stated previously, the main difference between mqttgram-cf and mqttgram-c is that the former fuzzer prioritizes topic-based messages during testing. mqttgram-r is based on mqttgram-cf because pub/sub fuzzers should in fact prioritize topic-based messages to trigger important publish operations of the broker.

The response engine and packet generator of the original MQTTGRAM satisfy only two
of the essential conditions for successful pub/sub fuzzing: two-way communication and topic awareness. The original packet generator supported only version 3.1.1 of MQTT. Figure 5.1 presents the refined architecture for `mqttgram-r`. The main difference between the architectures of `mqttgram-r` and the original MQTTGRAM is that the former supports versions 3.1.1 and 5.0 of MQTT. This means that, in contrast to its counterparts, `mqttgram-r` meets all three of the criteria shown in Table 5.1.

![Figure 5.1: Refined Architecture for MQTTGRAM-R](image)

`mqttgram-r` generates hexadecimal strings from two grammars based on MQTT 3.1.1 and 5.0 respectively. There are several notable differences between both versions, which in turn influenced the development of the grammar for MQTT 5.0. For example, the headers of MQTT packets vary across different versions of the standard. In fact, several MQTT 5.0 packets have reason codes, as part of their variable header, to indicate the result of an operation. The most notable difference or feature of MQTT 5.0, however, is the introduction of user properties, which are UTF-8 string key-value pairs defined by the user to set specific rules or settings such as session expiry intervals, topic aliases, maximum packet size, among others. A grammar based on MQTT 5.0 is developed that incorporates all changes made to the packet headers, including all 45 reason codes and 27 user properties introduced
in the latest version of the standard. A simple example of our MQTT 5.0 grammar and its expansion rules is shown below:

\[
\langle \text{start} \rangle ::= \langle \text{packets} \rangle \\
\langle \text{packets} \rangle ::= \langle \text{publish} \rangle \\
\langle \text{publish} \rangle ::= \langle \text{fheader} \rangle \langle \text{vheader} \rangle \langle \text{properties} \rangle \\
\langle \text{properties} \rangle ::= \langle \text{no-property} \rangle \mid \langle \text{propertyN} \rangle \\
\langle \text{no-property} \rangle ::= 'Number 0 in Hex notation' \\
\langle \text{propertyN} \rangle ::= \langle \text{length} \rangle \langle \text{id} \rangle \langle \text{value} \rangle \\
\langle \text{length} \rangle ::= 'length of id and value' \\
\langle \text{id} \rangle ::= 'id of property in Hex notation' \\
\langle \text{value} \rangle ::= 'value of property in Hex notation' \\
\langle \text{fheader} \rangle ::= 'Hexadecimal string of fixed header' \\
\langle \text{vheader} \rangle ::= 'Hexadecimal string of variable header' \\
\langle \text{payload} \rangle ::= 'Hexadecimal string of payload'
\]

Figure 5.2: Simple Example of the MQTT 5.0 Grammar in Backus-Naur Form

Rather than developing it from scratch, the MQTT 5.0 grammar is adapted from its 3.1.1 counterpart used by the original MQTGRAM. New and updated expansion rules were defined to describe the grammatical structure of MQTT 5.0 messages. The main difference between the MQTT 3.1.1 and 5.0 grammar is the latter’s nonterminal symbol \( \langle \text{properties} \rangle \), which describes the necessary expansion rules to generate all 27 properties considering their length, ID, and values defined in the standard. The end result of the grammar is the concatenation of hex strings for the fixed header, variable header, payload, and properties, which together make up an MQTT 5.0 message.

Algorithm 3 presents the pseudocode of the proposed multiversion grammar-based approach for a pub/sub protocol, which is incorporated into mqttgram-r.

Algorithm 3: Multiversion grammar-based fuzzing approach for MQTT

\[
\textbf{Input:} \text{Protocol State } S, \text{ Protocol Version } V \\
\textbf{while} \ stopping\_criterion == \text{false} \ \textbf{do} \\
\quad t \leftarrow \text{select\_packet\_type}(S, V) \\
\quad m \leftarrow \text{generate\_packet}(t) \\
\quad r \leftarrow \text{feed\_input}(m) \\
\quad S \leftarrow \text{evaluate\_response}(r) \\
\quad \textbf{if } S == \text{disconnected} \ \textbf{then} \\
\quad \quad V \leftarrow \text{choose\_protocol\_version}() \\
\quad L \leftarrow \text{monitor\_program}(S) \\
\textbf{Output:} L
\]

The algorithms proposed in Chapter 3 were adapted for mqttgram-r. More specifically, Algorithm 3 is adapted from Algorithm 1, and receives two inputs instead of one: (1) a
Protocol State $S$ and (2) a Protocol Version $V$, the latter of which can be either v3.1.1 or v5.0. Before establishing a connection with the broker, the algorithm randomly chooses between testing either version 3.1.1 or 5.0 of MQTT. $V$ is then assigned the version number chosen by the algorithm. mqttgram-r then selects (line 2 in Algorithm 3), generates, and sends MQTT messages based on version $V$ to the broker until a disconnection occurs, after which $V$ is assigned a random value for the next connection (lines 6-7 in Algorithm 3). This process is repeated until a stopping criterion has been satisfied. In other words, mqttgram-r tests only one version of MQTT for each connection with the broker. This design choice is based on the fact that the broker will disconnect a fuzzer from the network unless both parties exchange messages that belong to the same version. Maintaining version consistency therefore leads to fewer connection requests, which in turn leads to a greater focus on publication or subscription messages during testing.

Algorithm 2 remains largely unchanged for mqttgram-r, the only difference being that the calculate() function (line 6) now calculates the lengths of user properties after the hexadecimal string has been produced.

With these changes, mqttgram-r mitigates certain shortcomings found in MQTT 5.x compatible fuzzers and presented in Chapter 4. For example, in contrast to learner, mqttgram-r supports all 27 properties introduced in MQTT 5.0. In contrast to FUME, mqttgram-r sends messages based on the same MQTT version to the broker for each connection. This in turn allows mqttgram-r to have infrequent disconnections from the broker, thereby increasing the probability of triggering message publications during testing.

5.2 Performance Evaluation

This section evaluates the performance of the proposed multiversion grammar-based approach, which is incorporated into mqttgram-r. For the experiments, Mosquitto 1.6.8 is selected as the target broker. mqttgram-r and its counterparts exchange messages with Mosquitto until a stopping criterion is satisfied. Two separate experiments were conducted, each with a different stopping criterion. In the first experiment, the fuzzers communicate with Mosquitto for 30 minutes. In the second experiment, the fuzzers communicate with Mosquitto until 8000 packets have been exchanged between both parties. For each stopping criterion, the average statement coverage achieved by the MQTT fuzzers in 100 test runs is calculated. For comparison purposes, mqttgram-r is also evaluated when testing only MQTT 5.0. This variant is referred to as mqttgram5-cf for the remainder of the chapter. Among the 10 open-source fuzzers available to developers, six are completely automated, whereas the remaining four require user intervention. In order to provide a more fair comparison, only automated fuzzers are evaluated for this research study. The testbed shown in Figure 1.3 was used for the performance evaluation.

Statement coverage should always be analyzed in conjunction with the important features or functionalities of the protocol. In the case of MQTT implementations, the most important files are those related to features of the pub/sub design pattern, such as message publications and topic subscriptions. In that regard, the fuzzers are evaluated by measuring the number of statements executed in pub/sub-related files of Mosquitto:
handle_publish.c, handle_subscribe.c, and subs.c. The number of statements executed in handle_connect.c is also measured in order to determine the degree to which connection requests are favored over their pub/sub counterparts.

5.2.1 handle_connect.c

Figure 5.3 presents the number of statements (as a percentage) executed by each fuzzer in 30 minutes, and after exchanging 8000 packets with the broker. Despite the two different stopping criteria, coverage percentages shown in Figure 5.3 vary slightly and are in the same decreasing order.

![Figure 5.3: Statement Coverage in handle_connect.c](image)

FUME achieves the highest coverage in 30 minutes, executing approximately 67.96% of the statements. Furthermore, FUME executes 64.04% of the statements after exchanging 8000 packets with Mosquitto, outperforming its counterparts by a wide margin. The outperformance of FUME is directly linked to the overabundance of connection requests performed during testing. This is due to the following three reasons. First, FUME performs multiple tasks, including connection requests, simultaneously, after which it disconnects from the broker and repeats the entire process until a stopping criterion has been satisfied. For each test scenario, the interaction between FUME and Mosquitto is considerably short, meaning the latter handles connection requests much more frequently than pub/sub-related operations. For each scenario, FUME sends packets unrelated to connection and disconnection requests in random order, which does not comply with the rules provided by the standard. As a result, Mosquitto closes the connection with FUME immediately, halting message publications or any other pub/sub functionality in that particular test scenario. Connections are closed within a short time frame, meaning FUME reconnects to the broker very frequently. Third, FUME generates many PUBLISH packets incorrectly, which causes the broker to close the connection. FUME then establishes a new connection with the broker to continue the tests.

Similar to FUME, mqtt_fuzz performs connection requests frequently. As a result, mqtt_fuzz executes approximately 54.17% of the statements in 30 minutes, outperforming most of its counterparts, except for FUME. This outcome also holds true when mqtt_fuzz
and Mosquitto exchange 8000 packets, which results in 50.52% of the statements being executed. In contrast to FUME, mqtt_fuzz interacts with the broker *dynamically*, requesting and responding to messages accordingly. In order to accomplish this goal, mqtt_fuzz *correctly* generates and sends messages to the broker on most occasions. For example, mqtt_fuzz is incapable of generating test cases for username and password authentication, which cause Mosquitto to close the connection immediately. This in turn makes mqtt_fuzz reconnect more frequently to Mosquitto than most fuzzers, except for FUME. Connection requests are also more prevalent because mqtt_fuzz uses a mutation-based approach, which creates and sends test cases to Mosquitto faster than their generation-based counterparts. It is also worth noting that mqtt_fuzz performs disconnection requests at any given moment during testing. In that regard, the vast majority of disconnections mainly occur upon request of mqtt_fuzz rather than Mosquitto.

learner executes approximately 31% of the statements in 30 minutes and after exchanging 8000 packets with Mosquitto. Likewise, CyberExploit achieves the exact coverage performance when satisfying both conditions, executing approximately 24% of the statements. The coverage performance across both stopping conditions is identical because learner and CyberExploit perform exactly the same connection request for the entire test run. However, learner outperforms CyberExploit because of two reasons. First, learner performs more connection requests than CyberExploit. In fact, within the first four minutes of testing, learner performs only connection-related requests. The initial test period therefore plays a key role in the success of learner over CyberExploit. Second, learner uses a username and client ID to authenticate with the broker, whereas CyberExploit opts only for the latter. learner further differentiates itself from CyberExploit by performing connection requests with properties specific to MQTT 5.0. The aforementioned differences allow learner to outperform CyberExploit.

All variants of MQTTGRAM perform up to 4x fewer connection requests than their counterparts. Among all variants of MQTTGRAM, mqttgram-r achieves the highest coverage performance, executing approximately 49.97% of the statements in 30 minutes. mqttgram5-cf, mqttgram-cf, and mqttgram-c execute 48%, 46.81%, and 46.55% of the statements, respectively. Coverage performances are fairly similar after exchanging 8000 packets with the broker. mqttgram-r and mqttgram5-cf execute approximately 49.46% and 47.42% of the statements, respectively. mqttgram-cf and mqttgram-c achieve a nearly identical coverage performance, each executing 46.13% and 46.11% of the statements.

Among all fuzzers, fuzz() achieves the lowest coverage performance. fuzz() executes 6.96% and 6.56% of the statements in 30 minutes, and after exchanging 8000 packets with the broker, respectively.

### 5.2.2 handle_publish.c

Figure 5.4 presents the statement coverage percentage achieved by the MQTT fuzzers in the file handle_publish.c.

Despite executing the largest number of statements in handle_connect.c, FUME fails to outperform most fuzzers in handle_publish.c except for fuzz(), which executes no statements whatsoever. As a result, FUME achieves the second-lowest performance, execut-
ing approximately 11.1% of the statements in 30 minutes, and only 3.9% after exchanging 8000 packets with Mosquitto. The low coverage performance can be attributed to the fact that most of the publish requests performed by FUME are invalid.

CyberExploit executes 25% of the statements within the first three minutes of testing, and before exchanging 500 packets with Mosquitto. Despite outperforming FUME within a short time frame, CyberExploit executed no more than 25% of the statements across all of its test runs. The lack of a coverage increase is due to CyberExploit performing the same publish request for most of the test run.

mqttgram-cf and mqttgram-c slightly outperform CyberExploit, executing 31.09% and 31.02% of the statements respectively in 30 minutes. mqtt_fuzz surpasses the coverage performance of mqttgram-cf and mqttgram-c, executing 34.66% of the statements. Despite using fewer Publish flags than its aforementioned counterparts, learner achieves a higher coverage performance (35%) because it executes statements specifically written for MQTT 5.0. Aside from learner, only FUME, mqttgram5-cf, and mqttgram-r are capable of executing statements specifically for MQTT 5.0. mqttgram-r and mqttgram5-cf outperform their counterparts by executing at least 12% more statements. More specifically, mqttgram5-cf executes 47.51% of the statements, whereas mqttgram-r executes 47.2%.

Although learner outperforms mqtt_fuzz, mqttgram-c, and mqttgram-cf in 30-minute test runs, a different outcome occurs after exchanging 8000 packets with Mosquitto. All of the aforementioned four fuzzers achieve a near identical performance. However, mqttgram-cf and mqtt_fuzz slightly outperform both learner and mqttgram-c, executing 31.07% and 31.29% of the statements, respectively. It is worth noting that mqtt_fuzz achieves a peak coverage performance of 34%, whereas mqttgram-cf executes at most 33% of the statements. However, their peak coverage performance is considerably surpassed by mqttgram-r and mqttgram5-cf, which execute 47.07% and 47.19% of the statements respectively. mqttgram5-cf achieves a peak coverage performance of 51%, whereas mqttgram-r achieves 50%.
5.2.3 *handle_subscribe.c*

Figure 5.5 presents the statement coverage of the MQTT fuzzers in *handle_subscribe.c*.

![Coverage Percentage (%)](coverage.png)

**Figure 5.5: Statement Coverage in *handle_subscribe.c***

Among all fuzzers, `mqttgram-r` achieves the highest coverage performance, executing 71.76% of the statements in 30 minutes. This outcome is all the more surprising given that `mqttgram-r` exchanges at least 15x fewer packets than FUME, which executes 67.99% of the statements within the same time frame. This result sheds light on two common issues across existing MQTT fuzzers: (1) the abundance of repeated test cases and (2) the lack of domain knowledge, which in turn leads to the same statements being executed repeatedly regardless of how many packets are sent over time.

On average, `mqttgram5-cf` and FUME achieve a similar coverage performance despite the former supporting only MQTT 5.0 and exchanging the fewest packets with Mosquitto in 30 minutes. However, FUME manages to achieve a peak coverage performance of 75%, whereas `mqttgram5-cf` achieves 70%.

`mqtt_fuzz` achieves the fourth-best coverage performance, executing at most 64% of the statements. On average, `mqtt_fuzz` executes 62.67% of the statements, whereas `learner` executes 58%, which is also its best coverage percentage across 100 test runs. In fact, `learner` executes 58% of the statements in approximately 3 minutes, and remains the same for the remainder of the test run. It is worth noting that, at the time of writing, `mqttgram5-cf` and `learner` are the only fuzzers designed specifically for MQTT 5.0. However, between the two fuzzers, `mqttgram5-cf` is the most effective test suite for MQTT 5.0, outperforming `learner` by a margin of nearly 10%. Despite using Mosquitto to infer the message sequence and syntax of MQTT, `learner` underperforms considerably in terms of statement coverage.

The coverage performance between `mqttgram-c` and `mqttgram-cf` is nearly identical, averaging approximately 55.96% and 57.20% respectively in 30 minutes. However, their statement coverage peaks at 60% in 30-minute test runs, outperforming `learner` by 2%. It
is worth noting that the performance of \texttt{mqtt\_fuzz} surpasses that of \texttt{mqttgram\_cf} and \texttt{mqttgram\_c}. The reason behind this outcome is due to \texttt{mqtt\_fuzz} performing subscription requests with both invalid and valid QoS levels, whereas its grammar-based counterparts use only the former type.

\texttt{CyberExploit} achieves the second-lowest performance, averaging and peaking at 12\% of the statements. The reason for its low statement coverage is due to subscription requests being performed mostly in the same manner during testing. Most notably, \texttt{CyberExploit} uses the same identifier (0) and QoS level (0) for every subscription message it generates and sends to the broker. As a result, levels 1 and 2 of QoS are left untested. Subscription topics are the only exceptions to this rule, varying considerably across test cases.

\texttt{fuzz()} fails to execute any statement whatsoever in \texttt{handle\_subscribe.c} because it is incapable of connecting successfully to the broker. As a result, the broker does not receive nor handle any subscription request from \texttt{fuzz()}.

In terms of the packet-based experiments, \texttt{mqttgram\_r} achieves the highest coverage performance, outperforming its counterparts by a margin of at least 4\%. On average, \texttt{mqttgram5\_cf} and \texttt{FUME} execute 65.82\% and 64.1\% of the statements, respectively. Prior to 2000 packets, the statement coverage of \texttt{mqttgram5\_cf} peaks at 68\%, and afterwards increases to 70\%, which is exactly the same result across all of its 30-minute test runs. The statement coverage of \texttt{FUME} peaks at 68\% prior to exchanging 500 packets with Mosquitto, and remains the same for the remainder of the test run. Regardless of the stopping criterion, the coverage performance of both \texttt{mqttgram5\_cf} and \texttt{FUME} are fairly similar, despite the latter supporting multiple versions of the standard.

The remaining six fuzzers, including \texttt{mqtt\_fuzz}, achieve the same ranking when satisfying both stopping conditions. The statement coverage of \texttt{learner} peaks and remains at 58\% during most of the fuzzing campaign. The considerable lack of increase in statement coverage signals that \texttt{learner} performs the same type of subscription request constantly. This reinforces the fact that developers are neglecting the core characteristics of the pub/sub design pattern when building MQTT fuzzers. \texttt{mqttgram\_c} and \texttt{mqttgram\_cf} achieve similar performance, executing 55.04\% and 55.5\% of the statements respectively. \texttt{CyberExploit} underperforms considerably when compared to its counterparts, executing at most 12\% of the statements. Similar to \texttt{learner}, \texttt{CyberExploit} achieves peak performance very early during testing, meaning its subscription-related test cases lack variety. Unsurprisingly, \texttt{fuzz()} executed no statements whatsoever, as is the case in its 30-minute test runs.

5.2.4 \texttt{subs.c}

\texttt{subs.c} may very well be considered the most important file for Mosquitto because its purpose is to publish messages to interested subscribers. In that regard, \texttt{subs.c} is the file that is most associated with the core features of the pub/sub design pattern. MQTT fuzzers should therefore prioritize statement coverage in \texttt{subs.c}. Figure 5.6 reveals that the variants of MQTTGRAM are the only fuzzers capable of executing over 60\% of the statements across both stopping conditions.

\texttt{mqttgram\_r}, in particular, achieves the best performance among other variants of
MQTTGRAM, executing 66.25% of the statements in 30 minutes. mqttgram-cf and mqttgram-c execute 65.13% and 64.91% of the statements respectively. In fact, among all the variants of MQTTGRAM, mqttgram-cf triggers the highest average number of message publications in 30 minutes (319), followed by mqttgram-r (212), mqttgram5-cf (192), and mqttgram-c (178) respectively. The latter fuzzer triggers the lowest number of message publications because it is designed to generate topic-based packets less frequently than its counterparts (Araújo Rodrigues and Macêdo Batista, 2021). mqttgram5-cf triggers the second-lowest number of publications because its methodology for packet generation is more time consuming than that of its counterparts. This has to do with the fact that version 5.0 of MQTT packets adds more fields and user properties to the payload than past releases. For each MQTT 5.0 packet, mqttgram5-cf needs to randomly select a user property and define its corresponding value, the process of which affects the overall number of test cases generated and sent to the broker over time. mqttgram-r is affected by this issue to a lesser degree because it occasionally generates MQTT 3.1.1 packets, which demand less effort. However, despite triggering the second-highest number of message publications, mqttgram-r executes more statements in subs.c than mqttgram-cf. This result stems from the fact that mqttgram-r supports two versions of the standard, whereas mqttgram-cf supports only one. The superior performance of mqttgram-r serves as an indication of its aim and capabilities in triggering the broker into publishing messages using different parameters, whereas all of their counterparts fail in this regard.

For example, FUME triggers the broker into publishing at least 8x more messages (2769) to interested subscribers in 30 minutes than all variants of MQTTGRAM. Despite the large number of message publications, FUME executes 61.54% of the statements, slightly below the average performance of its grammar-based counterparts. This is surprising given the fact that FUME supports all major versions of MQTT, whereas its counterparts support one or two releases at most. Another surprising fact is the number of packets exchanged between FUME and Mosquitto. The higher the number of packets exchanged with the broker, the greater the chances of executing more statements and triggering additional message publications. In that regard, FUME exchanges 546,284.2 packets with
Mosquitto in 30 minutes, out of which only 268.6 (0.05%) are publish requests. The lack of message publications is further hindered by the fact that most publish requests performed during testing are invalid, hence its performance drop to 53.89% in 8000-packet test runs. As a result, FUME struggles to trigger Mosquitto into publishing messages to interested subscribers, meaning the core functionality of the pub/sub pattern is neglected to a certain degree during testing. Although FUME performs approximately 5745 subscription requests during the test run, these efforts also seem to be of little avail.

Among MQTT fuzzers, mqtt_fuzz performs the most subscription and publish requests, averaging 6868 and 4926 respectively in 30 minutes. This leads mqtt_fuzz to trigger 15012.1 message publications in 30 minutes, outperforming all of its counterparts by a wide margin. Despite the large number of message publications, mqtt_fuzz is slightly outperformed by FUME and MQTTGRAM-based fuzzers in terms of statement coverage, which averages approximately 58.97%. It is worth noting that mqtt_fuzz achieves a coverage performance of 58.02% in 3 minutes, meaning it increments its statement coverage by approximately 1% at the end of the test run.

learner, on the other hand, is incapable of increasing its coverage performance after 3 minutes, executing at most 56% of the statements. This is due to the fact that learner uses the same connection, publish, and subscription parameters for the entire test run. This is also the case for CyberExploit, which its coverage percentage never surpasses 25% after 3 minutes. In addition to the same parameters, CyberExploit and learner assign the same values to the majority of the fields in connection- and pub/sub-related packets, the sole exceptions being subscription topics and their corresponding messages. However, in contrast to learner, at no time during testing does CyberExploit trigger the broker into publishing messages. The lack of message publications causes CyberExploit to achieve the second-lowest coverage performance among MQTT fuzzers. Likewise, similar to CyberExploit, fuzz() executes 24% of the statements due to also being incapable of triggering message publications. It is worth noting that fuzz() achieves the same coverage percentage (24%) after exchanging 8000 packets with Mosquitto, as shown in Figure 5.6. CyberExploit executes 25% of the statements, which is also its same coverage percentage in 30-minute test runs.

A major performance difference shown in Figure 5.6 is with regards to FUME, which drops to 53.89% when executing 8000-packet test runs. The reason behind the performance drop has to do with the fact that FUME struggles to generate valid publish packets, requiring more time than its counterparts to execute certain statements. Unsurprisingly, learner achieves the same result as in 30-minute test runs, executing 56% of the statements. The coverage performance of mqtt_fuzz varies slightly across both stopping conditions, executing 57.79% of the statements in 8000-packet test runs. The variants of the MQTTGRAM are the only fuzzers capable of executing over 60% of the statements. mqttgram-r achieves the highest coverage performance (64.63%), followed by mqttgram-cf (64.51%), mqttgram-c (63.09%), and mqttgram-5cf (61.58%), respectively.

After evaluating the performance of mqttgram-r, the answer to RQ4 is as follows:
RQ4: How effective is an MQTT fuzzer when considering three essential elements for pub/sub fuzzing: two-way communication capabilities; topic awareness; and multiversion support?

Chapter 4 identified FUME as the most effective MQTT fuzzer proposed in the literature thus far. However, mqttgram-r manages to outperform all of its state-of-the-art competitors, including FUME, in terms of standardized testing metrics such as code coverage, executing up to 12x more statements in pub/sub-related files. This result is all the more surprising given that mqttgram-r supports fewer versions of MQTT than FUME, and still manages to achieve the highest statement coverage. mqttgram-r and FUME execute at most 2321 and 2311 statements, respectively. On average, mqttgram-r executes from 2131 to 2276 statements in 30 minutes, whereas FUME executes from 2138 to 2242 statements. The original MQTTGRAM underperforms considerably compared to mqttgram-r, executing at most 2140 statements in 30 minutes. These results therefore confirm the research hypothesis, which stated that the effectiveness of an MQTT fuzzer is higher when considering the three essential elements.

5.3 Concluding Remarks

Chapter 4 determined that all fuzzers have issues regarding pub/sub protocol testing. This chapter therefore presented and evaluated a new fuzzer based on an architecture that developers can use as a reference to test MQTT or other pub/sub protocols. The open-source fuzzer introduced in this chapter outperforms all of its state-of-the-art competitors in terms of pub/sub testing. Not only did this chapter present the new multiversion grammar-based approach for a pub/sub protocol, but it also reinforced the importance of the contributions presented in previous chapters of this thesis for the development of better testing tools. This chapter can therefore help raise awareness and motivate developers to better design their fuzzers considering the three essential elements for successful pub/sub fuzzing in future studies.
Chapter 6

Stress Test Evaluation of Fuzzing Techniques for MQTT Brokers

Centralized pub/sub architectures, such as the one shown in Figure 6.1, have been adopted for several smart city applications, including transportation systems, traffic control, and healthcare services.

![Centralized Broker Architecture for Pub/Sub Messaging](image)

As previously stated, a major disadvantage of centralized architectures, when compared to their distributed counterparts, is that the broker most often becomes a performance bottleneck because it is the single entity responsible for data transmission and storage (Johnsen et al., 2018). 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 (2006), 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 (MITRE Corporation, 2006). Furthermore, resource exhaustion is
one of the most common types of DoS attacks in IoT according to the literature (Mrabet et al., 2020).

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 (Oliveira et al., 2019). The MITRE Corporation (2006) lists fuzz testing as one of the three main detection methods for resource-exhaustion problems. However, in order to guarantee that the broker is robust and reliable enough against performance bottlenecks or issues in smart cities, a fuzzer should perform CPU- and memory-intensive tests (Săndescu et al., 2018). Despite the wide range of broker fuzzers available, there is a lack of information in the literature regarding their capabilities to perform resource-exhaustion attacks during the test run.

This chapter therefore aims to answer the following research question:

**RQ5: How effective are existing fuzzing strategies at impacting 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 (Mishra and Kertesz, 2021), none have evaluated the capabilities of existing fuzzing tools for stress-testing purposes. This is therefore the first research to investigate about the fuzzers’ capabilities in regards to resource exhaustion. This chapter also highlights questionable design choices that prevent existing fuzzers from consuming more system resources.

### 6.1 State of the Art

At the time of writing, broker fuzzers proposed in the literature are mostly for MQTT, which research studies have demonstrated is extremely vulnerable to resource-exhaustion attacks (Kim et al., 2017). 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. Despite these issues, resource-related metrics are barely used in the literature for comparing MQTT fuzzers. At the time of writing, only Hernández Ramos et al. (2018) evaluated their fuzzer by considering the CPU consumption of the broker.

Although none of the broker fuzzers presented in the previous chapters 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. A resource-guided fuzzer receives and analyzes resource-related feedback about the broker while fuzzing. Due to their nonexistence, 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
6.2 | PERFORMANCE EVALUATION

far preferred the latter, as evidenced by the research studies conducted by *Fehrenbach* (2017) and *Morelli et al.* (2021), which propose to use or extend an open-source MQTT fuzzer to perform more complex resource-exhaustion attacks. Second, the tools proposed by *Fehrenbach* (2017) and *Morelli et al.* (2021) are proprietary, meaning developers have no choice but to select one of the state-of-the-art fuzzing frameworks evaluated in this research study. 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 (*Wen et al.*, 2020) or ResFuz (*L. Chen et al.*, 2022), 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 this research study.

### 6.2 Performance Evaluation

This section evaluates six open-source MQTT fuzzers (*Philippe Biondi and the Scapy Community*, 2023; *F-Secure Corporation*, 2015; *Casteur et al.*, 2020; *Aichernig et al.*, 2021; *Araujo Rodriguez and Macêdo Batista*, 2021; *Pearson et al.*, 2022). Mosquitto 1.6.8 and Moquette 0.13 were chosen as the target brokers because of their popularity in the literature (*Hernández Ramos et al.*, 2018; *Bender et al.*, 2021; *Sochor et al.*, 2020b). The MQTT brokers and fuzzers were evaluated using default settings. The CPU and memory usage of the brokers are monitored throughout the fuzzing campaigns. The more CPU- and memory-intensive inputs are sent to the brokers, 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 each stopping criterion. Two separate experiments were conducted, 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 brokers (8000). Resource usage is measured after every three minutes and after 500, 1000, 2000, 4000, 6000, and 8000 packets. The fuzzers are referred to by either their designated name (if available) or main characteristic. Similar to the previous chapters, the fuzzer by *Aichernig et al.* (2021) is referred to as *learner* due to lacking a formal name. It is worth noting that Moquette lacks support for MQTT 5.0, hence the reason why it could not be tested with *learner* and *FUME*. The testbed shown in Figure 1.3 was used for the performance evaluation.

#### 6.2.1 Mosquitto

Figure 6.2 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 both grammar-based approaches remains at 0.10% throughout most of the fuzzing campaign.

Based on the 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 Figure 6.3, further confirm this limitation.

Figure 6.4 presents the memory usage of Mosquito when fuzzing for 30 minutes.

FUME peaks Mosquito’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 Mosquito 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 the 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 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 Figure 6.5.

![Figure 6.5: Average Memory Usage of Mosquitto When Exchanging 8000 Packets.](image)

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.

6.2.2 Moquette

All fuzzers manage to either maintain or increase Mosquitto’s CPU usage during the test run. In contrast, Moquette’s CPU usage gradually decreases over time, as shown in Figure 6.6.

![Figure 6.6: Average CPU Usage of Moquette During 30 Minutes.](image)

For example, mqtt_fuzz consumes the highest amount of CPU on Moquette, reaching up to 4.00%. However, CPU usage decreases from 3.66% to 1.84% on average at the end of 30 minutes. Moquette’s CPU usage also decreases when testing it with CyberExploit, lowering from 1.46% to 0.58%. Both variants of MQTTGRAM performed in a similar manner, consuming at most 1.50% of CPU usage, which decreases to roughly 0.60% at the end of the fuzzing campaign. Moquette’s CPU usage peaks at 6.70% when exchanging 8000 packets with mqtt_fuzz, and gradually decreases over time, as shown in Figure 6.7.

![Figure 6.7: Average CPU Usage of Moquette When Exchanging 8000 Packets.](image)
Moquette’s memory usage is fairly similar to that of Mosquitto, as shown in Figure 6.8.

In fact, fuzzers rank in the exact order across both brokers at the end of 30 minutes. For example, among 3.x compatible fuzzers, mqtt_fuzz consumes the most memory on both Mosquitto and Moquette. Due to the identical rank order at the end of 30 minutes, all explanations about Mosquitto’s fuzzing campaigns also apply to those of Moquette. In fact, Moquette’s memory usage throughout 30 minutes further confirms that most fuzzers, except for FUME, are incapable of performing memory-intensive tests for pub/sub brokers. The average memory usage by 3.x compatible fuzzers ranges from 5.90% to 9.00%, and peaks at 10.40%.

Fuzzers rank differently, however, when exchanging 8000 packets with Moquette, as shown in Figure 6.9.

Unexpectedly, fuzz() outperforms other fuzzers in terms of average memory usage, which ranges from 5.95% to 6.55%. Both fuzz() and mqttgram-cf consume at most 7.50% of memory, whereas the remaining fuzzers use no more than 6.60%.

Overall, the fuzzers consume few CPU and memory resources on Moquette. These results support the major finding of this research study, which is that existing MQTT fuzzers are incapable of intensively testing both a broker’s CPU and memory.

### 6.3 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 chapter, performing multiple requests simultaneously refers to 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 (See Appendix C, Figure C.4). Future work should attempt to develop fuzzers that apply both approaches to increase the broker’s workload.

Second, storing significant amounts 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, 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.

The 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 paper, it is strongly recommend that developers incorporate resource-exhaustion techniques into their MQTT fuzzers.

After analyzing the resource-consumption of existing MQTT fuzzers, the answer to RQ5 is as follows:

**RQ5: How effective are existing fuzzing strategies at impacting the CPU and memory usage of the broker during testing?**

The 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%, which is more than 20% when compared to its counterparts.

### 6.4 Concluding Remarks

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 chapter presented a performance evaluation of the most popular open-source fuzzers in the literature. Developers can use the guidelines presented in this chapter to either improve existing fuzzers or incorporate them into their own testing tool.
Chapter 7

Conclusions and Future Work

Fuzzing has made considerable progress across multiple domains. However, a research field that has been largely neglected is pub/sub protocol fuzzing, which is important to guarantee the reliability of message brokers deployed to the Internet of Things. The aim of this PhD research was therefore to study and develop effective fuzzing strategies for pub/sub protocols, aiming to contribute to the development of more robust applications in IoT and Smart Cities. This thesis therefore began by analyzing the research advancements and trends of pub/sub protocol fuzzing since its inception (Chapter 2). In that regard, the thesis focused primarily on MQTT, the only pub/sub protocol for which developers have proposed a wide variety of fuzzing techniques in the literature. Despite focusing specifically on MQTT, this research can serve as an initial step towards developing effective fuzzing strategies for pub/sub protocols in general, raising awareness and encouraging developers to better test brokers considering their message-publishing features and event-driven architecture.

It was evident after reviewing the literature that several research gaps needed to be filled in order to further advance the field of pub/sub protocol fuzzing. For example, information regarding established testing techniques such as grammar-based fuzzing for pub/sub protocols was mostly nonexistent in the literature, despite developers and researchers expressing interest. As a result, a new grammar-based approach for a pub/sub protocol (MQTT) was proposed and incorporated into a new fuzzer called MQTTGRAM, which developers can now use as a reference for testing purposes (Chapter 3).

MQTTGRAM joined several other fuzzers in providing a different testing strategy to developers and researchers, leading to a promising, yet complicated situation. On the one hand, developers now have several different and varied options at their disposal to test MQTT, but on the other hand the sheer number of techniques hinders decision-making in choosing the most appropriate or effective for a given situation. This hindrance will also have a negative impact on future studies, which will be unable to discern promising research directions or advancements in the field of pub/sub protocol fuzzing. As a result, a taxonomical classification (Chapter 4, Figure 4.4) and performance benchmark of fuzzing techniques for MQTT were presented in this thesis to clarify inquiries regarding their effectiveness. Due to

---

the sheer amount of fuzzing techniques proposed for MQTT, one could have easily assumed that developers have a wide range of options to test their brokers’ pub/sub functionalities. However, upon further investigation, the research findings indicated that all fuzzers have shortcomings regarding pub/sub protocol testing (Chapter 4). The root of the problem stems from the fact that all techniques were proposed without considering three essential elements for testing message-publishing features: (1) Two-way communication; (2) Topic awareness; and (3) Multiversion support. The algorithm and architecture of MQTTGRAM were refined to meet the three criteria (Chapter 5)\(^2\), outperforming all state-of-the-art fuzzing techniques in terms of statement coverage, which is an established metric to measure the quality of test suites. In addition to test metrics, the fuzzers were evaluated based on their stress-testing capabilities in order to incentivize future studies in that field (Chapter 6). The testbed\(^3\) used for the doctoral studies is also freely available for future work.

### 7.1 Future Work

Considering that pub/sub fuzzers are mostly for MQTT at the time of writing, the research findings presented in this thesis highlight the current state of both MQTT and pub/sub protocol fuzzing in general. This section, however, focuses more on the lessons learned and future research directions to move both fields forward.

#### 7.1.1 Improvements to Existing Fuzzing Techniques

**Learning-based fuzzing**

Automatic knowledge acquisition using learning-based algorithms has been extensively researched to mitigate development and configuration efforts for fuzzers. However, the main shortcoming of learning-based fuzzers is that their knowledge is based more on protocol implementations rather than formal specifications. This approach also comes at the risk of putting more effort and time into learning the protocol rather than actually testing it, as evidenced by the subpar performance of learner. Developing a successful learning-based algorithm tailored specifically to pub/sub protocols such as MQTT is a challenge that has yet to be mitigated. Future work should focus on improving learning-based algorithms, and learner specifically, in two areas. First, in its current state, the learning-based algorithm for learner is incapable of understanding several core pub/sub functionalities of MQTT. The algorithm needs to better infer the syntax rules and message sequence from a given MQTT implementation. Second, learner takes approximately 4 minutes to generate interesting test cases as it interacts with the broker. learner should therefore be improved to gain knowledge of MQTT either quicker, or before rather than during testing.

---


Scenario-based fuzzing

Generating different types of scenarios is a useful, promising, and suitable technique to test specific functionalities of a target system. Pub/sub protocols, in particular, tend to be more scenario-oriented than other network-based systems due to their feature-rich and requirement-heavy nature. For example, there are at least three possible scenarios related to message publication that are worth considering when testing pub/sub protocols. First, a broker will discard publish messages if there are no interested subscribers. Second, a broker will send topic messages immediately to connected subscribers. Third, the broker will retain and send topic messages after the subscriber reconnects to the network. Existing scenario-based fuzzers fail to generate the latter two scenarios, meaning the broker discards topic messages throughout the entire test run. The fuzzer proposed by Di Paolo et al. (2021) is currently incapable of performing subscription requests, whereas CyberExploit is unable to trigger the broker into publishing messages. Another common misstep found across MQTT fuzzers is their short-lived test scenarios, each consisting of a single pub/sub packet. Scenario-based fuzzers must be designed around the idea that a message will be published only if there is at least one interested subscriber. For each test scenario, the fuzzer should at least perform a valid subscription request, and then publish a message to an existing topic. For future work, scenario-based fuzzers should therefore be more tailored to the pub/sub design pattern.

Mutation-based fuzzing

A mutation-based fuzzer should trigger the broker into publishing messages regardless of how a packet is modified. Mutation-based fuzzing, in the context of pub/sub protocols, must therefore be performed carefully, otherwise the broker will very rarely send messages to interested subscribers. For example, a broker publishes messages only to existing topics. Mutating a single character in an existing topic could potentially generate a new value that may be inconsistent with prior subscription requests. As a result, the mutated message will be discarded by the broker, preventing it from reaching subscribers. Proxy fuzzers, in particular, are more prone to this issue because they mutate packets in transit without considering user subscriptions or sessions.

Coverage-guided fuzzing with mutation testing is an approach considered less effective for network protocols (Yurong Chen et al., 2019). In fact, coverage-guided fuzzers such as AFLNet and its variant AFLNet-MQTT are incapable of behaving as genuine MQTT clients because their current architecture is unsuitable for pub/sub communications (Zeng et al., 2020; Sneha Suhitha Galiveeti and Pranitha Malae, 2020). According to Sneha Suhitha Galiveeti and Pranitha Malae (2020), the architecture of their fuzzer hinders its coverage performance significantly, becoming a major issue when testing MQTT implementations. At the time of writing, only MultiFuzz attempts to mitigate the shortcomings of the architecture through the use of message-aware mutation operators. Future work should continue mitigating limitations of the architecture in order to better support the pub/sub design pattern.

Another major issue is the time-consuming nature of coverage-guided approaches to discover new code paths during testing, as evidenced by the performance evaluation of MultiFuzz (Zeng et al., 2020) and AFLNet-MQTT (Sneha Suhitha Galiveeti and Pranitha Malae, 2020).
Pranitha Malae, 2020). This task is far more complex for network-based fuzzers, which have to discover coverage-increasing test cases, while still maintaining an active connection with the broker. Future work should attempt to develop fast coverage-guided approaches capable of supporting several versions of MQTT.

**Hybrid-based fuzzing**

Combining mutation- and generation-based techniques seems promising for future research because it allows developers to take advantage of their strengths during the test run. However, in order to effectively leverage the strengths of both mutation- and generation-based techniques, a hybrid-based fuzzer has to be much more aware of messages and states than its counterparts. The main challenge lies in carefully choosing between mutation- or generation-based techniques during testing. Otherwise, the fuzzer will make poor decisions regarding the type of packet generation that is most appropriate for a given situation. In the context of pub/sub protocols, there are multiple factors to consider when making a decision, such as subscriptions or user sessions. Thus, a hybrid-fuzzer for MQTT has to be smart enough to choose the technique that will enable it to reach pub/sub states in a more effective and efficient manner. FUME fails to meet this requirement because it is unaware of most state transitions, except those necessary to connect to and disconnect from the broker. Future work should focus on improving both the finite state machine from which FUME is modeled by, and the decision-making process for the most appropriate packet generation technique during testing.

**Grammar-based fuzzing**

Future work should combine grammar-based fuzzing with multiple techniques presented in the taxonomy. For example, the grammar can be used to generate vulnerability-oriented packets. Coverage-guided fuzzing should also be used with the MQTT grammars in order to gain feedback during testing, and thus attempt to generate packets that traverse through undiscovered code paths.

### 7.1.2 Development of Built-In Fault Detection Mechanisms

Most MQTT fuzzers lack fault detection mechanisms, requiring external debugging tools to monitor the behavior of the broker during testing. For example, this thesis uses AddressSanitizer as a tool to detect run-time errors or memory leaks in Mosquitto throughout the fuzzing campaign. Casteur et al. (2020), Di Paolo et al. (2021), and Sochor et al. (2020b) analyze system logs after each test run for any sort of indication regarding crashes or unhandled exceptions. The Polymorph fuzzing framework by Hernández Ramos et al. (2018) consists of five modules, none of which monitor exceptions. The repository description of mqtt_fuzz states that brokers should be compiled using debugging tools such as AddressSanitizer or gdb to detect potential flaws during testing. It is impractical to determine if Defensics monitors exceptions during testing because its source code is unavailable. Future work should focus on developing fuzzers that are able to detect faults in the broker. At the time of writing, MultiFuzz, AFLNet-MQTT, FUME and learner are the only fuzzers that possess built-in fault detection mechanisms.
7.1.3 Development of a Resource-Intensive Fuzzer for MQTT Brokers

A resource-intensive fuzzer should be developed considering the lessons learned from the research study presented in Chapter 6. For future work, additional experiments should also be conducted with other broker-side implementations such as HiveMQ, VerneMQ, and RabbitMQ. A simple traffic generator should be developed in order to provide baseline results against which to compare the stress-testing capabilities of existing and subsequent fuzzers.

7.1.4 Development of Fuzzers Based on Hybrid and Machine Learning Approaches

Although promising, the fuzzing techniques proposed for MQTT thus far are either underdeveloped or unsuitable for pub/sub protocols, as evidenced by the research findings presented in the previous chapters. Future work should delve deeper into two promising techniques for pub/sub protocols. First, hybrid-based approaches that incorporate elements of both grammar-based and mutation-based fuzzing should be further studied for pub/sub protocols. Second, fuzzing techniques based on machine learning should also be further explored.

7.1.5 Performance Evaluation Over Longer Test Runs

It is worth clarifying that the broker was constantly monitored during the test runs in the case of an unexpected crash. However, none of the fuzzers were capable of crashing the broker within 30 minutes. Future work should attempt to evaluate the fuzzers over longer test runs in order to further identify shortcomings and areas of improvement.

7.1.6 Maintenance of MQTT Fuzzers

Most, if not all, research proposals lack information regarding their fuzzers’ flexibility to evolve in lockstep with MQTT itself. For example, in its current form, learner is only applicable to MQTT 5.0, meaning it requires adjustments for future versions. The emergence of FUME in early 2022 further diminishes the relevance, longevity, and necessity of learner due to the former’s built-in knowledge of the message syntax for version 5.0. Among other examples, Sochor et al. (2020b) developed an attack-based fuzzer that, while promising, requires further maintenance as more types of implementation flaws emerge. The simple mutations performed by mqtt_fuzz to a given packet will be less effective as MQTT implementations become more robust over time, requiring further improvement. FUME supports all major versions of MQTT, extending its lifespan even further. However, no information is given regarding its flexibility to evolve as new versions of MQTT are released. The effectiveness of proxy fuzzers, proposed by Hernández Ramos et al. (2018) and Eclipse Foundation (2018), is mostly unaffected by new version releases because they simply mutate packets that are transmitted from one endpoint to another, lacking packet generators or any sort of knowledge regarding state transitions. Future work should attempt to propose flexible approaches that can easily evolve over time.
7.1.7 Expand the Research Scope to Include Other Pub/Sub Protocols

MQTT is the most popular pub/sub protocol in terms of the number of fuzzing techniques. However, future work should focus on advancing the state of the art by studying fuzzers for other pub/sub protocols such as AMQP, DDS, and XMPP. Most importantly, future work should adapt existing fuzzing strategies for MQTT to other pub/sub protocols, analyzing their strengths and weaknesses more in depth.
Appendix A

Usage of MQTT in IoT Applications

MQTT is used for several technologies, including Facebook Messenger, Amazon Web Services, Arduino, and Mongoose OS (MAGGI et al., 2018). MQTT is also currently being integrated into several IoT-based applications, such as home automation systems; health monitoring systems; Intelligent Transportation Systems (ITS); and electricity metering systems.

A.1 Home Automation Systems

Future smart cities must integrate home automation systems into their infrastructure (SCHIEFER, 2015). A home automation system connects lighting, heating, ventilation, and electronic devices, which interact intelligently to automate domestic tasks. Over the last few years, home automation systems have grown in terms of popularity. Open source home automation software such as HomeAssistant ¹, HomeGear ², Domoticz ³, and OpenHab ⁴ are compatible with MQTT, allowing users to integrate custom-made or publicly available brokers ⁵.

A.2 Health Monitoring Systems

Over the last few years, the healthcare industry has shown interest in using IoT technologies to improve services (CASINO et al., 2015). Health monitoring systems use sensors to collect information, such as a patient’s pulse rate, which is then transferred to hospitals for further analysis. Health monitoring systems provide significant advantages over traditional

healthcare methods. In fact, studies have confirmed the benefits of MQTT in the healthcare sector (Yi et al., 2016; Georgi and Le Bouquin Jeannès, 2017). For example, MQTT offers real-time communication, which is necessary for efficient monitoring systems (Georgi and Le Bouquin Jeannès, 2017). Real-time communication allows medical practitioners to access data immediately and continuously. The ease of access to medical data reduces frequent visits of senior citizens; increases health awareness; and minimizes healthcare costs (Khan et al., 2019).

A.3 Intelligent Transportation Systems

ITS offer services that improve the overall transport and traffic management. MQTT has emerged as a popular and effective protocol for bus-tracking systems because of its reliability and scalability. Several research papers propose real-time bus-tracking systems using MQTT to transfer the bus’ location to an external server for further analysis (Sharad et al., 2016; Lohokare et al., 2017; Pianez et al., 2017; Rutke, 2019). Users can monitor the bus’ location with a website or mobile application. These recent research studies confirm the popularity and interest of MQTT for ITS.

A.4 Electricity Metering Systems

Smart Grids are applications that distribute and generate energy efficiently (Engel, 2013). A Smart Grid has three primary objectives: distribute energy efficiently; provide information and feedback to clients; and offer secure and integrated services (Curiale, 2014). A key component of Smart Grids is Smart Electricity Metering, which is used to capture information related to energy consumption, and transfer it to the service provider. Miškuf et al. (2017) propose an MQTT-based metering system that measures and analyzes energy consumption, highlighting the benefits of MQTT in electricity metering systems.
Appendix B

Comparison Between MQTT and Other IoT Protocols

This section highlights the differences between MQTT and other IoT protocols. The explanation of each protocol is in chronological order.

B.1 HTTP

The Hypertext Transfer Protocol (HTTP) emerged in 1991 for the World Wide Web. It is based on the client-server model and runs on top of the Transmission Control Protocol (TCP). The size of the header and message depends on the web server or programming technology (Naik, 2017). Unlike MQTT, HTTP is power inefficient and lacks QoS. Despite its popularity, HTTP’s slow performance and large memory footprint makes it unsuitable for IoT environments (Naik, 2017).

B.2 XMPP

XMPP was introduced in 1999, and designed originally for instant messaging. However, it has also been used as a pub/sub messaging protocol for IoT. XMPP runs on top of TCP and is standardized by the Internet Engineering Task Force (IETF) 1. An advantage of XMPP is its built-in support for Transport Layer Security (TLS), offering more security by default. Unlike MQTT, XMPP lacks levels of QoS, being less flexible and more prone to packet loss in unreliable networks as a result (Dizdarević et al., 2019). XMPP’s message size is larger than MQTT, hindering its performance in constrained networks (Dizdarević et al., 2019). In fact, a current challenge is optimizing XMPP for low-powered devices (Dizdarević et al., 2019; Wang et al., 2017).

B.3 AMQP

AMQP was introduced in 2003, and also runs on top of TCP. Similar to MQTT, AMQP is standardized by the Organization for the Advancement of Structured Information Standards (OASIS) \(^2\), and its control packets are divided into three fields. The *Frame Header* is required for every packet. However, unlike MQTT, the frame header has a size of eight bytes. The remaining two fields, *Extended Header* and *Frame Body*, are optional, and their size depends on the broker or programming technology. Despite offering three levels of QoS (Dizdarević et al., 2019), AMQP’s main drawback is its large code footprint, which makes it unsuitable for constrained devices (Dizdarević et al., 2019). Unlike MQTT, AMQP lacks features such as Last Will and Testament, Retained Messages, and Persistent Sessions.

B.4 DDS

DDS is a publish-subscribe messaging protocol introduced in 2004 by the Object Management Group (OMG). DDS’s distinguishing feature is its decentralized broker architecture (Al-Fuqaha et al., 2015), which enables peer-to-peer communications between the network clients (Dizdarević et al., 2019). In contrast to MQTT and AMQP, DDS offers 23 levels of QoS (Al-Fuqaha et al., 2015). However, the main drawback of DDS is its high bandwidth consumption compared to MQTT (Yuang Chen and Kunz, 2016).

B.5 CoAP

The Constrained Application Protocol (CoAP) is based on the client-server model for M2M communications (Maggi et al., 2018), and emerged in 2010 as a lightweight alternative to HTTP. Unlike MQTT, CoAP runs on top of the User Datagram Protocol (UDP), meaning it has low latency, but is more prone to packet loss. Message delivery is even more unreliable when considering that CoAP offers limited support for QoS. In contrast to MQTT, CoAP lacks levels of QoS, relying only on acknowledgment signals for message delivery confirmation. For added security, CoAP supports protocols such as the Datagram Transport Layer Security (DTLS) (Halabi et al., 2018).

B.6 Summary

Table B.1 presents a brief comparison of IoT protocols.

Among all protocols presented in Table B.1, MQTT is the preferred choice by IoT developers because of two reasons (Eclipse Foundation, 2022). First, it is lightweight compared to traditional protocols such as HTTP. A study by Yang Chen and Kunz (2016) further confirms that MQTT consumes less bandwidth than CoAP and DDS. Second, it can be used for real-time communication, which is appropriate for smart city applications such as home automation systems, health monitoring systems, and intelligent transportation systems.

\(^2\) http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-overview-v1.0.html. Accessed on March 23rd, 2023
systems (Jaloudi, 2019). Home automation systems tend to provide better support for MQTT than XMPP and AMQP. For example, HomeAssistant and Domoticz have extensive support for MQTT; limited support for XMPP; and no support for AMQP. HomeGear has built-in support for MQTT; and no support for XMPP and AMQP.

It is worth noting that among all pub/sub protocols discussed in this section, MQTT is considered the most popular for IoT (Zeng et al., 2020). Testing MQTT in depth is therefore necessary considering its popularity and benefits over its counterparts.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>QoS</th>
<th>Transport</th>
<th>Fixed Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>No</td>
<td>TCP</td>
<td>Undefined</td>
</tr>
<tr>
<td>XMPP</td>
<td>No</td>
<td>TCP</td>
<td>Undefined</td>
</tr>
<tr>
<td>AMQP</td>
<td>3 Levels</td>
<td>TCP</td>
<td>8 Bytes</td>
</tr>
<tr>
<td>DDS</td>
<td>23 Levels</td>
<td>TCP</td>
<td>Undefined</td>
</tr>
<tr>
<td>CoAP</td>
<td>Limited</td>
<td>UDP</td>
<td>4 Bytes</td>
</tr>
<tr>
<td>MQTT</td>
<td>3 Levels</td>
<td>TCP</td>
<td>2 Bytes</td>
</tr>
</tbody>
</table>

Table B.1: Comparison of IoT Protocols
Appendix C

MQTT Packets

MQTT clients and brokers communicate with one another using control packets. The delivery of these control packets is guaranteed by specifying a QoS level. Certain control packets require a specific level of QoS to be delivered. Table C.1 presents all of MQTT’s control packets.

<table>
<thead>
<tr>
<th>Control Packet Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td>Connection request</td>
</tr>
<tr>
<td>CONNACK</td>
<td>Connection acknowledgment</td>
</tr>
<tr>
<td>PUBLISH</td>
<td>Publish a message</td>
</tr>
<tr>
<td>PUBACK</td>
<td>Publish acknowledgment (QoS 1 &amp; 2)</td>
</tr>
<tr>
<td>PUBREC</td>
<td>Publish received (QoS 2)</td>
</tr>
<tr>
<td>PUBREL</td>
<td>Publish release (QoS 2)</td>
</tr>
<tr>
<td>PUBCOMP</td>
<td>Publish completed (QoS 2)</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Subscribe request</td>
</tr>
<tr>
<td>SUBACK</td>
<td>Subscribe acknowledgment</td>
</tr>
<tr>
<td>UNSUBSCRIBE</td>
<td>Unsubscribe request</td>
</tr>
<tr>
<td>UNSUBACK</td>
<td>Unsubscribe acknowledgment</td>
</tr>
<tr>
<td>PINGREQ</td>
<td>Ping request</td>
</tr>
<tr>
<td>PINGRESP</td>
<td>Ping received</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>Disconnect message</td>
</tr>
<tr>
<td>AUTH</td>
<td>Authentication Exchange (MQTT 5)</td>
</tr>
</tbody>
</table>

Table C.1: MQTT Control Packets

C.1 Structure

An MQTT control packet is divided into three fields: the fixed header; the variable header; and the payload, as shown in Figure C.1.

- **Fixed Header**: The size of the fixed header is only two bytes, thereby enabling lightweight messaging for resource-constrained devices. The bits 7-4 of the fixed
header consist of the type of packet being sent. The remaining bits (3-0) of the first byte are reserved for flags specific to each packet. For example, if a PUBLISH packet is sent to the broker, then the four bits are reserved for three flags: (1) duplicate messages (DUP); (2) QoS level (QoS); and (3) retained messages (RETAIN); as shown in Table C.2. All of the aforementioned features of MQTT, such as retained messages, will be further explained in the following sections. The second byte represents the number of bytes remaining, considering the variable header and the payload.

<table>
<thead>
<tr>
<th>Bits</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1</td>
<td>Control Packet Type</td>
<td>DUP</td>
<td>QoS</td>
<td>RETAIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte 2</td>
<td>Remaining Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table C.2: Fixed Header Format of the PUBLISH Packet

- **Variable Header**: The variable header is optional, being absent in MQTT control packets such as PINGREQ, PINGRESP, and DISCONNECT. The variable header varies depending on the packet type. For example, the variable header of the PUBLISH packet contains the topic length and topic name, whereas the variable header of the SUBSCRIBE packet contains a message identifier.

- **Payload**: The payload is also optional, and varies depending on the packet type. For example, the payload of the PUBLISH packet contains the topic message that will be sent to interested subscribers. In contrast, the payload of the CONNECT packet contains user-related information such as the ID, username, and password.

### C.2 Transmission

Figure C.2 and C.3 present sequence diagrams of MQTT using the control packets. Both figures present different types of scenarios involving a publisher, subscriber, and broker. Figure C.2 presents interactions related to the core functionality of the pub/sub design pattern, whereas Figure C.3 presents other scenarios.

The scenarios of Figure C.2 are as follows.

1. **Connection Request**: Both the publisher and subscriber request to connect to the broker by sending a CONNECT packet. The broker acknowledges the connection as successful by responding with a CONNACK packet.

2. **Pub/sub with QoS 0**: The subscriber sends a SUBSCRIBE packet, specifying the topic of interest (a) and QoS level (0). The broker acknowledges the subscription as successful by sending a SUBACK packet. Afterwards, the publisher sends a message about that particular topic to the broker, which then redirects the message to the
Figure C.2: Pub/Sub Messaging Sequence on MQTT
interested subscriber. Messages set to QoS level 0 are delivered to the subscriber \emph{at most once}. The publisher does not receive an acknowledgment from the broker.

3. **Pub/Sub with QoS 1**: This interaction is based on messages set to QoS level 1. In such a case, messages are delivered to the subscriber \emph{at least once} until receiving an acknowledgment from the broker (PUBACK).

4. **Pub/Sub with QoS 2**: If subscriptions are set to QoS level 2, then messages are delivered to subscribers \emph{exactly once} using a four-way handshake.

The scenarios of Figure C.3 are as follows.

5. **Connection Request with AUTH (MQTT 5.0)**: MQTT 5.0 supports enhanced user authentication through the use of AUTH packets, which can be exchanged between MQTT clients and brokers. The AUTH packets contain data based on the authentication method established in the CONNECT packet. Exchanging both CONNECT and AUTH packets adds an extra layer of security because the former only enables a basic authentication using usernames and passwords.

6. **Unsubscribe request**: MQTT clients can cancel topic subscriptions via UNSUBSCRIBE messages. The broker acknowledges unsubscribe requests as successful by responding with an UNSUBACK packet.

7. **PING utility**: Ping messages are used to verify the status of MQTT clients and brokers. Once a PINGREQ packet is sent out, a PINGRESP packet is expected. Undelivered responses could signal network connection issues.

8. **Disconnection Request**: Publishers and subscribers can close the network connection with the broker by delivering DISCONNECT packets.

Similar to HTTP, MQTT supports \emph{pipelined connections}, which means that multiple requests or responses can be sent over a single transmission. Due to their asynchronous nature, MQTT messages can also be sent successively without waiting for an immediate response from the broker or clients. Figure C.4 presents an example of simultaneous and successive messages.
5. Connection Request with AUTH (MQTT 5.0)

CONNECT

AUTH

AUTH

CONNACK

CONNECT

AUTH

AUTH

CONNACK

6. Unsubscribe request

UNSUBSCRIBE(topic=a)

UNSUBACK

7. PING utility

PINGREQ

PINGRESP

PINGREQ

PINGRESP

8. Disconnection Request

DISCONNECT

DISCONNECT

Figure C.3: Message Sequence for AUTH, UNSUBSCRIBE, PING, and DISCONNECT Packets
Figure C.4: Simultaneous and Successive Messages
Appendix D

Input Coverage of Fuzzing Techniques for the MQTT Protocol

*Input coverage* refers to the statement coverage achieved by an MQTT fuzzer after exchanging a specific number of packets with the broker. The input coverage achieved by all MQTT fuzzers was evaluated after exchanging 500, 1000, 2000, 4000, 6000, and 8000 packets. The input coverage of each fuzzer was evaluated because of the following reasons. First, grammar-based fuzzers are more aware of the input structure than their counterparts, making it necessary to analyze the quality of their test cases during the fuzzing campaign. Second, mutation-based approaches usually inject more test cases into target systems than their counterparts. This is largely due to mutation-based approaches using predefined test cases rather than generating new ones from scratch. The significant advantage of mutation-based approaches makes it necessary to measure the statement coverage after exchanging the same number of packets with the broker, thereby evaluating the fuzzers based on the quality, rather than quantity of the test cases. Third, an analysis of the input coverage will also provide a better understanding of how many packets are required by each fuzzer to execute a specific number of statements in the source code. The following sections present the input coverage by each MQTT fuzzer for Mosquitto and Moquette.

D.1 Mosquitto

Figure D.1 presents the input coverage of each MQTT fuzzer.

As expected, fuzz(), learner, and CyberExploit underperform because of their aforementioned shortcomings. fuzz() executes at most 1119 statements, and its coverage performance increases up to 0.37% during the test run. Although learner has higher input coverage than fuzz() and CyberExploit, its coverage performance varies the least, executing from 1793 to 1797 statements on average. The input coverage by learner increases up to 0.16% after exchanging 500 packets with Mosquitto. CyberExploit also
outperforms fuzz(), executing at most 1383 statements. The coverage performance by CyberExploit increases up to 1.39% during testing.

Among all fuzzers, learner increases input coverage the least during testing. This outcome is attributed to two main reasons. First, and as previously explained, learner reuses several packets and values constantly, rather than generating different test cases to execute more functionality and statements in the source code. Second, learner requires time to infer the message syntax and sequence of a network protocol, hindering its input coverage considerably at the beginning of the fuzzing campaign. For example, prior to exchanging 8000 packets with Mosquitto, learner alternates between two values for subscription topics until gaining a better understanding of the message syntax. However, regardless of the stopping criterion, learner struggles to execute more statements because it is currently incapable of inferring key features of network protocols for testing purposes.

As more packets are exchanged with the broker, the more statements are expected to be executed by an MQTT fuzzer. However, as shown in Figure D.1, the number of statements
rarely increases when testing with all three aforementioned fuzzers. After exchanging 500 packets with Mosquitto, learner executes only three additional statements, whereas fuzz() and CyberExploit execute four and nineteen respectively. This lack of increase during testing further confirms that the test cases generated by these fuzzers are of low quality. In contrast, the techniques used by FUME, mqtt_fuzz, and MQTTGRAM prove to be very well aware of MQTT's specifications, generating test cases of higher quality than its counterparts. The coverage performance of these fuzzers follows expected patterns, increasing at a steady pace over time.

On average, mqtt_fuzz executes from 1931 to 1998 statements after exchanging 500 packets with Mosquitto. The input coverage by mqtt_fuzz increases up to 3.46% during the test run, peaking at 2035 statements. mqttgram-c outperforms the peak input coverage achieved by mqtt_fuzz, executing 2044 statements before 6000 packets are exchanged with Mosquitto. mqttgram-cf outperforms both mqttgram-c and mqtt_fuzz, executing 2089 statements before 2000 packets. The input coverage of mqttgram-cf and mqttgram-c increases by up to 10.21% and 9.78% respectively, percentages of which are the highest among all MQTT fuzzers. The performance increase of mqttgram-c and mqttgram-cf serve as an indication of the quality of their test cases. On average, mqttgram-cf is slightly outperformed by FUME, despite the fact that the latter supports all major versions of MQTT, whereas the former supports only 3.1.1. On average, FUME executes from 2024 to 2066 statements after exchanging 500 packets with Mosquitto. However, the peak input coverage achieved by FUME is 2122 statements, outperforming all of its counterparts. FUME is therefore a better option to test MQTT implementations because it executes more statements with few test cases and low bandwidth consumption.

D.2 Moquette

Figure D.2 presents the input coverage of each MQTT fuzzer.

The results follow the same pattern as in Figure D.1, with fuzz() and CyberExploit performing poorly in terms of input coverage. fuzz() and CyberExploit execute at most 545 and 986 statements respectively. mqtt_fuzz outperforms both fuzz() and CyberExploit, executing at most 1531 statements. On average, the input coverage by CyberExploit, mqtt_fuzz, and fuzz() increases up to 3.53%, 1.15%, and 0.18% respectively. These coverage performances further highlight the lack of varied and full-featured test cases by most fuzzers except for both variants of MQTTGRAM. In fact, the input coverage by mqttgram-c and mqttgram-cf increases up to 14.44% and 18.14% respectively after exchanging 500 packets with Moquette. In addition to their considerable performance increase during testing, mqttgram-c and mqttgram-cf achieves the highest input coverage, executing at most 1558 and 1594 statements respectively.

Table D.1 presents the fuzzers’ average coverage performance after exchanging exactly 8000 packets with Moquette. However, the results by Sochor et al. (2020b) in Table D.1 are based on a single 30-minute test run due to their fuzzer being proprietary. Nevertheless, in spite of the small discrepancy, results are fairly similar with those in Table 4.1. Both variants of MQTTGRAM outperform their counterparts when exchanging the same number of packets with Moquette.
APPENDIX D

Figure D.2: Input Coverage (Moquette 0.13)

Table D.1: Input Coverage of Moquette (8000 packets)

<table>
<thead>
<tr>
<th>Source Code (Directory)</th>
<th>Fuzzed packets not based on existing vulnerabilities</th>
<th>Fuzzed packets based on existing vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scapy</td>
<td>Casteur et al. 2020</td>
</tr>
<tr>
<td>broker</td>
<td>21.00%</td>
<td>39.63%</td>
</tr>
<tr>
<td>broker security</td>
<td>12.00%</td>
<td>12.00%</td>
</tr>
<tr>
<td>broker subscriptions</td>
<td>8.00%</td>
<td>24.01%</td>
</tr>
<tr>
<td>persistence</td>
<td>9.00%</td>
<td>9.00%</td>
</tr>
<tr>
<td>interception</td>
<td>11.00%</td>
<td>26.60%</td>
</tr>
<tr>
<td>broker config</td>
<td>51.00%</td>
<td>51.00%</td>
</tr>
<tr>
<td>broker metrics</td>
<td>44.00%</td>
<td>68.00%</td>
</tr>
<tr>
<td>Logging</td>
<td>43.00%</td>
<td>43.00%</td>
</tr>
<tr>
<td>Total</td>
<td>19.23%</td>
<td>34.02%</td>
</tr>
</tbody>
</table>
Appendix E

Packet Exchange of Fuzzing Techniques for the MQTT Protocol

Figure E.1 presents the number of packets exchanged between the MQTT fuzzers and Mosquitto 1.6.8 in 30 minutes.

For simplicity purposes, results of MQTTGRAM shown in Figure E.1 are only from mqttgram-c because its performance is nearly identical to mqttgram-cf, the latter of which exchanges the lowest number of packets with the broker.
As shown in Figure E.1, FUME exchanges at least 2x more packets than its counterparts because of the following reasons. First, FUME performs multiple requests in a single packet due to MQTT supporting asynchronous communications. For each session, FUME sends most packets over a single TCP connection. All packets except for CONNECT are sent in random order to the broker. Connection requests are performed first to avoid immediate rejection by the broker. After sending multiple requests in a single packet, FUME receives a message from the broker acknowledging the connection as successful. FUME then reconnects to the broker intentionally to repeat the process until a stopping criterion is satisfied. Thus, the test run mostly consists of short-lived interactions with the broker.

mqtt_fuzz exchanges up to 9x more packets with Mosquitto than most fuzzers, except for FUME, because it slightly modifies predefined test cases rather than generating new ones from scratch. The latter approach is performed by the remaining four fuzzers shown in Figure E.1, hence their low network traffic during testing. learner exchanges more packets than the last three fuzzers because it generates smaller values for subscription topics in the test cases. CyberExploit generates subscription topics of considerable, but acceptable length. In contrast to learner and CyberExploit, fuzz() generates values of arbitrary length for several fields, including subscription topics. MQTTGRAM exchanges the fewest packets with the broker because of its careful generation-based approach.

In order to gain a better understanding of how quickly these fuzzers generate network packets, Figure E.2 presents the average time elapsed until exchanging 8000 packets with Mosquitto.

Developing fast packet generators brings several benefits to the testing phase, most notably frequent communications between the fuzzer and the broker. As a result, there is a higher probability of uncovering more bugs or reaching deeper states of the protocol within a short period of time. These benefits are overshadowed, however, if packets sent in rapid succession to the broker are of low quality. Thus, the effectiveness of MQTT fuzzers cannot be solely analyzed nor determined by the amount of time spent generating the packets.
necessary network packets (Figure E.2). Standardized metrics such as statement, input, and feature coverage must be analyzed alongside with Figure E.2 to determine both the effectiveness and efficiency of MQTT fuzzers. As expected, all variants of MQTTGRAM are the most time consuming because their aim is to carefully and correctly craft the network packets. More specifically, mqttgram-cf and mqttgram-c exchange 8000 packets in 642 and 630 seconds respectively. mqttgram-cf generates more topic-based packets than mqttgram-c, the task of which is time consuming due to the complexity and length of the message fields. fuzz() took roughly 8 minutes, whereas CyberExploit took 6 minutes. CyberExploit takes less time because it generates all the necessary network packets prior to testing an MQTT implementation. In contrast, fuzz() generates network packets during the fuzzing campaign, thereby having a longer response delay when it interacts with the broker. Furthermore, fuzz() assigns longer values than CyberExploit, resulting in larger packets that are more time consuming to generate. learner took slightly less than 4 minutes because of its mapper component, which rapidly translates abstract data into concrete data and vice versa for packet generation. The authors cited the complexity of generating MQTT packets as a motivation to develop the mapper component. Since the process of translating abstract data into concrete data is less complex than crafting entire packets from scratch, learner takes less time to exchange 8000 packets with the broker than its generation-based counterparts. mqtt_fuzz and FUME take the least amount of time, exchanging 8000 packets in approximately 1 and 2 minutes respectively. mqtt_fuzz is a much simpler (F-Secure Corporation, 2015) fuzzer than FUME, using only mutation-based approaches to craft the test cases. FUME, on the other hand, uses both mutation- and generation-based approaches, adding an extra layer of complexity to its testing capabilities. However, FUME takes less time to exchange 8000 packets than mqtt_fuzz because of its triage algorithm, which, according to the official repository description, uses simple alterations to craft the test cases. mqtt_fuzz, on the other hand, uses Radamsa, which applies various types of heuristics to modify the inputs, and can therefore take a longer time to generate the test cases.

Figure E.3 presents the number of packets exchanged with Moquette 0.13 in 30 minutes. Results are mostly similar to those achieved with Mosquitto. Unsurprisingly, mqtt_fuzz exchanges at least 7x, 6x, and 4x more packets than MQTTGRAM, fuzz(), and CyberExploit respectively. However, as explained in the previous sections, both variants of MQTTGRAM manage to outperform its counterparts in terms of statement coverage, despite exchanging the fewest packets with Moquette.
Figure E.3: Packets Exchanged with Moquette 0.13 in 30 Minutes
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


