# Vulnerability Analysis of IoT Platforms: A Case Study on Thinger.io

Jose M. Molina
Computer Science and Engineering Department/Applied Artificial Intelligence Group
Universidad Carlos III de Madrid
Colmenarejo, Madrid, Spain
molina@ia.uc3m.es

Alvaro Bustamante
Thinger.io Engineering Team
Las Rozas, Madrid, Spain
alvarolb@thinger.io

Miguel Angel Patricio
Computer Science and Engineering Department/Applied Artificial Intelligence Group
Universidad Carlos III de Madrid
Colmenarejo, Madrid, Spain
mpatrici@inf.uc3m.es

## Abstract

The rapid growth of the Internet of Things (IoT) has introduced significant security challenges due to the variety of communication protocols and the often-limited security measures in many devices. This paper presents a comprehensive set of techniques for evaluating the security of IoT protocols, including guided testing, fuzzing, symbolic execution, formal verification, and penetration testing. As a practical demonstration, guided testing and fuzzing are applied to the open-source Thinger.io platform to assess its communication protocols, uncovering critical vulnerabilities in WebSockets, MQTT, and proprietary protocols. The results illustrate the effectiveness of these methods in identifying security flaws and offer recommendations to improve the resilience of IoT systems. This work contributes to ongoing efforts to secure IoT ecosystems and cyber-physical systems by providing actionable insights and a methodology that can be applied across different platforms.

## Keywords

IoT, Vulnerability Analysis, Thinger.io, Security, Fuzz Testing, Protocols

**ACM Reference Format:**
Jose M. Molina, Alvaro Bustamante, and Miguel Angel Patricio. 2024. Vulnerability Analysis of IoT Platforms: A Case Study on Thinger.io. In *14th International Conference on the Internet of Things (IoT 2024), November 19–22, 2024, Oulu, Finland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3703790.3703821

## 1 Introduction

The rapid expansion of the Internet of Things (IoT) has led to an unprecedented number of devices connected to the Internet, communicating through a variety of protocols. According to estimates, the number of IoT devices worldwide is expected to exceed 75 billion by 2025, reflecting significant growth [23]. This massive interconnectedness offers advantages in automation, efficiency, and data-driven decision-making in sectors such as healthcare, manufacturing, smart cities, and cyber-physical systems.

This work is licensed under a Creative Commons Attribution International 4.0 License.

*IoT 2024, November 19–22, 2024, Oulu, Finland*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1285-2/24/11
https://doi.org/10.1145/3703790.3703821

However, this growth introduces significant security vulnerabilities. Securing IoT protocols is crucial because each exposed port and protocol, such as MQTT, HTTP, WebSockets, IOTMP, and CoAP—serves as a potential attack vector [13]. These protocols often prioritize performance over security, leading to inherent weaknesses [11]. In cyber-physical systems, vulnerabilities can have serious real-world consequences.

Manufacturers may prioritize functionality and cost over security features, resulting in devices lacking robust authentication and encryption mechanisms [26]. This openness can be exploited by attackers sending malformed input to disrupt systems [2], potentially causing denial of service (DoS) or severe exploits such as buffer overflows or unauthorized code execution [14]. Malformed inputs may result from erroneous implementations or deliberate attacks. For example, inadequate input validation can allow attackers to inject malicious payloads that the system cannot process safely [18].

These vulnerabilities are not merely theoretical. The Amnesia:33 vulnerability [12], discovered in 2020, demonstrated how insecure IoT devices can be exploited via protocol-level weaknesses. This set of vulnerabilities, which affect TCP/IP stacks in millions of IoT devices, included buffer overflows and memory corruption triggered by malformed packets, leading to remote code execution (RCE) or denial-of-service (DoS) attacks. Attacks exploiting these types of vulnerability highlight the dangers of improper input validation and weak protocol implementations. Additionally, attackers can exploit known protocol flaws without sophisticated tools simply by sending specially crafted messages designed to compromise device security.

The diversity of IoT devices and the lack of standardized security practices exacerbate the problem [7]. Many devices operate with limited computational resources, making it difficult to implement traditional security measures such as complex encryption algorithms [24]. Consequently, IoT ecosystems, including cyber-physical systems, often contain devices with varying security capabilities, creating an uneven security landscape [22].

This paper aims to explore multiple testing methodologies for identifying and mitigating protocol vulnerabilities, with a case study on Thinger.io, a cloud-based open-source platform for managing IoT devices. By applying fuzzing techniques, as well as guided tests based on protocol knowledge, this study seeks to uncover potential security flaws and provide recommendations to improve the security of IoT platforms. The goal is to contribute to the development of more secure IoT ecosystems and cyber-physical systems

by demonstrating effective vulnerability analysis methods that can be applied to other platforms and devices.

## 2 Protocol Vulnerability Testing Methods

The security of IoT platforms is highly dependent on the robustness of the communication protocols they employ. Protocols like MQTT, HTTP, WebSockets, CoAP, or IOTMP are integral to the functionality of IoT systems, but can also be entry points for attackers if not properly secured. Testing these protocols for vulnerabilities is crucial to ensure the resilience of IoT platforms against malicious activities.

Several methodologies are used to uncover potential weaknesses in protocol implementations. This section discusses key methods, including guided testing, fuzz testing, symbolic execution, formal verification, static and dynamic analysis, and penetration testing.

### 2.1 Guided Testing

Guided testing involves a systematic analysis of protocols based on in-depth knowledge of their specifications and expected behaviors [6]. Testers use their understanding of protocol structures, state machines, and typical use cases to design test cases that target specific components or features prone to vulnerability [25]. For instance, they may examine packet structures for incorrect parsing or handling, test state transitions to identify improper session management, or manipulate protocol flags and fields to uncover weaknesses in error handling.

This method is particularly effective in identifying vulnerabilities that may not be apparent through random testing, such as logical errors or improper implementation of protocol standards.

Several tools have been developed to facilitate guided testing of IoT protocols by leveraging detailed knowledge of protocol specifications. These tools enable testers to systematically analyze how devices handle expected and unexpected inputs, aiding in the identification of vulnerabilities related to improper input handling, authentication flaws, or deviations from protocol standards.

Protocol analyzers like **Wireshark** capture and analyze network traffic across various protocols, providing insights into packet structures and data flows. Web application testing suites such as **Burp Suite** and **OWASP ZAP** facilitate the interception and modification of HTTP/HTTPS requests and responses, helping to examine how IoT devices handle web-based interactions. For WebSocket testing, the **Autobahn Test Suite** performs automated compliance and robustness checks, systematically identifying deviations from the protocol specification.

In MQTT contexts, clients like **MQTT.fx** and **MQTT Explorer** allow testers to publish and subscribe with customized payloads, assessing the robustness of implementations. CoAP testing tools such as **Copper (Cu)** provide interfaces for sending custom CoAP requests, facilitating the examination of protocol handling. For proprietary or less common protocols like IOTMP, testers may use platform-specific SDKs or develop custom scripts based on protocol specifications.

### 2.2 Fuzz Testing

Fuzz testing, or fuzzing, is a dynamic testing technique that involves sending malformed or semi-random data input to a system to observe how it responds. The goal is to trigger unexpected behavior, crashes, or security vulnerabilities by exploring a wide range of input scenarios [17].

There are two main types of fuzz testing: general fuzz testing and protocol-aware fuzz testing. General fuzz testing involves sending completely random inputs to the system without any knowledge of the protocol. Although this approach can cover a broad spectrum of inputs, it is often inefficient for protocol testing because most random inputs are rejected by the protocol parser, failing to reach deeper levels of the application logic [9].

Protocol-aware fuzz testing enhances the efficiency of fuzzing by incorporating knowledge of the protocol's structure and syntax. By starting with a corpus of valid protocol messages, the fuzzer applies mutations to these messages to create new test cases that are syntactically correct but may contain unexpected values or sequences. This approach allows the fuzzer to generate inputs that are more likely to be processed by the system, to reach deeper execution paths within the application, and to identify vulnerabilities related to stateful interactions and protocol-specific logic. By focusing on valid protocol structures, protocol-aware fuzzing can more effectively discover critical vulnerabilities in protocol implementations without wasting resources on inputs that are immediately discarded by the system [21].

For protocol-specific fuzzing, tools like **MQTT-FUZZER** for MQTT, **Wfuzz** and the **Intruder** component of **Burp Suite** for HTTP, and **WSFuzzer** for WebSockets are commonly used. **MQTT-FUZZER** manipulates MQTT control packets to test the robustness of brokers and clients. **Wfuzz** and **Burp Suite's Intruder** enable the crafting and automation of custom HTTP requests to identify injection vulnerabilities and improper input handling. **WSFuzzer** sends malformed messages to WebSocket endpoints to identify vulnerabilities related to input handling and protocol compliance.

For the CoAP protocol, fuzzing tools such as the **Defensics CoAP Test Suite** and **Radamsa** are utilized. The **Defensics CoAP Test Suite** generates malformed messages to test how devices handle unexpected input. **Radamsa**, a general-purpose fuzzer, creates a variety of malformed inputs for any protocol by intercepting messages.

General-purpose fuzzers like **American Fuzzy Lop (AFL)**, **AFLNet**, and **Peach Fuzzer** are also used for various IoT protocols. **AFL** uses genetic algorithms to explore input spaces and can be adapted to network protocols. **AFLNet** extends AFL with state-aware gray-box fuzzing designed for network protocol implementations, helping to discover vulnerabilities in packet parsing and session handling. **Peach Fuzzer** supports protocol fuzzing through data and state models, allowing configuration for different protocols by defining their structures.

### 2.3 Symbolic Execution

Symbolic execution is a technique in which program execution is simulated with symbolic input instead of concrete values. It systematically explores program paths by representing input values

as symbols and solving constraints to find inputs that can trigger specific paths or conditions [5].

In the context of protocol testing, symbolic execution can be used to identify input values that cause buffer overflows or integer overflows [15]. It is also useful to explore edge cases in protocol handling logic and detect vulnerabilities that require specific input patterns. Although powerful, symbolic execution can be resource-intensive and may not scale well for large applications.

Several tools have been developed to facilitate symbolic execution in vulnerability analysis. Notably, **KLEE** is an open-source symbolic execution engine built on the LLVM compiler infrastructure. It is designed to generate high-coverage tests for complex programs written in C and C++, making it useful for analyzing protocol implementations in IoT devices. **S2E (Selective Symbolic Execution)** extends symbolic execution to entire software stacks, including operating systems and firmware, making it suitable for analyzing complex IoT systems where interactions between different software layers may introduce vulnerabilities. Additionally, **Angr** is a Python-based binary analysis framework that incorporates symbolic execution, among other techniques. It allows for the analysis of binaries and firmware images, which is particularly useful when source code is unavailable, a common scenario in proprietary IoT devices.

## 2.4 Formal Verification

Formal verification involves mathematically proving that a system adheres to its specifications. For protocols, this means verifying that the implementation conforms to the protocol standard and that there are no logical errors or vulnerabilities.

In formal verification, techniques such as model checking and theorem proving are often employed. Model checking systematically explores all possible states of a system to verify certain properties [10], while theorem proving uses logical proofs to demonstrate that specific conditions always hold.

Although formal verification provides strong guarantees about the correctness and security of protocol implementations, it can be complex to apply in practice, especially for large or highly dynamic systems, as they often require significant expertise in formal methods and mathematical reasoning.

Several tools have been developed to help in the formal verification of protocols. Model checking tools like **SPIN** and **UPPAAL** are widely used to verify finite-state concurrent systems, including communication protocols. **SPIN** utilizes the Promela modeling language to represent protocols and can verify properties such as deadlocks, livelocks, and assertion violations. **UPPAAL** is designed for real-time systems and is capable of verifying the timing properties and the reachability of the state. Interactive theorem provers such as **Coq**, **Isabelle/HOL**, and **PVS** enable the formalization and proof of protocol properties, requiring protocols and their properties to be specified in a formal language, with proofs often constructed interactively.

**TLA+** is a formal specification language developed for modeling and verifying concurrent and distributed systems, allowing protocols to be specified at a high level of abstraction. It includes tools like the **TLA+ Model Checker** (TLC) to verify that a protocol satisfies certain invariants and temporal properties.

## 2.5 Static and Dynamic Analysis

Although not exclusively focused on protocols, static and dynamic analysis tools can help identify vulnerabilities in protocol implementations. Static analysis tools examine source code without executing it, identifying potential issues such as buffer overflows, improper input validation, and insecure coding practices [16].

Dynamic analysis involves running the application and monitoring its behavior to detect anomalies. Techniques include monitoring memory usage for leaks or corruption, using instrumentation to detect improper use of APIs, and observing network communications for unexpected patterns. Dynamic analysis can uncover vulnerabilities that manifest during run-time, which may not be evident through static analysis.

Several tools are commonly used for static and dynamic analysis in protocol testing to identify vulnerabilities in implementations. For static analysis, **SonarQube** and **Coverity Scan** are widely adopted platforms that inspect code quality and detect potential security flaws, such as buffer overflows and improper input validation, in multiple programming languages. Dynamic analysis tools like **Valgrind** and **AddressSanitizer (ASan)** monitor applications at runtime to detect anomalies not apparent by static code examination. **Valgrind** includes tools for debugging and profiling, effectively identifying memory management issues such as leaks and corruption. **ASan**, part of the LLVM and GCC compiler suites, instruments code to catch memory errors such as buffer overflows and use-after-free bugs during execution.

For observing system interactions, utilities like **strace** and **ltrace** trace system calls and library calls made by a program, respectively, providing insights into how applications interact with the operating system and external libraries. This is beneficial for detecting unexpected behavior or security issues arising from improper resource usage within protocol implementations.

## 2.6 Penetration Testing

Penetration testing simulates real-world attacks to evaluate the security of protocol implementations within a system [20]. Ethical hackers attempt to exploit known and unknown vulnerabilities, providing information on possible attack vectors and the effectiveness of existing security measures [3].

This form of testing involves exploiting weaknesses in authentication or encryption mechanisms, performing man-in-the-middle attacks to intercept or modify communications, and leveraging protocol-specific vulnerabilities to gain unauthorized access. By mimicking the strategies used by malicious actors, penetration testing helps identify and remediate security gaps, thereby strengthening the overall security posture of IoT platforms.

Several tools are commonly used in the penetration testing of IoT protocols. The **Metasploit Framework** is an open-source penetration testing platform that provides information on security vulnerabilities and aids in penetration testing and the development of IDS signatures. **Nmap (Network Mapper)** is a powerful network scanning tool that is used to discover hosts and services on a computer network by sending packets and analyzing the responses. **Wireshark** is a network protocol analyzer that allows the capture and interactive browsing of network traffic, which is

invaluable for analyzing protocol implementations, detecting anomalies and performing passive reconnaissance during penetration testing. Additionally, **Burp Suite Professional** is an advanced web vulnerability scanner and penetration testing toolkit that can be used to test HTTP and WebSocket communications on IoT devices that offer web interfaces or APIs.

For IoT-specific penetration testing, tools like **MQTT-PWN**, **Firmalyzer**, and the **Attify IoT Security Framework** are employed. **MQTT-PWN** is a penetration testing toolkit specifically designed for the MQTT protocol, which allows testers to identify vulnerabilities in MQTT brokers by manipulating protocol fields, testing authentication mechanisms, and attempting unauthorized access. **Firmalyzer** is a cloud-based automated firmware security analysis platform that helps to identify vulnerabilities in the firmware of IoT devices, detecting issues such as hardcoded credentials, outdated libraries and insecure configurations. The **Attify IoT Security Framework** is a collection of tools for IoT penetration testing, including hardware interface, firmware analysis, and network testing components. Additionally, penetration testers may use platforms like **Kali Linux**, a Debian-derived Linux distribution designed for digital forensics and penetration testing, which comes pre-installed with numerous security tools suitable for testing IoT devices and protocols.

## 3 Case Study: Thinger.io Platform

Thinger.io is an open-source platform designed to simplify the integration and management of devices and applications from the Internet of Things (IoT) [8, 19]. It offers both cloud-based and on-premise solutions, providing flexibility for a wide range of deployment scenarios. This versatility has led to its widespread adoption in both academic research and industrial applications [1], serving as a foundation for the prototyping of IoT systems, conducting experiments, and deploying commercial IoT solutions [4].

Renowned for its protocol interoperability and flexibility, Thinger.io supports multiple communication protocols, including:

- **HTTP/HTTPS**: For RESTful API communications and web-based interactions.
- **WebSockets**: Enabling real-time, bidirectional communication between clients and servers.
- **MQTT**: A lightweight messaging protocol ideal for devices with limited bandwidth or processing capabilities.
- **CoAP**: Designed for constrained devices and networks, providing efficient communication in IoT environments.
- **IOTMP (IoT Message Protocol)**: A proprietary protocol developed by Thinger.io for optimized device management and data exchange.

This multi-protocol support allows Thinger.io to interface with a diverse array of devices, from simple sensors to complex industrial machines, accommodating various use cases and requirements. Figure 1 illustrates the general Thinger.io architecture, which supports multiple protocols and includes features that facilitate the development and deployment of IoT.

Despite its robust capabilities, security remains a critical concern for Thinger.io, as with any IoT platform. The platform supports secure communications through encryption protocols like TLS/SSL for HTTPS and MQTT over TLS. Authentication mechanisms such as API keys and token-based authentication are implemented to control access to devices and data. However, the diversity of supported protocols and the complexity of IoT ecosystems can introduce potential security vulnerabilities that require continuous assessment and mitigation.

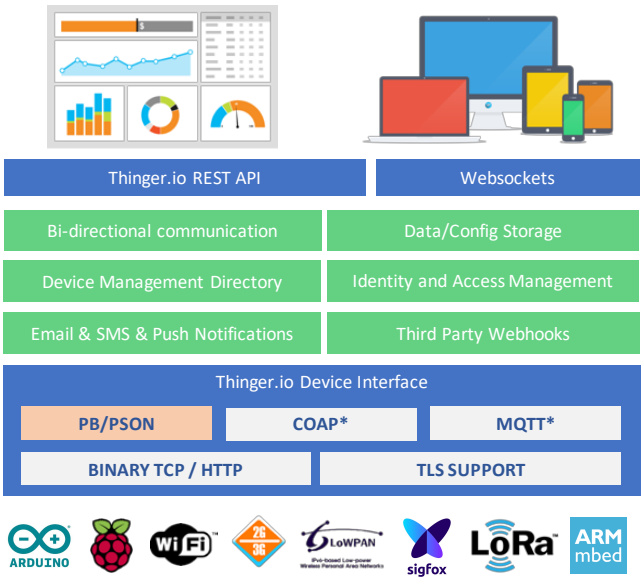


**Figure 1: Overview of Thinger.io platform.**

To demonstrate the practical application of the vulnerability techniques described in the protocol, a vulnerability analysis was performed on the Thinger.io platform. The platform was tested using a combination of guided testing, traditional fuzzing, and protocol-aware fuzzing, focusing on its most exposed communication protocols: WebSockets, MQTT, IOTMP, and HTTP.

These testing methods were selected based on their effectiveness in identifying vulnerabilities in various types of protocol in the Thinger.io platform. For widely understood protocols such as WebSockets, guided testing was employed due to the availability of detailed specifications, enabling targeted assessments. Protocol-aware fuzzing was applied to HTTP and MQTT, leveraging prior knowledge and corpus data to create test cases that explored a range of input scenarios and reached deeper execution paths. For lesser-known protocols such as IOTMP, general-purpose fuzzers were used to identify vulnerabilities through random input generation without relying on protocol-specific knowledge.

Other techniques such as symbolic execution, formal verification, static and dynamic analysis, and penetration testing were excluded due to resource limitations and scope constraints.

## 4 Protocol experimentation

This section presents the application of guided testing and fuzzing techniques on Thinger.io's key communication protocols, including WebSockets, MQTT, IOTMP, and HTTP. Each protocol was tested using relevant tools to uncover vulnerabilities, assess input validation, and detect potential security risks such as denial-of-service attacks or unauthorized access.

## 4.1 WebSocket Testing

The WebSocket protocol was evaluated using the **Autobahn Test Suite**, a widely recognized tool to validate WebSocket implementations. This suite performs more than 500 test cases, covering aspects such as protocol framing, data fragmentation, and UTF-8 handling. Figure 2 shows the results of applying this suite to Thinger.io. The tests revealed several areas for improvement, including the handling of fragmented packets and malformed UTF-8 sequences, which could potentially lead to denial-of-service attacks under certain conditions.

Further analysis involved using protocol-aware fuzzers to generate malformed WebSocket frames using **OWASP ZAP**. These tests identified that Thinger.io's WebSocket implementation properly closes connections when receiving invalid frames, mitigating potential security risks. However, continuous monitoring and updates are recommended to address any newly discovered vulnerabilities in the WebSocket protocol.

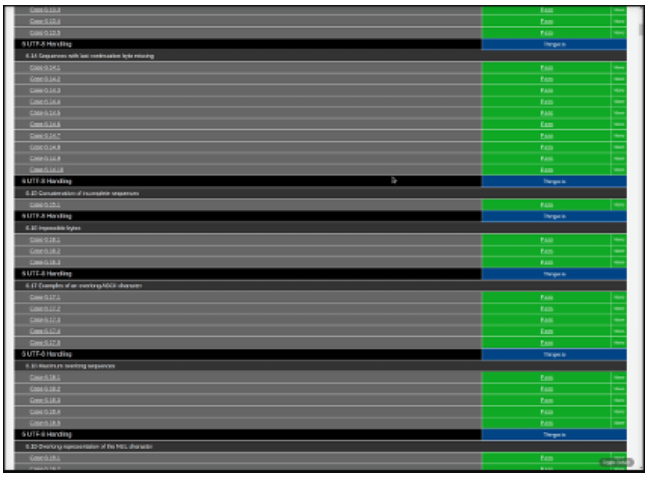

**Figure 2: Screen with results of passing Autobahn on Thinger.io.**

## 4.2 MQTT Protocol Testing

The MQTT protocol, another key component of Thinger.io, was tested using **mqtt_fuzz**, a fuzzer designed specifically for MQTT brokers. This tool employs **Radamsa**, which generates test cases by mutating valid MQTT packets to identify potential vulnerabilities. As shown in Figure 3, the fuzzer was able to elicit different network responses as it generated new inputs. Although the screenshot does not display errors in the protocol, since they were fixed following the tests, the fuzzer initially identified several edge cases where the system did not handle malformed packets properly, leading to possible denial-of-service conditions.

Additionally, static code analysis tools were used to examine the implementation of the MQTT broker within Thinger.io. The analysis highlighted areas where input validation could be strengthened, particularly in the parsing of MQTT control packets. Recommendations include implementing stricter validation of packet lengths and types, as well as improving error handling mechanisms to prevent potential crashes.

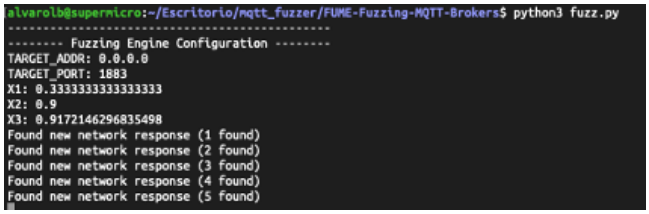


**Figure 3: Results of mqtt_fuzz testing on Thinger.io.**

## 4.3 IOTMP Protocol Testing

The IOTMP (IoT Message Protocol), used by Thinger.io, was tested using the **American Fuzzy Lop (AFL)** fuzzer. **AFL** is a powerful general-purpose fuzzing tool that uses code coverage analysis to guide the fuzzing process. Since there are no specific fuzzing tools tailored for IOTMP, AFL was chosen for its ability to explore a wide range of execution paths, increasing the likelihood of discovering critical vulnerabilities.

As shown in Figure 4, AFL exercised different execution branches during the test. Although the figure does not display any crashes, since the identified issues were fixed after testing, the fuzzer was able to find corner cases related to data type overflows. These overflows required the allocation of large amounts of memory, causing the process to crash. The use of AFL was invaluable in uncovering these vulnerabilities.

Addressing these issues involved improving input validation and implementing robust exception handling throughout the IOTMP implementation. Thorough internal testing using tools like AFL is crucial, especially for protocols such as IOTMP that may not be subjected to widespread scrutiny from the security community.


**Figure 4: Results of AFL testing on IOTMP.**

## 4.4 HTTP Protocol Testing

Finally, the HTTP protocol was tested using **libFuzzer**, another tool that focuses on code coverage-guided fuzzing. LibFuzzer generated a variety of malformed HTTP requests to test the robustness of the Thinger.io platform's HTTP handling mechanisms. Figure 5 shows an example of how libFuzzer works on the module that processes HTTP requests. In this case, the idea is to leave the fuzzer running

indefinitely until some incident occurs, such as a timeout, running out of memory, or a crash in the application. The results indicated that Thinger.io's HTTP implementation is robust against malformed requests, with appropriate error responses and no observed crashes.

The testing confirmed that Thinger.io's HTTP implementation is resilient, which is essential given the ubiquity of HTTP-based attacks. Continuous testing and adherence to secure coding practices are recommended to maintain this level of security.

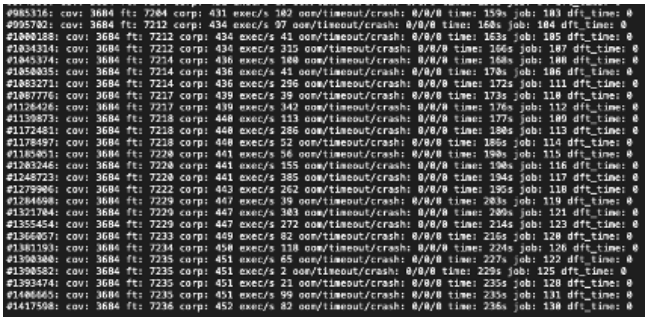


**Figure 5: Results of executing libFuzzer on Thinger.io.**

## 5 Conclusion

This study demonstrates the effectiveness of combining guided testing and fuzzing techniques to uncover vulnerabilities in IoT platforms, with Thinger.io serving as a practical example. Although Thinger.io proves robust and scalable, critical weaknesses were discovered in protocols such as WebSockets and MQTT, particularly in handling malformed packets. Tools like **AFL** and **mqtt_fuzz** successfully identified issues that could lead to denial-of-service attacks or unauthorized code execution.

To improve platform security, more stringent input validation, robust error handling mechanisms, and continuous security assessments are essential. In addition, integrating automated testing pipelines can ensure that vulnerabilities are detected early in the development process. Future research should explore the incorporation of AI-driven anomaly detection and adaptive security measures, allowing IoT platforms to proactively mitigate evolving threats in real-time.

## Acknowledgments

This study was funded by the Spanish company Thinger.io; the public research projects of the Spanish Ministry of Science and Innovation PID2020-118249RB-C22 and PDC2021-121567-C22 - AEI/10.13039/501100011033 and the project under the call PEICTI 2021-2023 with the identifier TED2021-131520B-C22.

## References

[1] Lawrence Oriaghe Aghenta and Mohammad Tariq Iqbal. 2019. Low-cost, open source IoT-based SCADA system design using thinger. IO and ESP32 thing. *Electronics* 8, 8 (2019), 822.

[2] MA Al Naeem, A Abubakar, and MMH Rahman. 2020. Dealing with well-formed and malformed packets, associated with point of failure that cause network security breach. *IEEE Access* (2020), 9241700.

[3] Özlem Aslan, Serkan S. Aktuğ, Merve Ozkan-Okay, Ahmet A. Yilmaz, and Erkan Akin. 2023. A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. *Electronics* 12, 6 (2023), 1333.

[4] Win Sandar Aung and S Aung Nyein Oo. 2019. Monitoring and controlling device for smart greenhouse by using Thinger. IO IoT server. *International Journal of Trend in Scientific Research and Development* (2019).

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39. https://doi.org/10.1145/3182657

[6] A Blasi, A Goffi, K Kuznetsov, and A Gorla. 2018. Translating code comments to procedure specifications. *Proceedings of the 2018 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018). https://doi.org/10.1145/3213846.3213872

[7] I Brass, L Tanczer, M Carr, M Elsden, and J Blackstock. 2018. Standardising a moving target: The development and evolution of IoT security standards. In *IET Conference Proceedings*. IET. https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0024

[8] Alvaro Luis Bustamante, Miguel A Patricio, Antonio Berlanga, and José M Molina. 2023. Seamless transition from machine learning on the cloud to industrial edge devices with thinger. io. *IEEE Internet of Things Journal* 10, 18 (2023), 16548–16563.

[9] Yuchen Chen, Tian Lan, and Guru Venkataramani. 2019. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Moving Target Defense*. ACM, 61–72.

[10] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of model checking*. Springer. https://link.springer.com/content/pdf/10.1007/978-3-319-10575-8.pdf

[11] Abebe Diro, Henok Reda, Naveen Chilamkurti, and Amjad Mahmood. 2020. Lightweight authenticated-encryption scheme for internet of things based on publish-subscribe communication. *IEEE* (2020). https://ieeexplore.ieee.org/abstract/document/9045934/

[12] Daniel dos Santos, Stanislav Dashevskyi, Jos Wetzels, and Amine Amri. 2021. Amnesia: 33 How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices. *Forescout Research Labs* (2021).

[13] AJ Hintaw, S Manickam, and MF Aboalmaaly. 2023. MQTT vulnerabilities, attack vectors and solutions in the internet of things (IoT). *IETE Journal of Research* (2023). https://www.tandfonline.com/doi/abs/10.1080/03772063.2021.1912651 Cited by 58.

[14] SD Hitefield, M Fowler, et al. 2018. Exploiting buffer overflow vulnerabilities in software defined radios. *2018 IEEE International ...* (2018).

[15] Zhen Huang and Xiaohong Yu. 2021. Integer overflow detection with delayed runtime test. In *Proceedings of the 16th International Conference on ....* ACM. https://dl.acm.org/doi/abs/10.1145/3465481.3465771

[16] Amandeep Kaur and Anand Nayyar. 2020. A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code. *Procedia Computer Science* 167 (2020), 2020. https://www.sciencedirect.com/science/article/pii/S1877050920312023

[17] George Klees, Anthony Ruef, Ben Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), 2123–2140. https://doi.org/10.1145/3243734.3243804

[18] Sujitha Lakshminarayana, Amit Praseed, and P Santhi Thilagam. 2024. Securing the IoT Application Layer from an MQTT Protocol Perspective: Challenges and Research Prospects. *IEEE Communications Surveys & Tutorials* (2024).

[19] A Luis Bustamante, MA Patricio, and JM Molina. 2019. Thinger. io: An open source platform for deploying data fusion applications in IoT environments. *Sensors* 19, 5 (2019), 1044.

[20] Gilberto Najera-Gutierrez and Juned Ahmed Ansari. 2018. *Web Penetration Testing with Kali Linux: Explore the methods and tools of ethical hacking with Kali Linux*. Packt Publishing.

[21] Van-Thuan Pham, Marcel Böhme, et al. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 449–454.

[22] E Schiller, A Aidoo, J Fuhrer, J Stahl, and M Ziörjen. 2022. Landscape of IoT security. *Computer Science* (2022). https://www.sciencedirect.com/science/article/pii/S1574013722000120

[23] Khurram Shafique, Bilal A Khawaja, Fozia Sabir, Shoab A Qazi, and Muhammad Mustaqim. 2020. Internet of things (IoT) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5G-IoT scenarios. *IEEE Access* 8 (2020), 23022–23040.

[24] S Singh, PK Sharma, SY Moon, and JH Park. 2024. Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions. *Journal of Ambient Intelligence and Humanized Computing* 15, 1 (2024), 123–145. https://doi.org/10.1007/s12652-017-0494-4

[25] MM Yamin, B Katt, and V Gkioulos. 2020. Cyber ranges and security testbeds: Scenarios, functions, tools and architecture. *Computers & Security* (2020). https://www.sciencedirect.com/science/article/pii/S0167404819301804 Cited by 273.

[26] Wei Zhou, Yiyang Jia, Anni Peng, Yong Zhang, and Peng Liu. 2018. The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. *IEEE Internet of Things Journal* 6, 2 (2018), 1606–1616.