POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

OCPPStorm: A Comprehensive Fuzzing Tool for OCPP Implementations

Supervisors:
Prof. Cataldo Basile
Prof. V.N. Venkatakrishnan
Prof. Rigel Gjomemo

Candidate:
Gaetano Coppoletta

Academic Year 2022/2023
Torino

Abstract

In the context of escalating electric vehicle (EV) adoption, the development of a robust charging infrastructure emerges as a critical enabler. At the heart of this infrastructure lies the Open Charge Point Protocol (OCPP), serving as a standardized communication interface between electric vehicle charging stations and central systems. Historically, the security scrutiny of OCPP has been executed in a sporadic and unsystematic fashion. Addressing this lacuna, this thesis presents OCPPStorm, a black-box fuzzer designed to navigate the security landscape of OCPP with precision and agility. OCPPStorm is architected to be indifferent to the programming language, thus making it inherently versatile for applications across various OCPP implementations. OCPPStorm implements a suite of fuzzing mechanisms, distinguished by their velocity and efficiency, to detect and delineate bugs and security vulnerabilities within OCPP systems. OCPPStorm is evaluated through rigorous testing across two different open source OCPP implementations, highlighting the tool's capability to transcend language and structural boundaries. The primary objective of this research is to elevate the methodological rigor in security testing of OCPP implementations, advancing the frontier of protocol security in the EV charging domain. OCPPStorm, with its rapid and comprehensive fuzzing techniques, serves as a vanguard in this endeavor, providing a scalable and effective framework for enhancing the security OCPP implementations. OCPPStorm utilizes information extracted from the official OCPP documentation and evaluates OCPP implementations sourced from public repositories on GitHub, ensuring the research did not involve human subjects.

Acknowledgements

I express my heartfelt gratitude to my thesis committee for their invaluable guidance and support throughout my research. Your insights have been crucial to my academic and personal growth.

A special thanks to my family for their unwavering love and encouragement, which have been fundamental to my success.

I am also grateful to everyone I've met along this journey. Your varied influences have been a source of inspiration and learning, enriching my experience in countless ways.

In sum, this journey has been more than an academic pursuit; it's been a period of personal growth and meaningful relationships. To all who have been part of this journey, thank you for your indispensable contributions to this thesis.

Table of Contents

List of Figures

Acronyms

OCPP	Open Charge Point Protocol
UIC	University of Illinois at Chicago
TLS	Transport Layer Security
EV	Electric Vehicle

Chapter 1

Problem Description

1.1 Background

Electric Vehicles (EVs) represent a significant leap towards sustainable and eco-friendly transportation. The proliferation of EVs necessitates a robust and efficient charging infrastructure to support their widespread adoption. The **Open Charge Point Protocol (OCPP)** emerges as a critical element in advancing EV charging technology, facilitating seamless communication and interoperability among charging stations and central management systems.

1.1.1 Electric Vehicles (EVs)

The rise of Electric Vehicles (EVs) marks a paradigm shift in the automotive industry towards greener and more sustainable transportation alternatives. EVs operate using electric energy stored in batteries, offering environmental benefits and reducing dependency on fossil fuels. The success of EVs relies on a comprehensive charging infrastructure that caters to diverse charging needs and ensures convenient and accessible charging points.

EV Charging Types

The EV charging landscape encompasses various charging levels, each with its specific characteristics and applications:

- **Level 1:** Provides standard 120-volt AC household outlets. Suitable for overnight charging and primarily used in residential settings.

- **Level 2:** Delivers power at 240 volts, significantly faster than Level 1 charging. Commonly found in public charging stations, workplaces, and commercial locations.
- **DC Fast Charging:** Offers high-voltage DC charging, enabling rapid charging and usually located along highways or in strategic areas to facilitate long-distance travel for EVs.

Efficient deployment of these charging levels is essential for catering to the diverse needs of EV users and enhancing the overall charging experience.

1.1.2 Open Charge Point Protocol (OCPP)

The **Open Charge Point Protocol (OCPP)**[1][2] stands as a pivotal element in the landscape of electric vehicle (EV) charging infrastructure. OCPP is an open and standardized communication protocol that facilitates communication between Electric Vehicle Charging Stations (EVCSs) and a Central System (CS), as shown in 1.1.

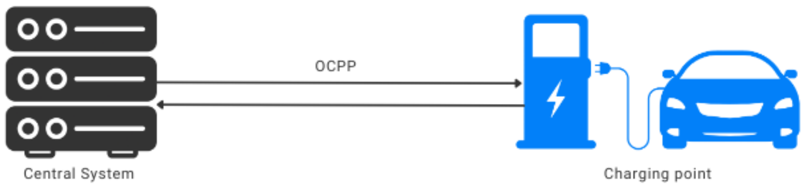


Figure 1.1: OCPP interaction between central system and charging point

OCPP was conceived with the primary objective of ensuring interoperability and compatibility among diverse EV charging equipment and management systems. It addresses the need for a uniform and robust communication standard in the burgeoning EV charging industry, which encompasses public charging stations, private charging facilities, and residential EV charging solutions.

Key aspects and features of OCPP include:

- **Vendor-Neutral**: OCPP is designed to be vendor-agnostic, allowing different manufacturers of charging stations and central management systems to implement the protocol, ensuring that EV infrastructure can support various hardware and software solutions.
- **Remote Management**: OCPP enables remote management and monitoring of EV charging stations, allowing CS operators to perform tasks like firmware updates, diagnostics, and monitoring of charging sessions from a centralized location.

- **Security**: Security is a paramount concern in the EV charging ecosystem. OCPP incorporates security features such as authentication, authorization, and data encryption to ensure the integrity and confidentiality of communications.
- **Scalability**: As the EV market continues to expand, OCPP's scalability becomes increasingly critical. The protocol is designed to support a growing number of charging stations and users.

OCPP implementations

To further illustrate the adoption and significance of OCPP in the EV charging landscape, 1.1 provides an overview of major charging companies that utilize OCPP. The adoption of OCPP has played a significant role in streamlining EV charging operations, enhancing user experience, and fostering innovation in the EV charging industry. It has enabled the development of a wide range of charging solutions, including public charging networks, fleet charging management, and smart grid integration.

1.1.3 Security in Open Charge Point Protocol (OCPP)

The rapid proliferation of Electric Vehicle Charging Stations (EVCSs) and the increasing reliance on the Open Charge Point Protocol (OCPP) underscore the importance of robust security measures within the protocol. As with any communication protocol, especially one that deals with critical infrastructure like EV charging, security is paramount to ensure the integrity, confidentiality, and availability of services.

Importance of Security in OCPP

With the rise of cyber threats targeting critical infrastructure, ensuring the security of OCPP communications is crucial for:

- Protecting user data and privacy.
- Ensuring the integrity of charging sessions.
- Preventing unauthorized access or control over charging stations.
- Safeguarding the broader electrical grid to which many charging stations are connected.

Table 1.1: MAJOR CHARGING COMPANIES USING OCPP

Company Name	**Region**	**OCPP Usage**	**Additional Notes**
ChargePoint	US	Yes	One of the largest EV charging networks in the US. Uses OCPP for interoperability.
EVBox	Europe	Yes	A leading EV charging solutions provider in Europe. Adopts OCPP for flexibility and compatibility.
Greenlots	Asia (Singapore-based)	Yes	Offers OCPP-compliant charging solutions, ensuring compatibility across different charging networks.
ABB	Global (Europe, US, Asia)	Yes	A major industrial equipment manufacturer that provides OCPP-compliant EV charging solutions.
Enel X	Europe	Yes	Provides OCPP-compliant charging solutions, allowing for easy integration with various management systems.
NewMotion	Europe	Yes	One of Europe's largest providers of EV charging services. Supports open standards like OCPP.

Security Features in OCPP

OCPP incorporates several security features designed to mitigate potential threats:

- **Authentication and Authorization:** OCPP supports mechanisms to authenticate both the charging station and the central system, ensuring that only authorized entities can establish a connection and communicate.
- **Data Encryption:** To protect the confidentiality of data during transmission, OCPP supports encrypted communication channels, typically using protocols like TLS (Transport Layer Security).
- **Message Integrity:** Ensures that the messages exchanged between the charging station and the central system have not been tampered with during transit.
- **Regular Security Updates:** As with any protocol, vulnerabilities may be discovered over time. The Open Charge Alliance regularly updates OCPP to address known security issues and enhance its security features.

Security Levels in Open Charge Point Protocol (OCPP)

OCPP has recognized the importance of security from its inception. However, as the protocol matured and the threat landscape evolved, the need for enhanced security became evident. The protocol has introduced varying levels of security across its versions:

- **OCPP 1.5 and Earlier: Basic Security**
  - Initial versions of OCPP operated primarily over HTTP, which does not inherently provide encryption.
  - Basic authentication mechanisms were in place, but they lacked the robustness required for critical infrastructure.
- **OCPP 1.6: Introduction of Enhanced Security**
  - OCPP 1.6 introduced the option to use WebSocket over TLS (WSS), adding an encryption layer to the communication.
  - This version also introduced more robust authentication mechanisms.
  - However, while these enhancements were available, they were optional, meaning that some implementations might still operate with basic security.
- **OCPP 2.0 and Later: Advanced Security**
  - OCPP 2.0 made significant strides in security, making many of the enhanced security features of 1.6 mandatory.

  - It introduced mutual authentication, where both the charging station and the central system authenticate each other, ensuring a higher level of trust in the communication.
  - The protocol also began emphasizing the importance of regular security updates and patches.

Implications of Security Levels

The varying security levels in OCPP have implications for both operators and users:

- **Trustworthiness:** Enhanced security ensures that users can trust the charging infrastructure, which is crucial for widespread adoption of electric vehicles.
- **Interoperability:** While security is paramount, it's essential that the introduction of new security features doesn't hinder the interoperability that OCPP aims to achieve.
- **Upgrade Considerations:** Operators using older versions of OCPP might need to consider upgrading to benefit from enhanced security features. This could involve both software and hardware upgrades.

In conclusion, as OCPP continues to evolve, its approach to security has become more sophisticated, addressing the challenges of a rapidly changing digital landscape. It's crucial for stakeholders to be aware of these security levels and ensure that their implementations are up-to-date and secure.

Challenges and Considerations

While OCPP has made significant strides in ensuring secure communications, there are challenges and considerations to be aware of:

- **Diverse Implementation:** The open nature of OCPP means that different vendors might implement the protocol differently, leading to potential inconsistencies in security measures.
- **Legacy Systems:** Older charging stations might be running outdated versions of OCPP that lack the latest security enhancements.
- **Physical Security:** While digital security is crucial, the physical security of charging stations is equally important to prevent tampering or unauthorized access.

In conclusion, while OCPP has incorporated robust security features, continuous vigilance, regular updates, and a holistic approach to security (considering both digital and physical aspects) are essential to ensure the safe operation of EV charging infrastructure.

1.2 Problem description and solution overview

1.2.1 Problem Description

The proliferation of Electric Vehicles (EVs) has led to a dramatic increase in the deployment of OCPP-based charging infrastructures worldwide. Given its vital role in the EV ecosystem, ensuring the security of OCPP implementations is of paramount importance. Yet, several challenges arise when attempting to test the security of these implementations systematically:

1. **Diverse Implementations:** OCPP implementations can vary greatly depending on the manufacturer, version, or specific requirements of the deployment. The heterogeneity of the system complexities precludes the formulation of a universally applicable testing methodology.
2. **Language and Platform Independence:** OCPP implementations can be found in a myriad of programming languages and platforms, further complicating the process of testing.
3. **Avoidance of Code Analysis:** A truly general testing method should not rely on specificities of the codebase, making traditional code analysis-based testing techniques less suitable.

The fundamental challenge resides in the development of a testing methodology that is simultaneously systematic and comprehensive, addressing the aforementioned complexities inherent to the system.

1.2.2 Solution Overview

To address the inherent challenges in testing OCPP's diverse landscape, a shift from traditional code-based analysis is necessary. The proposed solution revolves around the principle of *black box testing.*

Black Box Testing

Black box testing refers to a testing methodology where the internal workings of the item to be tested are not known by the tester. Instead, the tester knows only the inputs and the expected outcomes. This methodology is particularly suitable for OCPP testing for several reasons:

- **Implementation Independence:** Black box testing doesn't rely on the specifics of how the OCPP implementation is realized, making it suitable for diverse implementations.
- **Language Neutrality:** Since the internal code is not scrutinized, the programming language or platform on which the OCPP implementation resides becomes irrelevant, thus ensuring a broader application of the testing approach.
- **Focus on Functional Outcomes:** By emphasizing the expected behavior of the OCPP implementation rather than its internal mechanics, black box testing can identify discrepancies between intended and actual behaviors, highlighting potential security vulnerabilities.

In essence, by leveraging black box testing for OCPP, we can systematically test various implementations irrespective of their internal complexities or specificities. This method offers a holistic and robust security assessment that not only identifies vulnerabilities but also ensures the integrity and reliability of critical infrastructure systems.

Chapter 2

Approach

The endeavor to enhance the security and integrity of the Open Charge Point Protocol (OCPP) extends beyond the mere prevention of operational disruptions; it involves the reinforcement of critical infrastructure against potential security breaches. Although traditional testing methodologies contribute valuable insights, they may not be entirely comprehensive, with a propensity to overlook nuanced and unpredicted vulnerabilities.

Fuzz testing, commonly referred to as fuzzing, is a dynamic, automated software testing technique that systematically injects a broad spectrum of input data into a software system. Its primary aim is the detection of hidden anomalies and security weaknesses that are not evident during standard operational conditions. Fuzzing delves into an extensive array of inputs, striving to unearth vulnerabilities that may elude traditional testing methods.

It is imperative to recognize that fuzzing methodologies differ in their underlying principles and objectives. They can be generally classified into:

- **White-Box Fuzzing:** White-box fuzzing tools, such as AFL[3] and LibFuzzer[4], operate with in-depth knowledge of the internal workings of the target system. By accessing the system's source code, these fuzzers analyze code paths and utilize feedback mechanisms to generate inputs that traverse previously unexplored execution paths. Their primary advantage is their precision; by understanding the system's internal structure, white-box fuzzers can often uncover vulnerabilities that might elude other methods. However, they predominantly focus on executable files and might not be ideally suited to verify the accuracy and robustness of a protocol's implementation.
- **Black-Box Fuzzing:** Black-box fuzzing does not require knowledge of the system's internal logic or structure. It treats the software under test as a black box, inputting various data and observing outputs or system behavior. Its strength lies in its versatility; since black-box fuzzers don't rely on any specific knowledge about the

software's internals, they are universally applicable across different software systems. This approach can be particularly effective for testing the correctness and resilience of protocol implementations, where internal code paths might vary widely across different implementations, but the protocol behavior should remain consistent.

Given our specific goal of validating OCPP protocol implementations, the distinction between these two fuzzing methodologies accentuates the need for an approach precisely tailored to our requirements.

This chapter delves into the intricacies of our selected approach, black box fuzzing. We will elucidate its mechanisms, inherent challenges, and the individual components that constitute this method. In doing so, we aim to highlight its efficacy in verifying the correctness of OCPP implementations. Subsequent sections will provide a comprehensive exploration of our solution.

2.1 Our solution

2.1.1 Black-Box Fuzzing: An Overview

- **General Mechanism:** At its core, black-box fuzzing is an external, dynamic testing technique. Without any knowledge or bias of the system's internal code, structure, or algorithm, it solely focuses on the system's responses to a wide array of inputs. Black-box fuzzing essentially mimics an external attacker by probing and pushing the system to its limits. By assessing the input-output behavior, it evaluates the system's robustness and security posture. This way, the technique becomes instrumental in uncovering vulnerabilities that might remain hidden during conventional testing, such as edge cases not covered in manual tests or unforeseen input combinations.
- **Input Generation:** The success of black-box fuzzing significantly relies on its ability to generate a vast variety of input data. These inputs range from valid and expected ones to malicious, malformed, or anomalous variants. Using techniques like mutation (where existing data is modified) and generation (where entirely new data is created), black-box fuzzing seeks to challenge the system's defenses and logic. The intention is not just to see if the system can handle regular inputs, but to understand how it behaves under irregular, unexpected, or stress conditions. This comprehensive input generation is what allows black-box fuzzing to identify vulnerabilities that might be exploited in real-world attack scenarios.
- **Output Analysis:** Once inputs are fed into the system, the ensuing step is to meticulously analyze the outputs. In the context of black-box fuzzing, this doesn't just mean checking if the system crashes or behaves erratically. It's about deciphering the nuanced differences between expected and actual outcomes, be it in terms of the nature of the response, or even seemingly benign discrepancies. By interpreting

these outputs, researchers can pinpoint specific areas of concern, thereby guiding subsequent testing efforts or rectification measures.

2.1.2 The Rationale for Black-Box Fuzzing

Fuzzing has consistently proven its value in the realm of software security testing. Within the fuzzing spectrum, black-box fuzzing emerges as particularly apt for our endeavor, offering a trinity of key advantages that align seamlessly with the complexities of testing OCPP implementations:

- **Versatility:** Unlike white-box methods that require detailed insights into the software's internal code structure, black-box fuzzing thrives on its versatility. This form of testing is predicated on probing the external behaviors of a system. Hence, irrespective of the under-the-hood intricacies, modifications, or evolutions of the OCPP implementations, black-box fuzzing remains resilient and adaptive, ensuring it remains relevant and effective over time.
- **Independence from Internal Code:** One of the predominant challenges in validating various OCPP implementations is the myriad of internal code structures and languages used by different entities. Black-box fuzzing, by virtue of its design, circumvents this challenge altogether. The technique's ability to assess a system solely based on its inputs and outputs—without necessitating knowledge of its internal code—makes it an invaluable asset, especially when considering the diverse and proprietary nature of OCPP implementations.
- **Effectiveness in Protocol Validation:** Protocols, by their inherent nature, define a set of rules for data communication. Black-box fuzzing, in its essence, tests the robustness of these rules by feeding diverse and often unexpected inputs. In the context of OCPP, this means validating whether the implementations adhere to the protocol's stipulations consistently and securely, even under unanticipated conditions. This direct alignment of black-box fuzzing's capabilities with protocol validation requirements underscores its efficacy in our specific use-case.

Given these attributes, black-box fuzzing not only presents itself as a logical choice but emerges as a veritable toolset uniquely positioned to tackle the nuances and challenges associated with securing OCPP implementations.

2.1.3 Black-Box Fuzzing for OCPP

Black-box fuzzing emerges as an apt solution to evaluate the security posture of protocol implementations like OCPP.

- **Protocol-Centric Approach:** OCPP, being a protocol, has intricacies that are distinct from typical software applications. To craft an effective fuzzer for it, one must dive deep into its specification. By comprehending its operation mechanisms, message exchange patterns, and expected error responses, we can shape the fuzzing inputs in a way that they genuinely challenge the protocol's bounds.
- **Unbiased Testing:** The beauty of black-box fuzzing lies in its ignorance of the internal workings of the implementation. While we make our fuzzer aware of OCPP's rules and conventions, it remains oblivious to the specific details of any OCPP implementation. This ensures unbiased, genuine, and rigorous testing—a fuzzer probes the system as an external entity would, free from any preconceptions or biases about the internal logic.
- **Adherence to Specification:** The ultimate goal of our fuzzing endeavors is to validate that the OCPP implementation under test strictly adheres to the protocol's description. Any deviation, whether it results in an obvious malfunction or a subtle discrepancy in response patterns, is a potential vulnerability or point of non-compliance. Our protocol-aware fuzzer is designed to catch these deviations, ensuring that OCPP implementations are both secure and standard-compliant.

2.1.4 Key Challenges and Considerations

While black-box fuzzing presents a promising approach for validating OCPP implementations, it's not without its challenges. Deploying this testing technique in a protocol-centric context brings forth certain unique hurdles. The absence of insights into the internal workings of the target system, which is a defining feature of black-box testing, can simultaneously be a source of certain complexities. Before delving into the mechanics of our solution, it is essential to outline and understand these challenges, as they shape the strategies and methodologies we adopt. This section aims to spotlight these critical considerations and the inherent complexities they introduce into the fuzzing process.

- **Coverage:** One of the core challenges with black-box fuzzing lies in the realm of test coverage. Traditional fuzzing techniques, especially those that fall under the white-box category, often gauge coverage by monitoring the executed lines of code. This allows for a tangible measure of which parts of the application are tested and which remain untouched. However, in the context of black-box fuzzing, this metric is inaccessible due to the inherent nature of the method, we're in the dark about the internal intricacies of the system under test.

  Given this constraint, it's imperative to conceptualize an alternative metric for understanding test coverage. Our focus on OCPP, a protocol, provides us with a unique advantage in this regard. Protocols, by definition, dictate a series of potential states and transitions, often contingent on specific sequences of messages. Consider, for instance, the state of a charge point: it can either be authorized or unauthorized to initiate charging. This binary state representation, though rudimentary, illustrates the existence of distinct execution paths within the protocol's purview.

Therefore, to gauge coverage, we ought to shift our perspective from code paths to protocol paths, emphasizing the different states and state transitions as dictated by the sequences of messages. This strategy ensures that our fuzzer comprehensively tests all possible behaviors and responses of an OCPP implementation, thus providing a reliable measure of test coverage.

- **Response Interpretation:** A significant challenge in the domain of black-box fuzzing, especially when addressing protocol implementations, lies in the interpretation of system responses. The complexity is compounded by the subtlety of potential vulnerabilities or behaviors, which may not always manifest as overtly erroneous.

  To effectively discern the nature of a response, it's essential to have an intimate understanding of the protocol in question. This involves a thorough examination of the originating request: understanding its construction, identifying which properties were targeted for fuzzing, and recognizing the specific type of fuzzing applied. For instance, it's crucial to determine whether the fuzzer's payload involved sending a string exceeding the stipulated *maxLength* property or omitting a mandatory property before dispatching the message.

  By analyzing the constructed request, we can deduce the expected outcome or "predicted" response, grounded in the protocol's specifications. This predicted outcome then serves as a benchmark against which the actual server response is gauged. Discrepancies between the two can be indicative of potential issues—whether they are genuine vulnerabilities, deviations from the protocol, or other categories warranting further investigation.

- **Efficiency:** Efficiency stands as a paramount consideration for a fuzzer, particularly given the desire to dispatch a multitude of messages within a condensed timeframe. In the course of our experimentation, it became evident that the predominant contributor to inefficiencies was the message generation phase. This aspect emerged as the principal bottleneck for a fuzzer of this nature. Addressing this challenge and proposing solutions to enhance efficiency will be elucidated in subsequent sections.

2.1.5 Visualizing Our Solution

As we navigate the intricacies of our fuzzing solution, a visual representation can significantly aid in understanding the architecture and workflow. 2.1 provides a schematic view of our fuzzer, capturing its essential components and interactions.

In the next section we will dissect the components of the figure, elucidating their individual roles and their collective contribution to the overall fuzzing process.

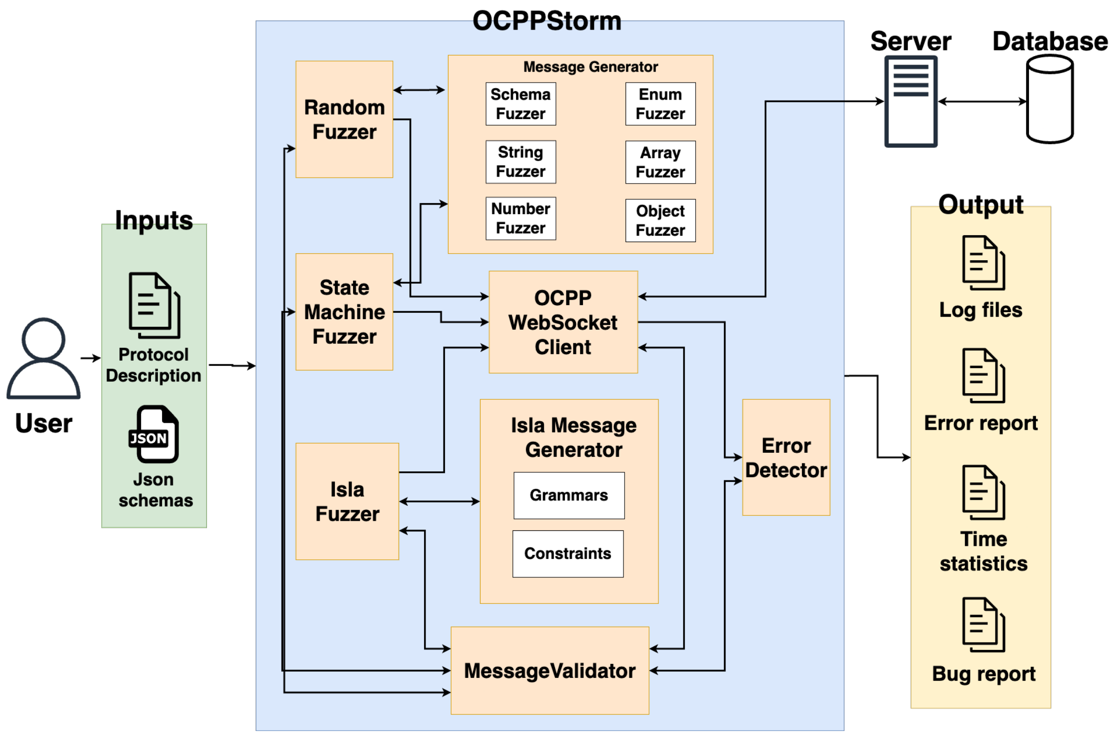


Figure 2.1: Schematic representation of our fuzzing solution.

2.2 Fuzzer Architecture and Workflow

2.2.1 Input

The effectiveness and precision of any fuzzer are largely contingent on the quality and specificity of its inputs. Our fuzzer, tailored to address the unique challenges of protocol fuzzing, primarily relies on two distinct but crucial inputs.

1. **Protocol Description**:
   - *Purpose*: The protocol description forms the foundational knowledge for our fuzzer. It encapsulates the protocol's rules, permissible message structures, and other intricate details. The fuzzer harnesses this description to gain a comprehensive understanding of the protocol, ensuring that its testing approach remains both rigorous and aligned with protocol specifications.
2. **JSON Schemas**:
   - *Purpose*: These schemas play a pivotal role in the fuzzing process, serving multiple essential functions.

(a) **Message Construction**: Both the *Random Fuzzer* and the *State Machine Fuzzer* utilize the JSON schemas to deduce how messages should be accurately constructed. This ensures that the generated messages, while intentionally malformed for testing purposes, are still rooted in the realistic structure of actual protocol messages.

(b) **Error Validation**: The JSON schemas also function as validators. Firstly, they are employed to validate the fuzzed input. By pinpointing which specific part of a message has been fuzzed, the *Error Detector* can deduce the expected error response from the server. Moreover, after a message is dispatched, the schemas assess the integrity and accuracy of the server's response. This dual use not only enables the detection of discrepancies but also offers a more intuitive understanding of how the server might react to malformed inputs, elucidating potential vulnerabilities or misconfigurations.

(c) **Grammar and Constraints for Isla Fuzzer**: The JSON schemas have been instrumental in formulating grammars and constraints for the *Isla Fuzzer*. Leveraging the Isla library [5], the *Isla Fuzzer* employs these grammars and constraints, rooted in the JSON schemas, to generate messages systematically.

In essence, these inputs not only guide the fuzzer in generating test cases but also aid in the intelligent interpretation of results, thereby ensuring a more refined and targeted fuzzing approach.

2.2.2 OCPPStorm

The OCPPStorm fuzzer is a modular and structured tool designed to examine and test the robustness of OCPP implementations. Through its intricately connected components, the fuzzer systematically challenges the protocol's security, reliability, and efficiency. The objective behind this is not only to uncover potential vulnerabilities but also to ensure that OCPP implementations are resistant to various unexpected or malicious inputs.

As illustrated in 2.2, the OCPP Fuzzer incorporates a structured workflow integrating three primary fuzzing modules: the "Random Fuzzer", "State Machine Fuzzer", and "Isla Fuzzer". These modules interact with specialized components such as the Message Generator—with its variety of fuzzing methods—and the Isla Message Generator that uses grammars and constraints. The communication with the external server is managed by the OCPP WebSocket Client. The generated messages undergo validation through the Message Validator and any identified anomalies or deviations are promptly detected by the Error Detector. This architectural synergy ensures meticulous crafting and dispatching of test messages, maximizing the probability of identifying anomalies, errors, and potential vulnerabilities in the targeted OCPP implementation.

In the following subsections, we delve into the individual components of OCPPStorm,

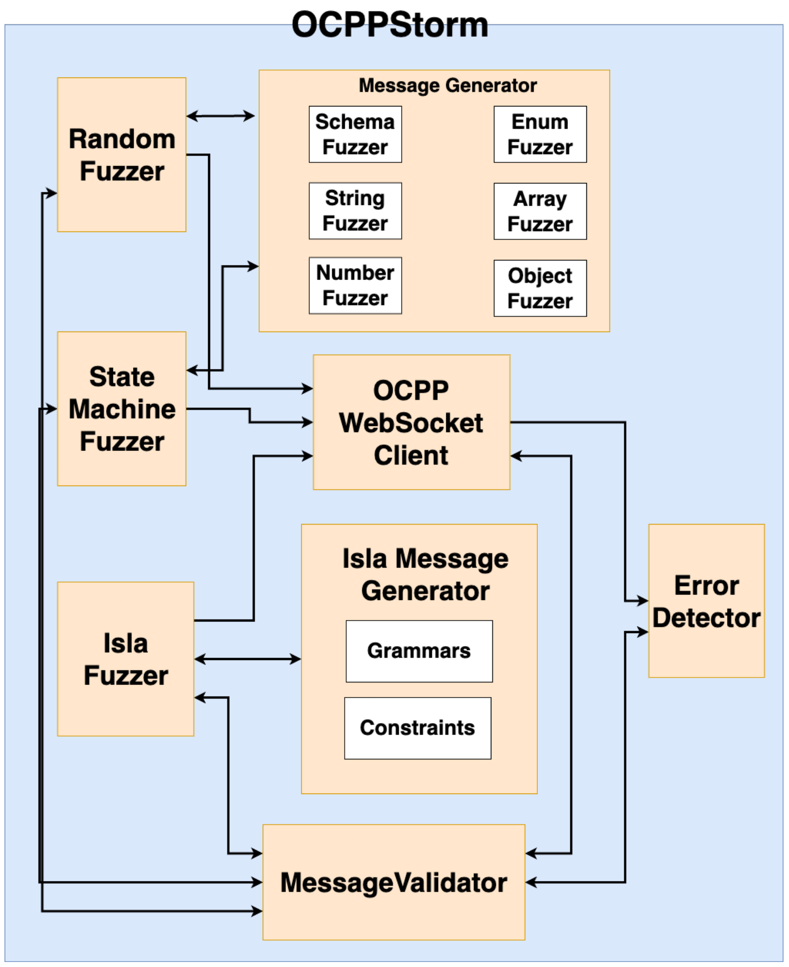


Figure 2.2: Architecture of OCPPStorm fuzzer

detailing their functionalities and significance within the system. In order to understand the components of OCPPStorm we need to provide details about the OCPP messages.

Message Type and Identification

To ensure accurate communication, it's essential to correctly identify the type and origin of each message.

The Message Type Each message bears a distinct *Message Type Number* that specifies its kind. 2.1 details the types and their respective directions.

If a server intercepts a message bearing an unrecognized *Message Type Number*, it is

Table 2.1: MESSAGE TYPES AND THEIR RESPECTIVE DIRECTIONS

MessageType	**MessageTypeNumber**	**Direction**
CALL	2	Client-to-Server
CALLRESULT	3	Server-to-Client
CALLERROR	4	Server-to-Client

instructed to discard the message payload, preserving its header for reference. Further, each message type may necessitate specific fields.

The Message ID A unique *Message ID* is indispensable for message tracking, especially for request identification. For *CALL* messages, the ID should be distinct from previously used IDs over the same WebSocket connection by the sender. However, for *CALLRESULT* or *CALLERROR* messages, the ID must echo the *CALL* message they are responding to. 2.2 illustrates the constraints for these unique identifiers.

Table 2.2: CONSTRAINTS FOR UNIQUE MESSAGE IDENTIFIERS

Name	**Datatype**	**Restrictions**
messageId	string	Limited to 36 characters

It's important to adhere to these standards for efficient communication and response management.

Message Structures in OCPP

The OCPP communication protocol initializes by establishing the identity of the charge point during a WebSocket handshake.

Understanding the 'Call' Structure Within the OCPP framework, a *Call* comprises four primary components:

- **MessageTypeId**: A standardized identifier.
- **UniqueId**: Serves as a distinctive tag ensuring congruence between requests and their corresponding results.
- **Action**: Specifies the desired remote operation.
- **Payload**: Contains pertinent arguments related to the action. In situations devoid of payload content, an empty JSON object ({}) is favored over the use of null, ensuring clarity and conciseness.

The *Call* syntax is typically represented as:

```
[MessageTypeId, UniqueId, Action, {Payload}]
```

Understanding the 'CallResult' Structure In the OCPP protocol, a successful handling of a call leads to the generation of a *CallResult*. It's important to note that certain error situations predefined in OCPP are perceived as regular results. Thus, they're treated as standard *CallResult*, even if they may not be favorably received by the recipient.

A typical *CallResult* is composed of three primary components:

- **MessageTypeId**: A standardized identifier.
- **UniqueId**: An identifier matching the one in the call request, ensuring the recipient can correlate the result with the corresponding request.
- **Payload**: Houses the outcome of the executed action. For instances where the payload lacks content, it's advocated to employ an empty JSON object ({}) as a substitute for null to maintain clarity and brevity.

The general representation of a *CallResult* syntax is:

```
[MessageTypeId, UniqueId, {Payload}]
```

Understanding the 'CallError' Structure A *CallError* is initiated under two primary conditions:

1. Perturbations in message transport such as network disruptions or service unavailability.
2. Receipt of a call with content discrepancies such as missing essential fields or duplicate unique identifiers.

The standard representation of a *CallError* entails five components:

```
[MessageTypeId, UniqueId, errorCode, errorDescription, {errorDetails}]
```

2.3 summarizes the key components of a *CallError*.

2.4 provides a comprehensive enumeration of valid error codes.

Table 2.3: KEY COMPONENTS OF A CALLERROR

Field	**Description**
UniqueId	A consistent identifier aligning with the initiating call request
ErrorCode	A specific string from the subsequent ErrorCode tabl.
ErrorDescription	Preferably populated; otherwise, a transparent empty string “”
ErrorDetails	A JSON object detailing the error; in the absence of details, an empty object ({}) is mandated

Table 2.4: ENUMERATION OF VALID ERROR CODES

Error Code	**Explanation**
NotImplemented	Receiver is unaware of the requested action
NotSupported	Receiver identifies but doesn’t support the requested action
InternalError	Receiver’s internal disruption prevented successful action execution
ProtocolError	Action’s payload is incomplete
SecurityError	Security complications during action processing
FormationViolation	Action’s payload has syntactical errors or non-compliance with PDU structure
PropertyConstraintViolation	Valid syntax but contains at least one incorrect field value
OccurenceConstraintViolation	Proper syntax but contravenes occurrence constraints in a field
TypeConstraintViolation	Correct syntax but violates data type constraints
GenericError	Captures errors not encapsulated above

OCPP Message Exchange Example

In this subsection, we explore a standard communication exchange between the charge point and the central system using the OCPP protocol.

Scenario:

1. *Initialization*: A charge point powers up and establishes its connection with the central system. Upon establishing the connection, the charge point dispatches a `BootNotification` to notify the central system of its operational status. In response, the central system verifies the identity of the charge point and sends back a `BootNotificationResponse`.

2. *Authorization*: Subsequently, when a car owner wishes to initiate a charging session, the charge point communicates an `Authorize` request to the central system. The central system then replies with an `AuthorizeResponse`.

3. *Start of Transaction*: Once authorized, the charge point begins the transaction by sending a `StartTransaction` message. The central system acknowledges with a corresponding response.

4. *Meter Reading*: During the charging session, the charge point periodically sends `MeterValues` messages to update the central system about the ongoing energy consumption.

5. *End of Transaction*: Upon completion of the charging process, the charge point sends a `StopTransaction` message to signal the end of the current session.

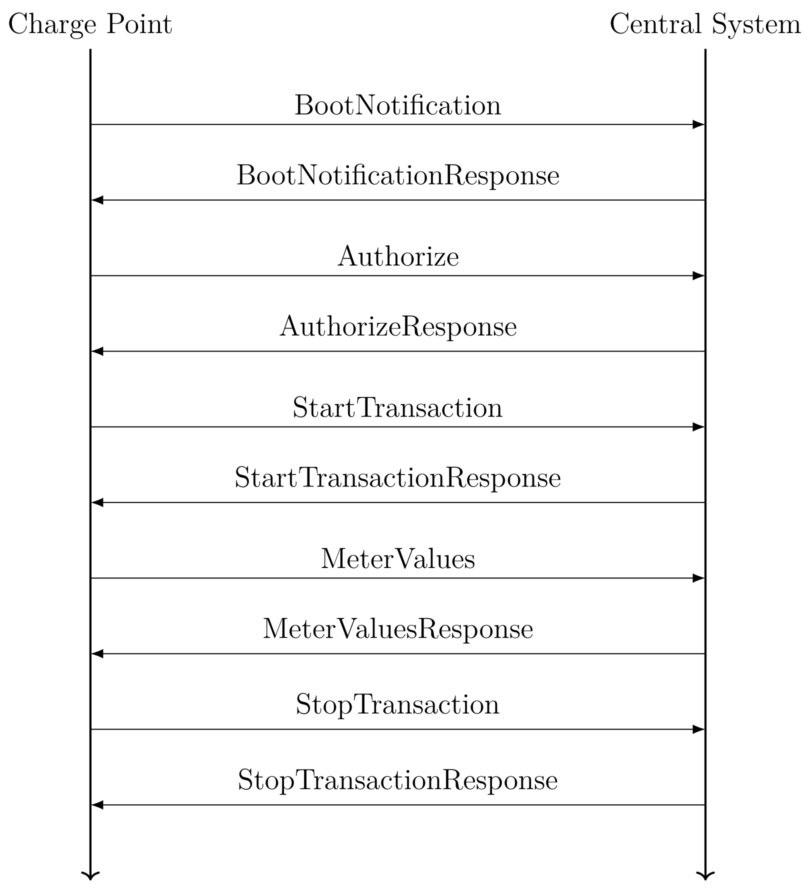


Figure 2.3: Sequence of OCPP messages between Charge Point and Central System

In Figure 2.3, we visually represent the sequence of messages exchanged between the Charge Point and the Central System.

Fuzzing Modules

OCPPStorm is equipped with three distinct fuzzing modules. Two of these modules intricately leverage the capabilities of the newly implemented Message Generator, while the third is rooted in the Isla library, meticulously utilizing the grammars and constraints devised specifically for fuzzing the OCPP protocol.

These fuzzing modules stand as the core components of OCPPStorm, orchestrating seamless interactions with their corresponding message generators, the Message Validator, and the OCPPWebSocketClient, ensuring a comprehensive and efficient fuzzing process. The three modules are aptly named "Random Fuzzer", "State Machine Fuzzer", and "Isla Fuzzer".

Random Fuzzer The Random Fuzzer is notably the simplest among the three. Its functionality is intertwined with the *Message Generator*, which is designed to produce messages randomly. During its operational cycle, this fuzzer goes through numerous iterations. In each loop, it selects a message type at random and generates it. The resultant message is then validated using the JSON schema and additional controls, aiming to identify which specific property has been fuzzed. This meticulous validation aids the fuzzer in predicting the server's response. Following validation, the message is transmitted, and the corresponding server feedback is assessed by the *Error Detector*.

State Machine Fuzzer The state machine fuzzer serves as a sophisticated adaptation of the rudimentary *Random Fuzzer*. Specifically designed for this context, the fuzzer enables users to define a series of OCPP messages, representing the intricate mechanics of a "state machine". The intention behind allowing the state machine to be user-defined is both deliberate and pivotal. Recognizing that many OCPP implementations remain proprietary and each uses distinct values for some properties, this capability empowers users to tailor the OCPP messages based on the unique parameters adopted by their individual servers. Consequently, users can either innovate new OCPP sequences or modify the ones we furnish by adjusting the parameters, ensuring compatibility with their specific instances and implementations.

Given that our fuzzer operates under the paradigm of a black-box mechanism, determining code coverage becomes an uncomplicated endeavor. Consequently, employing a state machine becomes an efficacious strategy to delve into all potential states of the server, theoretically encapsulating the entire codebase.

This state machine is bifurcated into two distinct segments:

1. **User-Defined State Machine:** This segment necessitates a user-provided sequence of OCPP messages, encapsulating an archetypal communication between the charge

point and the central system. Commencing its operation, the fuzzer transmits a predetermined set of fuzzed messages, curated by the *Message Generator*. Upon completion of this phase, the fuzzer picks the initial valid message from the state machine's input and forwards it. The subsequent response's accuracy is assessed by the *Error Detector*, operating under the assumption that the input sequence is flawless and thus expecting a pertinent reply. Post-receipt of this response, the fuzzer transitions to the ensuing state, reigniting the dispatch of fuzzed messages. This cycle perseveres until all input OCPP messages are duly processed.

As an example, consider setting the fuzzed message count at 10, drawing input from a file named `input.txt`, containing:

```
["BootNotification", {"chargePointVendor": "AAAVENDOR",
"chargePointModel": "Model X123","chargePointSerialNumber":
"CP-1234567890","chargeBoxSerialNumber": "CB-0987654321",
"firmwareVersion": "1.0.0","iccid": "12345678901234567890","imsi":
"98765432109876543210","meterType": "Electric Meter",
"meterSerialNumber": "M-9876543210"}]

["Authorize", {"idTag": "2"}]
```

Post the initial 10 fuzzed messages, the fuzzer forwards the `BootNotification` message, anticipating a positive affirmation of transition via the response. Subsequently, it shoots another series of 10 fuzzed messages, targeting the newfound server state. Progressing, it then transmits the next message from `input.txt`, identified as `Authorize`. After validating the response, an additional 10 fuzzed messages culminate this fuzzing phase.

2. **Status Notification Transition Fuzzer:** This component is designed specifically to fuzz every conceivable transition associated with the `StatusNotification` message.

   Drawing from the official documentation, this message is dispatched by a Charge Point to the Central System. Its primary purpose is to relay information regarding any status alterations or errors within the Charge Point. Notably, the transition from a preceding status to a new status might prompt the Charge Point to issue a `StatusNotification` request to the Central System.

   Having delineated the role of the `StatusNotification`, we must emphasize that all potential state transitions have been comprehensively defined. These state transitions encapsulate every possible scenario, mapping out every feasible state that might be encountered. In stark contrast to the user-defined state machine, the transitions here are explicitly stipulated by the protocol, obviating the necessity for user intervention and facilitating a fully automated process.

   Our foundational assumption posits the Charge Point's commencement state as "Available". Initiating our fuzzing from this state, our approach mirrors the technique employed in the user-defined state machine fuzzer. We predetermine the count of fuzzed messages designated for each server state and dispatch them sequentially. Subsequent to this, we opt for one potential transition from the current state, i.e. "Available", and subject the new state to fuzzing. Following this phase, a state, reachable from our current stance, is randomly selected and the transition is executed. This cyclical process perseveres until all plausible transitions are traversed.

Nevertheless, a conceivable challenge may arise: reaching an impasse where all transitions from a particular state have been explored, thus leaving the fuzzer in a static state indefinitely. To preclude this predicament, if we deplete unvisited transitions from the present state, we resort to a previously traversed transition. Progressing to this state, we then evaluate if it leads to any uncharted transitions. This iterative strategy persists until an unvisited transition is detected. Once identified, the fuzzing procedure is reinitiated, with this cycle continuing until every potential transition has been examined.

Isla Fuzzer The Isla Fuzzer is an integral component of our OCPP fuzzing solution, deriving its capabilities from the Isla library. A representation of this component is shown in 2.4.

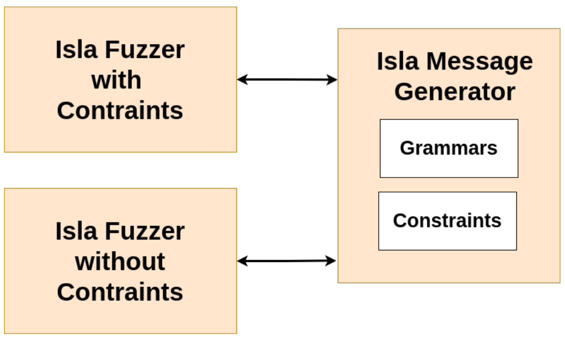


Figure 2.4: Components of the Isla Fuzzer

This fuzzer offers users two distinct options:

1. **Fuzzing With Constraints:** This mode integrates grammars with constraints to guide the fuzzing process. The constraints essentially serve as an Isla interpretation of the JSON properties of all OCPP messages. They play a pivotal role in determining which properties are eligible for fuzzing and provide guidelines on how the fuzzing should be executed. This type of fuzzing is the slowest among the others.

2. **Fuzzing Without Constraints:** Operating in this mode allows the generation of messages without taking into consideration the OCPP's constraints. This approach accelerates the message generation, enabling the creation of more fuzzed messages within the same timeframe. By disregarding the myriad combinations of constraints,

the generated messages exhibit increased randomness, with messages designed to fuzz all properties simultaneously.

It's worth noting that the Isla Fuzzer not only communicates seamlessly with the Isla Message Generator but also coordinates the fuzzing operations in synergy with the fuzzer's standard modules. This includes the Error Detector, Message Validator, and OCPPWebSocketClient, ensuring a comprehensive and efficient fuzzing process.

Message Generator

The Message Generator stands as a pivotal component within the fuzzer architecture, acting as an alternative to the Isla Message Generator, specifically tailored for the task of fuzzing the OCPP protocol. While the Isla library provides a more generalized approach to message generation, the Message Generator is intricately designed to cater to the nuances and specificities of the OCPP protocol. This component is shown in 2.5.

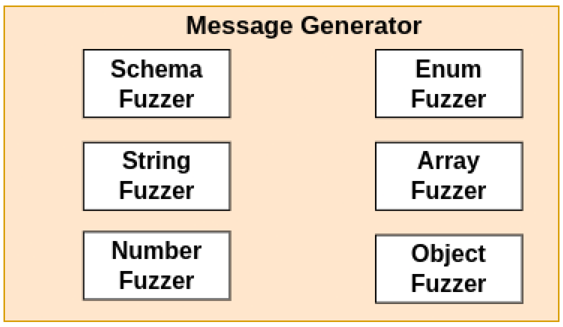


Figure 2.5: Components of the Message Generator

The Message Generator plays a pivotal role in the fuzzing process, encompassing a multifaceted approach to message creation and management. Its functionalities can be broadly classified into the following categories:

- **Schema-Conformant Message Creation:** The Message Generator leverages predefined schemas to generate messages that comply with various OCPP message types. This ensures that the generated messages strictly adhere to the expected structure of the protocol, making it instrumental in uncovering vulnerabilities linked to valid, schema-based inputs.
- **Randomized Message Generation:** Beyond schema-specific messages, the Message Generator has the capability to produce entirely random messages. This capability expands the scope of the fuzz testing, targeting potential vulnerabilities that might emerge from unforeseen and irregular inputs. The randomness is achieved by creating random actions and payload structures, increasing the unpredictability factor.

- **Interfacing with Other Components:** The Message Generator works in synergy with the other components of OCPPStorm, orchestrating the fuzzing operations harmoniously with common modules. Its output serves as an input for other components such as the error detector, message validator, and OCPP WebSocket client. This seamless integration ensures the timely dispatch, validation, and detection of any anomalies or errors in the generated messages, subsequently logging them for deeper analysis.

The Message Generator serves as an alternative to the Isla library, tailored specifically for fuzz testing the OCPP protocol. While the Isla library offers a generic platform allowing users to specify grammars for fuzzing, the Message Generator is a specialized tool designed exclusively fuzzing json schemas.

Isla Message Generator

The Isla Message Generator functions in tandem with the Isla Fuzzer, primarily focusing its operations on the generation of OCPP messages. This intricate component capitalizes on two primary strategies for message generation, leveraging both grammars and constraints, which are elucidated as follows:

- **Grammars:** They represent the foundational rules governing message generation, ensuring that the created messages maintain structural integrity. All the grammars have been written to comply with the json schemas defined for the ocpp protocol.
- **Constraints:** These are specific limitations or stipulations that need to be observed during the process of message creation. Their primary role is to ensure adherence to particular norms or to deliberately violate them, depending on the strategy.

Two distinct strategies delineate the operation of the Isla Message Generator:

1. **Generation with Constraints:** Adopting this strategy implies that during the message fuzzing process, a defined set of constraints is selected for the given message and harnessed in its generation. For instance, while generating the `Authorize` message, the only property it possesses is the `idTag`. This property, of the string type, is constrained by a maximum length of 20. Utilizing the constraints-driven generation, the Isla library would be instructed to create an `idTag` exceeding 20 characters, deliberately contravening the protocol's constraints.
2. **Generation without Constraints:** This approach sidelines the constraints delineated in the OCPP protocol's JSON schemas, placing emphasis solely on the grammars of these schemas. Consequently, while the generated messages will retain the appropriate schema, their properties will be randomly fuzzed. No specific constraints are enforced, allowing for a broader spectrum of generated outputs.

Error Detector

The Error Detector acts as a vital sentinel within the OCPPStorm fuzzer, meticulously analyzing the server's feedback in various scenarios. It bridges the gap between the expected behavior dictated by the OCPP protocol and the actual responses from the server, ensuring robustness and conformity. Given its significance, understanding the core functionalities of the Error Detector is paramount:

- **Server Response Analysis:** The Error Detector scrutinizes the server's reactions in light of the messages dispatched by the client. Two main scenarios emerge:
    1. *Valid OCPP Messages:* In instances where legitimate OCPP messages are relayed, the detector ascertains that the server reciprocates with appropriate responses and carries out the required operations.
    2. *Fuzzed Message Examination:* When fuzzed messages are introduced to the server, the detector's role amplifies. It delves into understanding the server's feedback, registering any irregularities or deviations from the expected behavior.
- **Fuzzed Message Error Prediction:** Beyond mere response analysis, the Error Detector wields predictive capabilities. By juxtaposing the fuzzed message against its corresponding JSON schema, and leveraging certain logical constructs, it can anticipate the errors the server might return. This stems from the ability to discern which constraints of the JSON schema are infringed upon. Such anticipatory prowess is instrumental in unveiling disparities between the protocol's guidelines and its real-world execution.
- **Error and Bug Statistics Collection:** Integral to the OCPPStorm's assessment toolkit, the Error Detector accumulates essential data regarding server errors and potential vulnerabilities. This statistical data is subsequently chronicled in log files, paving the way for in-depth analysis and discussions.

Referring to the provided architecture diagram (2.5), it's evident how the Error Detector interfaces seamlessly with various components, ensuring holistic and comprehensive fuzz testing of the OCPP protocol.

OcppWebSocketClient

The *OcppWebSocketClient* plays a central role in managing the communication with the charge point via the OCPP protocol. Acting as the primary interface to dispatch and receive messages, it ensures that all interactions conform to the OCPP standards, while also overseeing specific scenarios like initiating or concluding transactions. Here's a comprehensive breakdown of its primary responsibilities:

- **Connection Management:** The client seamlessly establishes a connection with the central system. By taking a charge point identifier and the base URL, it crafts the precise endpoint to interface with and initiates the handshake.
- **Message Dispatch:** The client has capabilities to send both fuzzed and genuine messages to the central system. For valid messages pertaining to transactions, it makes sure the transaction ID is appropriately appended, ensuring the correct flow of operations.
- **Transaction Handling:** The client exhibits a keen sense of transactional awareness. It recognizes the commencement of a transaction, recording the corresponding transaction ID, and acknowledges its conclusion. This transactional oversight is crucial for scenarios that involve multiple sequential operations with the charge point.
- **Connection Termination:** Post interaction, the client gracefully terminates the connection with the charge point, ensuring no residual open links.

Overall, the *OcppWebSocketClient* acts as a robust conduit for OCPPStorm to interact with central systems, ensuring the flow of valid and fuzzed messages, while also maintaining a vigilant eye on the server's responses.

Message Validator

The *Message Validator* is integral to ensuring the correctness and integrity of messages exchanged within the OCPPStorm framework. It functions as a gatekeeper, verifying that messages align with specific predefined structures and standards. Below are its primary responsibilities:

- **Timestamp Verification:** One of the validator's key checks involves scrutinizing the 'timestamp' property within messages. This attribute, if present, is validated against the RFC3339 standard, ensuring its format and value are consistent with the said specification. This adherence to a universally recognized timestamp format ensures synchronization and time-related consistency across messages.
- **Schema Compliance:** Beyond timestamp validation, the validator ensures that messages are compliant with their corresponding schemas. It compares the structure and values of the incoming message against the predetermined schema. Any deviation from this schema results in the message being marked as non-compliant.
- **Exception Handling:** The validator is not just limited to positive verifications. It is also equipped to handle scenarios where messages don't align with expectations. In instances where the message structure breaches its corresponding schema, the validator promptly identifies and reports this discrepancy without causing disruptions. This is necessary because a lot of messages are fuzzed and we use this module just to check wheter the message is fuzzed or not.

The *Message Validator* serves as an essential checkpoint within the OCPPStorm fuzzer. As shown in 2.2, various components of the fuzzer rely on the *Message Validator* to ensure protocol compliance. Every time an OCPP message is generated, or a response is returned from the server, it undergoes validation by the Message Validator. This ensures that each message adheres to the established protocol schemas, maintaining the integrity and correctness of the communication process. As observed in the system architecture, the Message Validator's centrality is evident, interfacing with multiple components like the *Random Fuzzer*, *State Machine Fuzzer*, *Isla Fuzzer*, the *OCPP WebSocket Client* and the *Error Detector.*

2.2.3 Output

OCPPStorm meticulously archives its output in a series of text files, each serving a distinct purpose in documenting the fuzzing process. These files capture detailed information on the nature of the tests conducted, the responses received, and any anomalies or noteworthy occurrences. Below is a summary of each output file and the insights it provides:

1. **correct_messages_logs.txt:** Records all messages considered correct by the fuzzer, including valid user-defined messages and status notification transitions. Useful for investigating incorrect server rejections.
2. **fuzzed_messages_without_errors.txt:** Logs fuzzed messages that unexpectedly do not result in server errors, indicating potential validation gaps.
3. **response_not_valid_with_protocol_log_file.txt:** Details responses that, while technically valid, do not comply with the expected OCPP JSON schema.
4. **statistics_file.txt:** Compiles general metrics about the fuzzing process, such as total iterations, correct and incorrect responses, and various types of errors detected.
5. **stats_for_error_status.txt:** Keeps track of the specific types of errors encountered during fuzzing, with a counter for each error type.
6. **stats_status_notification.txt:** Tracks the status transitions made during the fuzzing process, particularly focusing on the **StatusNotification** OCPP message.
7. **stats_time.txt:** Records the start and end timestamps of the fuzzing process, providing data on the duration and efficiency of the testing.
8. **valid_requests_causing_errors_log_file.txt:** Lists valid requests that, contrary to expectations, trigger server errors, suggesting issues in the server's handling.
9. **wrong_errors_received_log_file.txt:** Contains instances where the server's error responses differ from those predicted by the fuzzer, highlighting discrepancies in error handling.

Each of these files plays a crucial role in analyzing the performance of OCPPStorm and the robustness of the OCPP implementations under test. By providing detailed and segmented data, these logs enable a comprehensive assessment of the system's behavior in response to a wide array of fuzzed inputs.

Chapter 3

Implementation

In the previous "Approach" chapter, we discussed each component of OCPPStorm from a theoretical perspective. In this chapter, we will delve deeper, providing more technical details on how OCPPStorm has been implemented.

3.1 Acquiring the JSON Schema

3.1.1 Source of the Schema

The JSON schemas for each type of message are defined by OCPP 1.6[1]. We sourced these schemas directly from the official documentation provided by the Open Charge Alliance.

3.1.2 Processing and Storing the Schema

To facilitate easy access and integration within OCPPStorm, these acquired schemas have been organized and stored in the "schemas" folder of our fuzzer. This centralized storage approach was adopted as these schemas are extensively referenced by multiple components, notably for the validation of messages by the *Message Validator* component.

3.2 Fuzzing Implementations

3.2.1 Random Fuzzer

Design and Architecture

The Random Fuzzer adopts a straightforward approach towards fuzzing by emphasizing simplicity and volume. Its architecture is designed around the following core elements:

- **Initialization**: It sets up necessary prerequisites, such as logs, to ensure smooth execution and record-keeping. The fuzzer logs the start and end times of each fuzzing session for temporal tracking.
- **Random Message Generation**: At its heart, the *Random Fuzzer* continuously creates messages using the *MessageGenerator* class. The messages are generated in a random fashion, making sure to cover a diverse range of potential inputs.
- **Schema Validation**: Each generated message undergoes validation against its corresponding JSON schema. The schemas, sourced from the official Open Charge Alliance documentation, are housed within the 'schemas' directory of the fuzzer.
- **Message Dispatch**: Depending on the validation outcome, the message is labeled and dispatched for further processing. Valid messages are identified and sent as such, while the others are flagged for fuzzing.
- **Completion**: The fuzzer concludes its session by updating the logs with an end timestamp.

In essence, the architecture of the Random Fuzzer is streamlined to churn out a vast array of messages, validate them, and dispatch them efficiently, all while keeping track of its operations.

3.2.2 State Machine Fuzzer

Design and Architecture

The state machine fuzzer is designed to generate various types of OCPP (Open Charge Point Protocol) messages. A set of predefined valid message types exists, and the fuzzer can generate both valid and random messages to test the system.

Algorithm and Operation

Main Fuzzing Loop Explanation The main fuzzing loop is designed to test a system by sending a mixture of both fuzzed (or manipulated) and correct messages. Here is a detailed breakdown of each step:

1. **Begin recording start time:**
    - The start time is captured to measure the duration of the entire fuzzing process. This is used for performance analysis and to determine the efficiency of the fuzzing process.
2. **Read a list of correct messages from a file:**
    - A file containing predefined correct OCPP messages is loaded. These messages act as a reference during the fuzzing process.
    - The correct messages ensure that the system can still process valid inputs amidst random or fuzzed messages.
3. **Loop until all correct messages are sent:**
    - **Produce and send fuzzed messages:** For a predetermined number, fuzzed or altered messages are generated. Occasionally, a completely random OCPP message might be produced.
    - **Validate the generated message:** Each fuzzed message is validated against known message types and schemas. Valid messages adhere to these schemas even if they're fuzzed versions.
    - **Send one correct message:** After a batch of fuzzed messages, a valid message from the list is sent as a sanity check and to move to the next state.
4. **Record the end time:**
    - The fuzzing process concludes by recording the end time. This, combined with the start time, gives the duration of the fuzzing operation, aiding in performance analysis.

In summary, the main fuzzing loop is a stress-test for the system, combining fuzzed messages with valid ones to identify potential system vulnerabilities.

State Transition Fuzzing Explanation State Transition Fuzzing is a technique that focuses on testing the system's behavior when transitioning between different states. It aims to identify potential vulnerabilities or incorrect behaviors associated with state changes. The process is outlined as follows:

1. **Initialize the current state to "Available":**
   - This step sets the starting state of the system to "Available," ensuring a consistent starting point for the fuzzing process.
2. **Calculate the total number of possible state transitions:**
   - A comprehensive list of potential state transitions is generated to map out the entire state space that needs to be tested.
3. **Iterate through state transitions:**
   - **Produce and send fuzzed messages:** Altered messages are generated and sent to potentially trigger unexpected state transitions.
   - **Determine possible next states:** Based on the current state, possible state transitions are identified.
   - **Handle visited state transitions:**
     - If all transitions from the current state are known, a random next state is selected and set as the current state.
     - Otherwise, an unvisited transition is chosen randomly as the next state.
   - **Mark the transition:** After deciding the next state, the transition is marked as visited to avoid redundancy.
   - **Generate a notification message:** A message indicating the state transition is generated to communicate the change.
   - **Send the notification:** The previously generated message is sent, signaling the state transition.
   - **Update the current state:** The system's current state is updated to reflect the recent state transition.
4. **Record the end time:**
   - After all state transitions have been tested, the end time is recorded. This aids in evaluating the duration and efficiency of the state transition fuzzing process.

In essence, State Transition Fuzzing rigorously tests the system's behavior during state changes, emphasizing the discovery of potential vulnerabilities or misbehaviors during these transitions.

3.2.3 Isla Fuzzer

Design and Architecture

The Isla Fuzzer is designed as a critical component of the OCPPStorm tool, focusing on grammar-based fuzzing with integrated constraints to effectively test OCPP implementations. This component is architected to generate a wide range of messages based on pre-defined grammars that represent the valid structural patterns of OCPP messages. The Isla Fuzzer's architecture is modular, facilitating the addition or modification of grammars and constraints to accommodate updates in OCPP standards.

Algorithm and Operation

The Isla Fuzzer operates on a sophisticated algorithm that randomly selects a message type and its corresponding grammar, applying constraints to generate test messages. The operation begins with the selection of a message type from a predefined list, followed by the retrieval of its grammar and associated constraints. It employs a solver that incorporates the grammar and constraints to produce a message that is structurally correct but potentially anomalous, challenging the robustness of the OCPP implementation.

Grammar and Constraints in Isla

Grammar Creation and Usage The grammars in Isla are meticulously crafted using a dictionary-based structure, where each message type is associated with a specific pattern that accurately delineates its valid format. These grammars serve as blueprints for message generation, ensuring that all crafted messages are syntactically aligned with the structural norms of the OCPP protocol.

To elucidate the concept of grammar-based message generation, consider the simplest grammar defined in the Isla Fuzzer for an "Authorize" message. This grammar is represented as follows:

```
AUTHORIZE: Grammar = {
    "<start>": ["<authorize>"],
    "<authorize>": ["{<idTag>}"],
    "<idTag>": ["\"idTag\": <data>"],
    "<data>": ["\"<string>\""],  # maxLength = 20
    "<string>": ["<char><string>", ""],
    "<char>": (return_list_of_printable_char())
}
```

The above grammar is a foundational example used to demonstrate the mechanism of grammar creation. It defines a recursive pattern for constructing an **Authorize** message, starting with a **start** symbol that unfolds into an **authorize** structure, which further decomposes into an **idTag** and its associated **data**. The **data** is then a composition of **string** elements, recursively defined to be a sequence of **char** elements or an empty string, adhering to the maximum length constraint.

The **return_list_of_printable_char()** function is implied to provide a list of printable characters, which are used to build the **string** values. This approach enables the Isla Fuzzer to not only generate valid message formats but also to explore the robustness of the OCPP implementation by testing how it handles strings of varying lengths and compositions.

A potential output generated by this grammar, respecting the maximum length constraint of 20 characters for the **data** field, could look like the following JSON object:

```
{
    "idTag": "B1a2C3d4E5f6G7h8I9j"
}
```

This output represents a well-formed OCPP "Authorize" message with a 20-character ‘idTag‘. The Isla Fuzzer would generate such messages to verify that the OCPP implementation correctly processes valid inputs and to test its behavior with inputs that push the boundaries of the protocol's specifications.

Integrating Constraints Constraints are a pivotal aspect of the message generation process within the Isla Fuzzer, serving to refine and direct the fuzzing approach effectively. These constraints are explicitly defined for each message type and can impose various limitations, such as the length of a string or the range of permissible values for a given field. The integration of constraints ensures that, while the generated messages adhere to the grammatical structure prescribed by OCPP, the content within them is manipulated to test the limits of the implementation's handling capabilities.

For instance, the **Authorize** message type may have an associated constraint that specifies the length of the ‘data‘ field must exceed 22 characters, as shown below:

```
"str.len(<data>)>22"
```

This constraint is applied during the generation process to produce messages that deliberately violate the protocol's maximum length requirement for the **data** field, which is 20. By doing so, the Isla Fuzzer simulates scenarios that could potentially arise from malformed or malicious inputs, assessing the OCPP implementation's resilience to such irregularities.

An example of a message generated under this constraint for the ‘Authorize‘ type might be:

```
{
    "idTag": "Z1x2C3v4B5n6M7a8S9dQ0wE"
}
```

In this case, the **idTag** field contains 23 characters, which breaches the stipulated maximum length. The Isla Fuzzer would utilize this message to evaluate whether the OCPP implementation properly rejects this input, logs an error, or exhibits any unintended behavior, thereby revealing its robustness and compliance with the OCPP specification.

3.3 Extensibility

OCPPStorm's design philosophy emphasizes modularity and extensibility across all its components, ensuring that the tool remains relevant and effective as the Open Charge Point Protocol (OCPP) evolves. The modular nature of OCPPStorm's components allows for straightforward extensions and adaptations, catering to new OCPP versions and message specifications. Key components that exhibit this extensible architecture include:

- **Isla Message Generator:** The core functionality of the Isla Message Generator can be expanded by incorporating grammars corresponding to new OCPP message types or by adapting to the specifications of OCPP 2.0, for example. This enhancement would allow the fuzzer to cover a broader spectrum of the protocol's communication patterns.
- **Message Generator:** The Message Generator's capability is not confined to OCPP messages. It accepts any JSON schema as input, generating fuzzed messages that conform to the provided schema. This versatility ensures that OCPPStorm can be utilized for a diverse set of applications beyond OCPP, demonstrating its utility across various domains that utilize JSON-based communication.
- **Error Detector:** The Error Detector component has undergone several refinements and remains open to further enhancements. It can be extended to improve error prediction algorithms or to recognize new error patterns that may emerge from updates in the OCPP specifications.
- **State Machine Fuzzer:** Featuring a dual-state machine architecture with one user-defined and one protocol-defined state machine, the State Machine Fuzzer is designed for extensibility. As OCPP evolves, the state machines can be modified to align with new protocol behaviors and state transitions, ensuring that OCPPStorm remains an accurate testing tool for future protocol versions.

The extensibility of OCPPStorm is integral to its design, ensuring that the tool not only addresses the current needs of OCPP security testing but is also poised to meet future challenges. This foresight in design allows OCPPStorm to serve as a lasting resource in the continuous endeavor to secure EV charging infrastructure.

3.4 Code Metrics

3.4.1 Size and Structure

An essential aspect of understanding the complexity and maintainability of a software system is analyzing its code metrics, particularly the size and structure of its codebase. In the case of OCPPStorm, the codebase is distributed across various modules, each serving a specific role in the fuzzing process. Below is a summary of the size, in terms of lines of code (LOC), for each major component of OCPPStorm:

Table 3.1: OCPPSTORM CODEBASE SIZE

Component	Lines of Code
FuzzSchema.py	286
MessageGenerator.py	48
IslaMessageGenerator.py	214
ErrorDetector.py	325
message_validator.py	20
json_grammars.py	552
random_fuzzer.py	36
state_machine_fuzzer.py	225
isla_fuzzer.py	77
OcppWebsocketClient.py	75
main.py	57
Total	1915

The table delineates the LOC for individual files, reflecting the modular design of the tool. The largest file, ‘json_grammars.py‘, is indicative of the extensive range of grammars used for fuzzing OCPP messages. Conversely, the main entry point of the application, ‘main.py‘, is concise, denoting a well-structured codebase where functionalities are encapsulated within specific modules. This modularity facilitates ease of maintenance and the potential for future enhancements.

3.4.2 Languages and Libraries Used

OCPPStorm is developed entirely in Python, a versatile programming language well-suited for rapid development and prototyping. Python’s extensive standard library and the rich ecosystem of third-party packages have been instrumental in the implementation of OCPPStorm. Below is an overview of the key libraries and their roles within the tool:

- **json & jsonschema:** Central to the tool's operation, these libraries are employed for working with JSON formatted messages. The 'jsonschema' library is used to validate the structure of JSON messages against predefined schemas, ensuring their adherence to the OCPP protocol.
- **os:** This standard library module provides a portable way of using operating system-dependent functionality like reading or writing to the filesystem.
- **rfc3339_validator:** This library is used to validate date-time strings in the RFC 3339 format, ensuring that timestamps in messages conform to this standard.
- **datetime:** Part of Python's standard library, it is used to handle and manipulate date and time data, essential for timestamping.
- **Enum & random:** These modules are utilized to define enumeration classes and generate random selections, respectively, aiding in the stochastic nature of the fuzzing process.
- **websocket:** This library provides the capabilities for the OCPPStorm to establish and manage WebSocket connections, a key component for real-time communication with central system s.
- **Isla & fuzzingbook.Grammars:** The Isla library, along with components from fuzzingbook.Grammars, is utilized for grammar-based fuzzing, aiding in the creation and manipulation of the syntactic structure of test messages.

The combination of these libraries with Python's expressive syntax and powerful features has enabled the creation of a robust and efficient tool for fuzzing OCPP implementations.

Chapter 4

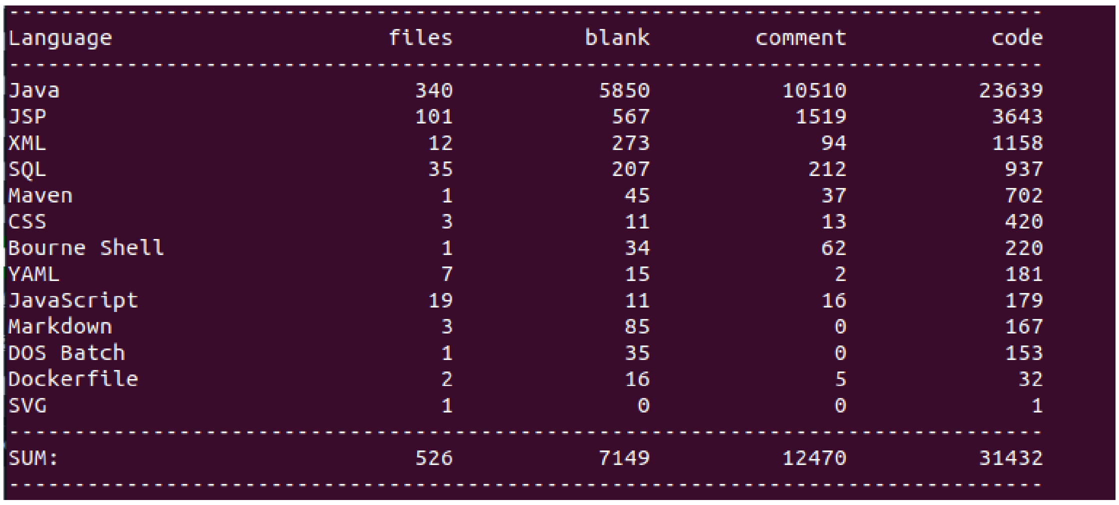
Evaluation

4.1 Setup

The evaluation of OCPPStorm was conducted against two distinct OCPP server implementations: Steve[6], an established open-source platform, and OCPP.Core[7], a .NET-based implementation. The evaluation aimed to measure the efficacy of OCPPStorm in identifying potential errors, security vulnerabilities, and to assess its performance and scalability.

4.1.1 Steve Implementation

Steve is an open-source OCPP server implementation written in Java. The following table shows the result of a line count analysis performed using the *cloc* tool on a Linux environment, providing insights into the size and composition of the codebase.



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           340           5850          10510          23639
JSP                            101            567           1519           3643
XML                             12            273             94           1158
SQL                             35            207            212            937
Maven                            1             45             37            702
CSS                              3             11             13            420
Bourne Shell                     1             34             62            220
YAML                             7             15              2            181
JavaScript                      19             11             16            179
Markdown                         3             85              0            167
DOS Batch                        1             35              0            153
Dockerfile                       2             16              5             32
SVG                              1              0              0              1
-------------------------------------------------------------------------------
SUM:                           526           7149          12470          31432
-------------------------------------------------------------------------------
```

Figure 4.1: Line count statistics for the Steve implementation as analyzed by the *cloc* tool

As indicated in 4.1, the Steve implementation comprises 526 files, with a significant portion written in Java. The analysis details the number of blank lines, comments, and lines of actual code, offering a comprehensive view of the code's structure.

Size and Structure:

- **Total Files**: 526
- **Total Lines of Code (LOC)**: 31432
- **Languages Used**: Primarily Java, with additional scripting and configuration languages such as JSP, XML, and SQL.

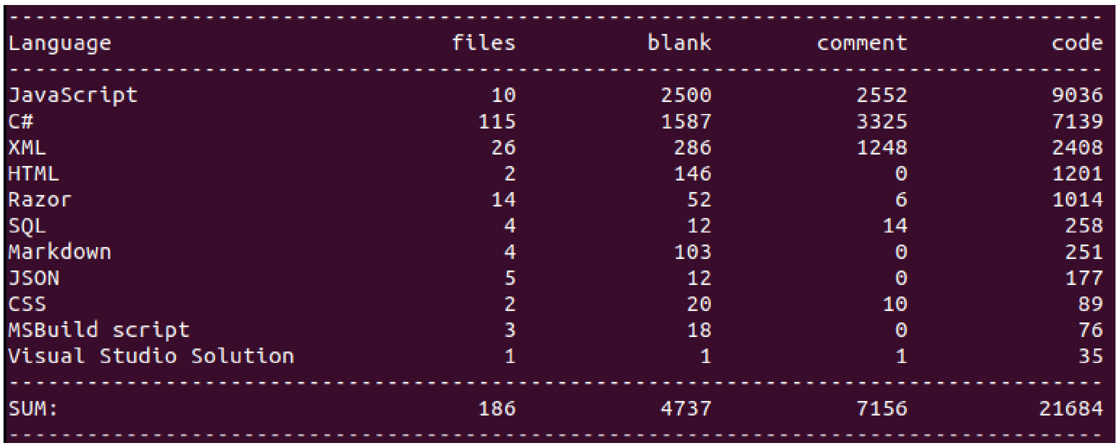
Comments and Documentation: The high number of comment lines suggests a well-documented codebase, which is beneficial for maintainability and further development.

Environment: For the purpose of controlled testing, the Steve server was installed on a local machine. This provided a stable environment, which is crucial for the accurate assessment of OCPPStorm's capabilities.

Database State: The database backing the Steve server was initialized with a set of valid states to simulate an operational charging network environment, allowing for tests that cover normal operational conditions as well as error handling and exceptional cases.

4.1.2 OCPP.Core Implementation

OCPP.Core is a .NET-based OCPP server implementation. Displayed below are the results from a line count analysis using the *cloc* tool, which gives a detailed breakdown of the codebase.



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
JavaScript                      10           2500           2552           9036
C#                             115           1587           3325           7139
XML                             26            286           1248           2408
HTML                             2            146              0           1201
Razor                           14             52              6           1014
SQL                              4             12             14            258
Markdown                         4            103              0            251
JSON                             5             12              0            177
CSS                              2             20             10             89
MSBuild script                   3             18              0             76
Visual Studio Solution           1              1              1             35
-------------------------------------------------------------------------------
SUM:                           186           4737           7156          21684
-------------------------------------------------------------------------------
```

Figure 4.2: Line count statistics for the OCPP.Core implementation as analyzed by the *cloc* tool

4.2 provides a snapshot of the OCPP.Core implementation's composition, encompassing 186 files and reflecting a diverse usage of languages within the .NET ecosystem.

Size and Structure:

- **Total Files**: 186
- **Total Lines of Code (LOC)**: 21684
- **Languages Used**: A mixture of C#, JavaScript, XML, and other languages for web development and database scripting.

Comments and Documentation: The analysis indicates a comprehensive documentation practice, as evidenced by a substantial number of comment lines, which underscores the implementation's maintainability.

Environment: Similar to the Steve implementation, OCPP.Core was deployed on a local machine. This ensured that the testing environment was consistent and that the results could be directly attributed to the differences in the server implementations rather than the underlying infrastructure.

Database State: The OCPP.Core server was paired with a database seeded with legal values, ensuring that the fuzzer's interactions could be as close to real-world operations as possible.

4.2 Evaluation results

The following section presents a comprehensive assessment of the performance and efficacy of the OCPPStorm tool when applied to two distinct OCPP implementations: Steve and OCPP.Core. The evaluation was structured to meticulously document the behavior of each fuzzer component under test conditions, with a focus on a variety of performance metrics that include the rate and accuracy of message processing, error detection capabilities, and overall system efficiency.

The evaluation hinged on a series of metrics, each designed to probe a different aspect of the fuzzing process. The metrics encompass the throughput of messages, gauged as Messages per Second (MPS), the Total Execution Time of the fuzzing session, the Average Time per Message, which provides insights into the efficiency of the system, and a suite of error-related statistics that illustrate the tool's precision in identifying and cataloging system vulnerabilities.

4.2.1 OCPPStorm Evaluation Metrics

In our comprehensive evaluation of the OCPPStorm's performance, a series of extensive fuzzing techniques were applied to scrutinize all possible properties of the OCPP messages. This endeavor aimed to cover a broad spectrum of test scenarios, ranging from schema violations and exceeding length constraints to the processing of unexpected input types. Due to the iterative nature of fuzz testing, some test messages were generated multiple times, leading to the same error being identified by different inputs or the repetition of identical inputs.

Before presenting the tables with detailed results, it is essential to understand the specific metrics used to assess the outcomes of our fuzzing efforts:

1. **Total Iterations:** The complete count of fuzzing tests executed.
2. **Correct Responses Rate:** The ratio of responses that were accurate, aligning with expectations for both valid and invalid inputs. Specifically, we consider a response as correct in two cases: 1) if the fuzzer sends an invalid message and we receive an error response, matching the type of error predicted by the fuzzer, 2) if the fuzzer sends a valid message and we receive a valid response. Ideally, this value should be as close to 100% as possible. This rate therefore reflects the system's accuracy in responding

correctly according to the expected outcome for each specific input. An example of correct message that falls into this category is:

```
[2,"1","Authorize","idTag":"abcde12345"]
```

which will receive this response:

```
[3,"1",{"idTagInfo":{"expiryDate":"2024-01-24T00:00:00-06:00"
,"status":"Accepted"}}]
```

An example of uncorrect message (*idTag* is not a string) that trigger the correct error and so falls into this ratio is:

```
[2,"1","Authorize","idTag": 123]
```

which will receive the predicted error (*TypeConstraintViolation*), here is a possible response:

```
[4,"1","TypeConstraintViolation","idTag longer than 20",{}]
```

3. **Uncorrect Response Rate:** This metric is the opposite of the Correct Response Rate. It is the ratio of responses that were different from the expected ones. This is composed of the following subcategories:

   - **Wrong Error Rate:** The ratio of errors differing from those anticipated by the fuzzer, suggesting potential issues in error handling. An example of message that cause the wrong error is:

     ```
     [2,"1","Authorize","idTag":123]
     ```

     which should trigger a *TypeConstraintViolation* error. If we receive another type of error, this ratio becomes higher. An example of wrong error for this message is:

     ```
     [4,"1","InternalError","",{}]
     ```

   - **Valid Requests Causing Error Rate:** The ratio of instances in which valid inputs erroneously triggered errors, indicating improper processing by the system. An example of message that falls into this category is:

     ```
     [2, ’15505’, ’StatusNotification’, {’connectorId’: 6,
     ’errorCode’: ’GroundFailure’, ’status’: ’Finishing’}]
     ```

The *StatusNotification* message comply with the constraints defined by its json schema and it should be accepted by the server. If it's not accepted and the server sends an error, this ratio becomes higher. An example of response is:

```
[4,"15505","InternalError","",{}]
```

- **Non-error-Causing Fuzz Rate:** The fraction of invalid messages that did not cause an error but ideally should have, indicating possible validation gaps. An example of message that falls into this category is:

  ```
  [2, '84', 'MeterValues', {'connectorId': 7}]
  ```

  the json schema of the *MeterValues* message has two required properties: *connectorId* and *meterValue*. The message in the example does not contain the *meterValue* property and so is flagged as fuzzed and should trigger an error. If we do not receive an error from the server this ratio will be higher. An example of response is:

  ```
  [3,"84",{}]
  ```

- **Responses Not Valid with Protocol:** The frequency of responses that did not adhere to the OCPP protocol's constraints. In particular the response is validated against its json schema and if it does not comply with it, the message falls into this category. An example of message for this ratio is:

  ```
  [2, '1', 'Authorize', {}]
  ```

  This message is not valid because the *idTag* property is required and here is missing. The server sends this response:

  ```
  [3,"1",{}]
  ```

  which does not comply with the json schema for the *AutorizeResponse* message, that must contain at least the *idTagInfo* property.

4. **Comprehensive Handling Rate:** This ratio evaluates the system's overall ability to correctly process inputs, but with a broader scope than the Correct Responses Rate. It includes all well-formed messages that receive an appropriate response and all uncorrect messages that successfully trigger any error response, regardless of the specific error type. This rate is indicative of the system's general resilience and effectiveness in handling a wide range of inputs, reflecting its capacity to appropriately respond under varied circumstances, without being restricted to the accuracy of error types. An example of message that falls into this category is:

   ```
   [2,"1","Authorize","idTag":123]
   ```

the predicted error for this message is *TypeConstraintViolation* but here we are just looking for an error without checking if it's the predicted one. Examples of possible responses for this category are:

```
[4,"1","TypeContraintViolation","idTag longer than 20",{}]
```

```
[4,"1","InternalError","",{}]
```

```
[4,"1","GenericError","",{}]
```

Following these metrics, the subsequent tables detail the performance metrics for each fuzzing approach and OCPP implementation. They highlight the system's proficiency in parsing and evaluating protocol communication, illuminating areas for potential improvement.

The ensuing tables, therefore, offer a holistic view of our testing process, reflecting both the meticulousness of the fuzzing techniques employed and the resilience of each OCPP implementation against the generated test cases.

4.1 (Random Fuzzer against Steve): This table highlights a Correct Responses Rate of approximately 22.44%, indicating that only a fraction of the responses from Steve matched the expected outcomes for both valid and invalid inputs. The Comprehensive Handling Rate, at 57.31%, suggests that while Steve can handle a broader array of inputs, there's still room for improvement, particularly in responding accurately to fuzzed inputs. The Uncorrect Response Rate reveals that a significant portion of responses deviated from the expected outcomes, including instances of wrong error responses and non-error-causing fuzzed messages. These findings underline areas in Steve's implementation that could benefit from further refinement for enhanced protocol adherence and error handling.

Table 4.1: EVALUATION METRICS FOR OCPPSTORM USING THE RANDOM FUZZER AGAINST STEVE

Metric	**Value**
Total Iterations	500,000
Correct Responses Rate	$\frac{112212}{500000} \approx 0.2244$ or 22.44%
Uncorrect Response Rate	$\frac{387788}{500000} \approx 0.7756$ or 77.56%
Wrong Error Rate	$\frac{255544}{500000} \approx 0.5111$ or 51.11%
Valid Requests Causing Error Rate	0 or 0%
Non-error-Causing Fuzz Rate	$\frac{104098}{500000} \approx 0.2082$ or 20.82%
Responses Not Valid with Protocol	$\frac{28146}{500000} \approx 0.0563$ or 5.63%
Comprehensive Handling Rate	$\frac{286560}{500000} \approx 0.5731$ or 57.31%

4.2 (Random Fuzzer against OCPP.Core): The data shows a Correct Responses Rate of 22.04%, indicating that about a fifth of the responses were as expected. The Comprehensive Handling Rate is notably higher at 86.37%, suggesting that while the system frequently responds correctly to both valid and invalid inputs, it often categorizes errors incorrectly, as reflected in the high Wrong Error Rate of 75.20%. This points towards a need for better error categorization. The low Non-error Causing Fuzz Rate of 2.75% indicates effective detection of malformed inputs.

Table 4.2: EVALUATION METRICS FOR OCPPSTORM USING THE RANDOM FUZZER AGAINST OCPP.CORE

Metric	**Value**
Total Iterations	500,000
Correct Responses Rate	$\frac{110209}{500000} \approx 0.2204$ or 22.04%
Uncorrect Response Rate	$\frac{389791}{500000} \approx 0.7796$ or 77.96%
Wrong Error Rate	$\frac{376025}{500000} \approx 0.7520$ or 75.20%
Valid Requests Causing Error Rate	0 or 0%
Non-error Causing Fuzz Rate	$\frac{13766}{500000} \approx 0.0275$ or 2.75%
Responses Not Valid with Protocol	0 or 0%
Comprehensive Handling Rate	$\frac{431870}{500000} \approx 0.8637$ or 86.37%

4.3 (Isla Fuzzer without constraints against Steve): The table indicates a fairly balanced Correct Responses Rate of 57.80%, suggesting that over half of the responses aligned with expectations. The Comprehensive Handling Rate stands at 58.79%, pointing to a relatively consistent handling of both correct and incorrect messages. However, the Uncorrect Response Rate at 42.19% and Wrong Error Rate at 23.18% highlight areas for improvement, particularly in error handling and response accuracy.

Table 4.3: EVALUATION METRICS FOR OCPPSTORM USING THE ISLA FUZZER WITHOUT CONSTRAINTS AGAINST STEVE

Metric	**Value**
Total Iterations	100,000
Correct Responses Rate	$\frac{57804}{100000} \approx 0.5780$ or 57.80%
Uncorrect Response Rate	$\frac{42196}{100000} \approx 0.4219$ or 42.19%
Wrong Error Rate	$\frac{23185}{100000} \approx 0.2318$ or 23.18%
Valid Requests Causing Error Rate	0 or 0%
Non-error Causing Fuzz Rate	$\frac{13256}{100000} \approx 0.1326$ or 13.26%
Responses Not Valid with Protocol	$\frac{5755}{100000} \approx 0.0575$ or 5.75%
Comprehensive Handling Rate	$\frac{58792}{100000} \approx 0.5879$ or 58.79%

4.4 (Isla Fuzzer without constraints against OCPP.Core): This table shows a Correct Responses Rate of 42.07%, indicating less than half of the responses were as expected. The

Comprehensive Handling Rate is higher at 61.99%, reflecting a reasonable degree of system resilience in processing both correct and uncorrect inputs. However, the high Uncorrect Response Rate of 57.93% and Wrong Error Rate of 36.58% highlight significant challenges in error handling and protocol adherence, necessitating further refinement.

Table 4.4: EVALUATION METRICS FOR OCPPSTORM USING THE ISLA FUZZER WITHOUT CONSTRAINTS AGAINST OCPP.CORE

Metric	**Value**
Total Iterations	100,000
Correct Responses Rate	$\frac{42069}{100000} \approx 0.4207$ or 42.07%
Uncorrect Response Rate	$\frac{57931}{100000} \approx 0.5793$ or 57.93%
Wrong Error Rate	$\frac{36577}{100000} \approx 0.3658$ or 36.58%
Valid Requests Causing Error Rate	$\frac{19134}{100000} \approx 0.1913$ or 19.13%
Non-error Causing Fuzz Rate	$\frac{2220}{100000} \approx 0.0222$ or 2.22%
Responses Not Valid with Protocol	0 or 0%
Comprehensive Handling Rate	$\frac{61990}{100000} \approx 0.6199$ or 61.99%

4.5 (Isla Fuzzer with constraints against Steve): This table reveals a relatively low Correct Responses Rate of 31.35%, suggesting significant challenges in Steve's ability to handle both valid and fuzzed inputs effectively. The high Uncorrect Response Rate at 68.65%, primarily driven by a Wrong Error Rate of 49.63%, indicates notable discrepancies in error handling. The Comprehensive Handling Rate of 69.63% implies a moderate level of system robustness, yet the underlying issues in error categorization and protocol adherence are evident and require attention.

Table 4.5: EVALUATION METRICS FOR OCPPSTORM USING THE ISLA FUZZER WITH CONSTRAINTS AGAINST STEVE

Metric	**Value**
Total Iterations	19,950
Correct Responses Rate	$\frac{6254}{19950} \approx 0.3135$ or 31.35%
Uncorrect Response Rate	$\frac{13696}{19950} \approx 0.6865$ or 68.65%
Wrong Error Rate	$\frac{9902}{19950} \approx 0.4963$ or 49.63%
Valid Requests Causing Error Rate	$\frac{37}{19950} \approx 0.0018$ or 0.18%
Non-error Causing Fuzz Rate	$\frac{3757}{19950} \approx 0.1883$ or 18.83%
Responses Not Valid with Protocol	0 or 0%
Comprehensive Handling Rate	$\frac{13892}{19950} \approx 0.6963$ or 69.63%

4.6 (Isla Fuzzer with constraints against OCPP.Core): Exhibits a Correct Responses Rate of 33.67%, indicating challenges in accurately processing inputs according to protocol specifications. The substantial Uncorrect Response Rate of 66.32%, largely due to a high Wrong Error Rate of 60.57%, suggests issues in error classification and handling. Notably, the Comprehensive Handling Rate is relatively high at 82.90%, suggesting that while the system is generally effective in processing a range of inputs, it struggles with accurate error categorization and adherence to protocol in certain instances.

Table 4.6: EVALUATION METRICS FOR OCPPSTORM USING THE ISLA FUZZER WITH CONSTRAINTS AGAINST OCPP.CORE

Metric	**Value**
Total Iterations	19,950
Correct Responses Rate	$\frac{6718}{19950} \approx 0.3367$ or 33.67%
Uncorrect Response Rate	$\frac{13232}{19950} \approx 0.6632$ or 66.32%
Wrong Error Rate	$\frac{12085}{19950} \approx 0.6057$ or 60.57%
Valid Requests Causing Error Rate	$\frac{196}{19950} \approx 0.0098$ or 0.98%
Non-error Causing Fuzz Rate	$\frac{951}{19950} \approx 0.0477$ or 4.77%
Responses Not Valid with Protocol	0 or 0%
Comprehensive Handling Rate	$\frac{16539}{19950} \approx 0.8290$ or 82.90%

4.7 (State Machine Fuzzer against Steve): This table reveals a Correct Responses Rate of only 22.74%, highlighting significant issues in adhering to the protocol and accurately processing inputs. The Uncorrect Response Rate is notably high at 77.24%, with a predominant Wrong Error Rate of 50.99%, indicating major challenges in error handling and response categorization. The Comprehensive Handling Rate, at 57.08%, suggests a moderate level of effectiveness in managing inputs, but also reflects substantial room for improvement in ensuring protocol compliance and error accuracy.

Table 4.7: EVALUATION METRICS FOR OCPPSTORM USING THE STATE MACHINE FUZZER AGAINST STEVE

Metric	**Value**
Total Iterations	1,114,800
Correct Responses Rate	$\frac{253558}{1114800} \approx 0.2274$ or 22.74%
Uncorrect Response Rate	$\frac{861182}{1114800} \approx 0.7724$ or 77.24%
Wrong Error Rate	$\frac{568591}{1114800} \approx 0.5099$ or 50.99%
Valid Requests Causing Error Rate	0 or 0%
Non-error Causing Fuzz Rate	$\frac{230361}{1114800} \approx 0.2066$ or 20.66%
Responses Not Valid with Protocol	$\frac{62229}{1114800} \approx 0.0558$ or 5.58%
Comprehensive Handling Rate	$\frac{636325}{1114800} \approx 0.5708$ or 57.08%

4.8 (State Machine Fuzzer against OCPP.Core): The Correct Responses Rate for OCPP.Core stands at 21.86%, pointing towards substantial issues in protocol adherence and response accuracy. The high Uncorrect Response Rate of 78.14% is predominantly due to a Wrong Error Rate of 75.41%, suggesting significant inaccuracies in error handling. However, the Comprehensive Handling Rate of 86.48% suggests that while there are issues with error categorization, the system shows a high level of resilience in handling a variety of inputs, indicating its overall robustness despite the noted deficiencies.

Table 4.8: EVALUATION METRICS FOR OCPPSTORM USING THE STATE MACHINE FUZZER AGAINST OCPP.CORE

Metric	**Value**
Total Iterations	1,114,800
Correct Responses Rate	$\frac{243672}{1114800} \approx 0.2186$ or 21.86%
Uncorrect Response Rate	$\frac{871068}{1114800} \approx 0.7814$ or 78.14%
Wrong Error Rate	$\frac{840691}{1114800} \approx 0.7541$ or 75.41%
Valid Requests Causing Error Rate	0 or 0%
Non-error Causing Fuzz Rate	$\frac{30377}{1114800} \approx 0.0272$ or 2.72%
Responses Not Valid with Protocol	0 or 0%
Comprehensive Handling Rate	$\frac{964041}{1114800} \approx 0.8648$ or 86.48%

4.2.2 Constraint Violation Analysis

The comprehensive analysis of constraint violations forms the cornerstone of our evaluation methodology. This segment aims to meticulously examine the system's adherence to the specified constraints under various testing conditions. Presented in the form of an extensive longtable, this analysis focuses on critical aspects such as Required Field Omission, Length Constraint Breach, and Data Type Discrepancy, which are pivotal in assessing the system's capacity to handle protocol deviations effectively.

Each of these aspects is carefully examined to gauge the system's compliance with the predefined protocol specifications and its resilience to anomalous input scenarios.

Table 4.9: CONSTRAINT VIOLATION RESULTS FOR STEVE

MessageType.Property	**Required Field Omitted**	**Length Constraint Breached**	**Data Type Discrepancy**
Authorize.idTag	X	X	X
BootNotification.chargePointVendor	X	X	X
BootNotification.chargePointModel	X	X	X
BootNotification.chargePointSerialNumber	-	X	X
BootNotification.chargeBoxSerialNumber	-	X	X
BootNotification.firmwareVersion	-	X	X
BootNotification.iccid	-	X	X
BootNotification.imsi	-	X	X
BootNotification.meterType	-	X	X
BootNotification.meterSerialNumber	-	X	X
DataTransfer.vendorId	X	X	X
DataTransfer.messageId	-	X	X
DataTransfer.data	-	-	X
DiagnosticsStatusNotification.status	X	-	X
FirmwareStatusNotification.status	X	-	X
MeterValues.connectorId	X	-	X
MeterValues.transactionId	-	-	X
MeterValues.meterValue[*].timestamp	X	-	X
MeterValues.meterValue[*].sampledValue	X	-	-
MeterValues.meterValue[*].sampledValue[*].value	X	-	
MeterValues.meterValue[*].sampledValue[*].context	-	-	X
MeterValues.meterValue[*].sampledValue[*].format	-	-	X
MeterValues.meterValue[*].sampledValue[*].measurand	-	-	X
MeterValues.meterValue[*].sampledValue[*].phase	-	-	X
MeterValues.meterValue[*].sampledValue[*].location	-	-	X
MeterValues.meterValue[*].sampledValue[*].unit	-	-	X
StartTransaction.connectorId	X	-	X
StartTransaction.idTag		X	X
StartTransaction.meterStart	X	-	X
StartTransaction.reservationId	-	-	X
StartTransaction.timestamp	X	-	X
StatusNotification.connectorId	X	-	X
StatusNotification.errorCode		-	X
StatusNotification.info	-	X	
StatusNotification.status		-	
StatusNotification.timestamp		-	
StatusNotification.vendorId			
StatusNotification.vendorErrorCode		X	
StopTransaction.idTag	-	X	
StopTransaction.meterStop	X	-	X
StopTransaction.timestamp	X	-	X
StopTransaction.transactionId	X	-	X
StopTransaction.reason	-	-	X
StopTransaction.transactionData[*].timestamp	X		
StopTransaction.transactionData[*].sampledValue	X		

Continued on next page

Table 4.9 continued from previous page

MessageType.Property	**Required Field Omitted**	**Length Constraint Breached**	**Data Type Discrepancy**
StopTransaction.transactionData[*].sampledValue[*] .value	X		
StopTransaction.transactionData[*].sampledValue[*] .context			
StopTransaction.transactionData[*].sampledValue[*] .format			
StopTransaction.transactionData[*].sampledValue[*] .measurand			
StopTransaction.transactionData[*].sampledValue[*] .phase			
StopTransaction.transactionData[*].sampledValue[*] .location			
StopTransaction.transactionData[*].sampledValue[*].unit			

Table 4.10: CONSTRAINT VIOLATION RESULTS FOR OCPP.CORE

MessageType.Property	**Required Field Omitted**	**Length Constraint Breached**	**Data Type Discrepancy**
Authorize.idTag		X	X
BootNotification.chargePointVendor		X	X
BootNotification.chargePointModel		X	X
BootNotification.chargePointSerialNumber	-	X	X
BootNotification.chargeBoxSerialNumber	-	X	X
BootNotification.firmwareVersion	-	X	X
BootNotification.iccid	-	X	X
BootNotification.imsi	-	X	X
BootNotification.meterType	-	X	X
BootNotification.meterSerialNumber	-	X	X
DataTransfer.vendorId		X	
DataTransfer.messageId	-	X	
DataTransfer.data	-	-	
DiagnosticsStatusNotification.status		-	
FirmwareStatusNotification.status		X	
MeterValues.connectorId		-	
MeterValues.transactionId	-	-	
MeterValues.meterValue[*].timestamp	X	-	
MeterValues.meterValue[*].sampledValue	X	-	-
MeterValues.meterValue[*].sampledValue[*] .value	X	-	
MeterValues.meterValue[*].sampledValue[*] .context	-	-	
MeterValues.meterValue[*].sampledValue[*] .format	-	-	
MeterValues.meterValue[*].sampledValue[*] .measurand	-	-	
MeterValues.meterValue[*].sampledValue[*] .phase	-	-	
MeterValues.meterValue[*].sampledValue[*] .location	-	-	
MeterValues.meterValue[*].sampledValue[*] .unit	-	-	
StartTransaction.connectorId		-	
StartTransaction.idTag			
StartTransaction.meterStart	X	-	
StartTransaction.reservationId	-	-	
StartTransaction.timestamp		-	
StatusNotification.connectorId		-	X
StatusNotification.errorCode		-	
StatusNotification.info	-	X	
StatusNotification.status		-	
StatusNotification.timestamp		-	
StatusNotification.vendorId			
StatusNotification.vendorErrorCode		X	
StopTransaction.idTag	-		
StopTransaction.meterStop		-	
StopTransaction.timestamp		-	
StopTransaction.transactionId		-	
StopTransaction.reason	-	-	

Continued on next page

Table 4.10 continued from previous page

MessageType.Property	**Required Field Omitted**	**Length Constraint Breached**	**Data Type Discrepancy**
StopTransaction.transactionData[*].timestamp			
StopTransaction.transactionData[*] .sampledValue			
StopTransaction.transactionData[*] .sampled-Value[*].value			
StopTransaction.transactionData[*] .sampled-Value[*].context			
StopTransaction.transactionData[*] .sampled-Value[*].format			
StopTransaction.transactionData[*] .sampled-Value[*].measurand			
StopTransaction.transactionData[*] .sampled-Value[*].phase			
StopTransaction.transactionData[*] .sampled-Value[*].location			
StopTransaction.transactionData[*] .sampled-Value[*].unit			

When comparing the constraint violation results for Steve (4.9) and OCPP.Core (4.10), a clear distinction in their handling of protocol constraints emerges.

Steve's implementation shows a significant degree of leniency in adhering to protocol specifications. It often accepts inputs that do not conform to the protocol's defined constraints, such as permitting fields longer than specified, overlooking required property omissions, and ignoring type discrepancies. This pattern is observed across numerous message properties and types. While this may indicate flexibility in input handling, it simultaneously raises potential concerns regarding security and system behavior predictability when faced with non-standard or malformed inputs.

On the other side, OCPP.Core exhibits a more rigorous adherence to protocol constraints. The system demonstrates a higher level of compliance, especially regarding the handling of required fields and data types, and is more stringent in accepting inputs that exceed the expected format or length. This approach suggests a heightened attention to protocol fidelity, enhancing the system's robustness against malformed inputs and maintaining consistent behavior.

In summary, Steve and OCPP.Core each display distinct methods of protocol adherence, with varying implications. Steve's more accommodating approach could offer user convenience and flexibility but at the risk of increased security vulnerabilities. Conversely, OCPP.Core's strict adherence to protocol standards may bolster security and consistency, ensuring that only inputs that precisely match the protocol specifications are accepted.

4.2.3 Performance Metrics

The first set of tables delineates the performance metrics derived from the fuzzing sessions conducted on both the implementations. It juxtaposes the results from the Random Fuzzer, State Machine Fuzzer, and Isla Fuzzer, both with and without constraints, offering a multi-faceted view of the tool's performance across different operational modes. In particular 4.11 shows the performance metrics comparison for Steve, while 4.12 shows the performance metrics comparison for OCPP.Core.

Table 4.11: PERFORMANCE METRICS COMPARISON FOR STEVE

Performance Metric	**Random Fuzzer**	**State Machine Fuzzer**	**Isla Fuzzer(w/o constraints)**	**Isla Fuzzer(w constraints)**
Total Messages	500000	1,114,800	100,000	4950
Messages per Second (MPS)	521.4	595.5	3.99	1.36
Total Execution Time	0h 15m 59s	0h 31m 12s	6h 57m 14s	1h 0m 38s
Average Time per Message (s)	0.0019	0.0017	0.2503	0.7349

Table 4.12: PERFORMANCE METRICS COMPARISON FOR OCPP.CORE

Performance Metric	**Random Fuzzer**	**State Machine Fuzzer**	**Isla Fuzzer(w/o constraints)**	**Isla Fuzzer(w constraints)**
Total Messages	500000	1,114,800	100,000	19950
Messages per Second (MPS)	20.69	20.59	3.36	1.37
Total Execution Time	6h 42m 42s	15h 2m 33s	8h 15m 48s	4h 2m 30s
Average Time per Message (s)	0.0483	0.0486	0.2975	0.7293

- **Comparison of Fuzzers:** 4.11 and 4.12 reveal notable differences in the performance of the State Machine, Random Fuzzer, and Isla Fuzzers. Both the State Machine and Random Fuzzers process messages at a much faster rate (approximately 521.4 to 595.5 MPS for Steve and 20.59 to 20.69 MPS for OCPP.Core), while the Isla Fuzzers (with and without constraints) operate more slowly (3.99 to 1.36 MPS for Steve and 3.36 to 1.37 MPS for OCPP.Core). This indicates that the isla library is much slower in generating messages compared to the other fuzzers.
- **Steve vs. OCPP.Core:** When comparing the two implementations, Steve generally exhibits a higher message processing speed across all fuzzers, indicating a more efficient handling of incoming messages. However, this does not necessarily reflect the overall robustness or security posture of the implementations. It is essential to consider this in the context of the fuzzing method employed.

4.3 Identified Vulnerabilities with OCPPStorm

Each identified issue was systematically reproduced, verified, and documented, with detailed logs and visual evidence collected to support the findings. The vulnerabilities identified are critical not only for the OCPP.Core and Steve implementations but potentially for other systems adhering to the OCPP standards, highlighting the need for rigorous protocol compliance and system validation.

4.3.1 OCPP.Core Vulnerabilities

Our comprehensive evaluation using OCPPStorm has uncovered several vulnerabilities within the OCPP.Core implementation. These issues span various aspects of the system, ranging from potential Denial of Service (DoS) exploits to logical inconsistencies that could affect transaction integrity and security.

- **DoS Vulnerability:** A significant flaw was discovered where the system fails to validate the length of the ’chargePointVendor’ field in ‘BootNotification’ message, potentially leading to server instability and DoS when processing excessively large inputs.

- **Negative Charging Transactions:** The system accepts ’StopTransaction’ messages with ’meterStop’ values lower than ’meterStart’ from ’StartTransaction’ messages, leading to incorrect logging of negative charging amounts and potential billing inaccuracies.
- **Unauthorized Transaction Termination:** A security lapse allows ’StopTransaction’ messages with any random ’transactionId’ to terminate active transactions, indicating insufficient validation procedures.
- **Concurrent Transaction Handling:** The system permits multiple transactions with the same ’connectorId’ and ’idTag’, contrary to the expected ’ConcurrentTx’ status, which could result in critical transaction management and billing errors.
- **Handling of Additional and Duplicate Properties:** A vulnerability where the server processes StartTransaction messages containing additional, arbitrary properties, or duplicate properties without proper validation. Particularly concerning is the server’s acceptance of the last occurrence of a duplicate property. This could be exploited to alter transaction records or impact system integrity, as the server does not reject messages with unknown additional properties or duplicate entries, potentially leading to unpredictable system behavior.
- **Repeated Use of Message ID:** The system exhibits a compliance issue with the OCPP specification, which mandates that each CALL message must have a unique message ID on the same WebSocket connection. Our tests revealed that the system fails to enforce this requirement, accepting multiple messages with the same message ID. This oversight could lead to confusion in request identification and processing, potentially impacting the integrity and traceability of transactions.

4.3.2 Steve Vulnerabilities

- **Invalid Timestamp Handling in StartTransaction:** A significant flaw was detected in how the server processes the ‘StartTransaction‘ OCPP message, specifically related to the handling of the ‘timestamp‘ parameter. When a ‘StartTransaction‘ message is sent with the ‘timestamp‘ set to 1000000, the server erroneously stores the ‘start_timestamp‘ in the database as ‘0000-00-00 00:00:00.000000‘. This incorrect handling leads to an SQL exception error when attempting to retrieve or display the transaction data in the system’s web interface. The expected behavior is for the server to either store a valid timestamp or reject the message if the timestamp is invalid, thus ensuring the integrity and accuracy of transaction records.
- **Handling of Multiple StopTransaction Messages:** The system displays a critical issue in maintaining transaction integrity. Despite expectations that a single

'StartTransaction' message should correspond to only one 'StopTransaction' message, the system erroneously accepts multiple 'StopTransaction' messages for the same transaction. This improper handling results in the creation of multiple stop records for the same transaction in the 'transaction_stop' table. Such behavior causes significant inconsistencies and potential errors in transaction management and reporting. The reproduction of this problem involves sending a 'StartTransaction' message followed by several 'StopTransaction' messages with varying 'meterStop' values for the same transaction, each of which is incorrectly recorded as a separate entry in the database.

- **Reprocessing of StartTransaction Messages:** The system exhibits a notable flaw in handling the lifecycle of transactions. Ideally, once a transaction is initiated with a 'StartTransaction' message and subsequently concluded with a 'StopTransaction' message, any further attempts to resend the same 'StartTransaction' message should be rejected, with the system indicating that the transaction has already been concluded. However, in its current behavior, the system erroneously accepts the repeated 'StartTransaction' message without creating a new record in the database. This could mislead users into believing a new transaction has commenced. Moreover, the system allows the initiation of a new transaction by merely altering a single field in the 'StartTransaction' message, such as the 'meterStart' value. This can lead to transaction duplication and data inconsistencies. To reproduce this issue, one needs to send a 'StartTransaction' message, stop the transaction, then resend the same 'StartTransaction' message, and finally send it again with a minor modification. This behavior poses significant challenges in accurately managing and tracking transactions, potentially impacting billing and auditing processes.

- **Unauthorized Transaction Termination via Predictable Transaction IDs:** A potential security vulnerability was identified in the system's handling of transaction IDs. The server issues new transaction IDs for incomplete 'StartTransaction' requests, and these IDs are auto-incremented, making them predictable. This predictability allows an unauthorized entity to terminate ongoing transactions. The issue was reproduced using two Docker containers simulating OCPP clients. The first container initiates a valid transaction, while the second container sends an incomplete 'StartTransaction' request to obtain a new transaction ID, then uses this ID (minus 1) to issue a 'StopTransaction' request, thereby unauthorizedly terminating the transaction started by the first container. This behavior was confirmed through the system's web interface. The expected behavior is for the server to not issue transaction IDs for incomplete requests and to authenticate 'StopTransaction' requests before processing them, thus preventing unauthorized transaction terminations.

- **Billing Discrepancies from Invalid MeterStop Values:** The server's handling of 'StopTransaction' messages has been found to potentially lead to billing discrepancies. Specifically, the server accepts 'StopTransaction' messages where the 'meterStop' value is less than the 'meterStart' value provided in the corresponding 'StartTransaction' message. This acceptance can result in incorrect billing calculations. The issue was identified through the following steps: initiating a 'StartTransaction' with a given 'meterStart' value, followed by a 'StopTransaction' message with a 'meterStop' value lower than the ‘meterStart‘. Despite the logical inconsistency, the server processes the transaction, impacting the billing accuracy. The expected behavior is for the server to validate the ‘meterStop‘ values, ensuring they are equal to or greater than the ‘meterStart‘ values, thereby maintaining billing integrity.

- **Repeated Use of Message ID:** The system exhibits a compliance issue with the OCPP specification, which mandates that each CALL message must have a unique message ID on the same WebSocket connection. Our tests revealed that the system fails to enforce this requirement, accepting multiple messages with the same message ID. This oversight could lead to confusion in request identification and processing, potentially impacting the integrity and traceability of transactions.

Chapter 5

Future work

5.1 Deployment on Additional Server Implementations

Expanding the deployment of OCPPStorm to additional server implementations is crucial for a more comprehensive understanding of its effectiveness. Future work in this area will focus on several key aspects:

- **Diverse Implementation Testing:** test OCPPStorm on a variety of server implementations, particularly those that are prevalent in the industry or possess unique architectural features. This will help understand the tool's adaptability and effectiveness across different server configurations.
- **Vulnerability Analysis:** The extended deployment aims to uncover a broader range of vulnerabilities or behavior inconsistencies in various server implementations, enriching our understanding of potential security flaws in OCPP ecosystems.
- **Architectural Impact Assessment:** By analyzing how different server designs and architectures influence fuzzing results, we can provide insights into design choices that enhance or weaken the security of OCPP implementations.

5.2 Integrating White Box Fuzzing

Incorporating white-box fuzzing techniques alongside our existing black-box methods can greatly enhance the depth and breadth of our security analysis of OCPP implementations:

- **Combining Fuzzing Techniques:** Employing both black-box and white-box fuzzing approaches allows us to leverage the strengths of each method. While black-box fuzzing offers a broad, surface-level assessment, white-box fuzzing can provide detailed insights into the internal logic and structure of the OCPP implementations.
- **Enhanced Code Coverage:** White-box fuzzing will enable us to achieve higher code coverage, particularly in areas that are less exposed or not typically triggered in normal operation. This approach is crucial for uncovering vulnerabilities that require specific conditions or sequences of actions to manifest.
- **Synergistic Approach:** The use of white-box fuzzing in conjunction with black-box techniques represents a synergistic approach to security testing. It combines the unpredictability and broad application of black-box testing with the targeted and detailed examination of white-box methods.

5.3 Source Code Static Analysis

The integration of static code analysis into the evaluation of OCPP server implementations offers a valuable complement to dynamic fuzzing techniques. This approach aims to enhance the overall security assessment framework:

- **Static Analysis Methodology:** Static code analysis techniques can be applied to OCPP server implementations to identify a range of potential vulnerabilities and code quality issues. These issues might not be evident through dynamic testing methods.
- **Types of Vulnerabilities Detected:** Static analysis is particularly effective in uncovering various types of security vulnerabilities, including but not limited to code injection, buffer overflows, and similar flaws that could remain concealed during dynamic testing phases.
- **Integration with Fuzzing:** The convergence of findings from static analysis and fuzzing data is expected to forge a more comprehensive and robust framework for security testing in OCPP implementations. Such an integrated approach ensures a thorough assessment, covering both dynamic and static aspects of code security.

Chapter 6

Related work

6.1 Introduction

This chapter presents a review of existing literature pertinent to the security and testing of the Open Charge Point Protocol (OCPP). It encapsulates various studies, ranging from security assessments of electric vehicle (EV) charging ecosystems to advanced testing methodologies for OCPP. This review sets the stage for understanding the current landscape of OCPP security and the innovative contributions of this thesis.

6.1.1 Electric Vehicle Charging Security

The rapid growth in the adoption of Electric Vehicles (EVs) has stimulated numerous studies examining the security aspects of EV charging infrastructures. Key studies in this domain include [8] and [9]. These papers provide an exhaustive examination of the vulnerabilities inherent in EV charging systems, extending from hardware components to critical communication protocols like OCPP.

In [8] the researchers conduct a multi-faceted analysis of the EV charging ecosystem. They explore the potential security risks associated with various components of EV charging infrastructure, including charging stations, network connections, and payment systems. The study highlights specific vulnerabilities in communication protocols, emphasizing the need for robust security measures in protocol design and implementation. This paper is particularly relevant as it underscores the complexity of the EV charging ecosystem and the intricate web of interactions that need to be secured.

Similarly, the [9] paper takes a comprehensive approach to scrutinize the security challenges facing EV charging systems. It delves into the potential cybersecurity threats that could compromise the integrity and reliability of these systems. The paper pays special attention to communication protocols like OCPP, discussing how vulnerabilities in these protocols could lead to unauthorized access, data breaches, and even disruption of charging services. This review is instrumental in understanding the broad spectrum of cybersecurity challenges that must be addressed to safeguard the burgeoning EV charging infrastructure.

Both studies underscore the criticality of secure communication protocols in the EV charging domain. They highlight how vulnerabilities in protocols like OCPP can have far-reaching implications, not just for individual charging stations but for the entire network of EV infrastructure. This aligns closely with the focus of our thesis, which seeks to enhance the security of OCPP implementations through rigorous testing methodologies. By understanding the potential security gaps identified in these studies, we can better tailor our fuzzing tool, OCPPStorm, to detect and mitigate similar vulnerabilities in OCPP implementations.

6.1.2 OCPP Security Vulnerabilities

The security of OCPP implementations has been a focal point of several research endeavors, particularly in the context of identifying and addressing specific vulnerabilities. Studies like [10] and [11] have been pivotal in uncovering and analyzing critical security threats within OCPP-based systems.

In [10], the researchers present a detailed analysis of the susceptibility of OCPP to man-in-the-middle (MitM) attacks. These attacks, where an unauthorized intermediary can intercept and potentially alter the communication between the charging station and the central system, pose a significant threat to the integrity of OCPP communications. The study not only identifies the potential for such attacks but also proposes mitigation strategies to enhance the security of OCPP implementations against MitM scenarios. This research is crucial as it directly addresses one of the fundamental security concerns in any communication protocol – the integrity and trustworthiness of the transmitted data.

Similarly, [11] explores the vulnerabilities in OCPP that could allow attackers to disrupt charging sessions or even gain remote code execution capabilities. This study delves into the practical implications of such vulnerabilities, demonstrating how they could be exploited to compromise the functionality of EV charging stations or gain unauthorized control. The findings of this study are particularly alarming as they expose the potential for severe disruptions in the EV charging process, underscoring the need for stringent security measures in OCPP implementations.

Both these studies highlight the multifaceted nature of security challenges in OCPP implementations. They reveal not only the potential for data interception and manipulation but also the possibility of direct attacks on the functionality and control of EV charging systems. These insights are invaluable for our research, as they provide a clear indication of the types of vulnerabilities that our tool, OCPPStorm, should be adept at detecting and mitigating. By incorporating the knowledge gleaned from these studies, we can enhance OCPPStorm's effectiveness in uncovering and addressing critical security vulnerabilities in OCPP implementations, thereby contributing to the overall resilience and reliability of EV charging infrastructures.

6.1.3 Testing and Compliance in OCPP

The evolution of Electric Vehicle (EV) charging systems, particularly regarding the Open Charge Point Protocol (OCPP), has underscored the necessity for robust testing mechanisms. Traditional approaches, often reliant on predefined test cases, have been foundational in ensuring protocol adherence, as seen in studies like [12]. However, the advent of fuzzing methodologies like those employed by OCPPStorm represents a significant paradigm shift in testing protocols.

The study mentioned above introduces a specialized tool for assessing OCPP 1.6 message conformance. This tool is crucial for validating the structural and content accuracy of messages against OCPP 1.6 specifications. While such tools are invaluable for confirming compliance with known standards, they operate within the boundaries of predefined scenarios and expected message formats.

In contrast, fuzzing, as employed by OCPPStorm, introduces a dynamic and unpredictable element into testing. Fuzzing does not rely on preset test cases; instead, it generates unexpected and often anomalous inputs. This approach is designed to uncover vulnerabilities and bugs that might elude conventional testing methods. Fuzzing's strength lies in its ability to simulate real-world scenarios where inputs may not always conform to expected patterns, thereby exposing potential weaknesses in protocol implementations.

The novelty of OCPPStorm lies in its ability to offer a more exhaustive testing landscape. By moving beyond the constraints of predefined test cases, it provides a broader and more realistic assessment of OCPP implementations. This is critical in a domain where the reliability and security of communication protocols are paramount for the safe operation of EV charging infrastructures.

In summary, while traditional testing tools like the one highlighted in the mentioned study play a crucial role in ensuring baseline compliance, the introduction of fuzzing methodologies through tools like OCPPStorm marks a significant advancement in the field. By embracing unpredictability and rigorously challenging protocol implementations, OCPPStorm sets a new standard in OCPP testing, ensuring that EV charging systems are not only compliant but also resilient against a wider array of potential threats and anomalies.

6.1.4 Fuzzing Techniques in Protocol Security

AutoFuzz[13] exemplifies the advancement in automated fuzzing techniques for network protocols. While AutoFuzz[13] focuses on automating the fuzzing process, OCPPStorm introduces novel aspects specifically tailored for the Open Charge Point Protocol (OCPP).

OCPPStorm distinguishes itself by its specialized approach to fuzzing in the context of OCPP. Unlike AutoFuzz[13], which targets a broad range of network protocols, OCPPStorm is meticulously designed to address the unique challenges and intricacies of the OCPP. It leverages specialized knowledge of the protocol to generate more targeted and relevant test cases. This results in a more effective identification of potential vulnerabilities specific to OCPP implementations.

Moreover, OCPPStorm incorporates a blend of techniques that are not just automated but also intelligently adapted to the protocol's structure and requirements. This includes understanding the specific message formats, error codes, and communication flows unique to OCPP. In doing so, OCPPStorm extends beyond the capabilities of generic automated fuzzers like AutoFuzz[13], providing a deeper, more focused analysis that is crucial for the nuanced environment of electric vehicle charging systems.

In essence, while AutoFuzz[13] sets the stage for automated fuzzing in network protocols, OCPPStorm builds upon these principles, introducing specific adaptations and enhancements for the OCPP. This represents a significant step forward in protocol-specific fuzzing, particularly in the evolving domain of electric vehicle charging infrastructure security.

6.2 Conclusion

The literature review reveals a growing focus on the security of EV charging systems and the need for rigorous testing of protocols like OCPP. While existing studies have laid a solid groundwork, there remains a need for advanced testing methodologies that can adapt to the diverse and evolving nature of OCPP implementations. Our research, through the development of OCPPStorm, aims to address this gap, building upon the existing body of knowledge and introducing innovative approaches to protocol security testing.

Chapter 7

Conclusion

This thesis introduced OCPPStorm, a sophisticated fuzzing tool tailored for the Open Charge Point Protocol (OCPP), aiming to enhance the security of electric vehicle charging systems. Through extensive testing, OCPPStorm has proven effective in uncovering a variety of vulnerabilities in OCPP implementations, demonstrating its potential as a valuable asset in the realm of EV charging system security.

7.1 Key Findings

OCPPStorm was rigorously applied to various OCPP implementations, revealing significant vulnerabilities, including Denial of Service (DoS) exploits, transaction integrity issues, and unauthorized transaction terminations. These findings highlight critical security lapses in current implementations and underscore the necessity for enhanced security measures and rigorous protocol compliance.

7.2 Contributions to the Field

The development of OCPPStorm represents a substantial contribution to the field of electric vehicle charging system security. By focusing specifically on OCPP, this tool addresses a niche yet crucial aspect of EV infrastructure, providing insights into potential security weaknesses and aiding in the development of more robust systems.

7.3 Limitations

Despite its effectiveness, OCPPStorm has limitations. Its current scope is confined to specific versions of OCPP and may not comprehensively cover all possible security threats. Future enhancements are required to broaden its applicability and effectiveness across different protocol versions and configurations.

In conclusion, OCPPStorm marks a significant step towards fortifying the security of electric vehicle charging infrastructures. It not only sheds light on existing vulnerabilities but also provides a robust framework for future research and development in this critical area.

Appendix A

OCPP.Core Vulnerabilities

A.1 DoS Vulnerability Due to Unrestricted 'chargePointVendor' Length

A.1.1 Attack Description

An exploitable Denial of Service (DoS) condition in OCPP.Core was identified, characterized by the absence of length validation for the `chargePointVendor` field within the `BootNotification` message.

A.1.2 Expected System Behavior

The system should adhere to the OCPP specification by enforcing a maximum length of 20 characters for the `chargePointVendor` field, rejecting any messages that exceed this threshold.

A.1.3 Actual System Behavior

In practice, the system accepts and attempts to process `BootNotification` messages containing a `chargePointVendor` field with lengths far exceeding the specified maximum, resulting in server instability and potential DoS attacks.

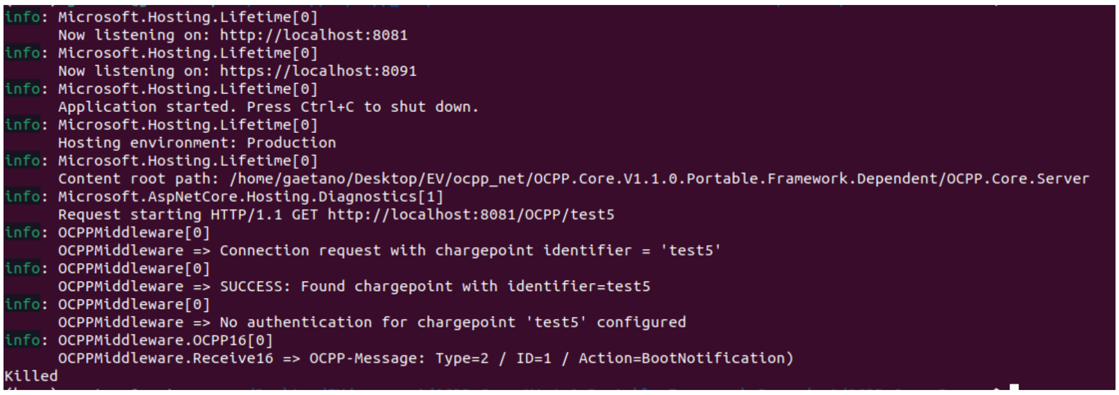
A.1.4 Reproduction Steps

1. Construct and transmit a `BootNotification` message with the `chargePointVendor` field containing 442 MB of data.

2. The server attempts to process the oversized message, which leads to a crash, demonstrating the DoS vulnerability.

A.1.5 Visual Evidence

Below is a screenshot from the server logs illustrating the system's response to the oversized `BootNotification` message:



```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:8081
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:8091
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: /home/gaetano/Desktop/EV/ocpp_net/OCPP.Core.V1.1.0.Portable.Framework.Dependent/OCPP.Core.Server
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET http://localhost:8081/OCPP/test5
info: OCPPMiddleware[0]
      OCPPMiddleware => Connection request with chargepoint identifier = 'test5'
info: OCPPMiddleware[0]
      OCPPMiddleware => SUCCESS: Found chargepoint with identifier=test5
info: OCPPMiddleware[0]
      OCPPMiddleware => No authentication for chargepoint 'test5' configured
info: OCPPMiddleware.OCPP16[0]
      OCPPMiddleware.Receive16 => OCPP-Message: Type=2 / ID=1 / Action=BootNotification)
Killed
```

Figure A.1: Server logs capturing the crash caused by the oversized `BootNotification` message.

A.2 Inconsistency in Transaction Meter Values Allowing Negative Charging

A.2.1 Attack Description

A logical discrepancy was detected in OCPP.Core's handling of transaction meter values. The system accepts `StopTransaction` messages with `meterStop` values that are lower than the `meterStart` values provided in the initial `StartTransaction` messages, resulting in the logging of negative charging amounts.

A.2.2 Expected System Behavior

The system should validate `meterStop` values to ensure they are greater than or equal to `meterStart` values, preserving the logical integrity of transaction data.

A.2.3 Actual System Behavior

The system logs transactions with negative charging amounts, which contradicts the principle of energy metering and could lead to billing inaccuracies.

A.2.4 Reproduction Steps

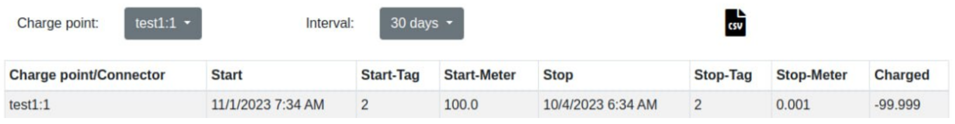
1. Initiate a `StartTransaction` with a specific `meterStart` value.

2. Proceed to send a `StopTransaction` message with a `meterStop` value that is less than the `meterStart` value.

3. Observe the server's acceptance of the transaction and the recording of a negative charged amount.

A.2.5 Visual Evidence

The screenshot below displays the negative charged amount recorded in the system database and shown in the web interface, highlighting the inconsistency:



Charge point/Connector	Start	Start-Tag	Start-Meter	Stop	Stop-Tag	Stop-Meter	Charged
test1:1	11/1/2023 7:34 AM	2	100.0	10/4/2023 6:34 AM	2	0.001	-99.999

Figure A.2: The web page showing a negative charging amount due to inconsistent meter values

A.3 StopTransaction: Unauthorized Termination with Random Transaction ID

A.3.1 Issue Description

A significant security flaw was found within the OCPP.Core server's transaction management process. The server incorrectly accepts a `StopTransaction` message with a random `transactionId`, disregarding the need for the specific `transactionId` allocated at the transaction's initiation.

A.3.2 Expected System Behavior

The server is expected to validate `StopTransaction` messages to confirm the inclusion of the initial `transactionId`, ensuring that transactions are conclusively tied to and terminated by authorized entities.

A.3.3 Actual System Behavior

The system erroneously terminates active transactions upon receipt of `StopTransaction` messages containing arbitrary `transactionId`s, indicating a lack of essential validation mechanisms.

A.3.4 Reproduction Steps

1. A transaction is initiated via `StartTransaction`, and a valid `transactionId` is provided by the server.

2. A `StopTransaction` message is submitted with a random `transactionId` not corresponding to the one initially provided.

3. The server processes the request, resulting in the unwarranted termination of the transaction.

A.4 Multiple Transactions Allowed with Same connectorId and idTag

A.4.1 Issue Description

An inconsistency with the OCPP server implementation was identified concerning the handling of concurrent transactions. According to OCPP documentation, the initiation of a new transaction with an ‘idTag‘ already engaged in an ongoing transaction should prompt an ‘AuthorizationStatus‘ of ‘ConcurrentTx‘. Deviations from this protocol were noted:

1. Server allows multiple transactions with the same ‘idTag‘ without issuing a ‘ConcurrentTx‘ status.
2. Server processes a ‘StopTransaction‘ message with a random ‘transactionId‘, effectively terminating the current transaction.

A.4.2 Expected System Behavior

The server should reject new transactions involving an ‘idTag‘ already in use, providing an ‘AuthorizationStatus‘ of ‘ConcurrentTx‘ as per OCPP standards.

A.4.3 Actual System Behavior

The server permits the creation of multiple transactions using identical ‘idTags‘ without the ‘ConcurrentTx‘ status and accepts ‘StopTransaction‘ messages with arbitrary ‘transactionIds‘.

A.4.4 Reproduction Steps

1. Transmit a ‘StartTransaction‘ message with a designated ‘idTag‘.
2. Send another ‘StartTransaction‘ message with the identical ‘idTag‘ during the active transaction.
3. The server fails to return a ‘ConcurrentTx‘ status.
4. Issue a ‘StopTransaction‘ with a non-specific ‘transactionId‘.
5. The server halts the most recent transaction.

A.4.5 Potential Impact

This issue may precipitate significant transaction management and billing discrepancies, compromising the system's functional integrity.

A.4.6 Proposed Mitigation Strategy

Revisions are suggested for the server's logic to encompass:

- The issuance of an 'AuthorizationStatus' of 'ConcurrentTx' for simultaneous 'StartTransaction' requests with the same 'idTag'.
- The validation of 'transactionId' within 'StopTransaction' requests to ensure accurate transaction termination.

A.5 Vulnerability in Handling Additional and Duplicate Properties in StartTransaction Messages

A.5.1 Issue Description

A vulnerability was discovered in the OCPP server's handling of the `StartTransaction` message. Specifically, the server incorrectly accepts additional, arbitrary properties and, in cases of duplicate properties, uses the value from the last occurrence without proper validation. For instance, if a `StartTransaction` message includes two different `connectorId` properties, the server processes the message using the value of the last `connectorId`.

A.5.2 Reproduction Steps

1. Send a `StartTransaction` message with a custom, additional property.
2. Send a `StartTransaction` message with duplicate properties, such as two different `connectorId` values.
3. Observe that the server accepts the message and processes it based on the last occurrence of the duplicate property.

A.5.3 Expected System Behavior

The server should reject messages with unknown additional properties and messages containing duplicate properties.

A.5.4 Actual System Behavior

The server processes the `StartTransaction` message, storing the value from the last occurrence of a duplicate property, which can lead to potential misconfigurations and system inconsistencies.

A.5.5 Potential Impact

Accepting additional and duplicate properties without proper validation can lead to unpredictable system behavior. This vulnerability could potentially be exploited to alter transaction records or impact the overall system integrity, posing significant risks to the reliability and security of the system.

Appendix B

Steve Vulnerabilities

B.1 Invalid Timestamp Handling in StartTransaction Messages

B.1.1 Issue Description

A critical vulnerability was identified in the handling of timestamp values in `StartTransaction` OCPP messages. Specifically, when the timestamp parameter is set to 1000000, the server incorrectly stores the `start_timestamp` in the database as 0000-00-00 00:00:00.000000, leading to an SQL exception when attempting to retrieve transaction data.

B.1.2 Reproduction Steps

1. Send a `StartTransaction` OCPP message with the timestamp set to 1000000.
2. Check the `start_timestamp` in the database, noting it is stored as 0000-00-00 00:00:00.000000.
3. Access the web interface to view the transaction data.
4. Encounter an SQL exception error due to the invalid timestamp storage.

B.1.3 Expected System Behavior

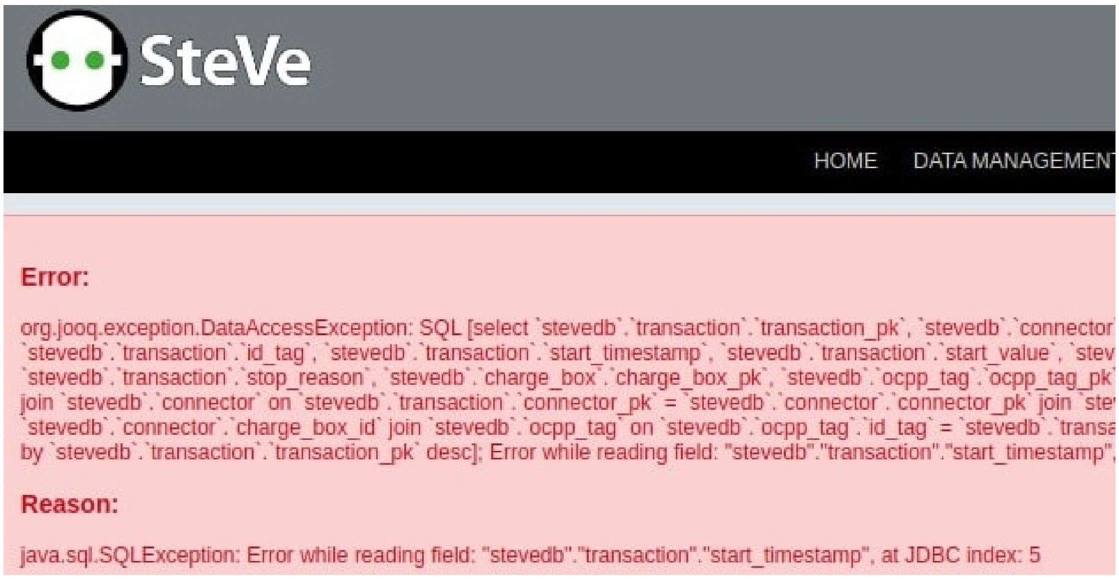
The server should accurately handle the timestamp value, either storing a valid timestamp or rejecting the message if the timestamp is determined to be invalid.

B.1.4 Actual System Behavior

The server stores an invalid timestamp, causing an SQL exception when reading the transaction data.

B.1.5 Visual Evidence

A screenshot is attached, demonstrating the SQL exception error encountered when the system tries to read transaction data with the invalid timestamp.



```
Error:

org.jooq.exception.DataAccessException: SQL [select `stevedb`.`transaction`.`transaction_pk`, `stevedb`.`connector
`stevedb`.`transaction`.`id_tag`, `stevedb`.`transaction`.`start_timestamp`, `stevedb`.`transaction`.`start_value`, `stev
`stevedb`.`transaction`.`stop_reason`, `stevedb`.`charge_box`.`charge_box_pk`, `stevedb`.`ocpp_tag`.`ocpp_tag_pk`
join `stevedb`.`connector` on `stevedb`.`transaction`.`connector_pk` = `stevedb`.`connector`.`connector_pk` join `ste
`stevedb`.`connector`.`charge_box_id` join `stevedb`.`ocpp_tag` on `stevedb`.`ocpp_tag`.`id_tag` = `stevedb`.`transa
by `stevedb`.`transaction`.`transaction_pk` desc]; Error while reading field: "stevedb"."transaction"."start_timestamp",

Reason:

java.sql.SQLException: Error while reading field: "stevedb"."transaction"."start_timestamp", at JDBC index: 5
```

Figure B.1: SQL exception error caused by invalid timestamp in StartTransaction message.

B.2 Handling Multiple StopTransaction Messages for a Single Transaction

B.2.1 Issue Description

A significant issue was identified in the system's handling of `StopTransaction` messages. The server erroneously accepts multiple `StopTransaction` messages for a single transaction, leading to the creation of multiple stop records for the same transaction in the `transaction_stop` table. This vulnerability results in inconsistencies and potential errors in transaction management and reporting.

B.2.2 Reproduction Steps

1. Send a `StartTransaction` message:

```
[2,"1","StartTransaction", {
  "connectorId": 2,
  "idTag": "6",
  "meterStart": 2,
  "reservationId": 11,
  "timestamp": "2023-10-03T12:34:56Z"
}]
```

2. Send a `StopTransaction` message multiple times for the same transaction, with varying `meterStop` values:

```
[2,"1","StopTransaction", {
  "idTag": "6",
  "meterStop": 200,
  "timestamp": "2023-11-06T15:54:23Z",
  "transactionId": 676897,
  "reason": "EVDisconnected"
}]
```

   (Repeat with different `meterStop` values and/or reason values)

3. Observe that each `StopTransaction` message creates a new entry in the `transaction_stop` table for the same `transaction_pk`.

B.2.3 Expected System Behavior

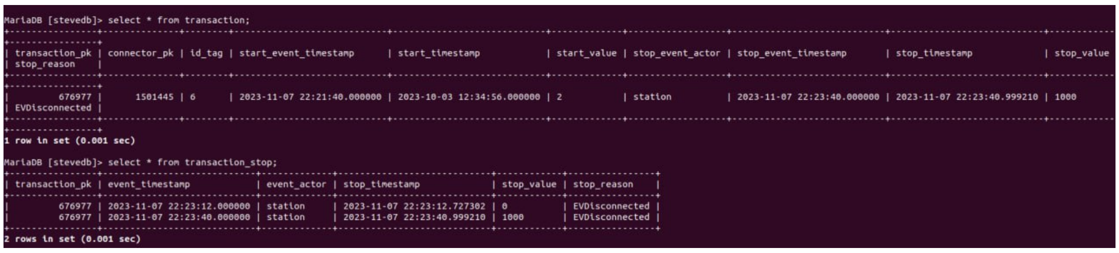
The system should enforce transaction integrity by only allowing a single `StopTransaction` message for each `StartTransaction`. Additional `StopTransaction` messages for a transaction that has already been stopped should be rejected, preventing duplicate entries in the database.

B.2.4 Actual System Behavior

The server accepts multiple `StopTransaction` messages for a single transaction, resulting in multiple records for the same transaction in the `transaction_stop` table, causing inconsistencies in transaction management.

B.2.5 Visual Evidence

A screenshot is provided, showing the state of the database after sending two different `StopTransaction` messages for the same transaction, one with `meterStop = 0` and another with `meterStop = 1000`.



```
MariaDB [stevedb]> select * from transaction;
+----------------+--------------+--------+----------------------------+----------------------------+-------------+------------------+----------------------------+----------------------------+------------
+----------------+
| transaction_pk | connector_pk | id_tag | start_event_timestamp      | start_timestamp            | start_value | stop_event_actor | stop_event_timestamp       | stop_timestamp             | stop_value
| stop_reason    |
+----------------+--------------+--------+----------------------------+----------------------------+-------------+------------------+----------------------------+----------------------------+------------
+----------------+
|         676977 |      1501445 | 6      | 2023-11-07 22:21:40.000000 | 2023-10-03 12:34:56.000000 | 2           | station          | 2023-11-07 22:23:40.000000 | 2023-11-07 22:23:40.999210 | 1000
| EVDisconnected |
+----------------+--------------+--------+----------------------------+----------------------------+-------------+------------------+----------------------------+----------------------------+------------
+----------------+
1 row in set (0.001 sec)

MariaDB [stevedb]> select * from transaction_stop;
+----------------+----------------------------+-------------+----------------------------+------------+----------------+
| transaction_pk | event_timestamp            | event_actor | stop_timestamp             | stop_value | stop_reason    |
+----------------+----------------------------+-------------+----------------------------+------------+----------------+
|         676977 | 2023-11-07 22:23:12.000000 | station     | 2023-11-07 22:23:12.727302 | 0          | EVDisconnected |
|         676977 | 2023-11-07 22:23:40.000000 | station     | 2023-11-07 22:23:40.999210 | 1000       | EVDisconnected |
+----------------+----------------------------+-------------+----------------------------+------------+----------------+
2 rows in set (0.001 sec)
```

Figure B.2: Database entries showing multiple stop records for the same transaction.

B.3 Improper Handling of Repeated and Slightly Modified StartTransaction Messages

B.3.1 Issue Description

The system exhibits improper handling of `StartTransaction` messages following a transaction's conclusion. After a transaction is stopped using a `StopTransaction` message, any subsequent attempts to resend the same `StartTransaction` message are erroneously accepted by the system, leading to potential confusion and mismanagement. Additionally, the system allows the initiation of a new transaction by altering a single field in a `StartTransaction` message, such as `meterStart`, which could result in transaction duplication and data inconsistencies.

B.3.2 Reproduction Steps

1. Send a `StartTransaction` message and receive a `transactionId`.
2. Send a corresponding `StopTransaction` message to conclude the transaction.
3. Resend the same `StartTransaction` message. Observe that the system accepts the message but does not create a new record in the database; the previous `transactionId` is returned.
4. Modify a single parameter (e.g., `meterStart`) in the `StartTransaction` message and send it again. Note that a new `transactionId` is generated, and a new transaction appears to start.

B.3.3 Expected System Behavior

Once a `StartTransaction` message is sent and subsequently concluded with a `StopTransaction` message, further attempts to resend the same `StartTransaction` message should be rejected, with the system responding with an error indicating that the transaction has already concluded. Additionally, new transactions should require unique initiation messages, rather than accepting messages with minor alterations.

B.3.4 Actual System Behavior

The system accepts the resend of a `StartTransaction` message after the transaction has been concluded, misleading users into believing a new transaction has started. Furthermore, the system permits the initiation of a new transaction by only altering a single parameter in the `StartTransaction` message, leading to potential transaction duplication and data inconsistencies.

B.3.5 Additional Context

This behavior can lead to significant issues in transaction management. It may cause confusion, complicate accurate transaction tracking, and could impact billing and auditing processes.

B.4 Unauthorized Transaction Termination Due to Predictable Transaction IDs

B.4.1 Docker Container Setup

Two Docker containers are configured to simulate two OCPP clients for testing purposes.

B.4.2 Issue Description

A security vulnerability was identified in the system related to unauthorized transaction termination. This vulnerability stems from the server's practice of issuing new transaction IDs for incomplete `StartTransaction` requests and the predictable, auto-incremented nature of these IDs, which allows an attacker to foresee and misuse them for terminating other transactions.

B.4.3 Reproduction Steps

1. From the first Docker container (Client 1), initiate a transaction:

```
[2,"1","StartTransaction", {
  "connectorId": 1,
  "idTag": "test",
  "meterStart": 20,
  "timestamp": "2023-09-03T12:34:56Z"
}]
```

2. From the second Docker container (Client 2), send a `StartTransaction` request with an empty `idTag` to receive a new transaction ID:

```
[2, "1", "StartTransaction", {"idTag": ""}]
```

3. Client 2 then sends a `StopTransaction` request using the new transaction ID minus 1:

```
[2, "1","StopTransaction", {
  "meterStop": 1,
  "timestamp": "2023-11-06T15:54:23Z",
  "transactionId": [Retrieved Transaction ID - 1],
  "reason": "EVDisconnected",
  "transactionData": [...]
}]
```

4. The unauthorized termination of Client 1's transaction is verified via the web interface.

B.4.4 Expected System Behavior

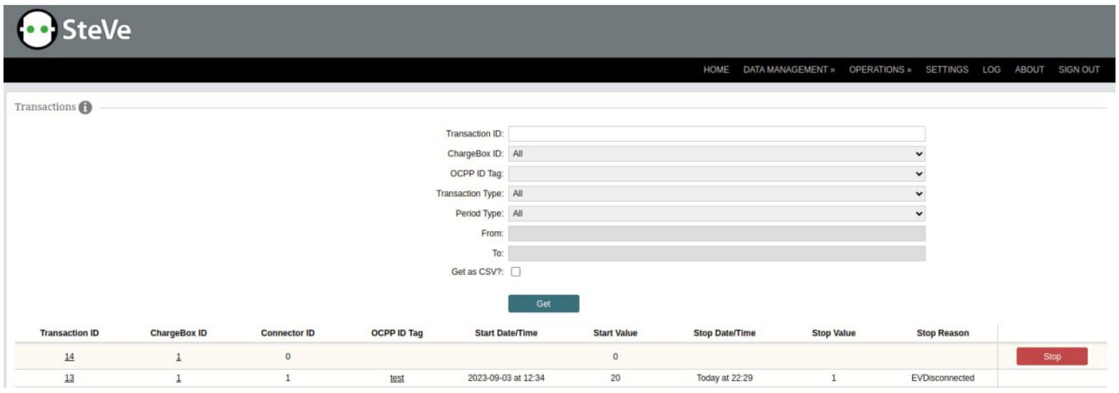
The server should not issue transaction IDs for incomplete `StartTransaction` requests and should authenticate `StopTransaction` requests before processing to prevent unauthorized transaction terminations.

B.4.5 Actual System Behavior

The server erroneously processes unauthorized `StopTransaction` requests using predictable transaction IDs, compromising transaction integrity.

B.4.6 Visual Evidence

A screenshot is provided, showing the unauthorized transaction termination confirmed through the web interface.



Transaction ID:	
ChargeBox ID:	All
OCPP ID Tag:	
Transaction Type:	All
Period Type:	All
From:	
To:	
Get as CSV?:	☐

Transaction ID	ChargeBox ID	Connector ID	OCPP ID Tag	Start Date/Time	Start Value	Stop Date/Time	Stop Value	Stop Reason	
14	1	0			0				Stop
13	1	1	test	2023-09-03 at 12:34	20	Today at 22:29	1	EVDisconnected	

Figure B.3: Web interface confirmation of unauthorized transaction termination.

B.5 Billing Discrepancies Due to Unvalidated Meter Values in StopTransaction

B.5.1 Issue Description

A significant issue has been identified in the Steve OCPP server's transaction handling process. Specifically, when a `StopTransaction` message is sent with a `meterStop` value that is less than the `meterStart` value from the corresponding `StartTransaction`, it could result in billing discrepancies.

B.5.2 Reproduction Steps

1. Send a `StartTransaction` message with a specified `meterStart` value.
2. Later, send a `StopTransaction` message with a `meterStop` value that is less than the `meterStart` value.
3. Observe the server's acceptance of the transaction and note the resulting billing calculation.

B.5.3 Expected System Behavior

The server should validate `meterStop` values to ensure they are equal to or greater than the corresponding `meterStart` values. This validation is essential to prevent errors in billing calculations.

B.5.4 Actual System Behavior

The server accepts `StopTransaction` messages with a `meterStop` value that is less than the `meterStart` value. This acceptance can lead to incorrect billing calculations, potentially resulting in significant financial inaccuracies.

B.6 Duplicate Message ID Handling in WebSocket CALL Messages

B.6.1 Issue Description

An issue was discovered in the Steve OCPP server regarding the handling of CALL messages over WebSocket connections. Contrary to the OCPP specification, which mandates that each CALL message's ID must be unique within the same WebSocket connection, the server was found to process multiple messages using the same message ID without rejection.

B.6.2 Reproduction Steps

1. Establish a WebSocket connection to the Steve OCPP server.
2. Send multiple OCPP CALL messages using the same message ID.
3. Observe that the server accepts and processes these messages, disregarding the uniqueness requirement for message IDs.

B.6.3 Expected System Behavior

According to the OCPP specification, the server should enforce the uniqueness of the message ID for each CALL message. Any messages with duplicate message IDs should be rejected to maintain message tracking integrity and avoid confusion.

B.6.4 Actual System Behavior

The server does not adhere to the OCPP specification regarding message ID uniqueness. It processes messages with duplicate message IDs, potentially leading to issues in message tracking and misinterpretation of responses.

B.6.5 Potential Impact

This issue could result in complications with message correlation and response handling, and potentially affect the transactional integrity of communications between charging stations and the central system.

Bibliography

[1] Open Charge Alliance. *Open Charge Point Protocol 1.6(OCPP) Documentation.* Version 1.6. URL: `https://www.openchargealliance.org/protocols/ocpp-16/`.

[2] Open Charge Alliance. *Open Charge Point Protocol 2.0.1(OCPP) Documentation.* Version 2.0.1. URL: `https://www.openchargealliance.org/protocols/ocpp-201/`.

[3] Michal Zalewski. *AFL: American Fuzzy Loop.* 2014-2017. URL: `https://lcamtuf.coredump.cx/afl/`.

[4] The LLVM Project. *LibFuzzer: a library for coverage-guided fuzz testing.* 2002-2018. URL: `https://llvm.org/docs/LibFuzzer.html`.

[5] Dominic Steinhöfel and Andreas Zeller. «Input Invariants». In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 583–594. ISBN: 9781450394130. DOI: `10.1145/3540250.3549139`. URL: `https://doi.org/10.1145/3540250.3549139`.

[6] Steve Community. *Steve - OCPP 1.6 Implementation.* `https://github.com/steve-community/steve`.

[7] Dallmann Consulting. *OCPP.Core - OCPP 1.6 Implementation.* `https://github.com/dallmann-consulting/OCPP.Core/tree/main`.

[8] Joseph Antoun, Mohammad Ekramul Kabir, Bassam Moussa, Ribal Atallah, and Chadi Assi. «A Detailed Security Assessment of the EV Charging Ecosystem». In: *IEEE Network* 34.3 (2020), pp. 200–207. DOI: `10.1109/MNET.001.1900348`.

[9] Jay Johnson, Timothy Berg, Benjamin Anderson, and Brian Wright. «Review of Electric Vehicle Charger Cybersecurity Vulnerabilities, Potential Impacts, and Defenses». In: *Energies* 15.11 (2022). ISSN: 1996-1073. DOI: `10.3390/en15113931`. URL: `https://www.mdpi.com/1996-1073/15/11/3931`.

[10] Juan E. Rubio, Cristina Alcaraz, and Javier Lopez. «Addressing Security in OCPP: Protection Against Man-in-the-Middle Attacks». In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS).* 2018, pp. 1–5. DOI: `10.1109/NTMS.2018.8328675`.

[11] David Elmo, George Fragkos, Jay Johnson, Kenneth Rohde, Sean Salinas, and Junjie Zhang. «Disrupting EV Charging Sessions and Gaining Remote Code Execution with DoS, MITM, and Code Injection Exploits using OCPP 1.6». In: *2023 Resilience Week (RWS)*. 2023, pp. 1–8. DOI: `10.1109/RWS58133.2023.10284654`.

[12] Dwidharma Priyasta, H. Hadiyanto, and R. Septiawan. «Ensuring Compliance and Reliability in EV Charging Station Management Systems: A Novel Testing Tool for OCPP 1.6 Messages Conformance». In: *Journal Européen des Systèmes Automatisés* 56 (Mar. 2023), pp. 121–129. DOI: `10.18280/jesa.560116`.

[13] Serge Gorbunov and Arnold Rosenbloom. «Autofuzz: Automated network protocol fuzzing framework». In: *Ijcsns* 10.8 (2010), p. 239.