OCPPuzz: Specification-Driven Fuzzing of Charging Station Management Systems with Large Language Model

JONGCHAN HONG, Sungkyunkwan University, Republic of Korea
JAEWON KIM, Sungkyunkwan University, Republic of Korea
SUNGJAE HWANG*, Sungkyunkwan University, Republic of Korea

Electric vehicles (EVs) are being rapidly adopted, with over 61,000 publicly accessible charging stations deployed across the United States as of 2024. A core component of this infrastructure is the Charging Station Management System (CSMS), which is responsible for security-critical tasks such as user authentication and billing. Given its importance, the CSMS has become a target of real-world attacks that have resulted in financial losses, data breaches, and denial-of-service (DoS) incidents. Nevertheless, research on CSMS security remains limited, and automated testing tools are lacking. Testing CSMS is challenging because they communicate with charging stations (CS) using the Open Charge Point Protocol (OCPP). Effective testing must contend with OCPP's complexity: ① messages containing up to 48 fields, ② inter- and intra-message field dependencies, and ③ its stateful nature, which requires tracking the states of both CS and CSMS during testing.

To address these challenges, we present OCPPuzz, a specification-based fuzzing framework for CSMS. OCPPuzz automatically extracts message structures, field constraints, and dependency rules from the OCPP specification, as well as valid CS-CSMS state transitions described in its use case diagrams. To handle specifications expressed in natural language and semi-formal diagrams, OCPPuzz combines heuristic rule-based extraction with a large language model (LLM). We evaluated OCPPuzz on four open-source CSMS implementations and uncovered numerous deviations from the OCPP specification that led to critical security issues, including DoS and free charging. We reported 930 implementation bugs to the corresponding vendors, of which 492 have been acknowledged so far. In addition, we reported 155 specification bugs in OCPP to the Open Charge Alliance (OCA); 78 have been committed for fixes and 82 acknowledged for further investigation. We expect additional acknowledgments and fixes in the near future.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: OCPP, fuzzing, GPT, automated validation, protocol testing

ACM Reference Format:
Jongchan Hong, Jaewon Kim, and Sungjae Hwang. 2026. OCPPuzz: Specification-Driven Fuzzing of Charging Station Management Systems with Large Language Model. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE063 (July 2026), 20 pages. https://doi.org/10.1145/3797091

1 Introduction

The rapid adoption of electric vehicles (EVs) has accelerated, reaching 17.3 million units produced worldwide in 2024 [1]. To support this expansion, over 61,000 publicly accessible charging stations were deployed in the United States in the same year [8]. A central component of each charging station is the charging station management system (CSMS), which coordinates charging sessions,

*Corresponding author.

Authors' Contact Information: Jongchan Hong, Sungkyunkwan University, Republic of Korea, jc1904@skku.edu; Jaewon Kim, Sungkyunkwan University, Republic of Korea, jaewon9912@skku.edu; Sungjae Hwang, Sungkyunkwan University, Republic of Korea, sungjaeh@skku.edu.

ACM 2994-970X/2026/7-ARTFSE063
https://doi.org/10.1145/3797091

authenticates users, and manages billing. Because the CSMS is responsible for these security-critical functions, its reliability and trustworthiness are essential to the security of the EV charging infrastructure. However, recent attacks have revealed significant weaknesses in CSMS, enabling adversaries to obtain unauthorized file access and bypass authentication to steal sensitive data [22], and even manipulate the system to steal electricity [29].

Testing CSMSs is inherently challenging because they communicate with charging stations (CSs) using the Open Charge Point Protocol (OCPP) [3]. OCPP introduces substantial complexity: individual messages may contain up to 48 fields, many of which exhibit inter- and intra-message dependencies. Furthermore, the protocol is highly stateful, requiring testers to simultaneously track and reason about the valid states of both the CS and the CSMS during execution.

In this work, we present OCPPuzz, a specification-based fuzzing framework tailored for CSMS. The OCPP specification is written in a hybrid form: natural language descriptions combined with structured information about message formats and field level constraints, as well as use case diagrams that define valid message sequences and protocol states. OCPPuzz automatically extracts structured message formats and field constraints from the specification using heuristic, rule-based parsing. In addition, it leverages a large language model (LLM) to interpret constraints expressed in natural language and to extract valid message sequences with state information from use case diagrams. By combining these data, OCPPuzz can both generate tests that deliberately violate specification constraints and evaluate valid scenarios across diverse protocol states.

We applied OCPPuzz to four open-source CSMS implementations and uncovered numerous deviations from the OCPP specification. Several of these deviations resulted in critical security flaws, including denial-of-service conditions, inconsistent billing behavior, and unauthorized free charging. In total, we reported 930 implementation bugs to vendors (492 acknowledged to date) and 155 specification issues to the Open Charge Alliance (OCA) [23], of which 78 have been committed for fixes and 82 acknowledged for further investigation. These findings highlight both the urgency of securing CSMSs and the effectiveness of our specification-driven fuzzing approach.

In summary, this study makes the following contributions:

- We design and implement OCPPuzz, a testing framework that automatically extracts message formats, constraints, and state transitions from the OCPP specification. We release OCPPuzz publicly for reproducibility and community use[1].
- We apply OCPPuzz to multiple CSMS implementations and uncover hundreds of security-critical flaws, many of which have been acknowledged and fixed by vendors.

2 Background and Challenge

This section introduces the CSMS and OCPP protocol, and highlights the motivation and challenges of our work.

2.1 Security of CSMS

CS is the physical system that allows EVs to connect and recharge their batteries. Each CS is connected to a backend system known as the CSMS, which performs security-critical tasks such as user authentication, firmware updates, and billing. Therefore, the security of the CSMS is crucial; however, several real-world security issues have been reported. Researchers have identified vulnerabilities in ABB's CSMS, including unauthorized file access, bypassing PIN code authentication, and hijacking charger connections [22]. These attacks can result in financial losses, data leakage, and DoS attacks. In addition, researchers have discovered security flaws in the OCPP, which allow attackers to steal energy and driver information [29]. These vulnerabilities arise from the lack

[1]https://github.com/jongchan-hong/OCPPuzz

of clear specifications for handling multiple connections and from weak authentication policies in OCPP. This highlights the need for an effective testing framework for CSMSs, yet no such framework currently exists.

2.2 OCPP Protocol

OCPP [3] is a widely adopted standard protocol that enables communication between CSs and CSMSs. OCPP, maintained by the OCA [23], ensures a consistent and scalable EV charging infrastructure across diverse environments.

Figure 1 illustrates an example of CS and CSMS communication using OCPP. OCPP defines three types of messages: `CALL`, `CALLRESULT`, and `CALLERROR`. A `CALL` message (denoted by the value 2) is used to issue requests. For instance, the CSMS sends a `RequestStartTransaction` message to the CS, instructing it to initiate charging. The first element, 2, specifies that the message is of type `CALL`; the second element is a unique identifier for the request; the third element denotes the message name; and the fourth element is the payload of the message, which consists of several fields. Upon receiving this request, the CS responds with a `CALLRESULT` message (denoted by the value 3 in the first element). The second element, 1, indicates that this response corresponds to request message with identifier 1, and the final element (Accepted) confirms that the CS accepts the CSMS request. Subsequently, the CS issues a new `CALL` message to the CSMS to notify it that charging will commence. This message is assigned a new unique identifier 2 as second element, and has the message name `TransactionEvent`. However, this message includes an invalid value for `remoteStartId` field (5), which differs from the value (1) originally provided by the CSMS in step 1. Consequently, the CSMS responds with a `CALLERROR` message (denoted by the value 4) to indicate that the `remoteStartId` is incorrect. This example illustrates that, in order to generate valid messages for testing the CSMS, both the correct message sequence and field values must be precisely specified.

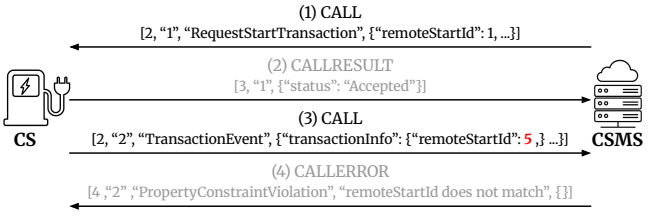


Fig. 1. Example of OCPP message exchanges (`CALL`, `CALLRESULT`, `CALLERROR`) between a CS and a CSMS.

2.3 OCPP Specification

OCPP protocol is specified using natural language descriptions as well as use case diagrams. Figure 2a illustrates the structure and semantics of the `TransactionEventRequest` message in the OCPP specification. According to the specification, the `TransactionEventRequest` message is defined for transmission from the CS to the CSMS and comprises 12 fields, including `transactionInfo`, `idToken`, and `meterValue`. The type of each field is specified in the second column (e.g., `TransactionType`), while the cardinality is indicated in the third column (e.g., `1..1` denotes that exactly one value must be provided; `0..1` indicates an optional field that can appear at most once; and `0..*` indicates zero or more values are permitted). The final column provides a description of each field and its intended value. Field types are classified as `Class`, `Enumeration`, or `Primitive` data types. For example, `IdTokenEnumType` is a class type with multiple fields defined in a separate subsection (see Figure 2d). An enumeration type defines a fixed set of values, also detailed in a separate subsection (Figure 2e). Primitive data types include string, integer, and identifierString, which may contain letters and digits ($[a-zA-Z0-9]$) as well as special characters ($[*, ., =, :, +, |, @, -, \_]$). These field values depend on `variables`, introduced in OCPP version 2.0.1, which serve as parameters for

1.60.1. TransactionEventRequest Message Direction

This section contains the field definition of the TransactionEventRequest PDU sent by the Charging Station to the CSMS.

transactionInfo	TransactionType	1..1	Required. Contains transaction specific information.
idToken	IdTokenType	0..1	This contains the identifier for which a transaction is (or will be) started or stopped. Variable Dependency
meterValue	MeterValueType	0..*	Depending on the EventType of this TransactionEvent the following Configuration Variable is used to configure the content: Updated: SampledDataTxUpdatedMeasurands

(a) `TransactionEventRequest` Definition

2.7.7. `SampledDataTxUpdatedMeasurands` Variable Dependency

Description	The Charging Station reports the list of supported Measurands in VariableCharacteristicsType.valuesList of this variable.

(b) `SampledDataTxUpdatedMeasurands` Definition

2.48. TransactionType Message Dependency

remoteStartId	integer	0..1	The ID given to RequestStartTransactionRequest.

(c) `TransactionType` Definition

2.28. IdTokenType

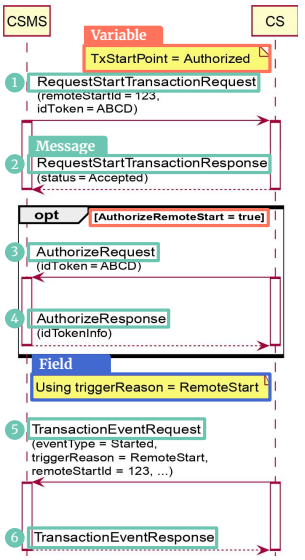
idToken	identifierString[0..36]	1..1	Required. IdToken is case insensitive.
type	IdTokenEnumType	1..1	Required. Enumeration of possible idToken types.

(d) `IdTokenType` Definition

3.43. IdTokenEnumType Field Dependency

NoAuthorization	IdToken field SHALL be left empty.
ISO14443	ISO 14443 UID of RFID card. It is represented as an array of 4 or 7 bytes in hexadecimal representation.
ISO15693	ISO 15693 UID of RFID card. It is represented as an array of 8 bytes in hexadecimal representation.

(e) `IdTokenEnumType` Definition



(f) Use Case Diagram

Fig. 2. Definition of the `TransactionEventRequest` message and the use case diagram for the remote charging scenario in the OCPP specification.

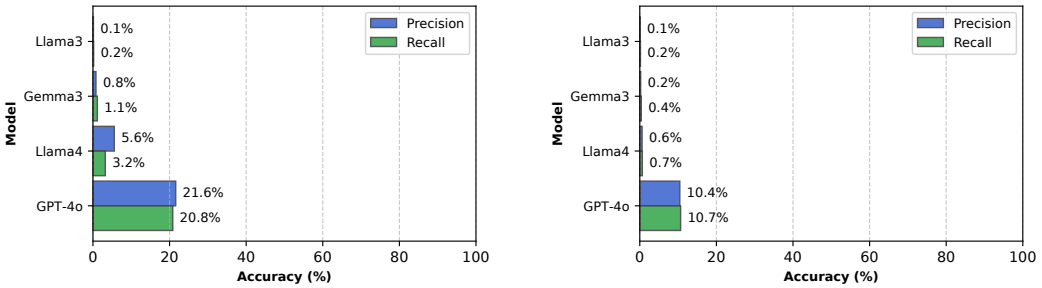
configuring CS (Figure 2b). They are also message-dependent, meaning their values must be set based on data received from specific messages (Figure 2c).

While field definitions, including types and allowable values, are provided in table, OCPP specifies valid message sequences and context-dependent field and variable values using use case diagrams. The specification includes 129 diagrams covering diverse EV-charging scenarios (e.g., remote charging, smart charging, and firmware updates), collectively referencing every OCPP message at least once. Figure 2f illustrates the remote charging scenario. Upon a user initiated request, the CSMS sends a `RequestStartTransactionRequest` to the CS, which verifies feasibility and replies with a `RequestStartTransactionResponse` message, setting the status field to `Accepted` if charging can proceed. Because `AuthorizeRemoteStart` is set to `true`, authentication is performed using `AuthorizeRequest` and `AuthorizeResponse` messages. Furthermore, with `TxStartPoint` configured as `Authorized`, the CS sends a `TransactionEventRequest` to inform the CSMS of session initiation only when the user is authenticated, and charging begins upon receiving a `TransactionEventResponse` from the CSMS. As illustrated in this example, the use case diagram clearly presents the valid OCPP message flow between the CSMS and CS, along with the scenario- and state-dependent variables and field values.

2.4 Challenges in OCPP Protocol Fuzzing

There are four main challenges in fuzzing the OCPP protocol.

C1. Complexity of message structure. OCPP messages exhibit a highly complex and hierarchical structure. For example, the `TransactionEventRequest` message in Figure 2a contains 12 top-level fields; however, when nested fields (*e.g.*, fields within class types) are considered, the total expands to 48 fields. This structural complexity makes it difficult to generate valid and effective test cases without an understanding of the message structure.



(a) Results of message structure inference

(b) Results of class structure inference

Fig. 3. Evaluation of LLMs on understanding the OCPP protocol.

C2. Dependencies in messages. Many OCPP fields and messages are interdependent as shown in Figure 2. For instance, identifiers such as `idToken` or `remoteStartId` are introduced in one message but referenced in subsequent ones, while certain field values are only valid if other configuration variables or prior responses satisfy specific constraints. These dependencies create a combinatorial explosion in valid message sequences, making naive fuzzing approaches ineffective.

C3. Stateful protocol. OCPP is inherently stateful, with valid message exchanges determined by the runtime states of both the CS and CSMS. Use case diagrams specify which messages are valid under particular conditions (*e.g.*, ongoing transactions, authorization status, or charger availability). Fuzzing without accurate state modeling leads to invalid inputs that fail early, reducing coverage of critical behaviors.

C4. Unstructured specification. Although the OCPP specification is extensive, it is primarily expressed in natural language, tables, and diagrams rather than in a formal specification. This lack of formalism hinders the automatic derivation of fuzzing strategies, requiring significant manual effort to extract protocol information. Also, as illustrated in Figure 2, understanding the protocol often requires cross-referencing multiple subsections. For example, to interpret the structure of `TransactionType` used in `TransactionEventRequest`, *subsection 2.48. TransactionType* must be consulted. Similarly, valid message sequences are only specified in use case diagrams rather than in tables, while details about variables such as `TxStartPoint` or `AuthorizeRemoteStart` are found exclusively in tables. This fragmentation makes understanding the OCPP specification both difficult and error-prone. In particular, based on the OCPP 2.0.1 specification, the entire document spans 1,693 pages and is accompanied by 135 separate data files (csv, xlsx, json), making its interpretation a highly complex task.

3 Large Language Model Guided Fuzzing

To address the challenges outlined in section 2, we leverage a large language model.

3.1 Large Language Models

LLMs are machine learning models trained on massive amounts of text data to understand and generate natural language. Recently, LLMs have evolved beyond simple text generation, demonstrating capabilities in extracting structured information from unstructured documents, inferring domain-specific rules, and generating context-aware responses [19, 31]. These advancements have drawn significant attention in practical domains such as technical specification interpretation and software testing, leading to a rapid expansion of LLM applications across nearly all fields [12, 27, 33]. However, LLMs still face limitations when dealing with highly specialized domains or knowledge that is underrepresented in their training data. To overcome this challenge, recent studies have

explored augmenting LLMs with external knowledge or domain-specific data, a direction commonly referred to as Augmented Language Models (ALMs) [18]. ALMs aim to complement the generalization ability of LLMs by incorporating domain knowledge, thereby enabling more accurate and reliable responses in specialized applications.

3.2 LLM Understanding of the OCPP Protocol

We first evaluate the LLM's capability to understand the OCPP protocol to determine its potential usefulness in the fuzzing process. Specifically, we designed two tasks for the LLM. ① Message structure inference: the LLM is given the name of an OCPP message and asked to output its field names along with their corresponding types. ② Class structure inference: the LLM is given the name of a class and asked to output its field names and corresponding types. The LLM's output is considered correct only if both the field names and their types are accurately inferred.

Figure 3 presents the evaluation results for four LLM models: GPT-4o [24], Gemma3 [13], Llama3 [20], and Llama4 [21]. Across both tasks, GPT-4o achieved the highest precision. For message structure inference, GPT-4o attained a precision of 21.6% and a recall of 20.8% over 128 OCPP messages. For class structure inference, it achieved 10.4% precision and 10.7% recall over 54 classes. These results indicate that LLMs do not possess sufficient understanding of the OCPP protocol. Therefore, we augment them with the OCPP specification before leveraging them for fuzzing.

4 OCPPuzz Design

This section provides an overview of OCPPuzz architecture, a fuzzing framework for evaluating CSMS implementations.

Overview. Figure 4 illustrates the overall approach of the OCPPuzz framework, which addresses the challenges outlined in subsection 2.4 as follows. First, to address C1, OCPPuzz extracts message structures by syntactically parsing the OCPP specification in both PDF and JSON formats (subsection 4.1). Second, for C2, we derive dependencies among messages, fields, and variables from natural language descriptions in the specification using the LLM, combined with an iterative feedback loop for improved accuracy (subsection 4.2 & subsection 4.3). Third, for C3, OCPPuzz performs CSMS fuzzing by accounting for CS–CSMS states, extracting valid message sequences, variables, and field values specific to each state of these entities from both the specification's tables and use case diagrams (subsection 4.4 & subsection 4.5). Finally, for C4, we provide the table of contents of the OCPP specification to the LLM so that it can automatically locate missing references. Moreover, to overcome token limitations, we leverage AWS S3 [5] to upload relevant page data and provide pre-signed URLs to the model. To ensure the reliability of the extracted data, OCPPuzz employs a self-correction mechanism that feeds parsing errors back into subsequent prompts to rectify format inconsistencies. Furthermore, to mitigate subjective bias and minimize human errors, we adopted a cross-validation procedure where multiple authors independently verified cases requiring manual inspection. The following subsections describe each component of OCPPuzz in detail.

4.1 Parser

The OCPP specification is available in two formats: JSON and PDF. In the JSON format, each message structure is specified in a separate JSON Schema file [15], which provides message descriptions and detailed field information, including names, types, and constraints on valid values (e.g., string length). The PDF specification [3] also defines message structures but as discussed in subsection 2.3, introduces more sophisticated constraints on field values, such as inter-message and variable dependencies that are not captured in the JSON version. Consequently, the *Parser*

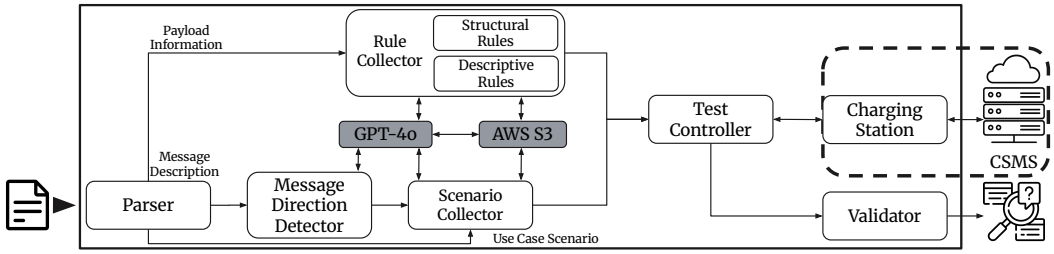


Fig. 4. Overview of OCPPuzz framework.

module extracts message structures from both formats. Parsing the JSON Schema is straightforward because it is written in a structured format. In contrast, extracting message structures from the PDF specification is non-trivial. To address this, we manually reviewed the PDF specification to identify relevant sections and then developed heuristic rules to extract information.

Table 1 summarizes the OCPP specification sections parsed by our system. For the *Table of Contents*, the *Parser* analyzes word positions and indentation to build a hierarchical tree of section titles, levels, and page numbers; this data is used by the *Rule Collector* (RC) and *Scenario Collector* (SC) to retrieve additional context. The *Use Case & Requirements* section provides scenario data for resolving message dependencies; tables containing a `scenario description` column are mapped to adjacent figures and passed to RC and SC. The *Messages, Datatypes, and Enumerations* section (Figure 2) defines message directions and constraints. It is parsed by detecting section boundaries, classifying content by heading depth, and extracting repeated table patterns, and the resulting data is used by the *Message Direction Detector* (MDD), RC, and SC. The *Referenced Components and Variables* section, which describes variables in natural language, is extracted using similar table-based heuristics and provided to RC and SC. Finally, the *Security Events* and *Standardized Units of Measure* sections define predefined event types (e.g., invalid firmware signature) and measurement units (e.g., kWh, Hertz), which are parsed to populate enumerations for the `SecurityEventNotificationRequest` message's type field. The *Parser* successfully extracted 128 messages, 54 classes, 88 enumerations, 8 primitive datatypes, 145 variables, 20 security events, and 33 standardized units of measure.

Table 1. Parsed OCPP specification sections.

Section Name	Module
Table of Contents	RC, SC
Use Case & Requirements	RC, SC
Messages, Datatypes & Enumeration	MDD, RC, SC
Primitive Datatypes	RC
Referenced Components and Variables	RC, SC
Security Events	RC
Standardized Units of Measure	RC

4.2 Message Direction Detector

As shown in Figure 2a, OCPP messages define both the sender (`from`) and the recipient (`to`) in unstructured natural language, and transforming this information into structured data requires considerable effort. To interpret such unstructured natural language, we manually analyzed the specification and identified three types of message descriptions. The first type is `well-defined descriptions`, as illustrated in Figure 2. The second type is `dependent descriptions`, such as "Response to a DeleteCertificateRequest." which refer to other messages. The final type is `missing descriptions`, where no direction-related information is provided.

To automate the extraction of message directions for these three types, we adopted an LLM. As illustrated in Figure 5, we designed the prompt to effectively handle the three categories of message descriptions identified through manual analysis. The system role was formulated to provide explicit instructions for consistently extracting message directions, while the developer role

specified the LLM's handling strategies for each of the three types. For `missing descriptions`, we specified signatures indicating that extraction was not possible, and for `dependent descriptions`, the LLM was instructed to additionally request a dependent description whenever a dependency relationship between messages was identified.

Examples for these three cases were included in the prompt, and in-context few-shot learning [28] was applied in constructing the prompt. In this setup, the user role receives the message descriptions parsed in subsection 4.1 as input. Finally, to ensure consistent and deterministic responses, we set the `temperature` to `0`. In cases where parsing errors occurred in OCPPuzz, the algorithm recorded the error information in the `context` and provided it to subsequent iterations, thereby preventing the same mistakes from recurring. To further support this process, we implemented a retry mechanism (e.g., up to 10 attempts). The remaining failure cases were manually analyzed to identify their causes, which enabled the detection of specification bugs.

Message Direction Detector Instruction Prompt

System

Extract Message Direction.

Developer

Rules:

- If undeterminable, set both to `End Signature`.
- Request reference descriptions when needed.

Context:

- + *{Previous Error Contents}* [exists]
- + *{Reference Descriptions}* [requested]

Assistant

Example Input/Output Pairs

User

Message Description

Fig. 5. Prompt structure for MDD.

4.3 Rule Collector

Effective fuzzing requires accurately extracting constraint information for each field in the OCPP message payload. We present a rule collection strategy that combines two complementary approaches: Structural Rules and Descriptive Rules.

OCPPuzz employs a rule collection data structure that can encapsulate multiple dependencies. Each rule is represented by an optional `Pre-Condition` and a `Constraint`. The `Pre-Condition` comprises the elements `Target`, `Attribute`, `Operator`, and `Values`. The `Target` denotes the entity on which the dependency is imposed. As described in Figure 2, this can represent dependencies on previous messages, variable dependencies, or field-level dependencies, and thus serves as the field identifying such references. The `Attribute` specifies the property to be examined, such as `required`, `type`, or `maxLength`. The `Operator` defines the logical relation to be applied, such as `equal` or `notEqual`. Finally, the `Values` represent the concrete values that satisfy the `Pre-Condition` and make it evaluate to true. The `Constraint` denotes the restrictions that apply when the corresponding `Pre-Condition` is satisfied. Its structure is identical to that of the `Pre-Condition`, except that it omits the `Target` element. We classified the constraints embedded in the extracted rules into four high-level categories: **Type** – field values must conform to basic data types such as string, integer, or decimal, ensuring type safety. **Presence** – specifies whether a field must be included in a message. **Semantics** – restricts acceptable values (e.g., enumerations, Base64/PEM formats). **Cardinality** – limits field length, byte size, or array items.

Rule Collector Instruction Prompt

System

Extract rules.

Developer

Rules:

- Avoid rule duplication.
- Respond `End Signature` when no rules remain.
- Request the corresponding page when information from the required `Table of Contents` is needed.

Context:

- *Reference Variables*
- *Table of Contents*
- + *{Saved Data (Attributes, Operators)}* [exists]
- + *{Previous Responses}* [exists]
- + *{Previous Error Contents}* [exists]

Assistant

Example Input/Output Pairs

User

Message or Class Field Descriptions
+ *{Additional Page Images}* [requested]

Fig. 6. Prompt structure for RC.

4.3.1 Structural Rule. Structural Rules refer to standardized constraints that can be extracted without the assistance of LLM. As mentioned in subsection 2.3, these rules are derived from both

the pdf documentation and the JSON Schema file. Typical constraints include field mandatory status, data type definitions, string length limits, array size restrictions, valid sets of field values, numeric ranges, string format requirements, and array element type restrictions.

During our analysis, we observed inconsistencies between the JSON and PDF specifications. For example, the minimum value of a field is defined as 0 in the PDF but as 1 in the JSON version. We also identified discrepancies in field descriptions across the two sources. Such inconsistencies can lead to developer errors and interoperability issues; we have reported these problems to the OCA.

4.3.2 Descriptive Rule. Figure 6 shows the prompt used in the *Rule Collector*. The `System` role defines the core task by instructing the LLM to extract rules. The `Developer` role aims to prevent duplication of previous responses, and instructing the LLM to request additional data when necessary. To maintain consistent terminology across responses, we provided `Saved Data (attributes,operators)`, introduced `Reference Variables` to distinguish variable names from natural language descriptions, and supplied the `Table of Contents` to assist in navigating the specification. The *Rule Collector* extracts descriptive rules from messages and classes. For the same message or class, rules from previous responses are included in the prompt to avoid duplicate rule extraction, and error information from prior responses is incorporated to prevent repeated mistakes.

The `Assistant` role provided diverse `Example Input/Output Pairs`, while the `User` role supplied field-level descriptions of each message and class, supplemented with descriptions from the JSON schema.

The extraction process was repeated until the LLM generated the `End Signature`, indicating that no further rules could be extracted. Additionally, we employed two approaches to validate the correctness of the extracted rules. If an extracted descriptive rule matched a structured rule, it was considered correct; otherwise, it was manually validated.

4.4 Scenario Collector

The *Scenario Collector* is a component designed to extract valid message flows from the OCPP specification. As shown in Figure 2f, OCPP messages often exhibit interdependencies. For example, configuration variables may need to be set in advance, and prerequisite messages must be exchanged beforehand. Thus, to construct valid and state-dependent tests, the system must first configure the necessary variables to reproduce the same state and establish message exchanges between the CS and the CSMS. In this way, specific test rules are valid only when the CS reaches a certain precondition state, making this state a mandatory requirement for validation.

Scenario Collector Instruction Prompt

System
Extract scenario flows.

Developer
Rules:
- Avoid scenario duplication.
- Combine descriptions and figures into one unit.
- Insert implicit messages from yellow annotations.
- Add prerequisites or reference values.
- Include pre-configuration variables as required
- Request the corresponding page when information from the required `Table of Contents` is needed.
- Return `End Signature` if no valid OCPP flows.

Context:
- *Message Directions*
- *Reference Variables*
- *Table of Contents*
- *+ {Previous Responses}* [exists]
- *+ {Previous Error Contents}* [exists]

Assistant
Example Input/Output Pairs

User
Scenario Figures Page Images
+ {Additional Page Images} [requested]

Fig. 7. Prompt structure for SC.

To automate this process using the LLM, we designed a prompt that effectively extracts valid states and configurations from scenario descriptions and use case diagrams (Figure 7). The `System` role provides explicit instructions for the LLM to consistently extract scenario flows from OCPP descriptions and figures. The `Developer` role consists of fixed guidelines (e.g., `Rules`) and a `Context` that either references specific data or previous responses.

The `Rules` are defined to combine each use case diagrams with its corresponding description, to infer messages from unstructured annotations such as yellow notes in the diagrams, and to include prerequisites when they exist. If no valid OCPP flow can be identified, the process terminates by returning the `End Signature`.

The `Context` includes `Message Directions` collected in subsection 4.2 to support direction analysis and provides `Reference Variables` so that the LLM can determine whether text in figures corresponds to variables. It also incorporates the `Table of Contents`, enabling the LLM to flexibly request sections when additional information is needed for scenario analysis. `Previous Responses` and `Previous Error Contents` are included in subsequent queries to improve consistency. To further support this process, the *Scenario Collector* applies a duplication-prevention mechanism that avoids unnecessary repetition or accumulation of identical outputs, thereby mitigating token limitation issues.

The `Assistant` role provides `Example Input/Output Pairs` to improve the consistency of LLM responses, while the `User` provides scenario figure page images as input and, when requested in `Previous Responses`, additional page images as well. For the extracted scenario data, manual inspection was performed only on items inconsistent with the data previously collected by our framework, including messages, classes, enums, variables, and message directions.

4.5 Test Controller

The *Test Controller* constructs valid message sequences to establish valid states of the CS and CSMS and deliver mutated test messages to the target CSMS for testing.

4.5.1 Scenario-Based Message Sequence Construction. Algorithm 1 describes how the *Test Controller* generates valid message sequences to establish valid states of CS and CSMS, and produce mutated messages for testing, based on the scenario data S collected by the *Scenario Collector* described in subsection 4.4. Since the target system under test is the CSMS, the algorithm considers only request messages satisfying `isCSToCSMSRequest`(S_i) as target messages. For each such message S_i, the algorithm iterates over all rule combinations obtained from rule data (R) by calling S_i.`getRuleCombinations`(R), and executes a separate test set for each combination. At the beginning of a test set, a connection to the CSMS is established through `createConCSMS(C)`, and the execution context c is initialized. If S.`variables` $\neq \emptyset$, the algorithm requests the CSMS to configure these variables by invoking `con.setVariables`(S.`variables`). Then, for all messages S_j where $0 \leq j \leq i$, the algorithm generates and sends payloads so that the state immediately prior to S_i is achieved. If `isCSToCSMSRequest`(S_j) holds and $j \neq i$, a valid message is generated using `genValidMessage`(S_j, R, c) and sent to the CSMS. If $j = i$, the algorithm generates a mutated message by applying the current rule combination via `genMutateMessage`(S_j, cmb, R, c). It applies an inverted constraint from *cmb* to a target field, ensuring the payload violates exactly one rule for a precise evaluation of the CSMS's response.

Algorithm 1 Scenario based mutation test

```
1: Input: Scenario Data S, Rule Data R, Target System Connection Info C
2: for i = 0 → S.length do
3:     if isCSToCSMSRequest(S_i) then
4:         for all cmb ∈ S_i.getRuleCombinations(R) do
5:             con ← createConCSMS(C)          ▷ create connection target system
6:             c ← ∅                           ▷ initalize test execution context
7:             if S.variables.length ≠ ∅ then
8:                 c.update(con.setVariables(S.variables))
9:             for j = 0 → i do
10:                if isCSToCSMSRequest(S_j) then
11:                    if j ≠ i then
12:                        c.update(con.send(genValidMessage(S_j, R, c)))
13:                    else
14:                        c.update(con.send(genMutateMessage(S_j, cmb, R, c)))
15:                else if isCSMSToCSRequest(S_j) then
16:                    c.update(con.send(callTriggerApi(S_j, R, c)))
17:            end for
18:        end for
19: end for
```

If instead `isCSMSToCSRequest`(S_j) holds, the system invokes `callTriggerApi`(S_j, R, c) to trigger a CSMS-initiated message and responds accordingly.

4.5.2 Payload Generation. The *Test Controller* generates valid and invalid mutated values for each field based on Algorithm 2 and assembles them into a valid payload. For each field, the algorithm iterates over the set of field rules R and checks their applicability. If a rule specifies a `Pre-Condition`, its validity is verified using `checkPreCondition`(r, C); only if the condition holds is the rule considered in value generation.

When a rule is selected for violation, other rules are preserved whenever possible, and the payload value is generated according to the following constraint categories. For `Type` constraints, the algorithm produces values of a random data type different from the expected one. For `Presence` constraints, if a field is mandatory, the generated value is empty, whereas if the field must be omitted, it is forcibly included. For `Semantics` constraints, the system inserts semantically invalid data; for example, if a field requires a specific character set, strings containing characters outside that set are generated. For `Cardinality` constraints, such as array size limits (e.g., `maxItems` = 4), the algorithm generates an array longer than the specified bound.

Algorithm 2 Field value generation from rule combinations

```
Input: Execution Context C, Combination CMB, Field rules R
Output: Field value V
RC ← createRuleConfig()
for all r ∈ R do
    if ¬ checkPreCondition(r, C) then
        continue
    if isReverseRuleCombination(r, CMB) then
        r.reverse()
    RC.add(r)
end for
return genRandomValue(RC, C)
```

Finally, all applicable rules are collected into the configuration object RC via `RC.add`(r), and the resulting field value is returned by `genRandomValue`(RC, C). In this way, OCPPuzz attempts to activate as many rules as possible whenever their conditions are met. When conflicting constraints arise, a priority-based approach is applied so that the generated values satisfy as many constraints as possible while ensuring internal consistency.

4.6 Validator

The *Validator* assesses system responses to test cases where individual constraints have been inverted during rule combination. As described in Algorithm 2, payloads violating specific constraints are generated and sent to the target CSMS, which responds with either a `CALLRESULT` or a `CALLERROR` message as shown in Figure 1. The *Validator* then verifies whether the response matches the expected outcome. In particular, it determines whether the system correctly returns an error for violated constraints or erroneously accepts invalid data. This process enables a quantitative evaluation of the robustness and consistency of the CSMS's validation logic, serving as an important indicator for identifying potential weaknesses and assessing protocol compliance in real-world environments.

5 Evaluation

To evaluate the effectiveness of OCPPuzz and the validation capability of CSMS open-source implementations based on OCPP 2.0.1, the most widely adopted version with available implementations at the time of our study, we selected four well-known projects: CitrineOS [9], OCPP.Core [11], MaEVe [25], and ocpp-go [17]. Given that OCPP is a relatively niche protocol and its behavior varies significantly across versions, the pool of available open-source implementations was limited. In our selection process, we excluded projects that only provide interfaces without an internal implementation. Notably, OCPP.Core has been deployed in actual field operations using KEBA P30 x-series charge points [16], demonstrating its use in real-world CSMS, while CitrineOS is developed based on the official OCTT [4], providing strong conformance with the standard.

To assess the effectiveness of OCPPuzz, we investigate the following research questions:

- **RQ1:** How well can OCPPuzz extract information from the OCPP specification?
- **RQ2:** How effective is the generation of mutation payloads?
- **RQ3:** What categories of bugs can OCPPuzz identify?
- **RQ4:** How effective is the test case generation strategy?

5.1 RQ1: How well can OCPPuzz extract information from the OCPP specification?

5.1.1 Message Direction Extraction. Based on our manual analysis of all 128 OCPP 2.0.1 messages, OCPPuzz correctly extracted the message directions for 117 messages, achieving an accuracy of 91.4%. The remaining 11 messages failed due to missing or misaligned descriptions in the official specification rather than errors in OCPPuzz. These inconsistencies, corresponding to a failure rate of 8.59%, were formally reported to the OCA (`OCA-3687`). The OCA responded that these issues will be addressed in a future Errata release. This demonstrates not only the accuracy of the proposed extraction approach but also its utility in improving the quality of the specification.

5.1.2 Rule Extraction. OCPPuzz analyzed 25 request messages transmitted from the CS to the CSMS, extracting 1,261 structural rules and 918 descriptive rules in total. On average, 7.04 classes are defined per message, and each message contains approximately 50.44 structural rules. Because descriptive rules were extracted using an LLM, we manually verified them; 865 out of 918 rules (94.2%) were confirmed to be valid. These results demonstrate the practical usefulness of LLM-based rule extraction for test automation. Notably, no type constraints were identified in the descriptive rules; all such constraints were captured by the structural rule collection, indicating that natural-language descriptions rarely specify data types explicitly.

In total, OCPPuzz collected 2,126 rules (1,261 structural and valid 865 descriptive rules), and descriptive rules account for 40.7% of all constraints. This shows that descriptive constraints are no less important than structural ones in OCPP protocol testing.

5.1.3 Scenario Extraction. To evaluate the accuracy of the scenario extraction results, the `ground truth` dataset was constructed using data parsed from the key sections of the OCPP specification listed in Table 1, consisting of 54 classes, 88 enumerations, and 145 variables, along with the transmission directions of 117 messages obtained by OCPPuzz, as described in subsubsection 5.1.1.

The extracted variable identifiers, transmission directions, constant values, and datatype specifications in the scenarios were then compared against this dataset to assess correctness and filter out inconsistencies. As a result of this filtering, a total of 105 valid scenarios were extracted.

During the validation process, several representative error types and their occurrence counts were identified. First, there were 6 cases where a variable name did not exist in the reference variable list, and 12 cases where the message direction in the scenario did not match the pre-collected direction information. In addition, 26 cases occurred where field names used in the payload was not defined in the message specification, and 8 cases where a string field of the `enum` type contained a value not defined in the specification. Finally, 15 cases were identified where the field's data type did not match the type defined in the message schema.

All 67 discrepancies were manually reviewed, which corresponded to 14 unique inconsistencies in the OCPP specification. These findings were formally submitted to the OCA (`OCA-6287`), contributing to improving the accuracy and consistency of the standard.

Table 2. Mismatch response rates observed for valid and invalid OCPP test messages.

Project	Valid		Invalid					Total
	Mismatch	Timeout	Mismatch				Timeout	
			Type	Presence	Semantics	Cardinality		
CitrineOS	1,691/101,022 (1.7%)	1,691	8,661/34,872 (24.8%)	1,718/13,238 (13.0%)	5,019/10,212 (49.1%)	6,120/14,937 (41.0%)	304	23,209/174,281 (13.3%)
MaEVe	54/101,022 (0.1%)	0	1/34,872 (0.0%)	1,699/13,238 (12.8%)	5,549/10,212 (54.3%)	5,703/14,937 (38.2%)	732	13,006/174,281 (7.5%)
ocpp-go	40,624/101,022 (40.2%)	0	10,990/34,872 (31.5%)	3,895/13,238 (29.4%)	2,930/10,212 (28.7%)	4,803/14,937 (32.2%)	3,374	63,242/174,281 (36.3%)
OCPP.Core	56,580/101,022 (56.0%)	0	63/34,872 (0.2%)	555/13,238 (4.2%)	1,786/10,212 (17.5%)	872/14,937 (5.8%)	0	59,856/174,281 (34.3%)

Mismatch is represented as $(n/N, \%)$ with n = mismatches, N = total cases. Timeout is set to 10s.

5.2 RQ2: How effective is the generation of mutation payloads?

Table 2 summarizes the alignment between test cases and expected responses, where valid messages are expected to yield valid responses and invalid mutated messages are expected to trigger errors. We manually analyzed four system responses (both valid and invalid) generated from unique test messages based on field-level inversion rules to verify whether the test messages were produced as intended and whether the target system returned a `match` or `mismatch` for the expected reason.

5.2.1 Valid Cases. OCPP.Core exhibited the highest mismatch rate of 56.0%. Two major causes were identified: 1) Timezone offset limitation: OCPP.Core only supports timezone offsets up to ±14 hours. When requests exceeded this range, internal errors occurred. 2) Restricted scenario handling: the variable `TxStartPoint` is essential for the CS to decide transaction initiation, but OCPP.Core limits it to the `Authorized` value only. Any other valid value for `TxStartPoint` results in error responses. For ocpp-go, the mismatch rate was 40.2%. Although the specification did not impose such constraints, ocpp-go enforced stricter validation: fields with item size 0 were rejected, and identifiers or numerical fields were validated to ensure values greater than zero. For CitrineOS, a non-compliant behavior was observed where `CALLRESULT` messages incorrectly contained `CALLERROR` contents. This led to ambiguity in whether the CS should interpret the server's response as success or failure, potentially resulting in improper handling of charging or authentication requests. Moreover, timeouts were most frequent in CitrineOS (1,691 cases), caused by the absence of any response to unsupported messages. Such behavior risks leaving the CS in indefinite waiting states, which is a severe operational issue.

5.2.2 Invalid Cases. We observed that semantic constraints caused the highest mismatch rates in three CSMS implementations: MaEVe (54.3%), CitrineOS (49.1%), and OCPP.Core (17.5%). In contrast, ocpp-go exhibited mismatches that were distributed approximately evenly across the four constraint categories.

Because semantic constraints are extracted from natural-language descriptions in the specification, these results suggest that open-source CSMS implementations are largely insufficient in validating such natural-language constraints.

ocpp-go exhibited the highest number of timeouts, which were caused by its inability to process payloads containing invalid enum string values. In addition, we confirmed that some invalid test cases resulted in error responses due to pre-validation checks that were also present in valid-case mismatches, rather than the inversion rules themselves.

5.2.3 Summary of Findings. In total, we identified unique mismatches in each system as follows: 328 cases in CitrineOS [9], 164 cases in OCPP.Core [11], 113 cases in MaEVe [25], and 325 cases in ocpp-go [17]. Among these, 492 mismatches were acknowledged by the vendors. A CitrineOS contributor remarked that **"the level of detail here is much higher than in the OCA's own testing tool. They don't test for primitive data type checking."** The contributor further noted

that the JSON schemas produced by the OCA for the protocol provide no validation of those data types either. This observation highlights both the limitations of existing testing tools and the contribution of our approach in uncovering deeper compliance issues.

5.3 RQ3: What categories of bugs can OCPPuzz identify?

5.3.1 Root Cause Analysis of Validation Failures. Table 3 summarizes the validation failures identified in various open-source implementations using OCPPuzz. Based on the analysis presented in subsection 5.2, it is not feasible to enumerate all observed failure cases; therefore, for each category, we selected a representative mismatch to illustrate the corresponding validation issue.

Type. `TR-1` represents the rule for type specification that is structurally extracted for each field. For example, if a field is defined as `string`, our framework inverts this constraint by generating inputs with other data types (e.g., `integer`, `null`, or `array`) to test the robustness of the implementation. In conclusion, MaEVe was the only implementation that successfully handled all type violation cases.

Persence. `PR-1` to `PR-12` correspond to rules governed by presence constraints. If a field is marked as `Required`, our framework generates payloads by omitting its value; if a field must be absent, we enforce its inclusion in the payload. Specifically, `PR-1` denotes unconditional required fields, `PR-2` to `PR-8` represent field dependency rules where the presence constraint depends on the value of another field, and `PR-10` to `PR-12` capture message dependency rules where the constraint is enforced by the context of prior message exchanges.

In `PR-5`, when the CS publishes a firmware, the CSMS must verify the deployment using the URI provided in the `location` field. If the CSMS fails to validate this field, the update authentication procedure can be bypassed, allowing adversaries to distribute malicious firmware or forge the update process without detection. This vulnerability demonstrates a practical attack vector that directly compromises the integrity of the firmware update process.

Finding 1: *Ignoring presence and inter-field dependency rules enables attackers to bypass update validation and distribute malicious firmware undetected.*

Semantics. `SR-1` to `SR-14` correspond to rules governed by semantic constraints. These rules enforce specific values or formats, and our framework tests violations by generating random values or inputs that do not conform to the specified format. `SR-1` and `SR-2` are unconditional rules applied to primitive data types: `SR-1` applies to all fields using `identifierString` as an identifier, while `SR-2` enforces the format of `dateTime` values. `SR-3` to `SR-5` address constraints in the `IdTokenType` class, where the presence and format of the `idToken` field depend on the `type` field. `SR-6` to `SR-8` refer to constraints described in the appendix, and `SR-9` to `SR-12` define rules on the format of signatures in energy metering data transmissions. `SR-13` and `SR-14` regulate the format of authentication data in certificate verification, `SR-15` to `SR-20` specify constraints where values exchanged in prior messages must be reused, and `SR-21` to `SR-22` enforce field/variable dependencies that determine the measurement basis for energy metering values.

`SR-8` is a field that reports security events occurring in the CS. If this specification is not followed, the CSMS cannot detect security events from the CS, potentially enabling attacks such as DoS or evasion of security monitoring.

Finding 2: *Implementations ignore security event reporting, allowing critical incidents to go undetected and enabling DoS or monitoring bypass.*

`SR-9` to `SR-12` reveal missing validation of signatures in the process of transmitting energy metering values, and `SR-21` to `SR-22` expose missing validation of compliance with metering configuration. As a result, an attacker could tamper with metering data in transit, omit required signatures, or

Table 3. Overview of missing validations in OCPP implementations.

Category	Reverse Rule	Report
Type	`[TR-1]` (type = *T*): value must be of type *T*	[C*,O*,G]
Presence	`[PR-1]` required (MUST be present)	[G]
	`[PR-2]` (Ⓒ*ChargingSchedulePeriodType.numberPhases* ≠ 1): Ⓒ*ChargingSchedulePeriodType.phaseToUse* field must be omitted	[C*,O*]
	`[PR-3]` (Ⓒ*VariableAttributeType.mutability* ≠ WriteOnly): Ⓒ*VariableAttributeType.value* field must be required	[C*,M,G]
	`[PR-4]` (Ⓒ*VariableCharacteristicsType.dataType* in ['OptionList', 'MemberList', 'SequenceList']): Ⓒ*VariableAttributeType.value* field must be required	[C*,M,G]
	`[PR-5]` (Ⓡ*PublishFirmwareStatusNotification.status* = Published): Ⓡ*PublishFirmwareStatusNotification.location* field must be required	[C*,G]
	`[PR-6]` (Ⓒ*ChargingProfileType.chargingProfilePurpose* ≠ TxProfile): Ⓡ*PublishFirmwareStatusNotification.location* field must be omitted	[C*]
	`[PR-7]` (Ⓡ*TransactionEvent.triggerReason* = ChargingStateChanged): Ⓒ*TransactionType.chargingState* field must be required	[O*,M,G]
	`[PR-8]` (Ⓡ*TransactionEvent.eventType* = Started): Ⓒ*TransactionType.chargingState* field must be required	[C*,O*,M,G]
	`[PR-9]` (Any Ⓡ*TransactionEvent* except the first one that occurs after the EV has connected): Ⓡ*TransactionEvent.evse* must be omitted	[C*,O*,M,G]
	`[PR-10]` (An Ⓡ*TransactionEvent.idToken* that has already been used for authorization in a Ⓡ TransactionEvent): Ⓡ*TransactionEvent.idToken* must not be used again	[C*,O*,M,G]
	`[PR-11]` (Ended by a Ⓡ*RequestStopTransaction* or a ⓇReset): Ⓡ*TransactionEvent.idToken* should not be present	[C*,O*,M,G]
	`[PR-12]` Ⓡ*TransactionEvent.reservationId* is only provided in the first ⓇTransactionEvent	[C*,O*,M,G]
Semantics	`[SR-1]` (type = identifierString): can only contain characters from `[A-Za-z0-9*_=:+|@.-]`	[C*,O*,M,G]
	`[SR-2]` (type = dateTime): shall follow the RFC3339 format	[O*,G]
	`[SR-3]` (Ⓒ*IdTokenType.type* = ISO14443): Ⓒ*IdTokenType.idToken* must be in hexadecimal format	[C*,O*,M,G]
	`[SR-4]` (Ⓒ*IdTokenType.type* = ISO15693): Ⓒ*IdTokenType.idToken* must be in hexadecimal format	[C*,O*,M,G]
	`[SR-5]` (Ⓒ*IdTokenType.type* = NoAuthorization): the Ⓒ*IdTokenType.idToken* must be empty	[C*,O*,M,G]
	`[SR-6]` Ⓒ*ComponentType.name* should use a value from Standardized Component Names	[C*,M,G]
	`[SR-7]` Ⓒ*VariableType.name* should use a value from Standardized Variable Names	[C*,G]
	`[SR-8]` Ⓡ*SecurityEventNotification.type* should use a value from Security Events list	[C*,M]
	`[SR-9]` Ⓒ*SignedMeterValueType.publicKey* Base64 encoding required	[C*,O*,M,G]
	`[SR-10]` Ⓒ*SignedMeterValueType.signedMeterData* Base64 encoding required	[C*,O*,M,G]
	`[SR-11]` Ⓡ*SignCertificate.csr* must be PEM-encoded	[C*,M]
	`[SR-12]` Ⓡ*SignCertificate.csr* RFC 2986 CSR format required	[C*,M]
	`[SR-13]` Ⓒ*OCSPRequestDataType.responderURL* URL format required	[C*,M]
	`[SR-14]` Ⓒ*OCSPRequestDataType.serialNumber* must be in hexadecimal format	[C*,M]
	`[SR-15]` Ⓡ*NotifyDisplayMessages.requestId* must match an earlier Ⓡ*GetDisplayMessages.requestId*	[C*,G]
	`[SR-16]` Ⓡ*NotifyMonitoringReport.requestId* must match an earlier Ⓡ*GetMonitoring.requestId*	[C*,G]
	`[SR-17]` Ⓡ*NotifyReport.requestId* must match an earlier Ⓡ*GetReport.requestId* or Ⓡ*GetBaseReport.requestId*	[C*,M,G]
	`[SR-18]` Ⓡ*PublishFirmwareStatusNotification.requestId* must match an earlier Ⓡ*PublishFirmware.requestId*	[C*,G]
	`[SR-19]` Ⓒ*ReportChargingProfiles.requestId* must match an earlier Ⓡ*GetChargingProfiles.requestId*	[C*]
	`[SR-20]` Ⓒ*TransactionType.remoteStartId* must match an earlier Ⓡ*RequestStartTransaction.remoteStartId*	[C*,O*,M,G]
	`[SR-21]` (Ⓡ*TransactionEvent.eventType* value is Started): meterValue configured by Ⓥ*SampledDataCtrlr.TxStartedMeasurands*	[C*,O*,M,G]
	`[SR-22]` (Ⓡ*TransactionEvent.eventType* value is Updated): meterValue configured by Ⓥ*SampledDataCtrlr.TxUpdatedMeasurands*	[C*,O*,M,G]
Cardinality	`[CR-1]` (type = string): value length must not exceed `maxLength`	[C*]
	`[CR-2]` (type = array): item size must not exceed `maxItems`	[C*]
	`[CR-3]` (type = integer): shall be represented as a 32-bit signed integer	[C*,M,G]
	`[CR-4]` (type = dateTime): shall have at most 3 decimal places	[C*,O*,M,G]
	`[CR-5]` (type = decimal): shall have at most 6 decimal places	[C*,O*,M,G]
	`[CR-6]` (Ⓒ*IdTokenType.type* = ISO14443): Ⓒ*IdTokenType.idToken* must be 4 or 7 bytes long	[C*,O*,M,G]
	`[CR-7]` (Ⓒ*IdTokenType.type* = ISO15693): Ⓒ*IdTokenType.idToken* must be 8 bytes long	[C*,O*,M,G]

Report C: CitrineOS [9] O: OCPP.Core [11] M: MaEVe [25] G: ocpp-go [17] *: Confirmed or Patched
Object Symbol Ⓡ: Request Message Ⓒ: Class Ⓥ: Variable

manipulate measurement bases, thereby compromising data integrity. Such vulnerabilities enable adversaries to obtain free charging or impose excessive billing on targeted users.

Finding 3: *Missing validation of metering signatures and configuration enables undetected tampering of energy data, leading to free charging or over-billing attacks.*

SR-13 and SR-14 indicate that insufficient validation of data formats in the OCSP-based certificate validation process can lead to improper handling of authentication data, allowing attackers to bypass or forge certificate-based authentication.

Finding 4: *Lack of semantic validation in the OCSP-based certificate validation process allows attackers to bypass or forge certificate-based authentication.*

SR-18 highlights the lack of validation for correlated request identifiers: the `requestId` in a `PublishFirmwareStatusNotification` must match a prior `PublishFirmwareRequest` issued by the CSMS. If this validation is omitted, a man-in-the-middle adversary could inject a malicious firmware update without detection, potentially resulting in DoS or privilege escalation.

Finding 5: *Lack of validation on firmware update identifiers allows adversaries to inject malicious firmware via man-in-the-middle attacks, potentially leading to device compromise or DoS.*

SR-20 corresponds to the case where the distinguishing value for a remote charging request from an authenticated user is not validated. In this case, a payload replay attack can be leveraged to enable unauthorized charging or manipulate billing.

Finding 6: *Failure to validate identifiers in remote charging requests, ignoring message dependency rules, enables replay attacks resulting in unauthorized charging or fraudulent billing.*

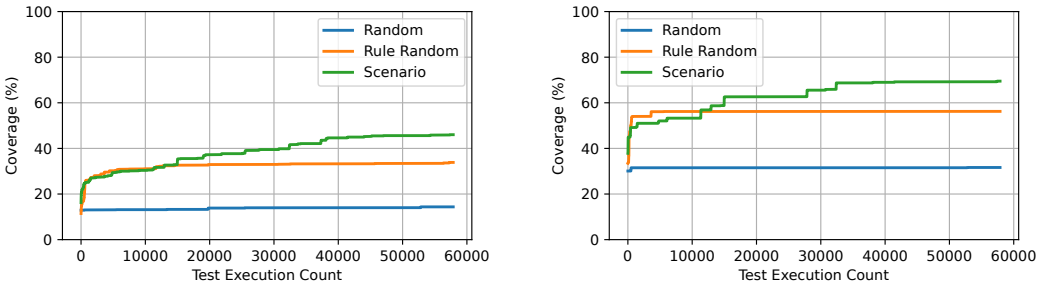
Cardinality. CR-1 to CR-7 correspond to rules governed by cardinality constraints. When the `maxLength` constraint is inverted, our framework generates random values that exceed the maximum length; similarly, for fields restricted to a maximum of three decimal places, we generate values with four or more decimal digits for testing. CR-1 to CR-5 apply to primitive data types and are enforced across all fields of the same type, while CR-6 and CR-7 specify field dependency rules in the `IdTokenType` class, where the length of the `idToken` value is determined by the associated `type` field.

Finding 7: *Ignoring field dependencies effectively nullifies ISO requirements and allows adversaries to compromise data integrity.*

5.3.2 Specification Violations and Documentation Errors. During our test result analysis, we identified ambiguities and constraint-related issues in seven fields. For example, `eventId`, `reservationId`, `amount`, and `seqNo` are not explicitly required by the specification to be positive, yet ocpp-go [17] enforces this constraint, leading to inconsistencies. In addition, fields such as `signingMethod`, `encodingMethod` and `SecurityEventNotificationRequest.type` lack clear enumerations or references for valid values, causing confusion for implementers and undermining interoperability. These issues were reported to the standardization body under issue ID `OCA-7110` and were agreed to be addressed following discussion.

5.4 RQ4: How effective is the test case generation strategy?

Figure 8 compares the cumulative code coverage (branch and statement) achieved by each testing strategy on CitrineOS [9]. The *Random* strategy generates purely random payloads for testing, while the *Rule Random* strategy generates payloads guided by the rules collected in subsection 4.3. Finally, the *Scenario* strategy builds upon the rules by following the scenarios collected in subsection 4.4, exploring all states reachable by the CS while systematically inverting the collected rules. The



(a) Branch coverage comparison

(b) Statement coverage comparison

Fig. 8. Cumulative coverage for three testing strategies.

results show that the *Random* strategy consistently achieves the lowest overall coverage, and although the *Rule Random* strategy explores faster in the early stages due to the absence of repeated messages, the *Scenario* strategy eventually surpasses both by discovering code paths unreachable by the other two strategies, leading to accelerated growth in the later stages and the highest final coverage.

These findings demonstrate the superiority of the scenario-based approach, particularly for protocols like OCPP that require handling of complex procedures and state dependencies. The results suggest that such an approach is essential for achieving comprehensive coverage in testing stateful systems.

6 Related Work

6.1 OCPP Verification

Since CSMS is responsible for security-critical tasks within EV ecosystem, such as authentication or billing, testing CSMS is essential. Several researchers have proposed testing methodologies that are applicable to OCPP implementations in CSMS. Sarieddine et al. [26] evaluated CSMS's implementation of OCPP against top OWASP security weaknesses such as injection and authentication vulnerabilities. Alcaraz et al. [2] demonstrated a systematic threat modeling based on STRIDE threat model by scrutinizing logical and structural components of OCPP. Boussaha et al. [7] implemented CheckOCPP, a protocol dissector that specializes in analyzing OCPP traffic and facilitating attacks such as MitM, replay and injection. They also proposed emuOCPP, a framework that can test vulnerabilities involving secure profiles[6]. Coppoletta et al. [10] devised OCPPStorm, which is also a fuzzer that uses structural specification to generate fuzzing input. OCPPStorm also utilizes user-defined sequences of valid messages to increase the coverage.

While previous contributions can emulate CS-CSMS network, manipulate intermediate packets, and conduct human-assisted fuzzification, no research exists that has used the LLM to implement automated testing of OCPP, addressing the challenge of unstructured specifications.

6.2 LLM Based Fuzzing

Today's research is rapidly adopting LLMs in software testing due to their excellence in generation and inference tasks [30]. Xia et al. [32] proposed Fuzz4All, a fuzzer in which LLM acts directly as an input generator and mutator, targeting diverse SUTs. On the other hand, Zhang et al. [34] proposed G^2Fuzz, which uses LLMs to generate grammar-aware input generators for non-textual formats. Ma et al. [19] proposed mGPTFuzz, the first fuzzer for the Matter IoT standard, which uses LLMs to automatically build a knowledge base from the specification to guide test input

generation. Similarly, Wang et al. [31] proposed LLMIF, which leverages LLMs to both parse the protocol specification and reason on input results of each fuzzing round to enrich fuzzing seed, demonstrating its approach on Zigbee.

OCPP includes constraints that require reaching specific states in order to perform testing, making state attainment a critical challenge. To address this, OCPPuzz incorporates use case figures from the unstructured specification into prompts to extract scenarios that can achieve these states. Such a methodology has not been proposed in prior research.

7 Discussion

7.1 Threats to Validity

7.1.1 Internal Validity. OCPPuzz is a specification-based automated testing framework; hence, any errors or inconsistencies in the OCPP specification may result in invalid test executions. Furthermore, the transformation of use case diagrams and textual descriptions into structured data using an LLM is subject to hallucinations, necessitating manual verification. In our experiments, we employed only the GPT-4o model, as the cost constraints of the GPT API prevented the evaluation with multiple models, and configuration parameters such as temperature were applied in a limited manner.

Additionally, we did not perform a direct empirical comparison between OCPPuzz and prior approaches discussed in subsection 6.1 due to tool availability limitations and fundamental differences in research goals and methodologies. EmuOCPP [6] focuses on reproducing and validating specific, pre-defined attack scenarios (e.g., MitM and CS impersonation), whereas OCPPuzz is a fuzzing framework that automatically explores a wide range of complex execution paths derived from the protocol specification. As a result, applying identical evaluation metrics would not provide a meaningful or fair comparison. Moreover, the broader testing scope and higher scenario complexity targeted by OCPPuzz would likely bias such a comparison in our favor, potentially compromising the objectivity of the evaluation. Regarding OCPPStorm [10], although it targets OCPP implementations, its research artifacts and key internal components-including the manually defined state machine fuzzer-are neither sufficiently described nor publicly available at the time of writing, preventing a reproducible comparison.

7.1.2 External Validity. At the time of writing, three OCPP versions (1.6, 2.0.1, and 2.1) coexist. Although our methodology can theoretically be applied to all versions, we restricted our evaluation to version 2.0.1. In addition, we conducted experiments on four open-source CSMS implementations. Although one of them has been actively developed and deployed in practice, our framework has not yet been directly tested against real-world commercial charging station servers. This limits the generalizability of our findings to all possible OCPP deployments.

7.2 Version Generalizability

Although our evaluation focused on OCPP 2.0.1, OCPPuzz is applicable to other OCPP versions. When applied to OCPP 2.1, it revealed 49 cases of missing, incorrect, or inconsistent message directions, which were reported to the OCA and are currently under discussion.

8 Conclusions

OCPP has long posed challenges for testing due to the complexity of its message structures, dependencies between messages, its stateful nature, and the unstructured specification. To address these challenges, this study proposes OCPPuzz, the first LLM-based fuzzing framework for OCPP. OCPPuzz utilizes the CS–CSMS state transitions described in OCPP use case diagrams to transform specification scenarios into testable data, ensuring that generated messages drive the system toward

specific states. Furthermore, based on message structures, field constraints, and dependency rules extracted from the OCPP specification, OCPPuzz systematically inverts field constraints to verify whether implementations properly validate them. We evaluated OCPPuzz on four open-source CSMS implementations. The experiments uncovered numerous violations of the OCPP specification that could lead to severe security issues, including DoS and free-charging attacks. In total, the corresponding vendors acknowledged **492 implementation bugs**, while **155 specification-related issues** were reported to the OCA, of which 78 were committed to fixes, 82 were acknowledged for further investigation and 49 did not respond. These results show that OCPPuzz effectively enhances EV charging security and contributes to improving the OCPP specification.

9 Data Availability

To foster further fuzzing research on OCPP implementations, we publicly release all data and artifacts necessary to reproduce the results of this paper [14].

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was partly supported by the three Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT; Ministry of Science and ICT) (No.2024-00337703; Development of satellite security vulnerability detection techniques using AI and specification-based automation tools, No.2024- 00398745; Proofs and responses against evidence tampering in the new digital environment, No. 2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes, IITP-2025-RS-2024-00437849; ICT Challenge and Advanced Network of HRD).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] The International Energy Agency. 2025. Trends in the electric car industry. https://www.iea.org/reports/global-ev-outlook-2025/trends-in-the-electric-car-industry-3. Accessed: 2025-09-11.

[2] C. Alcaraz, J. Cumplido, and A. Trivino. 2023. OCPP in the spotlight: threats and countermeasures for electric vehicle charging infrastructures 4.0. *International Journal of Information Security* (2023), 1395–1421. https://doi.org/10.1007/s10207-023-00698-8

[3] Open Charge Alliance. 2025. Open Charge Point Protocol (OCPP). https://openchargealliance.org/protocols/open-charge-point-protocol/ Accessed: 2025-08-23.

[4] Open Charge Alliance. 2025. Test Tool (OCTT). https://openchargealliance.org/test-tool/. Accessed: 2025-08-23.

[5] Amazon Web Services, Inc. 2025. Amazon Simple Storage Service (S3). https://aws.amazon.com/s3/. Accessed: 2025-08-23.

[6] S. Boussaha, V. F. Gómez, T. Barber, and D. Antonioli. 2025. EmuOCPP: Effective and Scalable OCPP Security and Privacy Testing. In *3rd USENIX Symposium on Vehicle Security and Privacy (VehicleSec '25)*. https://www.usenix.org/conference/vehiclesec25/presentation/boussaha

[7] S. Boussaha, V. F. Gómez, T. Barber, and D. Antonioli. 2025. CheckOCPP: Automatic OCPP Packet Dissection and Compliance Check. In *2025 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 335–342. https://doi.org/10.1109/EuroSPW67616.2025.00044

[8] Pew Research Center. 2024. Electric Vehicle Charging Infrastructure in the U.S. https://www.pewresearch.org/data-labs/2024/05/23/electric-vehicle-charging-infrastructure-in-the-u-s/. Accessed: 2025-09-11.

[9] CitrineOS Contributors. 2025. CitrineOS. https://github.com/citrineos/citrineos. Accessed: 2025-08-23.

[10] G. Coppoletta, R. Gjomemo, A. Kaur, N. Valizadeh, V. Venkatakrishnan, and O. Rana. 2024. OCPPStorm: A comprehensive fuzzing tool for OCPP implementations. In *Symposium on Vehicle Security and Privacy (VehicleSec '24)*. https://doi.org/10.14722/vehiclesec.2024.23069

[11] dallmann-consulting. 2025. OCPP.Core. https://github.com/dallmann-consulting/OCPP.Core. Accessed: 2025-08-23.

[12] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. 423–435. https://doi.org/10.1145/3597926.3598067

[13] Google DeepMind. 2025. Gemma 3. https://deepmind.google/models/gemma/gemma-3/. Accessed: 2025-08-23.
[14] J. Hong. 2025. OCPPuzz. https://github.com/jongchan-hong/OCPPuzz. Accessed: 2025-12-25.
[15] JSON Schema. 2017. JSON Schema: Draft-06 Specification. http://json-schema.org/draft-06/schema Accessed: 2025-08-20.
[16] KEBA Group AG. 2025. Kecontact P30 x-series. https://www.keba.com/en/emobility/products/x-series/x-series. Accessed: 2025-08-23.
[17] lorenzodonini. 2025. ocpp-go. https://github.com/lorenzodonini/ocpp-go. Accessed: 2025-08-23.
[18] Z. Luo, C. Xu, P. Zhao, X. Geng, C. Tao, J. Ma, Q. Lin, and D. Jiang. 2023. Augmented Large Language Models with Parametric Knowledge Guiding. https://doi.org/10.48550/arXiv.2305.04757
[19] X. Ma, L. Luo, Q. Zeng, and G. Mason. 2024. From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices. In *33rd USENIX Security Symposium (USENIX Security '24)*. https://www.usenix.org/conference/usenixsecurity24/presentation/ma-xiaoyue
[20] Meta. 2024. Llama 3. https://www.llama.com/models/llama-3/. Accessed: 2025-08-23.
[21] Meta. 2025. Llama 4. https://www.llama.com/models/llama-4/. Accessed: 2025-08-23.
[22] D. M. Ofek. 2023. A Case Study on the Importance of Security and Operations by Design – ABB ChargerSync Platform. https://c2a-sec.com/a-case-study-on-the-importance-of-security-validation-done-right-abb-chargersync-platform/ Accessed: 2025-08-29.
[23] Open Charge Alliance. 2025. Promoting Open Standards - Connecting the EV Industry. https://www.openchargealliance.org Accessed: 2025-08-23.
[24] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2025-08-23.
[25] D. Peakall. 2025. MaEVe. https://github.com/thoughtworks/maeve-csms. Accessed: 2025-08-23.
[26] K. Sarieddine, M. A. Sayed, S. Torabi, R. Attallah, D. Jafarigiv, C. Assi, and M. Debbabi. 2024. Uncovering Covert Attacks on EV Charging Infrastructure: How OCPP Backend Vulnerabilities Could Compromise Your System. In *19th ACM Asia Conference on Computer and Communications Security*. 977–989. https://doi.org/10.1145/3634737.3644999
[27] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. https://doi.org/10.1109/TSE.2023.3334955
[28] S. Sun, Y. Liu, D. Iter, C. Zhu, and M. Iyyer. 2023. How Does In-Context Learning Help Prompt Tuning? https://doi.org/10.48550/arXiv.2302.11521
[29] SaiFlow Research Team. 2023. Vulnerabilities in ChargePoint EV Charging Network Expose Sensitive Data. https://www.securityweek.com/ev-charging-management-system-vulnerabilities-allow-disruption-energy-theft/ Accessed: 2025-08-23.
[30] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208
[31] J. Wang, L. Yu, and X. Luo. 2024. LLMIF: Augmented Large Language Model for Fuzzing IoT Devices. In *2024 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP54263.2024.00211
[32] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. 2024. Fuzz4ALL: Universal Fuzzing with Large Language Models. In *IEEE/ACM 46th International Conference on Software Engineering*. 1547–1559. https://doi.org/10.1145/3597503.3639121
[33] C. Yang, Z. Zhao, and L. Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In *30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3676641.3716022
[34] K. Zhang, Z. Li, D. Wu, and S. Wang. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *34th USENIX Security Symposium (USENIX Security '25)*. https://doi.org/10.48550/arXiv.2501.19282

Received 2025-09-12; accepted 2025-12-22