# Token Time Bomb: Evaluating JWT Implementations for Vulnerability Discovery

Jingcheng Yang*‡, Enze Wang*†‡, Jianjun Chen*✉,
Qi Wang*, Yuheng Zhang*, Haixin Duan*, Wei Xie†, Baosheng Wang†
* Tsinghua University
† National University of Defense Technology

***Abstract*—JSON Web Tokens (JWT) have become a widely adopted standard for secure information exchange in modern distributed web applications, particularly for authentication and authorization scenarios. However, JWT implementations have introduced various vulnerabilities, such as signature verification bypass, token spoofing, and denial-of-service attacks. While prior research has reported individual such vulnerabilities, there is a lack of systematic study for JWT implementations.**

**In this paper, we propose JWTeemo, a novel testing methodology to effectively discover JWT vulnerabilities in JWT implementations. We evaluated JWTeemo against 43 JWT implementations across 10 popular programming languages and discovered 31 previously unknown security vulnerabilities, 20 of which have been assigned CVE numbers. We demonstrated the security impact of these vulnerabilities, such as enabling authentication bypass in Kubernetes and denial-of-service attacks against Apache James. We further categorized these vulnerabilities into five types, and proposed several mitigation strategies. We discussed our mitigation strategies with the IETF, which has acknowledged our findings and suggested that they would adopt our mitigations in a new RFC document. We have also reported those identified vulnerabilities to the affected providers and received acknowledgments and bug bounty rewards from Apache, Connect2id, Kubernetes, Let's Encrypt, and RedHat.**

## I. Introduction

JSON Web Token (JWT) is a compact and self-contained standard for securely transmitting information as a JSON object between parties. As each JWT is digitally signed or encrypted, it allows the recipient to verify the token's authenticity and integrity. This property enables stateless authentication, as the recipient does not need to contact the issuing server for validation. Owing to its efficiency and security features, JWT has become a cornerstone for authentication in modern web applications and distributed systems and has seen widespread adoption by prominent systems such as CloudFlare [1], Let's Encrypt [2], and Kubernetes [3].

However, recent studies have uncovered vulnerabilities in JWT implementations that can lead to severe security implications. For instance, a recently disclosed vulnerability in Microsoft SharePoint Server (CVE-2023-29357 [4]) allowed attackers to bypass authentication and achieve administrative privileges using a maliciously crafted JWT. Furthermore, the potential impact of flawed JWT implementations extends to Denial-of-Service (DoS) attacks. Given the widespread adoption of JWT, these vulnerabilities pose a significant threat to web ecosystems.

Despite the discovery of individual vulnerabilities, current methodologies for identifying JWT-related security flaws rely on manual, ad hoc approaches. While techniques such as static and dynamic analysis have proven effective for identifying taint-style vulnerabilities (e.g., SQL injection and XSS), their application to JWT implementations is challenging. The diverse nature of JWT vulnerabilities and the absence of clear sources and sinks for taint tracking hinder their detection. Prior work, such as JWTKey [5], has focused on key management issues in applications using JWTs but it cannot identify security vulnerabilities within JWT implementations. To date, no systematic study has yet been conducted for evaluating JWT implementations.

In this paper, we aim to bridge this gap by investigating three research questions:

- RQ1: How can we generate JWTs to systematically trigger vulnerabilities in JWT implementations?
- RQ2: How do we detect JWT vulnerabilities automatically?
- RQ3: What is the current prevalence of vulnerability among real-world JWT implementations?

To answer these questions, we introduce a novel fuzzing tool JWTeemo to systematically detect vulnerabilities in JWT implementations. For RQ1, we introduce a novel grammar Function-extended Backus-Naur Form (FBNF) to model JWTs and utilize a feedback-driven approach for effective test case generation. For RQ2, we develop a differential analyzer that employs both parsing discrepancy and resource exhaustion analysis strategies to effectively detect vulnerabilities. For RQ3, we evaluate JWTeemo against 43 popular JWT implementations across 10 different programming languages.

In total, we discovered 31 vulnerabilities in 17 popular JWT libraries that could lead to authentication bypasses or DoS attacks. We further analyzed these vulnerabilities and identified five primary attack categories targeting JWT implementations: (1) Signature/Encryption Confusion, (2) Algorithm Confusion,

‡ Both authors contributed equally to this work.
✉ Corresponding author: jianjun@tsinghua.edu.cn.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240697
www.ndss-symposium.org

(3) JWT Format Confusion, (4) Billion Hashes Attack, and (5) Compression DoS. We responsibly disclosed all discovered vulnerabilities to the affected vendors, resulting in 20 CVE assignments. We received acknowledgments and bug bounties from entities including RedHat, Kubernetes, Apache, and Connect2id. We reported our findings and mitigation strategies to the IETF, which acknowledged our work and would incorporate our proposals into the new JWT Best Current Practices (BCP) RFC document.

In summary, we make the following contributions:

- We introduced JWTeemo [1], a novel testing methodology to automatically uncover vulnerabilities in JWT implementations.
- We conducted the first large-scale, systematic security evaluation of JWT implementations, covering 43 widely-used libraries across 10 popular programming languages. Our analysis uncovered 31 previously unknown security vulnerabilities, all of which are exploitable to perform authentication bypass or DoS attacks. To date, 20 vulnerabilities have been assigned CVE IDs, and our findings have been acknowledged by prominent vendors and projects.
- We analyzed the root cause of these vulnerabilities and categorized the root cause into three types. We propose several mitigation strategies and discussed them with the IETF, which has acknowledged our work and would incorporate our proposals into a new RFC document.

## II. Background

### A. JSON Web Token

The widespread adoption of distributed architectures and Single-Page Applications (SPAs) has exposed the limitations of traditional server-side session-based authentication. Such mechanisms require servers to store and synchronize user login states, presenting significant scalability challenges in multi-server distributed environments. To address this, the Internet Engineering Task Force (IETF) proposed the JSON Web Token (JWT), which was standardized in 2015 through a series of core specifications [6], [7], [8].

The JWT ecosystem is not a monolithic specification but a modular framework designed to meet diverse security objectives, offering two primary JWT types. The first is *JSON Web Signature* (JWS, RFC 7515 [7]), which ensures integrity and authenticity through digital signatures or Message Authentication Codes (MACs). Although JWS guarantees that the information has not been tampered with or originates from a trusted source, its payload remains visible to any intermediary. The second is *JSON Web Encryption* (JWE, RFC 7516 [8]), which provides confidentiality protection. When claims contain sensitive data, JWE encrypts the content, ensuring that it can only be read by the intended recipient. This separation of concerns, complemented by JSON Web Algorithms (JWA, RFC 7518 [9]) for defining cryptographic algorithms and JSON Web Key (JWK, RFC 7517 [10]) for representing cryptographic keys, forms a flexible and comprehensive framework for modern security credentials. This allows developers to precisely select the level of security required for their specific use case.

[1]JWTeemo is available at https://github.com/JWTeemo/JWTeemo

### B. JWT Structures

Depending on the security mechanism employed, a JWT can be structured in one of two primary types: JWS or JWE, as illustrated in Figure 1a and Figure 1b, respectively.

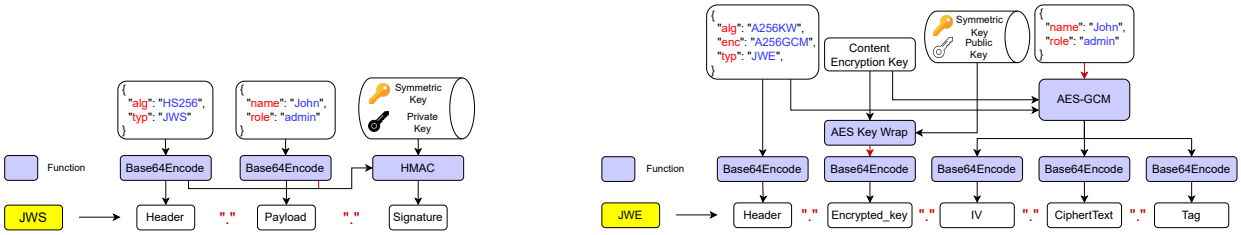
A JWS, as shown in Figure 1a, is composed of three parts separated by dots: `Header`, `Payload`, and `Signature`. The `Header` is a JSON object containing cryptographic metadata, including the `alg` claim that specifies the signing algorithm. The `Payload` is another JSON object that carries the core claims to be transferred, such as the username and roles. The Signature is computed over the Base64Url-encoded `Header` and `Payload` using the algorithm specified in the `alg` claim, providing an integrity and authenticity value. This signature provides data integrity and authenticity, ensuring that any tampering with the header or payload can be detected. The final token consists of the Base64Url-encoded `Header`, `Payload`, and `Signature`, concatenated in order with dot separators.

A JWE, as depicted in Figure 1b, consists of five parts separated by dots: `Header`, `Encrypted_key`, `IV`, `Ciphertext`, and `Authentication Tag`. The `Header` is a JSON object containing cryptographic metadata, including the `alg` claim that specifies the key encryption algorithm (e.g., A256KW) and the `enc` claim that defines the content encryption algorithm (e.g., AES-GCM). The encryption process is a two-step procedure. First, the JSON object containing the core claims (e.g., username and roles) is serialized into plaintext, and a random single-use `Content Encryption Key` (CEK) is generated. The plaintext is then encrypted with the CEK under the `enc` algorithm, producing the `Ciphertext`, `IV`, and `Authentication Tag`. Second, the CEK is then encrypted with the developer-provided JWK under the `alg` algorithm, producing the `Encrypted_Key`. Finally, each of these five components is Base64Url-encoded and concatenated with dots to form the final JWE token. This encryption ensures confidentiality of the payload, so that only authorized recipients possessing the correct key can recover the original claims.

### C. JWT Vulnerabilities

In recent years, attacks targeting JWT implementations have exposed significant security risks. A prominent example is the vulnerability in Microsoft SharePoint Server (CVE-2023-29357 [4]), where attackers exploited a flaw in the JWT validation logic to forge JWS signatures, bypassing authentication and ultimately achieving privilege escalation and even remote code execution.

Current research on JWT vulnerabilities can be broadly categorized into two directions. One line of work focuses on key management, such as JWTKey [5], which aims to address security issues throughout the lifecycle of cryptographic keys,




(a) JWS Structure

(b) JWE Structure

Fig. 1: Two Typical Types of JWT

including their generation, storage, transmission, and use. However, a vast number of security flaws stem not from improper key management but from the specific processes within JWT validation and parsing implementations. The inherent flexibility of the JWT specification, while offering broad design space for developers, inadvertently creates opportunities for security vulnerabilities due to a lack of mandatory constraints on implementation details. The other research direction explores these implementation-level issues. For instance, researchers at security conferences like DEFCON [11] and BlackHat [12] have disclosed multiple vulnerabilities that exploit implementation weaknesses.

Although prior work has identified several vulnerabilities in JWT, the discovery process heavily relies on manual analysis. While some tools exist [13], [14], they are mostly dictionary-based scanners that only detect known vulnerabilities, and there remains a lack of systematic evaluation of JWT implementations, which has motivated our study.

## III. Overview

### A. Threat Model

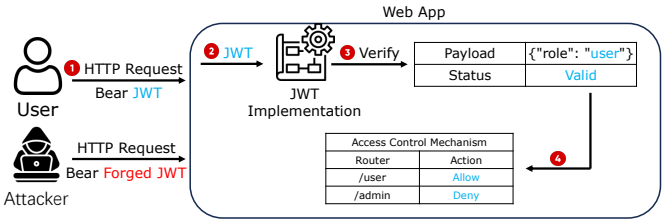



Fig. 2: JWT-based Web Application and Threat Model

In this paper, we consider a typical web application that employs JWT for authentication and authorization. After a user logs in to the web application, they receive a JWT. The user then includes this JWT token with each request to access protected endpoints. Behind the scenes, a JWT implementation handles cryptographic checks (e.g., signature verification or decryption) and decodes the token to extract its claims. The application enforces access rules based on these claims, as illustrated in Figure 2.

We assume a remote attacker with black-box access to the target application. The attacker possesses the capability to submit arbitrarily structured JWTs and observe the corresponding responses from the web application. The attacker aims to exploit vulnerabilities in the JWT processing for various attacks, such as signature verification bypass, token spoofing, and denial-of-service attacks.

### B. Research Questions

This paper answers the following research questions:

**RQ1: How do we generate JWTs to systematically trigger vulnerabilities in JWT implementations?**

The first question is the efficient generation of high-quality and diverse JWTs. These tokens must satisfy two requirements: (1) The generated JWTs must be syntactically valid according to their specification, as any malformed token would be rejected outright during initial parsing, thus failing to exercise deeper code paths. (2) The generated JWTs must exhibit sufficient diversity to ensure a comprehensive security assessment of the JWT implementation.

To the best of our knowledge, no prior work has specifically focused on fuzzing JWTs, and conventional generation methods, such as those based on Augmented Backus-Naur Form (ABNF) widely used in protocol fuzzing, are ill-suited for the JWT context for two primary reasons. First, JWT generation involves numerous functional operations, such as HMAC and RSA signing; however, ABNF, being a formal language for defining syntax, lacks the intrinsic capability to perform such computations. Second, while ABNF is suitable for generating context-free grammars, JWT generation is inherently context-sensitive. For instance, the JWS signature operation is dependent on the value pre-defined in the alg header field.

To address this question, we introduce Function-extended Backus-Naur Form (FBNF), an extension of ABNF that incorporates FUNC and IF constructs. The former handles functional computations between nodes, while the latter manages contextual dependencies between the current generation content and preceding node values. During the JWT generation phase, we first construct an FBNF grammar graph based on the relevant RFC documents and then traverse this graph to generate test cases. Furthermore, the JWT RFCs define a large number of claims, leading to an exponential growth in the combinatorial space. A naive random generation strategy would suffer from combinatorial explosion and produce a high volume of invalid samples that are immediately rejected for violating semantic constraints (e.g., a missing `alg` claim),

severely hampering fuzzing efficiency and effectiveness. To this end, we model the "`node selection`" process within the FBNF grammar graph as a Monte Carlo Tree Search (MCTS) and employ a UCT-Rand algorithm to guide the generation process. During fuzzing, we leverage the parsing feedback from the target implementation on previously generated JWTs to dynamically adjust the selection weights of nodes in the grammar graph, thereby progressively "learning" the internal semantic logic of the target and enabling more directed testing.

**RQ2: How do we detect JWT vulnerabilities automatically?**

Prior fuzzing methodologies typically detect vulnerabilities by observing program exceptions, such as crashes. However, JWT security issues manifest as logic bugs and often do not trigger explicit program exceptions.

To overcome this limitation, we have developed a novel differential analyzer for detecting JWT vulnerabilities based on differential testing [15], [16], [17]. This analyzer consists of two key components: (1) Cross-library differential analyzer: This component compares the output behavior of different JWT libraries to detect logical vulnerabilities. When processing the same JWT input, distinct libraries with identical functionality should yield the same output (e.g., a valid or invalid JWT verification result) . A discrepancy in outcomes across different implementations for the same input is flagged as a potential vulnerability. (2) Resource consumption detector: This component continuously records the performance of each library during fuzzing, including CPU utilization and memory consumption. First, we obtain the average values of resource consumption from baseline tests. Then we track the resource usage before and after each fuzzing session, paying attention to whether it exceeds the threshold generated by the baseline tests. If the resource consumption significantly exceeds the baseline, the implementation is flagged as having a potential Denial of Service (DoS) vulnerability.

**RQ3: What is the current prevalence of vulnerability among real-world JWT implementations?**

To investigate security issues in real-world JWT implementations, we conduct the first large-scale, systematic evaluation across 43 widely-used libraries spanning 10 popular programming languages. Our analysis reveals 31 previously unknown vulnerabilities. These vulnerabilities fall into five categories—three enabling authentication bypass and two leading to Denial-of-Service (DoS) attacks. The authentication bypass issues can be exploited to forge tokens that are mistakenly accepted as valid, while the DoS vulnerabilities can exhaust server resources through crafted inputs, affecting availability. We categorize the root causes of JWT vulnerabilities into three classes and propose corresponding mitigation strategies. These measures target both the JWT specification itself and the implementations by library developers. We have shared our findings and proposals with the IETF, which has acknowledged our work and plans to incorporate our recommendations into a new RFC document.

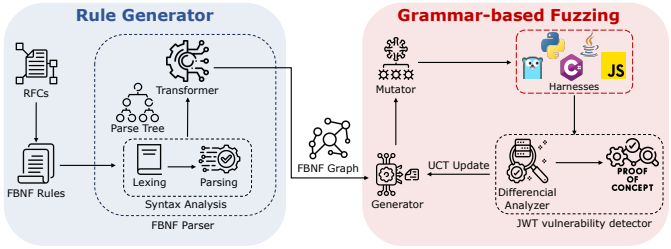



Fig. 3: JWTeemo Workflow

## IV. JWTeemo: Design and Implementation

### A. Workflow

We design and implement JWTeemo, a framework designed to detect security vulnerabilities in JWT implementations through automated and efficient fuzzing. Figure 3 presents the workflow, which consists of two primary modules: the *Rule Generator* and the *Grammar-based Fuzzing* module. In the Rule Generator module, we first manually craft a set of grammar rules based on the relevant RFC specifications for JWT. For this, we use Function-extended Backus-Naur Form (FBNF), a novel descriptive language we developed by extending the traditional Augmented Backus-Naur Form (ABNF). An FBNF parser then processes these rules to construct an FBNF parsing graph. The Grammar-based Fuzzing module traverses this graph to generate an initial JWT corpus, and subsequently utilizes a mutator to apply a series of mutation strategies to this corpus. The resulting JWTs are then sent to the target JWT implementations. Based on the feedback from the JWT implementations, a generator adjusts the weights of the nodes in the FBNF generation graph. Finally, a differential analyzer is employed to identify and report discrepancies in parsing results and resource consumption among different JWT implementations.

### B. Rule Generator

Listing 1: An example of JWT's FBNF

```
JWT = JWS / JWE
JWS = CompactJWS / FlattenJWS
CompactJWS = b64header "." b64payload "." base64_encode(
    signature)
signature = if(alg_value,{
                "HS256": HMACUsingSHA256(key, b64header "."
                     b64payload),
                "RS256": RSAUsingSHA256(key, b64header "."
                     b64payload),
            }
```

**FBNF Rules** The Augmented Backus-Naur Form (ABNF) is widely adopted for describing the context-free grammar of Internet protocols. Its rules are typically expressed in the format `Rule = Definition`, where the left side is the rule name and the right side is its definition. ABNF supports fundamental operations such as concatenation, selection, and repetition. However, the generation process for protocol formats like JWT often involves functional operations and contextual dependencies, the semantics of which lie beyond the expressive capabilities of traditional ABNF.

To address this limitation, we propose FBNF (Function-extended BNF), an extended formalism that introduces two novel constructs to ABNF: function invocation and context-aware function selection. Within FBNF, a grammar rule can not only reference static symbols but also embed the semantics of function calls and contextual choices in its definition. Specifically, the right-hand side of a rule can formally represent an invocation of a function and dynamically select which function to call conditioned on the values of other syntactic nodes. This allows for a more accurate modeling of the semantic dependencies inherent in the protocol generation process.

A concrete example of an FBNF grammar for JWT is presented in Listing 1. Here, the construct `base64_encode(signature)` applies Base64 encoding to the output of the signature rule. Furthermore, a conditional `if` construct dynamically dispatches to the correct cryptographic function based on the `alg_value` parameter: it selects `HMACUsingSHA256` for `HS256` and `RSAUsingSHA256` for `RS256`. These functions operate on arguments including a key and a message body constructed by concatenating `b64header`, a dot (`"."`) separator, and `b64payload`.

The formal ABNF-style definitions for our extensions, function invocation and function selection, are presented in Appendix Listing 3. These extensions cover the syntactic structures for function invocation and selection while maintaining the readability and consistency of the original ABNF. Consequently, FBNF significantly enhances the expressive power for defining protocol formats. This makes it particularly well-suited for generation tasks involving security protocols that require contextual processing and functional execution, such as Security Assertion Markup Language (SAML) [18].

**FBNF Parser** The FBNF Parser consists of two primary components: Grammar-Based Input Analysis and a Transformer. The former is responsible for parsing each FBNF rule into a syntax tree to extract its internal structural relationships. The latter further transforms these syntax structures into a unified FBNF parsing graph, which facilitates subsequent semantic modeling and test case generation.

In the Grammar-Based Input Analysis stage, we employ the Lark parser generator for lexical and syntactic analysis of the input FBNF rule set, thereby constructing a Concrete Syntax Tree (CST) for each rule. Within the CST, each internal node represents a specific syntactic operator (e.g., concatenation, selection, function call), while the leaf nodes correspond to either terminals or invocations of other grammar rules. The CST preserves the hierarchical structure, compositional relationships, and semantic logic of the rule, fully encapsulating its internal structural logic.

The Transformer component then converts all CSTs into a single, directed FBNF graph, as illustrated in Fig. 4, to explicitly model inter-rule dependencies. To achieve this, the Transformer first interprets the specific operator nodes within each CST. Drawing an analogy from conventional ABNF parsing, where the three fundamental operations, concatenation, selection, and repetition, are mapped to AND, OR, and RAND nodes, respectively, we extend this principle for FBNF. Specifically, the two new operation types introduced in our FBNF paradigm are translated into FUNC nodes for direct semantic function calls and IF nodes for conditional function selection.

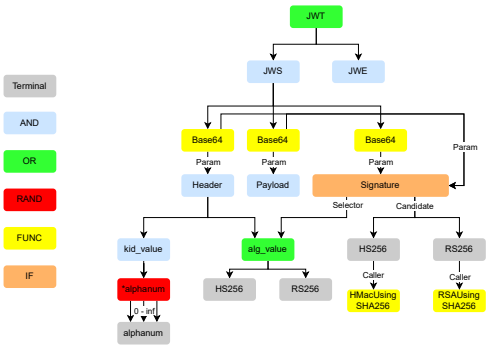



Fig. 4: FBNF Graph Example

After interpreting these intra-rule structures, the Transformer builds the complete graph by linking the individual trees. It traverses the non-terminal leaf nodes of each CST and, based on their names, adds a directed edge to the root node of the rule it references. These edges not only represent structural references but, more critically, capture the contextual dependency paths within the semantic generation process. For instance, the `alg_value` rule is used to construct the `alg` field in a JWT Header and also dictates the generation logic for the `signature`. To model this semantic dependency, the Transformer introduces a directed edge in the parsing graph from the `signature` node to the `alg_value` node. This signifies that the generation of the `signature` is contingent upon the resolved value of the algorithm field. Such dependency edges ensure the sequential integrity of semantic generation: `alg_value` must be generated and resolved before the `signature` can be produced, thus enabling context-aware semantic modeling.

### C. Grammar-based Fuzzing

JWTeemo leverages grammar-based fuzzing, augmented by the Upper Confidence Bounds Applied to Trees (UCT) Rand algorithm [19], to improve the efficiency of identifying security issues in JWT implementations, regardless of their implementation language. It generates more effective test cases through a feedback mechanism implemented for JWTeemo and the UCT-Rand algorithm. Additionally, a differential analyzer is used to find vulnerabilities by comparing the parsing results for the same JWT test case between different implementations and analyzing the resource consumption of each implementation individually.

**JWT Generator** The JWT Generator is responsible for producing a large volume of syntactically valid test cases based on the FBNF parsing graph. The graph consists of five node types, each representing a distinct operation that guides the

generator's depth-first traversal. The JWT Generator initiates the process by targeting the JWT rule as the root node and recursively traverses the FBNF parsing graph down to its terminal nodes (e.g., string and numeric literals). Furthermore, to enhance the efficiency of JWTeemo, we implement a caching mechanism. During each generation pass, the value produced by any non-RAND node is cached. This strategy prevents redundant generation operations that would otherwise occur when multiple traversal paths converge on the same node.

The generation strategy is determined by the type of node being visited during the traversal: (1) AND Node: When visiting an AND node, the generator must recursively visit all of its child nodes to proceed with the generation. (2) OR Node: Upon visiting an OR node, the generator recursively visits one of its child nodes. (3) RAND Node: This signifies that its child node is visited a randomized number of times. (4) FUNC Node: The generator first recursively visits all child nodes to gather their generated values. These values are then passed as arguments to a corresponding function, which the user must predefine in Python. The JWT Generator invokes this function, and its return value becomes the value of the FUNC node. (5) IF Node: The traversal first visits the designated condition child node. The resulting value is then compared against the keys of a selection-map to determine which function to select. Subsequently, the generator recursively visits all child nodes that serve as arguments for the chosen function. The values from these argument nodes are passed to the selected function, and its return value becomes the value of the IF node.

**Mutator** To discover a wider range of potential parsing ambiguities in JWTs, JWTeemo incorporates a Mutator module that performs random mutations at two distinct levels: structural and content. For structural mutation, the Mutator randomly selects a non-terminal node from the FBNF graph and either deletes the subgraph rooted at that node (Node Deletion) or replaces it with a different node (Node Replacement). For content mutation, it targets a randomly selected terminal node and either inserts a random character at a random position within its value or deletes a single character from a random position. By design, only a small number of these mutations are applied to any single JWT. This approach allows for the generation of a greater variety of JWT formats while ensuring that the token's fundamental structure is not entirely corrupted, thereby maintaining its potential to be parsed by target implementations.

**Differential Analyzer** The Differential Analyzer is responsible for identifying anomalous behaviors when different implementations parse JWTs. Our analysis focuses on two key aspects: (1) the consistency of parsing results across implementations, and (2) whether resource consumption within a single implementation exceeds established thresholds. To address these, we designed two corresponding strategies: Parsing Discrepancy Analysis and Resource Exhaustion Analysis.

The overall workflow, illustrated in Algorithm 1, iterates through each target JWT implementation. Within each iteration, the RuleGenerator module takes the FBNF parsing graph and enriches the syntax tree T with declarative values extracted from RFCs. The JWTGenerator then produces a JWT, which the Mutator alters to create a mutatedJwt. This mutatedJwt is sent to the implementation under test, and we collect its parsing result and resource consumption. The Differential Analyzer then invokes the appropriate strategy. Finally, the UCT-Rand algorithm updates the FBNF graph, using the successful parsing by the current implementation as a feedback signal.

**Algorithm 1:** Grammar-based Fuzzing Algorithm

**Input:** $G$: The initial FBNF graph constructed based on the RFC specification.
**Input:** $H$: An array of JWT Implementations, $H = \{H_1, ..., H_n\}$.
**Input:** $k$: Resource Usage metric (a constant factor).
**Output:** *differences*: An array of captured differences.

$\mu \leftarrow \mathbf{0}$; $\sigma \leftarrow \mathbf{0}$;
**repeat**
  $jwt\_seed \leftarrow$ JWTGENERATOR$(G)$;
  $jwt \leftarrow$ MUTATOR$(jwt_seed)$;
  $outputs, RU \leftarrow$ RUNFUZZ$(H, jwt)$;
  **foreach** *pair* $(i, j)$ *such that* $1 \leq i < j \leq n$ **do**
    **if** $outputs[i] \neq outputs[j]$ **then**
      /* Differences between implementations */
      $differences$.APPEND$((H_i, H_j, jwt))$;
  **for** $i \leftarrow 1$ **to** $n$ **do**
    **if** $RU[i] > \mu[i] + k \cdot \sigma[i]$ **then**
      /* Differences within the same implementation */
      $differences$.APPEND$((H_i, jwt))$;
    $\mu[i], \sigma[i] \leftarrow$ RESOURCEMONITORUPDATE$(RU[i])$;
  $G \leftarrow$ UCTUPDATE$(G, outputs)$;
**until** ENDCONDITIONS();

In the Parsing Discrepancy Analysis strategy, we assess result consistency based on two conditions: (1) whether one implementation successfully validates a given JWT while another fails, and (2) whether both implementations validate the JWT but return inconsistent parsed content. These checks are designed to detect vulnerabilities such as authentication bypass, which can lead to privilege escalation. Any JWT that triggers such a discrepancy is flagged as anomalous.

The Resource Exhaustion Analysis strategy monitors for abnormal usage, a common indicator of resource exhaustion Denial of Service (DoS) vulnerabilities. We focus on two primary metrics: excessive server CPU utilization and memory consumption. Adopting the methodology from Rampart [20], we continuously monitor these metrics and use Chebyshev's inequality (Equation 1) to identify statistically significant deviations. The inequality states that the probability of a random variable (X) differing from its mean by more than $k$ standard

deviations is at most $1/k^2$. Therefore, if a measured resource usage value R deviates from the mean by more than a threshold of $k \times \sigma$ (Equation 2), JWTeemo flags the responsible JWT as anomalous.

Finally, all JWTs flagged as anomalous by either strategy are archived in a database for subsequent analysis.

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{1}$$

$$R > \mu + k \times \sigma \tag{2}$$

**UCT Update** To enable the fuzzer to learn the internal validation logic of the target JWT implementation, we introduce a feedback-driven update mechanism based on the UCT (Upper Confidence bound for Trees)-Rand algorithm [19]. This approach iteratively optimizes the selection of nodes in the FBNF graph under limited feedback, while balancing the trade-off between exploration and exploitation to steadily improve the quality of generated test cases.

In each iteration, a generated JWT is submitted to multiple widely-used JWT implementations for validation. If more than 50% of the implementations accept the input, the current path is considered successful; otherwise, it is marked as failed, and the FBNF graph is updated accordingly.

In UCT-Rand, the selection of the next node $v'$ given the current node $v$ follows the strategy shown in Equation 3:

$$\pi(v) := \underset{v' \in v.\text{children}}{\text{weighted rand}} \left( Q(v, v') + c \cdot \sqrt{\frac{\ln N(v)}{N(v, v')}} \right) \tag{3}$$

where:

- $Q(v, v')$ denotes the empirical success rate of generating a valid JWT by choosing node $v'$ after $v$;
- $N(v)$ is the number of times node $v$ has been visited;
- $N(v, v')$ is the number of times node $v'$ has been selected following $v$;
- $c$ is an exploration coefficient to balance exploitation and exploration.

This feedback mechanism is particularly effective in identifying semantic dependencies between claims during generation. For instance, in a JWE using the PBES2 algorithm, both `p2s` and `p2c` fields are typically required to appear together for the token to be considered valid. Specifically, the node selection process over the FBNF grammar is modeled as a Monte Carlo Search Tree. Suppose the fuzzer selects a path like "`JWE` $\rightarrow$ `p2s`", meaning it intends to generate a JWE token containing the `p2s` field. The next possible nodes may include `p2c`, `kid`, or `exp`, yielding candidate paths such as "`JWE` $\rightarrow$ `p2s` $\rightarrow$ `p2c`" and "`JWE` $\rightarrow$ `p2s` $\rightarrow$ `kid`". If the path with both `p2s` and `p2c` leads to successful validation, while the one with only `p2s` fails, the algorithm records a higher empirical success rate for the former. In subsequent iterations, when revisiting the state "`JWE` $\rightarrow$ `p2s`", the algorithm is more likely to select `p2c` as the next node. This allows the fuzzer to adaptively learn and preserve inter-claim dependencies during generation.

## V. Evaluation and Findings

### A. Experiment Setup

**Dataset.** To test the effectiveness of JWTeemo, we considered the top 16 programming languages in the TIOBE [21] ranking, and selected the JWT libraries with GitHub stars $\geq$ 100 from jwt.io [22] for testing, a total of 43 libraries. The detailed contents are shown in Appendix Table IV. We wrote a harness for each library as an implementation for receiving and verifying JWTs.

**Setup.** JWTeemo is deployed on an Ubuntu server with a 4.1GHz 32-core CPU and 512G RAM. We designed and implemented a corresponding harness for each target library as the JWT implementation to receive and verify JWT. Each harness independently listens to a port, through which JWTeemo can send a JWT to the corresponding implementation. If the parsing is successful, the Harness returns the parsing result of the JWT, otherwise the Harness returns the error message during the parsing for further analysis.

### B. Discovering Real-World JWT Vulnerabilities

During the experiment, JWTeemo generated a total of 100,000 JWT test cases. Among them, 9,383 JWTs triggered parsing inconsistencies across different implementations. To avoid counting the same type of ambiguity multiple times, we applied a deduplication step: if several test cases failed on the same set of JWT implementations and produced the same error messages, we treated them as the same issue and grouped them together. After grouping the cases, we obtained 442 distinct JWT test cases, each representing a unique type of parsing error. We then manually analyzed these 442 cases and examined the pairwise parsing discrepancies among the JWT implementations. In total, JWTeemo revealed 1,804 observable differences across pairs of implementations.

Through our manual analysis, we identified five types of discrepancies across different implementations, three of which indicate concrete security issues in JWT implementations. Additionally, we discovered two types of intra-implementation resource consumption differences, both of which also reflect security problems in JWT implementations. Overall, JWTeemo detected 31 security issues from 43 implementations, including 2 Sign/Encryption Confusion vulnerabilities, 2 Algorithm Confusion vulnerabilities, 10 Billion Hashes Attack vulnerabilities, 13 Compression DoS vulnerabilities, and 4 implementations supporting the parsing of JWS in JSON format, which had the risk of JWT Format Confusion. Notably, the first three vulnerability categories can lead to token spoofing, enabling attackers to forge or tamper with tokens and bypass authentication, while the latter two expose implementations to JWT-based DoS attacks. In addition, 71.9% of the implementations supporting the PBES2 algorithms have Billion Hashes Attack flaws, and 86.7% of the implementations supporting JWE have Compression Attack flaws. All the vulnerabilities we found are shown in Table I. In modern Internet systems, JWTs are widely used for service authentication and authorization, and both token spoofing and DoS vulnerabilities can severely undermine

| Language | Library | Github Stars | Version | LLoC | Vulnerability | CVE Number |
|---|---|---|---|---|---|---|
| Python | python-jose | 1.5k | 3.3.0 | 3,726 | Compression DoS | CVE-2024-29370 |
| | jwcrypto | 430 | 1.5.0 | 6,393 | Billion Hashes Attack<br>Compression DoS<br>JWT Format Confusion | CVE-2023-6681<br>CVE-2024-28102<br>Fixed |
| | authlib | 4.5k | 1.2.1 | 22,586 | Compression DoS | Unassigned |
| C | latchset/jose | 170 | 11 | 9,274 | Billion Hashes Attack<br>JWT Format Confusion | CVE-2023-50967<br>Unassigned |
| | libjwt | 368 | 1.15.3 | 5,175 | Algorithm Confusion | CVE-2024-57453 |
| C++ | cpp-jwt | 387 | 1.4 | 3,357 | Algorithm Confusion | CVE-2024-57454 |
| Java | jjwt | 10.1k | 0.12.3 | 18,621 | Billion Hashes Attack<br>Compression DoS<br>Compression DoS (in JWS) | CVE-2024-39960<br>Unassigned<br>Unassigned |
| | jose4j | N/A | 0.9.3 | 30,282 | Billion Hashes Attack<br>Compression DoS | CVE-2023-51775<br>CVE-2024-29371 |
| | nimbus-jose-jwt | N/A | 9.37.1 | 50,324 | Billion Hashes Attack<br>Compression DoS | CVE-2023-52428<br>Unassigned |
| C# | jose-jwt | 933 | 4.1.0 | 19,576 | Sign/Encrypt Confusion<br>Compression DoS | CVE-2024-24238<br>CVE-2024-27663 |
| JavaScript | jose | 5.3k | 5.1.3 | 20,598 | Compression DoS | CVE-2024-28176 |
| | node-jose | 699 | 2.2.0 | 17,128 | Billion Hashes Attack<br>Compression DoS | CVE-2024-39960<br>Unassigned |
| PHP | jwt-framework | 881 | 3.2.8 | 16,949 | Billion Hashes Attack<br>Compression DoS | Fixed<br>Unassigned |
| Go | jose2go | 186 | 1.5.0 | 3,580 | Billion Hashes Attack<br>Compression DoS | CVE-2023-50658<br>CVE-2025-63811 |
| | go-jose | 2.3k | 3.0.1 | 16,256 | Compression DoS<br>JWT Format Confusion | CVE-2024-28180<br>Unassigned |
| | jwx | 1.9k | 2.0.17 | 37,837 | Billion Hashes Attack<br>Compression DoS<br>JWT Format Confusion | CVE-2023-49290<br>CVE-2024-28122<br>Fixed |
| Ruby | json-jwt | 297 | 1.16.3 | 2,883 | Sign/Encrypt Confusion | CVE-2023-51774 |

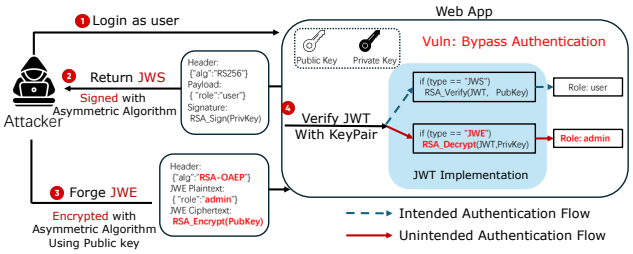
TABLE I: New JWT vulnerabilities discovered by JWTeemo. N/A indicates the library is not open source on GitHub.

these security guarantees. In particular, token forgery attacks allow adversaries to gain unauthorized access or escalate privileges, while DoS attacks can be triggered at minimal cost by maliciously crafted tokens that exhaust memory or CPU resources on authentication servers, potentially causing service outages.

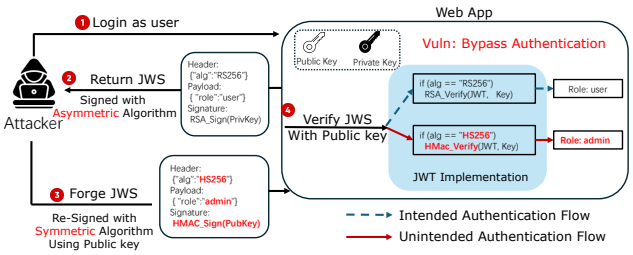
*1) Differences between implementations:* The final results of the differences found are shown in Appendix figure 12. We classified these differences into five categories based on root causes: (1) Sign/Encryption Confusion; (2) Algorithm Confusion; (3) JWT Format Confusion; (4) Different Claims Checker; (5) Different Algorithm Support. The first three differences can be exploited for signature verification bypass and token spoofing.

**Sign/Encryption Confusion** During fuzzing, we discovered that when using a JWS public key to encrypt and generate a JWE-type JWT, the JWT could be validated by two different implementations. This parsing discrepancy leads to an attack scenario as shown in Figure 5a. The attacker first obtains the public key used for validating the JWS signature. Then, they log in normally to get a JWS containing their role. The attacker next modifies the payload's role to admin. Finally, they use the obtained public key to encrypt the payload and create a forged JWE. Vulnerable JWT implementations, when parsing this JWT, determine it to be a JWE based on the number of dots `"."` in the JWT, and then use the corresponding private key to decrypt it. This results in successful authentication, allowing the attacker to escalate privileges. As shown in Appendix Figure 13a, the Generator module of JWTEEMO starts from the JWT node, selects the JWE subnode and the subsequent opaque nodes shown in the FBNF Graph according to its generation strategy, including the `alg_value` node with the value `RSA-OAEP`, and uses the non-terminal nodes as generation roots to further traverse their subnodes and instantiate the corresponding productions, thereby generating the resulting payloads as test inputs.

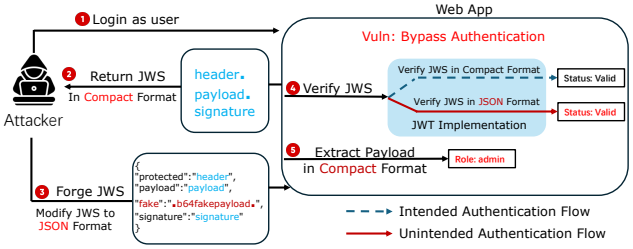
**Algorithm Confusion** During fuzzing, we found that when the JWS's `alg` claim is set to HS256, and the public key




(a) Sign/Encrypt Confusion: A vulnerable JWT Implementation misidentifies the JWT type, causing private key to be used for RSA decryption and enabling arbitrary token forgery.




(b) Algorithm Confusion: A vulnerable JWT Implementation misidentifies the JWT algorithm, causing public key to be used as HMAC secret and enabling arbitrary token forgery.




(c) JWT Format Confusion: A vulnerable JWT Implementation misidentifies the JWT Format. This allows attackers to convert legitimate Compact-format JWT into JSON-format JWT and insert forged payload and enabling arbitrary token forgery.

Fig. 5: Three types of JWT vulnerabilities discovered by differences between implementations

is used as the HMAC secret for signing, the JWS can be validated by two implementations. This parsing discrepancy leads to an attack scenario as shown in Figure 5b. The attacker first obtains the public key used for validating the RSA signature and logs in normally to get a JWS containing their role. The attacker then modifies the `alg` claim in the JWT Header to HS256 and changes the payload's role to admin. Finally, the attacker uses the obtained public key as the HMAC secret to sign the JWT and create a forged JWT. Vulnerable JWT implementations, when parsing this JWT, select the HMAC algorithm based on the `alg` claim in the Header and use the public key as the secret to validate the signature, allowing the attacker to bypass authentication and escalate privileges. As shown in Appendix Figure 13b, the Generator module of JWTEEMO starts from the JWT node, selects the JWS subnode and the subsequent opaque nodes shown in the FBNF Graph according to its generation strategy, including the `alg_value` node with the value `HS256`, and treats the selected non-terminals as roots to synthesize test payloads.

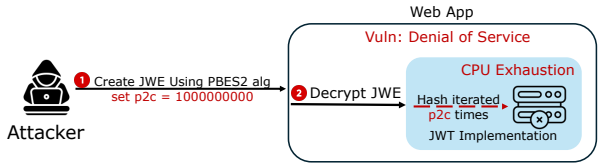
**JWT Format Confusion** During fuzzing, we discovered that when generating a JSON-type JWS, the JWS could pass validation from four different implementations. This parsing discrepancy could lead to an attack scenario as shown in Figure 5c. The attacker first logs in normally to obtain a JWS containing their role. They then convert this JWT into a JSON JWT type as shown in the figure, inserting a forged claim `"fake":".eyJyb2xlIjoiYWRtaW4ifQ."` (URL-safe base64-encoded of `{"role":"admin"}`) in the payload. As shown in Appendix Figure 13c, the Generator module of JWTEEMO starts from the JWT node, selects the JSON JWS subnode allowed by the JWS RFC [7], even though such a format is not permitted by the JWT RFC [6], and uses the non-terminal nodes as generation roots to instantiate the corresponding productions, thereby generating the resulting payloads that violate the expected format constraint.

Vulnerable JWT implementations, when parsing this JWT, check the `protected`, `payload`, and `signature` fields and wrongly determine it to be a valid JWT. However, when the Web App extracts the payload by splitting the JWT by periods (`"."`), they retrieve the attacker's forged `eyJyb2xlIjoiYWRtaW4ifQ` payload, which results in privilege escalation. A specific case of this vulnerability was discovered in Kubernetes, detailed in Case Study 1 in Section V-E.

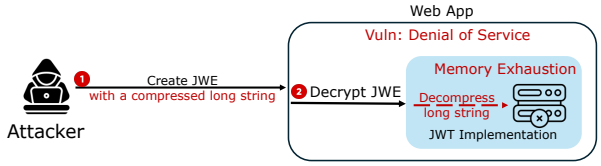
**Different Claims Checker** This issue poses a potential security risk. We found that after JWT signature verification, 20 implementations automatically check whether the exp claim is expired, etc., while other implementations do not check, or only provide an interface but do not call it automatically. However, application developers may not notice the expiration of exp claim. If the implementation does not provide automatic checking function, JWT may still be used after expiration, increasing the risk of session hijacking.

**Different Algorithm Support** The reason for this difference is that different implementations support different algorithms, and this difference is common between implementations. After our analysis, such differences will affect the compatibility of JWT in different applications, but will not directly cause security issues.

**False Positive Analysis.** A comprehensive examination of the 1,804 parsing discrepancies illustrated in Appendix Figure 12 revealed that 635 of these cases were attributable to false positives, resulting in an overall false-positive rate of 35.1%. All of these false positives stem from implementation-specific support differences among libraries, particularly inconsistencies in how supported algorithms and JWS/JWE features are handled. For instance, some implementations throw an exception when encountering an unsupported algorithm, whereas another implementation that does support that algorithm proceeds to parse successfully. Such behavioral asymmetry leads the differential analysis to misclassify these benign, capability-driven divergences as meaningful discrepancies.



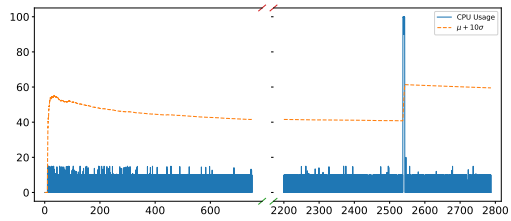

(a) Billion Hashes Attack



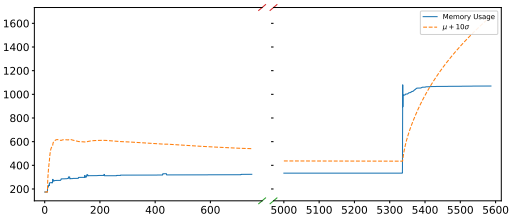

(b) Compression DoS

Fig. 6: Two Types of JWT vulnerabilities discovered by differences within the same implementations

*2) Differences within the same implementation:* In order to detect differences within the JWT implementation, we continuously monitored and compared the CPU usage and memory consumption of the JWT implementation when parsing the JWT generated by JWTeemo. We used formula $|R - \mu| > k\sigma$ to determine whether the current CPU usage and memory consumption significantly exceeded the baseline.



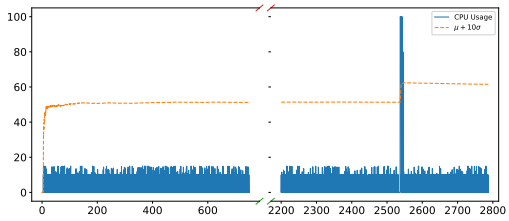

(a) CPU Usage (%) Over Time (s)



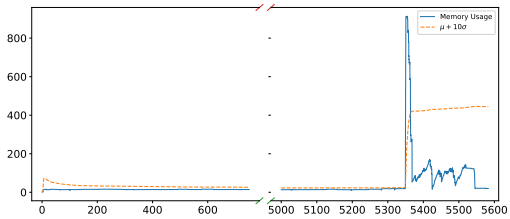

(b) Memory Usage (MB) Over Time (s)

Fig. 7: Resource Consumption of JJWT Under JWTeemo

Figures 7 and 8 are the test results for the jjwt library [23] (Java, 10.1k Stars) and the jwx library [24] (Go, 1.9k Stars), respectively. In these two charts, the blue line represents the resource consumption of the server caused by the JWTs sent by JWTeemo, and the blue line represents the calculated baseline. According to our analysis, the sharp increase in CPU usage caused by JWTeemo is due to the fact that the JWT we generated caused the target server to perform a large number of hash operations, that is, *Billion Hashes Attack*. The sharp increase in memory consumption caused by JWTeemo is due




(a) CPU Usage (%) Over Time (s)




(b) Memory Usage (MB) Over Time (s)

Fig. 8: Resource Consumption of JWX Under JWTeemo

to the fact that the JWT we generated caused the target server to decompress a long string, that is, *Compression DoS*.

**Billion Hashes Attack** This vulnerability scenario is illustrated in Figure 6a. The attacker first creates a JWE encrypted using the `PBES2` algorithm, then modifies the `p2c` claim in the JWT header to a large value and sends the modified JWT to the Web App. The `p2c` claim defines the number of iterations applied to the underlying hash function (e.g., `SHA-256`) during key derivation. When decrypting the JWE, the JWT implementation uses this value to derive the Content Encryption Key (CEK), meaning that a large `p2c` value results in a highly expensive key derivation process. As a result, an attacker can exploit this claim to force the server into performing excessive hash computations, leading to high CPU usage and causing a Denial-of-Service (DoS) attack. As shown in Appendix Figure 13d, the Generator module of JWTEEMO starts from the JWT node and first generates a JWE using the PBES2 algorithm in the relevant subnode by recursively expanding non-terminal nodes. Then, through the mutation module, it mutates the `p2c` claim to a large number, thereby producing concrete payloads that serve as test inputs.

**Compression Attack** The attack scenario is illustrated in Figure 6b. The attacker first constructs a JWE containing a long string in the payload. In the header, the attacker sets `{"zip":"DEF"}`, indicating that the payload is compressed using the Deflate algorithm. When the server parses this JWT, it decompresses the payload, which results in high memory consumption. As shown in Appendix Figure 13e, the Generator module of JWTEEMO starts from the JWT node and first generates a JWE with zip compression in the relevant subnode. Then, through the mutation module, it mutates the payload to include long strings, which are then fed to the fuzzing harness.

Notably, we found that the JJWT library accepts the `zip`

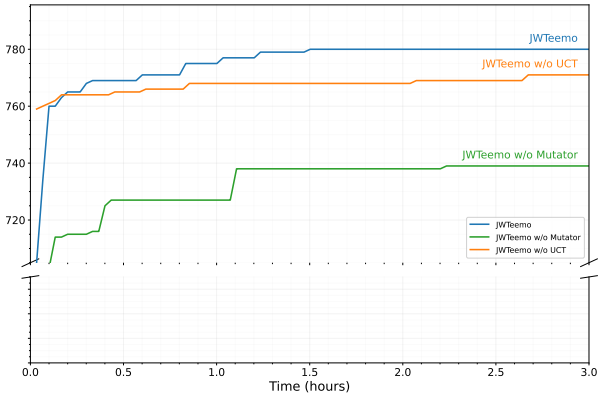



Fig. 9: Covered Edges over Time

claim within a JWS, even though the RFC explicitly states that the `zip` claim is only allowed in a JWE. This RFC-violating payload was discovered through mutation-based fuzzing in JWTeemo, highlighting the necessity of grammar-based fuzzing in uncovering such edge cases. As shown in Appendix Figure 13f, the Generator module of JWTEEMO starts from the JWT node and first generates a JWS. Then, through the mutation module, it inserts the zip subnode from the JWE subgraph into the header node, producing mutated tokens that are then fed to the fuzzing harness.

### C. Ablation Experiments

To evaluate the contribution of each component in JWTeemo, we conducted ablation experiments and designed two fuzzer variants, each disabling one of its key mechanisms: the UCT Update and the Mutator. We measured two metrics—Covered Edges and Time-to-Discover—to assess exploration efficiency and vulnerability detection capability, where Time-to-Discover denotes the time (in seconds) at which each vulnerability was first identified. We ran five fuzzing sessions per configuration and reported the median as the final result.

**Assessing the UCT Update.** We disabled the UCT-based exploration update (JWTeemo w/o UCT Update) to analyze its impact on exploration efficiency. We measured the temporal evolution of covered edges over time on the JJWT implementation. As shown in Figure 9, JWTeemo w/o UCT Update exhibited slower growth in edge coverage, indicating that the absence of guided exploration reduced the diversity of explored inputs. Moreover, in the Time to Discover Vulnerabilities evaluation as shown in Table II, JWTeemo w/o UCT Update required a longer time to identify known issues, confirming that UCT updates accelerate effective exploration and improve fuzzing stability.

**Assessing the Mutator.** We disabled the mutation engine (JWTeemo w/o Mutator) to evaluate its effect on input diversity and vulnerability discovery. In this configuration, JWTeemo w/o Mutator relied solely on FBNF graph traversal without dynamic node mutation. We observed a noticeable slowdown in coverage growth on the JJWT library and a reduced ability to reach deep parsing paths. In the Time to Discover Vulnerabilities analysis, JWTeemo w/o Mutator failed to uncover two vulnerability types detected by the full version. These results suggest that mutation-based exploration is essential for maintaining input diversity and improving overall fuzzing effectiveness. The comparative results for coverage and vulnerability discovery are summarized in Figure 9 and Table II.

### D. Comparison with Other JWT Tools.

To evaluate the effectiveness of JWTeemo, we conducted a comparative experiment with two state-of-the-art (SOTA) tools: JWT_Tool [13] and the JWT Editor plugin for Burp Suite [14]. To ensure a fair comparison, we modified our fuzzing harness to expose a simple web-based interface that accepts JWT tokens via HTTP requests and returns status code 200 when a token is successfully parsed and validated, or 403 otherwise. This unified interface allowed all tools—including JWTeemo, JWT_Tool, and JWT Editor—to be tested under consistent conditions. For JWT_Tool, executed a full scan, interacting entirely with the harness via the web interface. The tool generated and submitted various crafted tokens using its built-in payloads and heuristics. If any of the injected tokens bypassed the validation logic (i.e., triggered a 200 response), we considered the corresponding vulnerability detected. For JWT Editor, which is a semi-automated Burp Suite extension, we manually tested each of its exploit modules against the same web interface. For each vulnerability type, we loaded a sample token into the editor, applied the relevant attack options, and submitted the result to the server.

After applying all three tools to the harness, we recorded each tool's branch coverage on the JJWT library and the time required to discover vulnerabilities. The results are summarized in Table III. As shown, JWT_Tool and JWT Editor both achieved limited coverage and detected only the Algorithm Confusion vulnerability, with JWT Editor further requiring manual effort to operate. In contrast, JWTeemo attained higher coverage and discovered vulnerabilities more efficiently through its automated and systematic fuzzing process.

Our analysis of the results reveal that both existing tools only collect known exploits and rely on predefined payloads for detection, and therefore their ability to identify vulnerabilities is limited, making them less effective for comprehensive vulnerability discovery.

### E. Case Study

When we manually analyzed the detected vulnerabilities, we found two noteworthy cases, which affected the identity authentication service of kubernetes, and the SMTP authentication service of Apache/James-project.

**Case Study 1: Authentication Bypass in Kubernetes**

This vulnerability corresponds to the JWT Format Confusion issue discussed earlier. Kubernetes [3] (K8s) is a widely adopted open-source system for automating the deployment, scaling, and management of containerized applications. In such distributed systems, authentication and authorization are frequently built upon JWTs, which serve as portable,

| Configuration | Sign/Encrypt Confusion | Algorithm Confusion | JWT Format Confusion | Billion Hashes Attack | Compression Attack |
|---|---|---|---|---|---|
| **JWTeemo** | 43.2 | 8.7 | 27.2 | 2535.6 | 5349.4 |
| **JWTeemo w/o UCT Update** | 60.2 | 56.3 | 66.9 | 3204.4 | 20106.2 |
| **JWTeemo w/o Mutator** | 46.8 | 9.7 | 31.3 | N/A | N/A |

TABLE II: Time to Discover Vulnerabilities (in seconds) under different ablation settings.

| Tools | Time to Discover (s) | | | | | Covered Edges |
|---|---|---|---|---|---|---|
| | Sign/Encrypt Confusion | Algorithm Confusion | JWT Format Confusion | Billion Hashes Attack | Compression Attack | |
| **JWTeemo** | 43.2 | 8.7 | 27.2 | 2535.6 | 5349.4 | 780 |
| **JWT_Tool** | N/A | 1.1 | N/A | N/A | N/A | 473 |
| **JWT Editor** | N/A | 60✱ | N/A | N/A | N/A | 509 |

TABLE III: Comparison of JWTeemo with existing tools. ✱ indicates manual assistance is required.

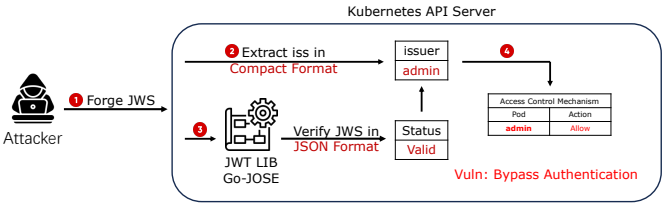



Fig. 10: Authentication Bypass in Kubernetes

```
{
    "fakeiss": ".eyJpc3MiOiJmYWtlaXNzdWVyIn0.",
    "protected": "real_header",
    "payload": "real_payload",
    "signature": "real_signature"
}
```

Listing 2: Simplified payload for exploiting the JWT vulnerability in Kubernetes.

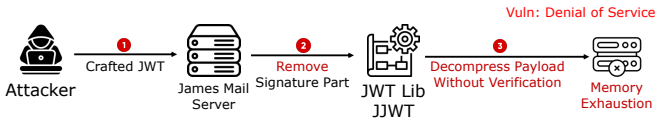



Fig. 11: Denial Of Service in Apache/James

cryptographically verifiable identity credentials. For example, Kubernetes uses JWT-based ServiceAccount tokens to allow pods to authenticate to the API server and to other services.

When verifying a ServiceAccount token, the Kubernetes API server must ensure the token was issued by the cluster's internal identity provider. This typically involves checking the `iss` (issuer) claim against the configured expected issuer. This validation step is crucial for establishing trust boundaries between workloads and control-plane components.

However, we discovered a logic flaw in Kubernetes token verification flow that leads to bypassing authentication, as shown in Figure 10. This vulnerability corresponds to the JWT Format Confusion issue discussed earlier, which arises from a discrepancy between how Kubernetes extracts the `iss` claim and how it verifies the JWT's signature. Specifically, the Kubernetes API server attempts to parse incoming tokens as if they are always in Compact format JWT—splitting on dots (`"."`) and base64-decoding the middle segment to extract claims like `iss`. In contrast, Kubernetes delegates cryptographic validation to the `go-jose` library [25], which also supports JSON format JWT, where the payload is stored in a named field of a JSON object. This mismatch can be exploited by crafting a JWT in JSON format that embeds a spoofed payload string (e.g., in the `fakeiss` field), as demonstrated in Listing 2.

Here, the `fakeiss` field contains a base64url-encoded string that decodes to `{"iss":"fakeissuer"}`. When Kubernetes's validation logic naively splits and decodes this string, it mistakenly interprets the forged issuer as valid. Meanwhile, the actual JWT signature remains intact and is verified by `go-jose` using the legitimate payload, which does not contain the spoofed issuer. As a result, the attacker can pass issuer validation while controlling the effective identity used for authorization. We reported this vulnerability to the Kubernetes security team and received a bug bounty for our disclosure.

Beyond Kubernetes, we identified a similar JWT vulnerability in another widely deployed distributed system: OpenShift's Telemeter [26], a Red Hat-operated telemetry component responsible for securely collecting and forwarding cluster metrics. Telemeter uses JWTs to authenticate incoming telemetry data sources. By applying the same exploitation technique, we were able to bypass Telemeter's authentication mechanism using a forged JSON-serialized token. This vulnerability was responsibly disclosed and has been assigned CVE-2024-5037 [27]. These findings illustrate that JWT format confusion in token processing is not an isolated mistake but a recurring class of bugs in real-world cloud-native systems, highlighting the importance of maintaining semantic consistency in token parsing and validation to ensure end-to-end security.

**Case Study 2: Compression DoS in Apache James-**

**Project.**

This vulnerability corresponds to the Compression DoS issue discussed earlier. Apache James-Project [28] is a complete, stable, secure, and scalable mail server developed by Apache. During SMTP login, it supports OAuth authentication where users provide a JWT token verified against an internally configured JWKs.

However, we discovered a logic flaw in Apache James-Project's token verification that leads to Denial-of-Service (DoS) attacks, as shown in Figure 11. In its JWT implementation, James-Project directly extracts the `kid` claim from the JWT payload to locate the verification key before signature verification. The JWT implementation used JJWT library [23], which supports the use of the `zip` claim in the header of JWS to indicate payload compression. Attackers can exploit this feature by crafting a JWT with a highly compressed long string. Because James-Project decompresses the payload before verification to extract the `kid`, an attacker can trigger excessive resource consumption without knowing the key, resulting in Denial-of-Service (DoS) attacks. We reported this vulnerability to the developers, who acknowledged and fixed this vulnerability.

## VI. Discussion

### A. Root Cause Analysis

The JWT security we discovered can be attributed to three major factors. First, JWT implementations often misunderstand the proper use of different JWT algorithms. For instance, in the case of Algorithm Confusion, developers allow public keys, intended for verifying asymmetric signature, to validate symmetric JWS signature without verifying whether the key is appropriate for symmetric algorithms. Similarly, in Sign/Encryption Confusion, while developers restrict keys to asymmetric algorithm, they fail to enforce further constraints limiting the key's use to either signature verification or decryption. These issues stem from a lack of enforcement in restricting keys to specific cryptographic functions.

Second, while the JWT RFC standard [6] supports a broad range of features, implementations often fail to implement these features in strict compliance with the specification. For example, in JWT Format Confusion, developers provide support for JSON Format JWS, which is explicitly not allowed in JWT RFC. Additionally, as discussed in Section V-E, developers implement the `zip` claim in JWS but it is also not allowed in JWT RFC.

Third, the JWT specification lacks sufficient risk warnings for certain claims, and many implementation developers may lack the necessary cryptographic expertise. Specifically, in the Billion Hashes Attack, the specification does not caution that the p2c claim could lead to excessive CPU consumption. Similarly, the specification does not address the risks associated with the zip claim, which can be exploited for a Compression DoS attack. RFC8725 [29] defines the Best Current Practices (BCP) for JSON Web Tokens. However, this standard is not actively maintained and has become outdated, failing to address the security issues identified in our research. By exposing these vulnerabilities, our work highlights the need for stronger security practices in JWT library development and calls for enhancements in the JWT specification to provide clearer guidance on potential risks.

### B. Mitigation

While RFC 8725 provides valuable document for JWT best current practice, we found four types of vulnerabilities we discovered are not covered in RFC 8725. We proposed new mitigations based on our findings: (1) Billion Hashes Attack: Recommend limiting the p2c claim size in PBES2-encrypted JWEs to prevent denial of service. (2) JWT Format Confusion: Advise against parsing JSON-type JWS in JWT implementations to mitigate authentication bypass risks. (3) Compression DoS: Suggest an upper limit on JWE payload lengths to prevent resource exhaustion attacks. (4) Sign/Encryption Confusion: Recommend excluding both public and private keys in JWKs, enforcing `use` claim checks, and clarifying JWS/JWE handling.

For JWT implementation developers, adherence to JWT standards is crucial. Developers should focus on the following aspects: (1) The usage of a JSON Web Key (JWK) must be strictly constrained to enhance security. According to RFC 7517, the `use` claim should be specified to distinguish whether a public key is intended for encrypting JWE or verifying JWS, reducing the risk of Sign/Encryption Confusion attacks. Additionally, the `alg` claim should also be implemented to restrict the key to specific algorithms, thereby mitigating Algorithm Confusion attacks. Developers must ensure that these claims are properly handled when processing JWKs. (2) JWT implementation developers should avoid supporting excessive or unnecessary features, such as parsing JWT in JSON format. By implementing these recommendations, both the JWT standard and JWT library developers can significantly reduce the risk of security vulnerabilities.

We contacted the IETF with the proposed mitigations to RFC 8725. The RFC 8725 authors have acknowledged our work are worthy of inclusion in the RFC and recognized that they would draft a new Best Current Practices (BCP) document to include our proposed mitigations.

### C. Limitation

Although JWTeemo can automatically detect parsing discrepancies among different JWT implementations, determining whether these discrepancies constitute security issues still requires manual analysis. In addition, since different JWT implementations support different algorithms and JWT structures (JWS/JWE), false positives may arise due to functional differences rather than security flaws. As large language models (LLMs) can understand and summarize discrepancies described in natural language, future work could leverage LLMs (e.g., GPT [30], Gemini [31], LLaMA [32]) to analyze and cluster these discrepancies, thereby reducing manual effort and improving analysis efficiency.

## VII. Related Work

### A. JWT Security

Previous work on JWT security can be broadly classified into two categories. On the one hand, tools like JWTKey [5] focus on identifying vulnerabilities related to key management, but overlook security risks arising from JWT implementations. On the other hand, approaches that attempt to address security issues caused by the characteristics of JWTs, such as the rule-based scanner JWT_Tool [13], are often ad-hoc and rely on manually crafted rules. This method lacks the ability to uncover new variants of attacks, as rules are often static and do not cover the evolving landscape of novel vulnerability patterns. To date, no systematic study has been conducted for JWT vulnerability. The attack in JWT Parkour [11] aims to "bypass the signature mechanism", and 5 attack methods and five mitigation measures are proposed; however, the attack methods proposed in this work are relatively scattered, and only target JWS, and no automated detection solution is proposed. In Three New Attacks Against JSON Web Tokens [12], the author proposed three new types of attack methods and proposed mitigation measures for JWT specification developers, JWT library developers, and JWT application developers. However, this work did not propose an automated detection solution. In summary, there is currently a lack of automated methods for exploiting vulnerabilities implemented by JWT, and most studies do not fully consider JWT's security issues. In contrast, our work systematically covers all forms of JWT and implements automated detection.

### B. Grammar-based Fuzzing

Grammar-based fuzzing has also been used in many works to automate vulnerability detection [19], [33], [34], [35], [36], [17]. One of the works is Wafmanis [33], which uses grammar-based fuzzing to discover the differences in HTTP protocol parsing between WAF and web applications, and bypass the WAF. REQSMINER [19] uses grammar-based fuzzing and UCT-Rand algorithm to detect inconsistencies in CDN forwarding. Both of the above works have achieved good results through Grammar-based Fuzzing.

## VIII. Conclusion

In this paper, we conducted the first systematic study of JWT vulnerability. We have introduced JWTeemo, a novel automated tool designed to detect JWT application vulnerabilities. We evaluated JWTeemo on 43 real-world JWT libraries. JWTeemo discovered 31 new JWT vulnerabilities and 20 new CVE IDs were assigned, demonstrating the practical utility of JWTeemo in JWT vulnerability detection. We responsibly disclosed our vulnerabilities and provided mitigation measures for different JWT users. We hope our work can aid the community in addressing the rising threats of JWT vulnerabilities.

## IX. Ethics Considerations

First, our research sets up testing environments on our local server to avoid interfering with real-world websites and networks. Second, we responsibly reported all identified vulnerabilities in detail to the corresponding vendors and CVE Numbering Authorities, and received positive responses and acknowledgments. Specifically, we followed the responsible disclosure procedures described in each vendor's Security Policy when available. If a vendor did not provide a public Security Policy, we reported the issue through their official repository channels, such as GitHub or Bitbucket issue trackers. Among them, 23 vulnerabilities have been patched with 20 CVEs assigned. The remaining cases are still under evaluation by developers. Up to now:

- **Apache** acknowledged our vulnerability report to James-Project [28] and released a fix.
- **Connect2id** acknowledged our report to nimbus-jose-jwt [37] and has assigned a CVE identifier for the reported vulnerabilities.
- **Kubernetes** accepted our report, fixed the vulnerability, and awarded us a bug bounty.
- **Latchset** acknowledged our report to jwcrypto [38] and jose [39] and has assigned CVE identifiers for the reported vulnerabilities.
- **Let's Encrypt** acknowledged our report to go-jose [25] and has assigned a CVE identifier for the reported vulnerability.
- **RedHat** acknowledged our reports to OpenShift Telemeter [26], a RedHat-operated telemetry service, and assigned a CVE identifier for the reported vulnerabilities.

In addition, we have reported the mitigation strategies proposed in this paper to the authors of RFC 8725. These mitigations have been formally presented and discussed at an IETF meeting, encouraging feedback from participating experts. The working group is currently incorporating our findings into the new JWT Best Current Practice (BCP) Draft RFC, which has reached version 02[2] and is expected to enter Working Group Last Call soon.


## Acknowledgment

We would like to express our sincere gratitude to the anonymous reviewers for their valuable comments and suggestions that greatly enhanced the quality of this paper. This work is supported by the National Natural Science Foundation of China (grant #62272265).


## References


[1] "Cloudflare," https://www.cloudflare.com/.

[2] "Let's encrypt," https://letsencrypt.org/.

[3] "Kubernetes," https://kubernetes.io/.

[4] "Cve-2023-29357," https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-29357.

[5] B. Xu, S. Jia, J. Lin, F. Zheng, Y. Ma, L. Liu, X. Gu, and L. Song, "Jwtkey: Automatic cryptographic vulnerability detection in jwt applications," in *Computer Security – ESORICS 2023*, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds. Cham: Springer Nature Switzerland, 2024, pp. 263–282.

[6] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7519

[7] ——, "JSON Web Signature (JWS)," RFC 7515, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7515


[2]https://datatracker.ietf.org/doc/draft-ietf-oauth-rfc8725bis/02/

[8] M. B. Jones and J. Hildebrand, "JSON Web Encryption (JWE)," RFC 7516, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7516
[9] M. B. Jones, "JSON Web Algorithms (JWA)," RFC 7518, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7518
[10] ——, "JSON Web Key (JWK)," RFC 7517, May 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7517
[11] L. Nyfenegger and L. Jahnke, "Jwat - attacking json web tokens," https://media.defcon.org/DEF%20CON%2026/DEF%20CON%2026%20workshops/DEF%20CON%2026%20-%20Workshop-Louis-Nyfenegger-and-Luke-Jahnke-JWAT-Attacking-JSON-WEB-TOKENS.pdf, Aug. 2018, dEF CON 26 Workshop.
[12] "Three new attacks against json web tokens," https://i.blackhat.com/BH-US-23/Presentations/US-23-Tervoort-Three-New-Attacks-Against-JSON-Web-Tokens.pdf.
[13] "A toolkit for testing, tweaking and cracking json web tokens resources," https://github.com/ticarpi/jwt_tool.
[14] "Jwt-editor," https://github.com/portswigger/jwt-editor.
[15] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632.
[16] J. Möller, F. Weißberg, L. Pirch, T. Eisenhofer, and K. Rieck, "Cross-language differential testing of json parsers," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1117–1127. [Online]. Available: https://doi.org/10.1145/3634737.3657003
[17] Y. You, J. Chen, Q. Wang, and H. Duan, "My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers," in *34th USENIX Conference on Security Symposium*, 2025.
[18] OASIS Security Services Technical Committee, "Security assertion markup language (saml) v2.0," https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf, 2005, accessed: 2025-08-07.
[19] L. Zheng, X. Li, C. Wang, R. Guo, H. Duan, J. Chen, C. Zhang, and K. Shen, "Reqsminer: Automated discovery of cdn forwarding request inconsistencies and dos attacks with grammar-based fuzzing."
[20] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from {CPU-Exhaustion}{Denial-of-Service} attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 393–410.
[21] "tiobe," https://www.tiobe.com/tiobe-index/.
[22] "jwt.io," https://jwt.io/.
[23] "jjwt library," https://github.com/jwtk/jjwt.
[24] "jwx," https://github.com/lestrrat-go/jwx.
[25] "go-jose," https://github.com/go-jose/go-jose.
[26] R. Hat, "Openshift telemeter," https://github.com/openshift/telemeter, 2025, accessed: 2025-08-07.
[27] "Cve-2024-5037," https://nvd.nist.gov/vuln/detail/CVE-2024-5037, Jun. 2024, national Vulnerability Database; CVSS v3.1 base score 7.5.
[28] A. S. Foundation, "Apache james project," https://james.apache.org, 2025, accessed: 2025-08-07.
[29] Y. Sheffer, D. Hardt, and M. B. Jones, "JSON Web Token Best Current Practices," RFC 8725, Feb. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8725
[30] OpenAI, "Chatgpt," https://chatgpt.com, 2022.
[31] G. DeepMind, "Gemini - google deepmind," https://deepmind.google/technologies/gemini, 2024.
[32] M. AI, "Introducing llama: A foundational, 65-billion-parameter large language model," https://ai.meta.com/blog/large-language-model-llama-meta-ai/, 2023.
[33] Q. Wang, J. Chen, Z. Jiang, R. Guo, X. Liu, C. Zhang, and H. Duan, "Break the wall from bottom: Automated discovery of protocol-level evasion vulnerabilities in web application firewalls," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 129–129.
[34] J. Zhang, J. Chen, Q. Wang, H. Zhang, C. Wang, J. Zhuge, and H. Duan, "Inbox Invasion: Exploiting MIME Ambiguities to Evade Email Attachment Detectors," in *31th ACM Conference on Computer and Communications Security*, 2024.
[35] K. Mu, J. Chen, J. Zhuge, Q. Li, H. Duan, and N. Feamster, "The Silent Danger in HTTP: Identifying HTTP Desync Vulnerabilities with Gray-box Testing," in *34th USENIX Conference on Security Symposium*, 2025.
[36] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations," in *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022.
[37] "nimbus-jose-jwt," https://bitbucket.org/connect2id/nimbus-jose-jwt.
[38] "jwcrypto," https://github.com/latchset/jwcrypto.
[39] "latchset/jose," https://github.com/latchset/jose.
[40] "pyjwt," https://github.com/jpadilla/pyjwt.
[41] "python-jose," https://github.com/mpdavis/python-jose.
[42] "authlib," https://github.com/lepture/authlib.
[43] "python-jwt," https://github.com/GehirnInc/python-jwt.
[44] "libjwt," https://github.com/benmcollins/libjwt.
[45] "l8w8jwt," https://github.com/GlitchedPolygons/l8w8jwt.
[46] "poco," https://github.com/pocoproject/poco.
[47] "jwt-cpp," https://github.com/Thalhammer/jwt-cpp.
[48] "cpp-jwt," https://github.com/arun11299/cpp-jwt.
[49] "java-jwt," https://github.com/auth0/java-jwt.
[50] "fusionauth-jwt," https://github.com/fusionauth/fusionauth-jwt.
[51] "jose4j," https://bitbucket.org/b_c/jose4j.
[52] "jose-jwt," https://github.com/dvsekhvalnov/jose-jwt.
[53] "jwt," https://github.com/jwt-dotnet/jwt.
[54] "System.identitymodel.tokens.jwt," https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet.
[55] "jose," https://github.com/panva/jose.
[56] "jsonwebtoken," https://github.com/auth0/node-jsonwebtoken.
[57] "node-jose," https://github.com/cisco/node-jose.
[58] "aws-jwt-verify," https://github.com/awslabs/aws-jwt-verify.
[59] "jose-php," https://github.com/nov/jose-php.
[60] "jwt-framework," https://github.com/web-token/jwt-framework.
[61] "lcobucci/jwt," https://github.com/lcobucci/jwt.
[62] "adhocore/php-jwt," https://github.com/adhocore/php-jwt.
[63] "cdoco/php-jwt," https://github.com/cdoco/php-jwt.
[64] "namshi/jose," https://github.com/namshi/jose.
[65] "firebase/php-jwt," https://github.com/firebase/php-jwt.
[66] "golang-jwt/jwt," https://github.com/golang-jwt/jwt.
[67] "jose2go," https://github.com/dvsekhvalnov/jose2go.
[68] "gbrlsnchs/jwt," https://github.com/gbrlsnchs/jwt.
[69] "cristalhq/jwt," https://github.com/cristalhq/jwt.
[70] "kataras/jwt," https://github.com/kataras/jwt.
[71] "pascaldekloe/jwt," https://github.com/pascaldekloe/jwt.
[72] "sjwt," https://github.com/brianvoe/sjwt.
[73] "json-jwt," https://github.com/nov/json-jwt.
[74] "ruby-jwt," https://github.com/jwt/ruby-jwt.
[75] "Jsonwebtoken.swift," https://github.com/kylef/JSONWebToken.swift.
[76] "jwt-kit," https://github.com/vapor/jwt-kit.

## APPENDIX

### A. Definitions for Function Invocation and Function Selection

Listing 3 defines the formal ABNF-style grammar used to specify the structure of function invocations and function selection expressions. The rules describe how a function name, its parameters, conditional selection constructs, and key–value selection mappings are syntactically formed.

Listing 3: Formal ABNF-style Definitions for Function Invocation and Function Selection

```
func-invocation   = func-name "(" [ parameters ] ")"
func-name         = ALPHA *( ALPHA / DIGIT / "_" )
parameters        = parameter *( "," parameter )
parameter         = rulename

func-selection    = "if" "(" condition "," selection-map ")"
condition         = rulename
selection-map     = "{" selection-entry *( "," selection-entry ) "}"
selection-entry   = quoted-key ":" rulename

quoted-key        = DQUOTE *( %x21-21 / %x23-5B / %x5D-7E ) DQUOTE
```

### B. Evaluation Dataset

Table IV lists the JWT libraries included in the evaluation dataset. For each library, the table provides its programming language, version, and GitHub star count where available.

| Language | Library | Version | Stars |
|---|---|---|---|
| Python | pyJWT[40] | 2.8.0 | 4.8k |
| Python | python-jose[41] | 3.3.0 | 1.5k |
| Python | jwcrypto[38] | 1.5.0 | 430 |
| Python | authlib[42] | 1.2.1 | 4.5k |
| Python | python-jwt[43] | 1.3.1 | 140 |
| C | jose[39] | 11 | 170 |
| C | libjwt[44] | 2.1.0 | 368 |
| C | l8w8jwt[45] | 2.2.1 | 111 |
| C++ | poco[46] | 1.12.5p2 | 7.6k |
| C++ | jwt-cpp[47] | 0.7.0 | 681 |
| C++ | cpp-jwt[48] | 1.4 | 387 |
| Java | jjwt[23] | 0.12.3 | 10.1k |
| Java | java-jwt[49] | 4.4.0 | 5.5k |
| Java | fusionauth-jwt[50] | 5.3.0 | 153 |
| Java | jose4j[51] | 0.9.3 | N/A |
| Java | nimbus-jose-jwt[37] | 9.37.1 | N/A |
| C# | jose-jwt[52] | 4.1.0 | 933 |
| C# | jwt[53] | 10.1.1 | 2.1k |
| C# | System.IdentityModel.Tokens.Jwt[54] | 7.2.0 | 1k |
| JavaScript | jose[55] | 5.1.3 | 5.3k |
| JavaScript | jsonwebtoken[56] | 9.0.2 | 17.1k |
| JavaScript | node-jose[57] | 2.2.0 | 699 |
| JavaScript | aws-jwt-verify[58] | 4.0.0 | 518 |
| PHP | jose-php[59] | 2.2.1 | 138 |
| PHP | jwt-framework[60] | 3.2.8 | 881 |
| PHP | jwt[61] | 5.2.0 | 7.1k |
| PHP | adhocore/php-jwt[62] | 1.1.2 | 271 |
| PHP | cdoco/php-jwt[63] | 1.0.0 | 232 |
| PHP | jose[64] | 7.2.3 | 1.8k |
| PHP | firebase/php-jwt[65] | 6.10.0 | 1.4k |
| Go | golang-jwt/jwt[66] | 5.2.0 | 16.4k |
| Go | jose2go[67] | 1.5.0 | 186 |
| Go | go-jose[25] | 3.0.1 | 2.3k |
| Go | jwx[24] | 2.0.17 | 1.9k |
| Go | gbrlsnchs/jwt[68] | 3.0.1 | 442 |
| Go | cristalhq/jwt[69] | 5.4.0 | 626 |
| Go | kataras/jwt[70] | 0.1.12 | 188 |
| Go | pascaldekloe/jwt[71] | 1.0.12 | 339 |
| Go | sjwt[72] | 0.5.1 | 114 |
| Ruby | json-jwt[73] | 1.16.3 | 297 |
| Ruby | ruby-jwt[74] | 2.7.1 | 3.5k |
| Swift | JSONWebToken[75] | 2.2.0 | 763 |
| Swift | jwt-kit[76] | 4.13.4 | 180 |

TABLE IV: The targets evaluated by JWTeemo. N/A indicates the library is not an open source on GitHub.

### C. Differences between implementations

Figure 12 illustrates the five types of differences between JWT implementations found by JWTEEMO, including Sign/Encryption Confusion, Algorithm Confusion, JWT Format Confusion, Different Claims Checker, Different Algorithm Support.

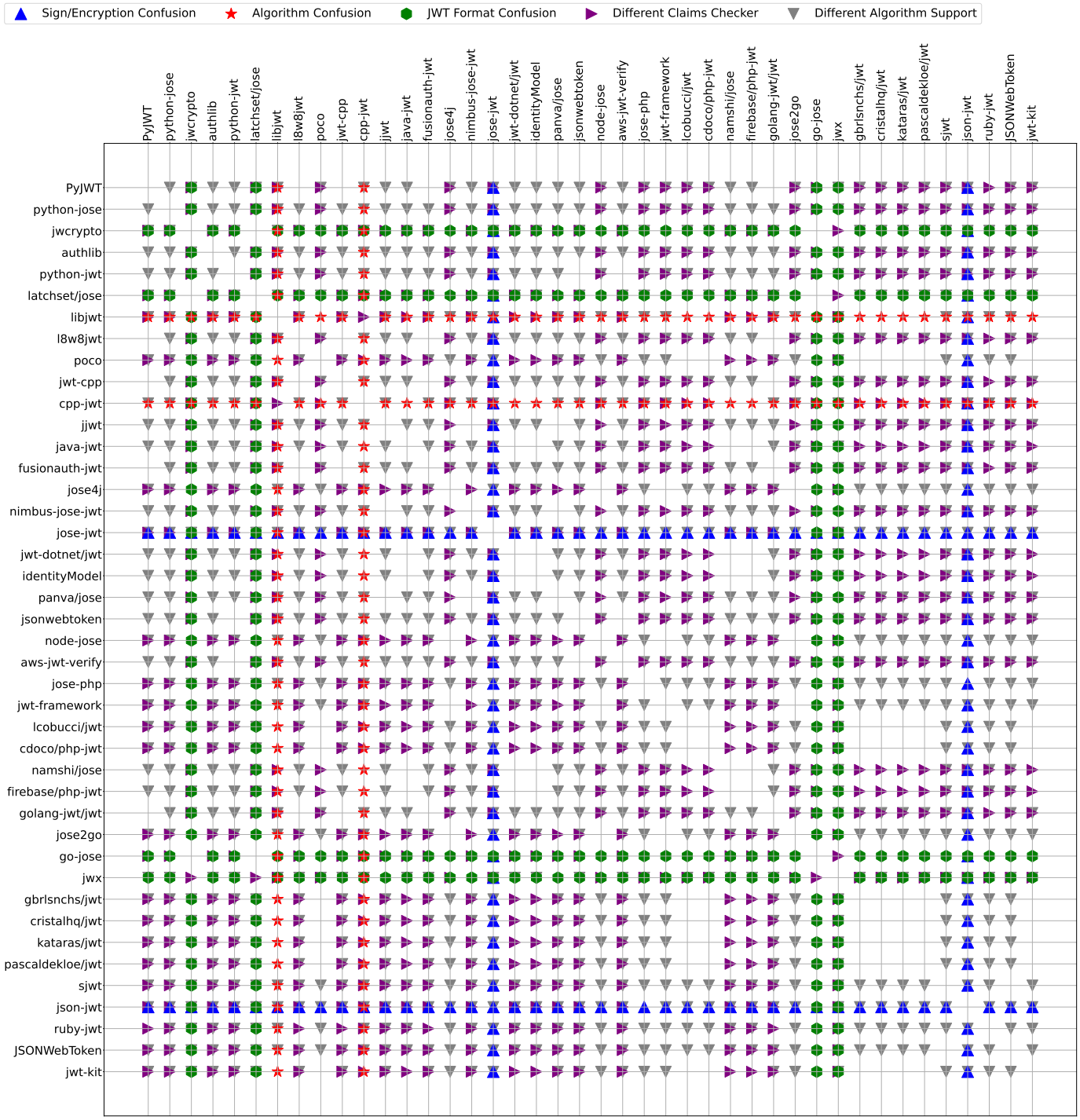
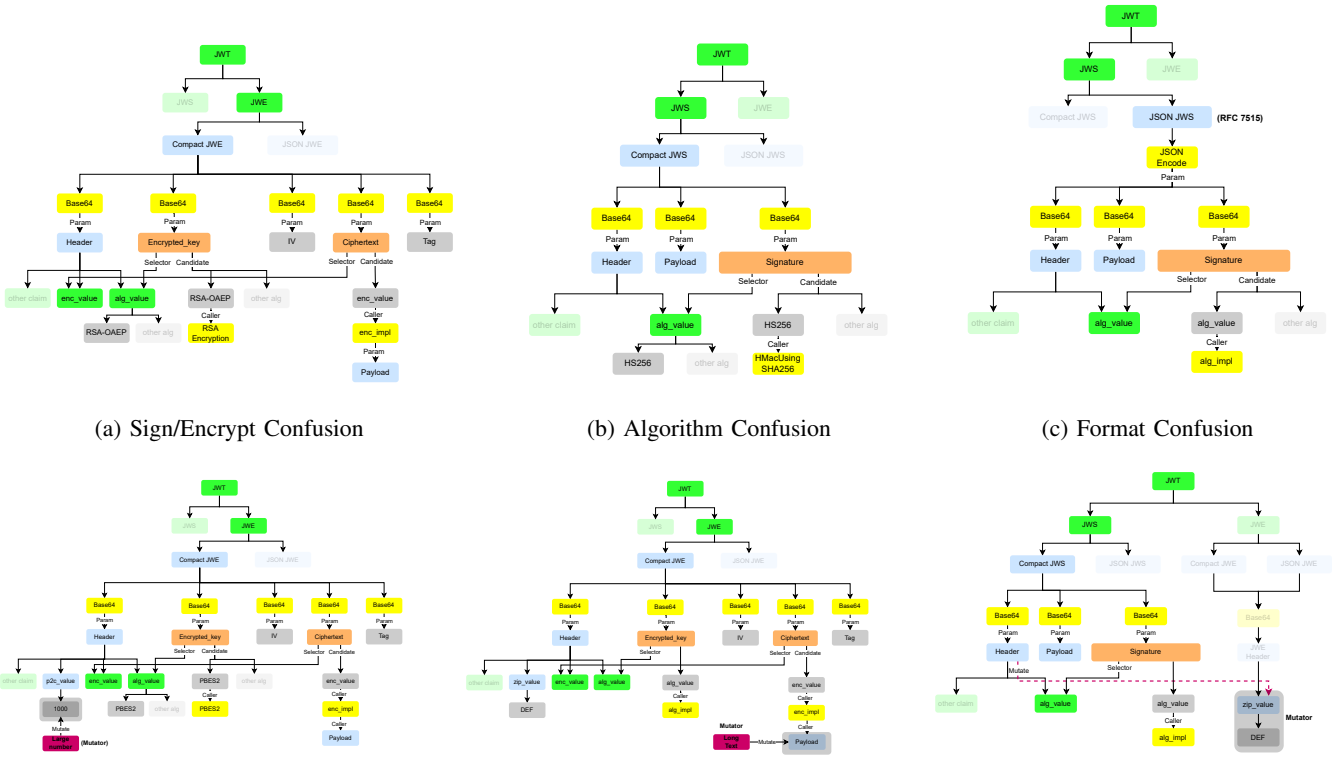



Fig. 12: The five types of differences between JWT implementations.

### D. Payload Generation via FBNF Graph

Figure 13 shows how the JWTs that trigger vulnerabilities are derived from the FBNF definitions and how the mutator module operates on them. Each subfigure shows the FBNF subgraph used by JWTEEMO, where opaque nodes mark the elements selected by the generator and used to construct the resulting JWTs.




(a) Sign/Encrypt Confusion (b) Algorithm Confusion (c) Format Confusion

(d) Billion Hashes Attack (e) Compression DoS (JWE path) (f) Compression DoS (JWS path)

Fig. 13: Payload generation from the FBNF graph.