PathFault: Automated Exploit Generator for Web Services via HTTP Message Parser Discrepancies

Juryeok Kim[0009−0001−1457−7631] and Youngjoo Shin[0000−0003−4831−7392]⋆

School of Cybersecurity, Korea University, Seoul, 02841, Republic of Korea
{ryan9423,syoungjoo}@korea.ac.kr

Abstract. Modern web services utilize complex architectures involving diverse HTTP processing components. These components include CDNs, WAFs, proxies, and web servers (e.g., Nginx, Apache, Cloudflare, and GCP). Independent parsing rules across these components lead to discrepancies in interpreting identical HTTP requests, including the URI. This phenomenon, known as Path Confusion, leads to critical security vulnerabilities, such as authentication bypass and sensitive data exposure. However, existing penetration testing methods fail to identify these vulnerabilities adequately. This limitation stems from limited system-wide visibility and operational risks in live environments. To address these challenges, we present PathFault, an automated methodology leveraging a Predicate Logic-based Surrogate Model. This model systematically abstracts component-specific parsing behaviors, enabling precise vulnerability analysis with minimal invasive testing. Furthermore, PathFault incorporates SMT solving techniques to automatically generate targeted exploit payloads. It combines security researchers' domain expertise with web service-specific parsing logic. Consequently, our approach significantly improves detection accuracy. It comprehensively assesses dynamic system interactions and addresses limitations of traditional root-cause analysis. Evaluation demonstrates PathFault's efficacy in identifying previously overlooked Path Confusion vulnerabilities.

Keywords: Web Security · Automated Exploit Generation · Path Confusion Vulnerability · HTTP Parser Discrepancy

1 Introduction

Modern web services, such as content delivery networks (CDNs), web application firewalls (WAFs), and proxies, increasingly rely on complex architectures that consist of multiple HTTP processing components [9, 17, 36]. Each web service component independently implements parsing rules for HTTP messages [7, 35, 38, 41, 47]. This independence can lead to discrepancies in the interpretation of identical HTTP requests, including URI [2, 18, 19, 48]. Such discrepancies in parsing, known as *Path Confusion*, create significant security vulnerabilities

⋆ Corresponding author: syoungjoo@korea.ac.kr

[14, 23, 24, 29, 30]. Specifically, path confusion vulnerabilities pose serious security threats such as authentication bypass [20, 33] and unauthorized disclosure of secret data [14, 15, 24]. For example, the CVE-2025-0108 vulnerability shows how different HTTP message parsing allows attackers to bypass authentication mechanisms [20]. Web cache deception (WCD) attacks similarly exploit these discrepancies, exposing sensitive information through incorrect caching behavior [14, 15, 23, 24, 29, 30]. These real-world examples underscore the considerable need to address path confusion vulnerabilities.

However, identifying and mitigating path confusion vulnerabilities pose significant challenges. First, authentication and caching in web services depend on service-specific URI patterns and the attributes that trigger them. But legacy penetration testing and vulnerability detection tools [23, 24] are unable to integrate such dynamic URIs and the security researcher's domain knowledge. That is, these tools cannot adapt to the unique characteristics of each service. Consequently, since they fail to capture how these mechanisms are triggered and influenced by component-level configurations, the scope and precision of vulnerability detection are restricted [3, 43, 50]. Furthermore, conducting penetration testing in live production environments poses considerable risks [34, 40]. These risks include potential exposure of user data and operational instability, which severely restricts the scope and depth of testing [16, 23].

To overcome these limitations, this work proposes *PathFault*, an automated approach to systematically detect path confusion vulnerabilities across complex web services. PathFault introduces a Predicate Logic-based Surrogate Model. This model abstracts the HTTP parsing rules employed by each web component. Such abstraction enables precise system-level vulnerability analysis without extensive direct testing on live systems. Furthermore, leveraging the expressive power of SMT (Satisfiability Modulo Theories) [12] solving techniques, PathFault automatically generates targeted exploit payloads. It integrates domain knowledge from security researchers with the service-specific parsing behaviors.

Our surrogate model facilitates comprehensive identification and precise analysis of the root causes underlying path confusion vulnerabilities. It significantly reduces reliance on invasive testing methods. Simultaneously, the SMT-based Exploit Payload Generator substantially enhances detection accuracy and scope. It systematically addresses dynamic aspects such as caching policies and proxy configurations, typically overlooked by conventional methods.

In summary, PathFault addresses critical gaps in current vulnerability detection practices through the following key contributions:

- This work enables precise white-box analysis of HTTP parsing behaviors in complex web systems by developing a predicate logic-based surrogate model.
- It broadens the scope and precision of vulnerability detection by designing an SMT-solving-based exploit payload generator that integrates a security researcher's domain knowledge with component-specific parsing logic.
- It overcomes limitations in traditional root cause analysis of path confusion vulnerabilities through a systematic methodology that models and evaluates system-level component interactions.

The source code for PathFault is publicly available at `https://github.com/koreacsl/PathFault`.

2 Background

2.1 HTTP message's URI

A Uniform Resource Identifier (URI) is a standardized method used to identify resources and facilitate their access within web [13]. RFC 3986 [8] defines a URI as consisting of several distinct components, including scheme, authority, path, query, and fragment. The scheme component specifies the protocol used for retrieving the resource, such as HTTP or HTTPS. The authority typically includes the host and port, defining the location of the server hosting the resource.

The path, query, and fragment components can be demonstrated using an example URI: `https://example.com/products/electronics/laptop?item=123&sort=price#details`

In this example, the path component `"/products/electronics/laptop"` represents the hierarchical structure of the resource on the server. It consists of segments separated by slashes (`"/"`), with each segment typically corresponding to a directory or a specific resource. The query component is utilized for the resource retrieval with prefixed `"?"` and parameters such as `"item=123"` and `"sort=price"`. The fragment component with prefixed `"#"` points to a subsection or element within the primary resource, such as `"details"` in the example.

Meanwhile, proper encoding and decoding of URIs are crucial to avoid misinterpretation and ensure accurate resource identification and access in web services. For these, percent encoding is used to encode reserved and non-ASCII characters within a URI, ensuring their correct transmission over networks. For example, the percent encoding converts a space as `"%20"` and a slash as `"%2F"`.

2.2 Path confusion-based attacks

Path confusion vulnerabilities can be exploited by attackers to compromise web security [31]. Path confusion arises when different web service components differently interpret the same URI path. For example, by percent-encoding the forward slash `"/"` as `"%2F"`, an attacker can craft a URI like `"/A%2FB"`, which may be interpreted as either two segments `"/A/B"` or a single segment `"/A%2FB"` depending on how each component decodes the path. Such discrepancies can be abused to bypass security controls or induce unintended service behavior.

Two prominent attack types leveraging path confusion are web cache attacks and authentication bypass attacks.

Web cache attack. Web cache attacks abuse discrepancies between cache servers and origin servers in interpreting paths to compromise web services. These attacks are typically known as either cache poisoning [21, 27, 32] or cache deception [14, 23, 24]. In a cache poisoning attack, attackers manipulate cache servers to store and serve malicious HTTP responses, thereby distributing harmful content like malicious scripts to end-users. In cache deception attacks, attackers abuse caching mechanisms to unintentionally store sensitive user data

Table 1. Notation of Predicate Logic

Notation	Definition	Example
$P(x)$	Object x has property P	$ContainsDotSegment(x)$: True if the string x contains the substring "/./".
$R(x, y)$	Relationship R between objects x and y	$Contains(x, \text{"/./"})$: True if the string x contains the substring "/./".
$\wedge$	Logical AND	$Contains(x, \text{"/./"}) \wedge Contains(x, \text{"\#"})$: Both substrings "/./" and "#" appear in x.
$\vee$	Logical OR	$Contains(x, \text{"/./"}) \vee Contains(x, \text{"\#"})$: At least one of the substrings "/./" or "#" appears in x.
$\neg$	Logical NOT	$\neg Contains(x, \text{"\#"})$: The string x does not contain the substring "#".
$\rightarrow$	Logical implication	$Contains(x, \text{"\#"}) \rightarrow RemoveSuffix(x, \text{"\#"})$: If the string x contains "#", then the suffix after "#" should be removed.
$\leftrightarrow$	Logical equivalence	$ContainsDotDot(x) \leftrightarrow Contains(x, \text{".."})$: Defines a predicate as logically equivalent to a binary relation.

in publicly accessible cache servers, which the attackers subsequently retrieve. Both attacks rely on the discrepant handling of paths between cache servers and origin servers, demonstrating the severe implications of path confusion.

Authentication bypass. Authentication bypass attacks leverage path confusion to evade security checks. For example, the CVE-2025-0108 [20, 33] vulnerability demonstrates how double percent-encoded dot segments (`"%252E%252E"`) can allow certain components, such as Nginx, to bypass authentication checks. As a result, protected resources hosted by components such as the Apache HTTP Server can be accessed without proper authorization. Such attacks highlight the critical security risks posed by path confusion, underscoring the necessity for consistent and robust path parsing across web service components.

2.3 Formal logical modeling and SMT-based analysis of path confusion

To effectively analyze path confusion, it is crucial to represent the transformation logic in a form that is both interpretable and inferable. To achieve this clarity, predicate logic notation is adopted, clearly expressing component interactions and transformation conditions. Table 1 summarizes the notation conventions applied throughout this paper. Specifically, the notation $P(x)$ indicates that an object x satisfies a certain property, whereas $R(x, y)$ expresses a relationship between two objects. Logical connectives, such as $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$, are utilized to construct predicates and articulate implications or equivalences explicitly.

Using this representation grounded in predicate logic, SMT solvers are employed to systematically assess these logical conditions for their satisfiability. SMT solvers specifically handle string constraints, enabling automatic generation of exploit payloads. This automated process allows for precise and reproducible testing of discrepancies in path interpretation.

3 System Model

This section formally defines the foundational concepts underpinning PathFault. First, Section 3.1 introduces the concept and formal definition of Path Confusion Patterns, which classify discrepancies of the path interpretation of web service components. Next, Section 3.2 presents the Predicate Logic-based Surrogate

Model and its formal definition for analyzing discrepancies in real web services. Finally, Section 3.3 formally defines Path Confusion Vulnerability and explains how PathFault detects such vulnerabilities through the surrogate model.

3.1 Path confusion patterns

Path confusion occurs when web service components interpret the same URI differently, leading to discrepancies. These discrepancies, stemming from diverse causes, are classified into distinct patterns. This section introduces *Path Confusion Patterns*, derived from a systematic analysis of their causes, and formally defines each pattern.

Path confusion patternization To systematically identify path confusion patterns, this study analyzes responses of web service components utilizing three types of confusable URIs:

- `"/tmp1/tmp2{char}tmp3/tmp4"`
- `"/tmp1/{char}/tmp2"`
- `"/a/../b"`

Here, `"{char}"` represents a single-byte character from `0x00` to `0xff`, covering the full ASCII range. This approach ensures detection of edge cases, such as null bytes and control characters that affect path interpretation. Consequently, this study classifies path confusion patterns in four distinct forms, as detailed below.

Full subsequent path removal. This pattern occurs when a web component truncates all segments following a specific character. For example, Nginx interprets the URI `"/tmp1/tmp2#tmp3/tmp4"` by truncating segments after the hash (`"#"`) , resulting in `"/tmp1/tmp2"`. However, Apache HTTP Server rejects paths containing `"#"`, highlighting discrepant handling.

Partial current segment removal. This pattern involves removing portions of a segment after specific characters. For instance, Apache Tomcat processes `"/tmp1/tmp2;tmp3/tmp4"` by truncating the segment after the semicolon (`";"`), resulting in `"/tmp1/tmp2/tmp4"`.

String substitution. This pattern involves the replacement of path substrings. For example, Apache Tomcat replaces `"//"` with `"/"` and `"/?/"` with `"?/"`.

Single previous segment removal. This pattern involves normalizing relative paths with dot-dot sequences (`".."`), as specified in RFC 3986 [8]. For instance, `"/a/../b"` should normalize to `"/b"`.

Path confusion formalization This section formalizes the definitions of *Path Confusion*, *Path Confusion Function*, and *Path Confusion Pattern* in sequence to model path confusion behaviors in web services.

Definition 1 (Path confusion). *A Path Confusion $\mathcal{P}$ is a set of pairs $\langle t, c \rangle$, where $t \in \mathbb{T}$ is a transformation function performing string manipulations, and*

Table 2. Transformation and Condition Types

Notation*	Description
$T_r(t, r, s)$	Replace all occurrences of t in s with r.
$T_o(o, s)$	Extract the substring of s starting from offset o.
$T_u(t, s)$	Extract the prefix of s up to the first occurrence of t.
$T_n(t, d, s)$	Remove from the previous d to the first t, inclusive, in s.
$T_p(t, s)$	Prepend string t to the beginning of s.
$T_s(t, s)$	Append string t to the end of s.
$T_d(t, d, s)$	Remove characters from t to the next d in s.
$C_e(t, s)$	s is exactly equal to t.
$C_s(t, s)$	s ends with substring t.
$C_p(t, s)$	s starts with substring t.
$C_c(t, s)$	s contains substring t.
$C_d(t, d, s)$	There is a d following the first occurrence of t in s.

*Variables t, r, and d represent strings and belong to Σ^*, the set of all possible strings; $o \in \mathbb{Z}_{\geq 0}$ is a non-negative integer offset.

$c \subseteq \mathbb{C}$ is a condition predicate evaluating the truth value of an input string $s \in \Sigma^*$. Formally,

$$\begin{aligned} \mathcal{P} &= \{\langle t, c\rangle \mid t \in \mathbb{T}, c \subseteq \mathbb{C}\}, \\ \mathbb{T} &= T_r \cup T_o \cup T_u \cup T_n \cup T_p \cup T_s \cup T_d, \\ \mathbb{C} &= C_e \cup C_s \cup C_p \cup C_c \cup C_d \end{aligned} \tag{1}$$

Each transformation $T_* \subset \mathbb{T}$ performs string manipulations, such as substitution or deletion, while each predicate $C_* \subset \mathbb{C}$ determines transformation eligibility, enabling formal analysis of path confusion patterns. Details of $\mathbb{T}$ and $\mathbb{C}$ components, designed for path confusion patterns, are provided in Table 2.

Definition 2 (Path confusion function). *The Path Confusion Function $\mathcal{P}(s)$ for an input string $s \in \Sigma^*$ applies a transformation $t \in \mathbb{T}$ if all predicates $c_i \in c \subseteq \mathbb{C}$ are satisfied. Here, $t(s)$ denotes the result of performing the transformation corresponding to t on s. Formally,*

$$\mathcal{P}(s) = \begin{cases} t(s) & \text{if } \bigwedge_{c_i \in c} c_i(s) \\ s & \text{otherwise} \end{cases} \tag{2}$$

Definition 3 (Path Confusion Pattern). *The four Path Confusion Patterns P previously identified are formalized through the $\mathcal{P}(s)$, where each pattern $P \subset \mathcal{P}$ is defined by a transformation and condition pair. Formally,*

$$\begin{aligned} P_f(t, s) &= \{\langle T_u(t, s), \{C_c(t, s)\}\rangle\} \\ P_s(t, r, s) &= \{\langle T_r(t, r, s), \{C_c(t, s)\}\rangle\} \\ P_c(t, s) &= \left\{ \begin{array}{l} \langle T_d(t, \text{“/”}, s), \{C_c(t, s), C_d(t, \text{“/”}, s)\}\rangle, \\ \langle T_u(t, s), \{C_c(t, s), \neg C_d(t, \text{“/”}, s)\}\rangle \end{array} \right\} \\ P_p(s) &= \{\langle T_n(\text{“/../”}, \text{“/”}, s), \{C_c(\text{“/../”}, s)\}\rangle\} \\ &\textit{where } t, r, s \in \Sigma^*, \quad o \in \mathbb{Z}_{\geq 0} \end{aligned} \tag{3}$$

Each pattern represents a distinct path manipulation:

– Full subsequent path removal $P_f(t, s)$ denotes a transformation that eliminates all path segments occurring after a specified target string t inside the input string s if t is present.

- String substitution $P_s(t, r, s)$ denotes a transformation that replaces all occurrences of the target string t in the input string s with the replacement string r.
- Partial current segment removal $P_c(t, s)$ denotes a transformation that either removes a segment after t until that slash if a slash follows t, or truncates a prefix up to t otherwise.
- Single previous segment removal $P_p(s)$ denotes a transformation that normalizes and eliminates the directory sequence `"/../"` within the input string s if present.

Table 7 in Appendix A presents a concise set $\mathcal{P}$ and whether percent decoding is performed, constructed from common web service components.

3.2 Predicate logic-based surrogate model

As modern web services comprise multiple components [5, 9, 17, 36, 44], identifying vulnerability causes precisely is challenging [24]. Moreover, direct testing on live systems poses risks of data exposure or disruptions [23, 24, 34, 40]. To address these, *Predicate Logic-based Surrogate Model* abstracts a web service while retaining only essential details for analyzing path confusion vulnerabilities. Thus, this model enables systemic path confusion analysis with minimal live system interaction. Built on $\mathcal{P}$ from Section 3.1, the model represents the service as a component sequence.

Definition 4 (Surrogate model). *The Surrogate Model $\mathcal{S}$ and its components $\mathcal{W}$ are defined as follows:*

$$\begin{gathered}\mathcal{S} = [\,\mathcal{W}_1,\ \mathcal{W}_2,\ \ldots,\ \mathcal{W}_n\,],\\ \mathcal{W} = (\,\mathit{isNorm},\ \mathit{isDecode},\ \mathit{Conditions},\ \mathit{Transformations}),\\ \text{where } \mathit{isNorm}, \mathit{isDecode} \in \{\,\mathit{True},\ \mathit{False}\} \text{ and } \mathit{Conditions} \subseteq \mathbb{C},\ \mathit{Transformations} \subseteq P\end{gathered} \tag{4}$$

These attributes characterize the component's behavior in processing requests:

- *isNorm* $\in \{\text{True}, \text{False}\}$ indicates whether the component performs dot-segment normalization, resolving relative path sequences like `"/../"`.
- *isDecode* $\in \{\text{True}, \text{False}\}$ indicates whether the component performs percent decoding, such as converting `"%2F"` to `"/"`.
- *Conditions* $\subseteq \mathbb{C}$ means the conditions under which it can process the request.
- *Transformations* $\subseteq P$ denotes the subset of path confusion patterns contained by a component.

Consequently, it models web services, where requests are processed by chaining the results interpreted by each component to subsequent components.

3.3 Path confusion vulnerability

Although path confusion inherently has the potential to trigger vulnerabilities [38], not every path confusion results in an exploitable vulnerability. The exploitability depends on the implementation details of each web service.

For example, each web service has a different URI set, routing rules, and caching rules. Thus, path confusion leading to sensitive resources varies across services. Consequently, identical path confusion results in one service being vulnerable, while another is unaffected due to routing differences.

To formally analyze path confusion considering these scenarios, this section defines *Path Confusion Vulnerability* and *Exploit Payload* by leveraging $\mathcal{S}$.

Definition 5 (Output of web service). *The output of a web service component $\mathcal{W}$, denoted $\mathcal{W}(x)$, for an input $x \in \Sigma^*$ is defined as:*

$$\mathcal{W}(x) = x_m \quad \textit{where} \quad \begin{cases} x_0 = x \\ x_j = P_j(x_{j-1}) \quad \textit{for } 1 \leq j \leq m \end{cases} \tag{5}$$

Given $\mathcal{W}$.Transformations $= \{\langle t_1, c_1\rangle, \ldots, \langle t_m, c_m\rangle\} \subseteq \mathcal{P}$, each P_j corresponds to the transformation pair $\langle t_j, c_j\rangle$. The initial input $x \in \Sigma^*$ serves as x_0, where each P_j transforms the previous output x_{j-1} to produce x_j. This formalizes the execution of $\mathcal{W}.Transformations$ to an input URI x, enabling modeling of the component's output for URI.

Definition 6 (Path confusion vulnerability). *A $\mathcal{V}$, which represents the existence of a path confusion vulnerability, is defined as follows:*

$$\mathcal{V} \leftrightarrow \exists s_0 \in \Sigma^* \left(\mathbb{C}_t(s_t) \wedge C_e(o_t, s_n)\right) \tag{6}$$

Path confusion vulnerabilities arise from interactions among various web service components. The exploit payloads must meet intermediate servers' caching rules in web cache attack, and authentication skip conditions in authentication bypassing. Also, both must reach the attacker's intended URI to the target server.

Thus, a path confusion vulnerability exists if and only if an initial input, processed by the web service components, is transformed to results satisfying attacker-specified conditions. In short, this vulnerability requires two kinds of conditions. First, intermediate results must satisfy the attacker's objectives. Second, the final output must match the attacker's intended target.

Equ. 6 expresses the predicate logic for $\mathcal{V}$, where an input string $s_0 \in \Sigma^*$ undergoes transformation through components $\mathcal{W}_1, \ldots, \mathcal{W}_n \in \mathcal{S}$, producing results $s_1, \ldots, s_n$ such that $s_i = \mathcal{W}_i(s_{i-1})$ for $i \in [1, n]$. The predicate $\mathbb{C}_t(s_t)$, an attacker-specified condition, requires an intermediate result s_t at a step $t \in [1, n]$ to meet the attacker's requirement for the component $\mathcal{W}_t$. Meanwhile, $C_e(o_t, s_n)$ ensures that the final output s_n matches the attacker-specified target o_t. An input s_0 satisfying both conditions is an *Exploit Payload*, triggering the desired intermediate state and achieving the targeted final output.

The formal predicate logic presented above not only precisely defines the existence of path confusion vulnerability but also provides a provable foundation for systematically identifying, constructing, and validating the exploit payloads.

4 PathFault

4.1 Overview

PathFault is a tool designed for defender-oriented security researchers, including penetration testers, service developers, and infrastructure architects, to automatically detect path confusion vulnerabilities in web services. As shown in Fig 1,

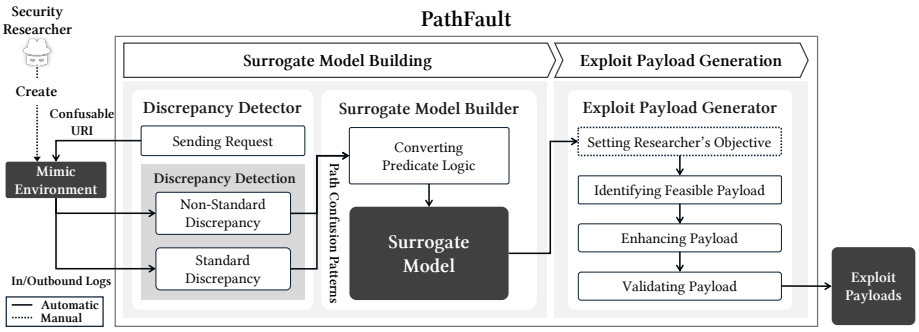

Fig. 1. Overview of PathFault.

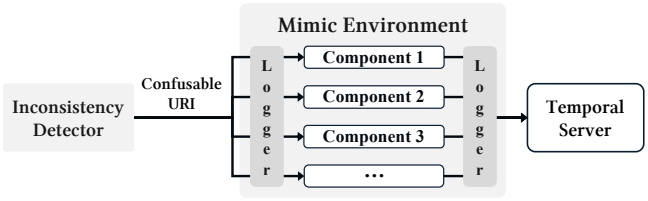

Fig. 2. Overview of Mimic Environment.

it takes inbound and outbound logs of the target web service as input, builds a surrogate model in the *Surrogate Model Building* stage, and generates an exploit payload as output in the *Exploit Payload Generation* stage.

In the Surrogate Model Building stage, the *Discrepancy Detector* analyzes logs to identify path confusion patterns using URI processing discrepancies, including RFC standard transformations. Then, the *Surrogate Model Builder* translates these patterns into predicate logic expressions, constructing a surrogate model for exploit payload generation.

In the *Exploit Payload Generation* stage, the security researchers incorporate domain knowledge, such as specific functionalities or component behaviors, into the surrogate model for reducing the model gap. Based on the model, PathFault employs SMT solving to identify viable $\mathcal{P}$ combinations that meet tester-specified conditions. Next, it enhances payloads using identified $\mathcal{P}$ and percent decoding behaviors. Finally, to ensure reliability, it disables optimizations and validates each payload against the surrogate model.

Meanwhile, to generate inputs for PathFault, inbound and outbound packets of the target web service components must be observable and recordable. Thus, the security researcher manually configures a *Mimic Environment*, as shown in Fig 2, using known components of the target web service. Unlike the real service, the components of the mimic environment operate independently, processing only their own requests without chained forwarding. If the PathFault receives the mimic environment, it automatically sends predefined HTTP requests to the mimic environment to generate the input logs on its own. PathFault provides a built-in module to simplify this configuration process.

In summary, PathFault enables automated detection through two stages:

– Surrogate Model Building: Builds a predicate logic-based surrogate model.

Algorithm 1 Non-standard discrepancy detection

```
1: function AnalyzeConfusablePaths(char)
2:     inpath1 ← "/tmp1/tmp2" + char + "tmp3/tmp4"
3:     inpath2 ← "/tmp1/" + char + "/tmp2"
4:     outpath1 ← getParsingResult(inpath1)
5:     outpath2 ← getParsingResult(inpath2)
6:     if outpath1 = NULL or outpath2 = NULL then
7:         return BadCondition(char)
8:     else if inpath1 ≠ outpath1 then
9:         if outpath1 = "/tmp1/tmp2" then
10:            return FullSubsequentPathRemoval(char)
11:        else if outpath1 = "/tmp1/tmp2/tmp4" then
12:            return PartialCurrentSegmentRemoval(char)
13:        end if
14:    else if inpath2 ≠ outpath2 then
15:        targetStr ← inpath2
16:              .remove("/tmp1")
17:              .remove("tmp2")
18:        replaceStr ← outpath2
19:              .remove("/tmp1")
20:              .remove("tmp2")
21:        return StringSubstitution(targetStr, replaceStr)
22:    end if
23: end function
```

– Exploit Payload Generation: Generates and validates exploit payloads using SMT solving in polynomial time.

The Section 4.2 and Section 4.3 explain the details of each stage.

4.2 Stage 1: Surrogate model building

This stage involves the creation of a mimic environment and two methods for detecting discrepancies.

Step 1: Mimic environment creation. The Mimic Environment imitates web service components, configured to behave independently without inter-component interactions. In short, each component processes only its own requests, unlike the interconnected structure of real services. The security researcher creates a mimic environment that satisfies two conditions:

– Each component has an accessible endpoint for receiving HTTP requests.
– Inbound and outbound packets are obtainable for all components.

Security researchers can easily create this environment with PathFault's utility modules. In this work, mimicking the 7 components in Table 7 requires an average of 3.86 files (102.4 lines of code) per component, and creating a component took 30 minutes to 2 days. However, the reuse of setups significantly reduces subsequent efforts. In fact, PathFault's open-source case study guide requires only executing copy-and-paste commands to create and run the mimic environment.

Step 2: Discrepancy detection. PathFault first analyzes non-standard discrepancies to identify P_f, P_c, or P_s.

Algorithm 1 analyzes non-standard discrepancies by sending confusable URIs such as `"/tmp1/tmp2{char}tmp3/tmp4"`. If the response is `"/tmp1/tmp2"`, it indicates P_f. Alternatively, if the response is `"/tmp1/tmp2/tmp4"`, it indicates P_c. If

neither pattern is detected, an alternative URI `"/tmp1/{char}/tmp2"` identifies P_s when the input and output paths differ. If the component fails to process the confusable URIs containing specific characters (Line 7), leaving no log, it appends $\neg C_c(char, s)$ to $\mathcal{W}.Conditions$.

Next, PathFault analyzes RFC3986-based [8] discrepancies using two algorithms detailed in Appendix B. Algorithm 3 deems P_p if `"/tmp1/../tmp2"` interprets to `"/tmp2"`. Algorithm 4 determines $\mathcal{W}.isDecode$ = True if `"/%21"` is decoded to `"/!"`

Step 3: Building a surrogate model. Finally, PathFault formalizes detected path confusion patterns with predicate logic into $\mathcal{S}$. Upon completion of this step, the $\mathcal{S}$ offering a provable foundation for the generation and verification of exploit payloads is built.

4.3 Stage 2: Exploit payload generation

This stage involves four steps: setting the researcher's objective, identifying feasible payloads via SMT solving, enhancing exploit payloads, and validating them.

Step 1: Setting the researcher's objective. The security researcher sets the objective on $\mathcal{S}$ for targeted path confusion attack scenarios. These objectives specify four components:

- Target precondition: Conditions the URI must meet before transformation.
- Target postcondition: Conditions the URI must satisfy after transformation.
- Essential transformation: Transformations applied to the URI during testing.
- Internal processing representation: Duplication of a component $\mathcal{W}$ for internal processing in $\mathcal{S}$.

For example, to test the feature related to the transformation that performs appending `".gz"` to the URIs ending with `".css"`, the researcher can formalize the path confusion $P_{gz} \subset \mathcal{P}$ as:

$$P_{gz} = \{\langle T_s(\text{“.gz”}, s), \{C_s(\text{“.css”}, s)\}\rangle\} \tag{7}$$

Also, internal processing such as rewriting is modeled by duplicating $\mathcal{W}$ in $\mathcal{S}$. This duplication is efficiently performed by using PathFault's surrogate model-based strategy without reanalysis. Consequently, it reduces the model gap and improves the payload's accuracy. And the security researcher's domain knowledge enhances the $\mathcal{W}$ to $\mathcal{W}_{pen}$, changing $\mathcal{S}$ to $\mathcal{S}_{pen}$ for precise payload generation.

Step 2: Identifying feasible payloads via SMT solving. In this step, PathFault identifies possible combinations of P that satisfy the conditions specified by each $\mathcal{W}_{pen}$ to achieve the security researcher's objective.

The conditional transformations $P \subset \mathcal{P}$ typically force SMT solvers to explore more paths, expanding the search space exponentially. To address this, *Assumption-based Estimation* simplifies SMT solving by presuming only the conditions for a selected combination of P are satisfied, reducing the search space. For instance, if a P is performed when containing `"/../"` in the URI, this approach assumes the condition is satisfied.

Algorithm 2 SMT-based exploit payload generation

```
1: function GeneratePayload(S^act_pen List, o_t)
2:     success S^act_pen List ← [ ]
3:     for each S^act_pen ∈ S^act_pen List do
4:         solver ← NewSMTSolver()
5:         s_0 ← NewString()                          ▷ Original input
6:         s_i ← s_0                                  ▷ Current input
7:         for each W^act_pen ∈ S^act_pen do
8:             preC ← ApplyPreCond(W^act_pen, s_i)
9:             (tIn, tC) ← ApplyTrans(W^act_pen, s_i)
10:            (nIn, nC) ← ApplyNorm(W^act_pen, tIn)
11:            postC ← ApplyPostCond(W^act_pen, nIn)
12:            cond ← Combine(preC, tC, nC, postC)
13:            solver.addCond(cond)
14:            s_i ← nIn
15:        end for
16:        solver.addCond(s_i = o_t)
17:        if solver.check() = sat then
18:            if EssentialSkipped(S^act_pen) then
19:                Alert("Essential transformation skipped")
20:                continue
21:            end if
22:            payload ← solver.model(s_0)
23:            success S^act_pen List.append(S^act_pen, payload)
24:        end if
25:    end for
26:    return success S^act_pen List
27: end function
```

Here, the P combinations, denoted as $\mathcal{S}^{act}_{pen}$, are generated by the *Selective Transformation Combination Algorithm* (STCA). STCA limits transformations (*maxTrans*) applied to each $\mathcal{W}_{pen}$ and the number of components that must perform normalization (*maxNorm*) in $\mathcal{S}_{pen}$, using Cartesian product and permutation operations (see Appendix C for details). Consequently, assumption-based estimation enables exploit payloads to be generated in polynomial time.

Building on this, PathFault generates payloads targeting a specified output o_t with SMT solving. A $\mathcal{S}^{act}_{pen}$, comprising multiple $\mathcal{W}^{act}_{pen}$ with preconditions, transformations, normalization, and postconditions, is used as input.

Algorithm 2 processes $\mathcal{S}^{act}_{pen}List$ comprising selected P combinations by the STCA. It initializes s_0 and s_i for each $\mathcal{S}^{act}_{pen}$ (Line 5-6) and applies preconditions ($\subseteq \mathbb{C}_t$), transformations, normalization, and postconditions ($\subseteq \mathbb{C}_t$) to generate constraints (Line 8-12). Combined constraints verify if s_i satisfies $C_e(o_t, s_n)$ (Line 16). If the SMT solver finds a solution (Line 17) and essential transformations are included (Line 18), the payload s_0 is stored (Line 22); otherwise, it is discarded.

Consequently, grounded in the predicate logic of path confusion vulnerability, payload and $\mathcal{S}^{act}_{pen}$ detailing the P in each $\mathcal{W}^{act}_{pen}$ that satisfies the security researcher's objective are stored in $success\mathcal{S}^{act}_{pen}List$.

Step 3: Enhancing exploit payload. Prior components' P and *Conditions* can block performing the P in the target component. To bypass this, PathFault enhances exploit payloads using two system-wide expansion methods.

First, *Normalization expansion* identifies strings in $\mathcal{S}_{pen}$ that enable normalization in $\mathcal{W}_{pen}$. It analyzes prior components to expand the strings via percent encoding and P_s. Then, it merged with prior $\mathcal{W}_{pen}$'s normalization string candidates to bypass prior $\mathcal{W}_{pen}$'s normalization. Applied to $success\mathcal{S}^{act}_{pen}List$, they create exploit variants. Finally, it creates diverse exploit payloads by applying them to $success\mathcal{S}^{act}_{pen}List$. This Algorithm 7 is detailed in Appendix D.

Next, *Contains-condition expansion* first identifies P requiring $C_c(t, s)$ in $\mathcal{W}^{act}_{pen}$. This analyzes prior components to expand strings satisfying $C_c(t, s)$ via percent encoding and P_s. Finally, these extend $success\mathcal{S}^{act}_{pen}List$ by expanding the target string t of P in $\mathcal{W}_{pen}$. Algorithm 10 is also detailed in Appendix D.

Step 4: Exploit payload validation. This step validates payloads on the surrogate model $\mathcal{S}_{pen}$ to ensure reliability. Due to the expansion mechanism and the assumption-based estimation ignoring $\mathcal{W}_{pen}.badCondition$, some of the payloads may be invalid. To address these, this step represents all P as if-statements and validates all $\mathcal{W}_{pen}.Conditions$ to obtain the payload satisfying all conditions.

Since payload validation is performed on a given string against constraints, its search space is smaller than payload generation, which explores all possible strings. Therefore, this enables obtaining a reliable payload efficiently.

4.4 Implementation

PathFault is implemented in Python 3.12.3 with approximately 3,940 lines of code and generates exploit payloads by converting web service parsing discrepancies into logical constraints using the Z3 solver's Python API[11, 39]. Because PathFault adopts a modular architecture, it supports the community's contributions and future extensions [45]. Additionally, to enhance performance, PathFault provides parallel SMT solving by allowing users to configure the number of worker processes.

5 Case Study: Generating Real-World Exploit Payloads

Path confusion vulnerabilities depend on multi-component interactions, unlike single-component vulnerabilities. Thus, path confusion vulnerability detection requires wide applicability, effectiveness, and scalability for diverse web services and objectives [4].

However, current security methodologies have struggled to analyze these interactions, resulting in limited case studies [24]. Given these challenges, this work aimed to analyze all possible cases. The possible case studies were gathered based on the following criteria: well-documented multi-component path confusion vulnerabilities (Section 5.1, Section 5.2); addressing path confusion vulnerabilities and emphasizing the importance of multi-component analysis (Section 5.4). Regarding Section [22], only Azure provided specific paths for caching rules(reviewed CDNs: Akamai, Azure, Cloudflare, Amazon, Fastly, etc.).

Specifically, Section 5.1 and Section 5.2 demonstrate PathFault's applicability by modeling exploit payloads from reported vulnerabilities. Section 5.3 compares PathFault with the state-of-the-art WCD vulnerability detection tool in terms of detection performance. Section 5.4 reveals that complying with vendor-recommended CDN configuration [22] can lead to path confusion vulnerability, underscoring potential security risks.

All case studies were performed on Ubuntu 20.04.6 LTS with an 11th Gen Intel Core i5-11600K CPU at 3.90GHz and 16GB RAM. PathFault employs a random seed for the SMT solver, with each instance configured to a 10-second timeout, and the max transformation argument is one per $\mathcal{W}$.

5.1 Authentication bypass: CVE-2025-0108

This section explains the trigger mechanism of CVE-2025-0108 [20, 33], which was caused by discrepant URI processing in a multi-tier web service comprising Nginx and Apache HTTP Server. The known exploit payload is shown as follows: `/unauth/%252E%252E/php/ztp_gate.php/PAN_Help/x.css.`

First, Nginx appends a header disabling authentication to requests for URIs prefixed with `"/unauth/"` and sends it to Apache HTTP Server with the URI unchanged.

And then, Apache HTTP Server decodes `"%252E%252E"` to `"%2E%2E"` and applies the rule that appends `".gz"`. After that, it internally redirects the request as follows: `/unauth/%2E%2E/php/ztp_gate.php/PAN_Help/x.css.gz.`

Subsequently, Apache HTTP Server percent-decodes `"%2E%2E"` to `".."`, resulting in: `/unauth/../php/ztp_gate.php/PAN_Help/x.css.gz.`

At the same time, it normalizes this URI to: `/php/ztp_gate.php/PAN_Help/x.css.gz.`

Consequently, the combination of path confusions enables unauthorized access to the protected endpoint `"/php/ztp_gate.php"`, bypassing authentication.

Summarizing these, the exploit leverages a chain of discrepancies in percent decoding, URI rewriting, and normalization across components. Stage 1 and Stage 2 in Section 5.1 detail the modeling process of the vulnerability using PathFault.

Stage 1: surrogate model building First, a security researcher creates mimic environments for a multi-tier web service with Nginx and Apache HTTP Server, using knowledge of web services, enabling PathFault to automatically build the surrogate model.

The researcher requests PathFault to create a surrogate model, generating $\mathcal{S}$ in Fig 3(a). Specifically, the "conditions" field, denoted in (4), specifies constraints of requests identified as *BadCondition* in Algorithm 1. These are expressed as negated predicates, indicating URIs that the component cannot process. The "transformations" field explained in (4), lists P identified by PathFault through discrepancies. The "is_normalize" and "is_decode" fields, denoted in (4), indicate whether the component performs dot-segment normalization and percent decoding. The "target_pre_conditions", "target_post_conditions"

(a) Automatically generated surrogate model by PathFault

(b) Surrogate model with security researcher-defined conditions and internal component duplication.

Fig. 3. Surrogate models for CVE-2025-0108. Changes are gray and additions are red.

and "essential_transformations" fields, denoted in Section 4.3, indicate the researcher's objective.

PathFault exports the surrogate model in Python format to facilitate analyzing and integrating payload generation and validation.

Stage 2: exploit payload generation At this stage, the security researcher reduces the model gap by integrating the domain-specific knowledge into $\mathcal{S}$. Thus, the targeted surrogate model $\mathcal{S}_{pen}$ becomes more similar to a real service.

This vulnerability requires internal redirection of the second Apache HTTP Server, unmodeled in $\mathcal{S}$. To represent the behavior, the researcher duplicates the $\mathcal{W}$ for Apache HTTP Server, leveraging PathFault's Python-based $\mathcal{W}$. This duplication, enabled by PathFault's Python-based representation, efficiently generates internal $\mathcal{W}$ without reanalysis.

Additionally, the researcher integrates the test objective by modifying $\mathcal{W}$ attributes. For Nginx, the precondition and postcondition are added, requiring the URI to prefix `"/unauth/"`, formalized as $C_p(\texttt{"/unauth/"}, s)$. For Apache HTTP Server, P_{gz} that indicates appending `".gz"` to URIs ending with `".css"` is added to the essential transformation. These updates build $\mathcal{S}_{pen}$, shown in Fig 3(b).

The researcher then asks PathFault to generate a payload that produces the final output as: `/php/ztp_gate.php/PAN_Help/x.css.`

Finally, PathFault sequentially integrates $\mathcal{W}$.Conditions and executes each component $\mathcal{W}_{pen}.Transformations \in \mathcal{S}_{pen}$ to each generated payload for validation. As outlined in prior sections, this is achieved through if-statements defining conditional transformations, resulting in the validated exploit payload.

Case study result PathFault identified 168 payloads with a 70% inclusion rate (10 runs) for known payloads using reasonable resources. It includes the payloads

Table 3. Resource-Related Performance for CVE-2025-0108

Average Time(s)			Memory Peak(MB)	
Generation	**Validation**	**Total**	**Generation**	**Validation**
84.55	38.64	123.19	434.18	110.40

that combine semantically separated segments into one using percent encoding `"%252F"`, such as: `/unauth/.%252E%252Fphp/ztp_gate.php/PAN_Help/x.css.`

The result further shows that $P_s(\text{“///”}, \text{“/”}, s)$ can be utilized for enhancing payloads, such as: `/unauth///%252E./php/ztp_gate.php/PAN_Help/x.css.`

Table 3 summarizes the PathFault's performance of modeling the exploit payload used in CVE-2025-0108. It shows that PathFault's ability to accurately model disclosed vulnerabilities within reasonable time and resource limits.

5.2 Web cache deception: ChatGPT account takeover

This section shows that PathFault can model the WCD vulnerability included in ChatGPT's chat sharing service despite the limitation in precise knowledge of the target [30].

This vulnerability, which exposes user authentication information to unauthorized external users, stems from an improper caching policy. The public proof-of-concept demonstrates how the vulnerability is triggered in scenarios where the attacker lacks information about the target web service's structure. The only required information is that caching occurs for `"/share/*"` and the final URI is `"/api/auth/session"`.

Statistical approach for surrogate model building Evaluating all possible combinations of $\mathcal{W}$ could generate exploit payloads without knowledge of the web service's structure, but doing so in polynomial time is challenging. Instead, in the normal distribution of attributes across all existing $\mathcal{W}$, selecting a $\mathcal{W}$ close to the mean value increases the relative likelihood that it is included in the target service compared to other $\mathcal{W}$. Based on this insight, building surrogate models using widely used open-source web servers increases the likelihood of approximating the target service's structure. Therefore, this study selects widely used components (Nginx, Apache HTTP Server, Apache Tomcat, and Apache Traffic Server) as the foundation for representative surrogate models.

Case study result Table 4 summarizes PathFault's results in generating exploit payloads across combinations of four open-source web servers. Note that a success rate of 1.0 indicates that the SMT solving timeout is sufficient for the string search space.

The known exploit payload is generated when the final server applies percent decoding and normalization, while the prior component skips percent decoding. Meanwhile, other exploit payloads are generated when both components perform percent decoding and normalization, as observed in request forwarding between `apachehttpserver` and `apachetomcat`, even in the absence of known payloads.

Table 4. Summary of performance of PathFault for ChatGPT account takeover

Θ: apachehttpserver ϒ: apachetomcat Φ: apachetrafficserver Ω: nginx

Web Service	Success Rate	# of Payloads	Known Payload	Average Time(s)			Memory (MB)	
				Generation	Validation	Total	Generation	Validation
Θ → ϒ	1.0	525	✗	80.22	44.69	124.91	598.29	110.36
Θ → Ω	0.0	0	✗	0.25	0.00	0.25	110.27	109.88
ϒ → Θ	1.0	479	✗	91.70	50.28	141.98	634.93	110.43
ϒ → Φ	0.0	0	✗	7.69	0.00	7.69	126.59	109.90
Φ → Θ	1.0	270	✓	20.42	39.98	60.40	176.85	110.27
Φ → ϒ	1.0	270	✓	31.12	43.17	74.30	381.83	110.38
Ω → Θ	1.0	135	✓	10.23	24.92	35.15	174.78	110.30
Ω → ϒ	1.0	135	✓	11.87	25.36	37.23	292.48	110.35

PathFault demonstrates that using a non-normalizing parser, such as Nginx or Apache Traffic Server, as the final component mitigates these vulnerabilities. That is, despite limited information, PathFault excels in providing insights for analyzing $\mathcal{W}$'s behaviors and identifying conditions to eliminate exploits.

Thus, PathFault can reproduce known exploits and detect unknown vulnerabilities as well as identify their causes and propose mitigation, demonstrating its practical utility for vulnerability remediation. Full results are in Appendix E.

5.3 Enhancing heuristic payloads

PathFault's goal is that defenders identify and mitigate vulnerabilities with a surrogate model and patterns before attackers, reducing risks [26]. Unlike black-box WCDE where attackers and defenders have equal footing [24], PathFault is a whitebox tool leveraging defenders' accessibility about service details as a strength for the goal. However, no whitebox research identifies such vulnerabilities from the defender's perspective. Despite this challenge, this work evaluates PathFault with a state-of-the-art WCDE, an attacker-focused tool, addressing such vulnerabilities about the accuracy of PathFault-generated payloads by comparing them with heuristic payloads from prior work on WCD vulnerabilities.

Overview The number of valid exploit payloads correlates with the likelihood that an adversary could trigger the vulnerability [42, 49]. This is because the probability of exploitation increases proportionally with the number of effective payloads across the entire string domain. In this regard, this evaluation shows that PathFault outperforms state-of-the-art tools in detecting and validating WCD vulnerabilities in modern web services by comparing results per number of components (depth).

Also, this work doesn't conduct a large-scale evaluation used in WCDE. This is because, despite the importance of identifying and mitigating vulnerability causes [24], current research focuses on attacker-driven detection. Therefore, this work focuses on vulnerability modeling performance, which supports defenders in identifying causes and mitigations through the results of diverse service-specific exploit payloads. Specifically, this study evaluates payload diversity, detected vulnerable combinations, successful exploit counts, and time per depth, comparing outcomes from prior work [24], which assumed security researcher objectives (first server processes URI ending with `"not a file.css"`,

Table 5. Comparison by Depth for WCDE and PathFault

Depth	WCDE [24]			PathFault			
	Exploit Payloads	Vulnerable Pairs	TP	Exploit Payloads	Vulnerable Pairs on WCDE	TP	Average Time(s)
2	2	3/12	4	**79**	**3/3**	**156**	**25.32**
3	3	12/24	21	**586**	**12/12**	**3161**	**104.05**
4	3	18/24	35	**1982**	**18/18**	**11544**	**298.43**

last server processes URI as `"/profile"`), with those automatically generated by PathFault under identical objectives.

It tests ordered combinations of two to four web server components, using four open-source web servers from Section 5.2. The state-of-the-art tool, WCDE, uses a fixed set of 12 payloads across all combinations, while PathFault uses generated payloads across combinations by depth.

Case study result Table 5 outlines the results of WCDE and PathFault per depth. Specifically, WCDE supports two valid payload types at depth-2 and three at depths 3 and 4. It detects vulnerabilities in 3/12 component combinations at depth-2, yielding 4 true positives, and in 12/24 and 18/24 combinations at depths 3 and 4, resulting in 21 and 35 true positives, respectively. In contrast, PathFault generates diverse exploit payloads with higher true positives than WCDE, while detecting all vulnerable combinations detected by WCDE. Also, the greater the depth, PathFault can find more exploit payloads, proving that risk mitigation requires considering components' complex interactions in vulnerability detection.

Time-wise, WCDE uses 12 predefined payloads, taking negligible time. PathFault's generation and validation times average per combination are 25.32s, 104.05s, and 298.43s at depth 2, 3, and 4. WCDE has a time advantage in verifying vulnerability existence, but its non-service-specific payloads result in low coverage, evident from payload diversity. Also, this time includes the vulnerability, its cause, and mitigation identification so that PathFault can support vulnerability patch plans quickly than WCDE.

Nevertheless, the detection methodology of WCDE remains effective. Therefore, integrating PathFault's exploit payload generation methodology, which integrates researchers' domain knowledge and service-specific characteristics, with WCDE's detection methodology can significantly enhance vulnerability detection performance across diverse web services [3, 43, 50].

5.4 Arbitrary web cache deception in the CDN configuration guide

This section demonstrates that WCD vulnerabilities can arise from caching rules recommended by commercial CDN providers, using PathFault. The guideline [22] advocates caching with URI prefixed with `"/images/"` and suffixed with `".jpg"` as best practices. These vendor-endorsed configurations overlook inherent security risks, leading users to adopt them in web services without addressing potential threats. This experimental validation aims to enhance security awareness regarding such risks. Specifically, this study uses ordered combinations of three web server components, using four open-source web servers from Section 5.2.

Table 6. Arbitrary Web Cache Deception in the CDN Configuration Guide

Risky Web Component Combinations
(a) apachehttpserver → apachetomcat → apachetrafficserver
(b) apachehttpserver → apachetrafficserver → apachetomcat
(c) apachetomcat → apachehttpserver → apachetrafficserver
(d) nginx → apachehttpserver → apachetrafficserver
(e) nginx → apachetrafficserver → apachehttpserver
(f) nginx → apachehttpserver → apachetomcat
(g) nginx → apachetrafficserver → apachetomcat
(h) nginx → apachetomcat → apachetrafficserver
Some of Exploit Payloads
(1) /images/%2E%2E/%2E/secret;.jpg
(2) /images/./%2E%2E/secret%3B.jpg
(3) /images/%2E%2E//secret%253B.jpg
(4) /images/../secret%23.jpg
(5) /images/.%252E%252Fsecret%25%32%33%25%32%33.jpg

Case study result Table 6 shows risky combinations of web components and generated exploit payloads by PathFault. The caching mechanism can be exploited by P_p and P_f. For example, by triggering $P_p(s)$ and $P_f(\text{“\#”}, s)$ at the final server, an attacker can reach sensitive endpoints like “/secret”, while triggering the caching mechanism.

Specifically, PathFault generates payloads using either `";"` or encoded `"%3B"` to strip the `".jpg"` suffix at Apache Tomcat (1,2). It also used encoded `"#"` as `"%23"` or double-encoding to bypass the server's *badCondition* about `"#"` (3,4,5). It further expands $P_s(\text{“/./”}, \text{“/”}, s)$ with encoded dots (`"%2E"`), generating diverse exploit payloads (1,2).

These demonstrate the feasibility of multiple valid exploit payloads and highlight the security risks inherent in the CDN vendor's caching guidelines. Although tested with four web service components at depth-3, other configurations may also be vulnerable, potentially exposing broader security risks. Specifically, although the default caching guidelines provided by CDN vendors can introduce security risks, such threats aren't mentioned in the official documentation. This suggests the issue within the security community, where system-level vulnerabilities are overlooked in deployment recommendations. Given this oversight, the community must prioritize collaborative research in system-level risk analysis and mitigation to strengthen deployment recommendations and enhance security practices.

6 Limitation

Resource-dependent payload generation. Exploit payload generation relies on the SMT solver's computational resources, causing variations in the number and diversity of payloads [6]. Because solving logic constraints for multiple transformations increases complexity exponentially [10], constrained resources may miss some payloads in polynomial time. Nevertheless, the prior case studies demonstrate that PathFault is useful in vulnerability detection through reproducing known exploit payloads and generating unreported payloads with reasonable resources.

Surrogate model gap. PathFault may fail to capture minor configurations such as the order of $\mathcal{W}.Transformations$, impeding completeness. Also, it excludes character-level percent encoding from P_s because the consideration of it increases the complexity of SMT solving. This may lead to widening the model gap.

Despite the model gap, prior case studies show that striking a balance between model completeness and generating payloads in polynomial time remains critical for practicality, as models successfully reproduce known exploit payloads and yield valid results in diverse environments. Therefore, future work on capturing fine-grained logic and path processing while maintaining the balance represents a promising direction.

7 Related Work

HTTP message parser discrepancy detection. Prior studies [1, 2, 7, 18, 19, 25, 35, 38, 41, 47, 48] identify discrepancies in HTTP parser implementations, highlighting servers' discrepant interpretations of identical requests. While these efforts introduced techniques for efficiently detecting parsing discrepancies, they do not explore how such discrepancies can be leveraged to construct exploits in a systematic manner. In contrast, our work formalizes path confusion patterns, introduces a methodology for systematically generating exploit payloads with discrepancies, and bridges the gap between discrepancy detection and exploitation.

System-level analysis via webserver fingerprinting. Prior work on web server fingerprinting [28, 37, 46] leverages system-wide discrepancies to infer web service structures, termed a passive attack. These methods, however, neither verify exploitability nor generate attack payloads. PathFault, in contrast, performs an active attack that produces exploit payloads utilizing discrepancies from a security researcher's perspective. Integrating fingerprinting with this approach enables adversarial system-level analysis, a promising direction for future work.

8 Conclusion

This research proposes a system-level framework for detecting path confusion vulnerabilities with high precision and coverage, leveraging predicate logic and SMT solving. It contributes to improving detection scope and accuracy via component interaction modeling by surrogate model capturing web service behavior with minimal real-world reliance, an exploit payload generator producing formally verified payloads to achieve efficient penetration testing, and a system-wide analysis methodology. These enable practical vulnerability detection in complex web services, thus paving the way for automated mitigation of discrepancy-based threats.

Acknowledgments. This research was supported by a National Research Foundation of Korea (NRF) grant, funded by the Korean government (MSIT) (RS-2023-NR077166, RS-2023-00227165).

References

1. Ajmani, D.K., Koishybayev, I., Kapravelos, A.: you are a liar://a unified framework for cross-testing url parsers. In: 2022 IEEE Security and Privacy Workshops (SPW). pp. 51–58. IEEE (2022)
2. Akhavani, S.A., Jabiyev, B., Kallus, B., Topcuoglu, C., Bratus, S., Kirda, E.: Waffled: Exploiting parsing discrepancies to bypass web application firewalls. arXiv preprint arXiv:2503.10846 (2025)
3. Alhamed, M., Rahman, M.H.: A systematic literature review on penetration testing in networks: future research directions. Applied Sciences **13**(12), 6986 (2023)
4. Antunes, N., Vieira, M.: Soa-scanner: an integrated tool to detect vulnerabilities in service-based infrastructures. In: 2013 IEEE International Conference on Services Computing. pp. 280–287. IEEE (2013)
5. Azad, B.A., Laperdrix, P., Nikiforakis, N.: Less is more: Quantifying the security benefits of debloating web applications. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1697–1714 (2019)
6. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
7. Benhabbour, I., Attia, M., Dacier, M.: Http conformance vs. middleboxes: Identifying where the rules actually break down. In: International Conference on Passive and Active Network Measurement. pp. 155–181. Springer (2025)
8. Berners-Lee, T., Fielding, R., Masinter, L.: Rfc 3986: Uniform resource identifier (uri): Generic syntax (2005)
9. Bi, Y., Yang, M., Fang, Y., Mi, X., Guo, S., Tang, S., Duan, H.: Dissecting Open Edge Computing Platforms: Ecosystem, Usage, and Security Risks . In: 2024 Annual Computer Security Applications Conference (ACSAC). pp. 1139–1155. IEEE Computer Society, Los Alamitos, CA, USA (Dec 2024). https://doi.org/10.1109/ACSAC63791.2024.00092
10. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2), 1–38 (2008)
11. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
12. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
13. Fielding, R.T., Reschke, J.: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Jun 2014). https://doi.org/10.17487/RFC7230
14. Gil, Omer: Web cache deception attack. presented at Black Hat USA 2017, Web AppSec Track, Las Vegas, NV, USA, Jul. 26, 2017. [Online]. Available: `https://www.blackhat.com/us-17/briefings.html/#web-cache-deception-attack` (2017), accessed: May. 19, 2025
15. Gil, Omer: Web cache deception attack. `https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html` (2017), accessed: May. 19, 2025
16. Hasegawa, A.A., Watanabe, T., Shioji, E., Akiyama, M.: I know what you did last login: inconsistent messages tell existence of a target's account to insiders. In: Proceedings of the 35th Annual Computer Security Applications Conference(ACSAC). pp. 732–746. ACSAC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3359789.3359832

17. Intelligence, M.: Content delivery network (cdn) market growth, trends, covid-19 impact, and forecasts (2023 - 2028). https://www.mordorintelligence.com/industry-reports/content-delivery-market (2022), accessed: May 19, 2025
18. Jabiyev, B., Gavazzi, A., Onarlioglu, K., Kirda, E.: Gudifu: Guided differential fuzzing for http request parsing discrepancies. In: Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 235–247 (2024)
19. Jabiyev, B., Sprecher, S., Onarlioglu, K., Kirda, E.: T-reqs: Http request smuggling with differential fuzzing. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1805–1820 (2021)
20. Kues, A.: Nginx/Apache path confusion to auth bypass in PAN-OS (CVE-2025-0108). https://slcyber.io/blog/nginx-apache-path-confusion-to-auth-bypass-in-pan-os (2025), accessed: May. 19, 2025
21. Liang, Y., Chen, J., Guo, R., Shen, K., Jiang, H., Hou, M., Yu, Y., Duan, H.: Internet's invisible enemy: Detecting and measuring web cache poisoning in the wild. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 452–466 (2024)
22. Microsoft: Control Azure Content Delivery Network caching behavior with caching rules. https://learn.microsoft.com/en-us/previous-versions/azure/cdn/cdn-caching-rules (2017), accessed: May 20, 2025
23. Mirheidari, S.A., Arshad, S., Onarlioglu, K., Crispo, B., Kirda, E., Robertson, W.: Cached and confused: Web cache deception in the wild. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 665–682 (2020)
24. Mirheidari, S.A., Golinelli, M., Onarlioglu, K., Kirda, E., Crispo, B.: Web cache deception escalates! In: 31st USENIX Security Symposium (USENIX Security 22). pp. 179–196 (2022)
25. Mu, K., Chen, J., Zhuge, J., Li, Q., Duan, H., Feamster, N.: The Silent Danger in HTTP: Identifying HTTP Desync Vulnerabilities with Gray-box Testing. In: 34th USENIX Security Symposium (USENIX Security 25) (2025), to appear
26. Nayak, K., Marino, D., Efstathopoulos, P., Dumitraş, T.: Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In: International Workshop on Recent Advances in Intrusion Detection. pp. 426–446. Springer (2014)
27. Nguyen, H.V., Iacono, L.L., Federrath, H.: Your cache has fallen: Cache-poisoned denial-of-service attack. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1915–1936 (2019)
28. Nmap Project: Nmap: The Network Mapper - Free Security Scanner. https://nmap.org/ (2025), accessed: May 21, 2025
29. Nokline: Caching the un-cacheables - Abusing URL parser confusions (Web cache poisoning technique). https://nokline.github.io/bugbounty/2022/09/02/Glassdoor-Cache-Poisoning.html (2022), accessed: May. 19, 2025
30. Nokline: ChatGPT account takeover - Wildcard web cache deception. https://nokline.github.io/bugbounty/2024/02/04/ChatGPT-ATO.html (2024), accessed: May. 19, 2025
31. OWASP Foundation: Web Security Testing Guide v4.2. https://owasp.org/www-project-web-security-testing-guide/ (2020), accessed: May 19, 2025
32. OWASP Foundation: Cache Poisoning Attack. https://owasp.org/www-community/attacks/Cache_Poisoning (2022), accessed: May 19, 2025

33. Palo Alto Networks, Inc.: CVE-2025-0108: Authentication bypass vulnerability in PAN-OS management web interface. https://nvd.nist.gov/vuln/detail/CVE-2025-0108 (2025), accessed: May. 19, 2025
34. Penetration Test Guidance Special Interest Group PCI Security Standards Council: Information Supplement: Penetration Testing Guidance, Version 1.1. https://docs-prv.pcisecuritystandards.org/Guidance%20Document/Penetration%20Testing/Penetration-Testing-Guidance-v1_1.pdf (2017), accessed: May. 19, 2025
35. Pisu, L., Loi, F., Maiorca, D., Giacinto, G.: Http/3 will not save you from request smuggling: A methodology to detect http/3 header (mis) validations. In: 2024 22nd International Symposium on Network Computing and Applications (NCA). pp. 97–104. IEEE (2024)
36. Pletinckx, S., Kruegel, C., Vigna, G.: A large-scale measurement study of the proxy protocol and its security implications. In: Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS) (2025)
37. Qin, Y., Zhu, Y., Zhang, L., Li, B., Ding, Y., Liu, Q.: Layyerx: Unveiling the hidden layers of doh server via differential fingerprinting. In: 2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 354–361. IEEE (2024)
38. Rautenstrauch, J., Stock, B.: Who's breaking the rules? studying conformance to the http specifications and its security impact. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. pp. 843–855 (2024)
39. Research, M.: Z3: A high-performance theorem prover. https://github.com/Z3Prover/z3 (2025), accessed: May 20, 2025
40. Scarfone, K., Souppaya, M., Cody, A., Orebaugh, A.: Technical guide to information security testing and assessment. NIST Special Publication **800**(115), 2–25 (2008)
41. Shen, K., Lu, J., Yang, Y., Chen, J., Zhang, M., Duan, H., Zhang, J., Zheng, X.: Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–13. IEEE (2022)
42. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE transactions on software engineering **37**(6), 772–787 (2010)
43. Singh, N., Meherhomji, V., Chandavarkar, B.: Automated versus manual approach of web application penetration testing. In: 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT). pp. 1–6. IEEE (2020)
44. of Standards, N.I., (NIST), T.: Security and privacy controls for information systems and organizations. Draft NIST Special Publication 800–53 Revision 5 (2017)
45. Thaiya, M.S., Julia, K., Mbugua, S.: On software modular architecture: Concepts, metrics and trend. International Journal of Computer and Organization Trends **12**(1), 3–10 (2022)
46. Topcuoglu, C., Onarlioglu, K., Jabiyev, B., Kirda, E.: Untangle: Multi-layer web server fingerprinting. In: Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS) (2024)
47. Wang, Q., Chen, J., Jiang, Z., Guo, R., Liu, X., Zhang, C., Duan, H.: 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). pp. 185–202. IEEE (2024)

48. Zheng, L., Li, X., Wang, C., Guo, R., Duan, H., Chen, J., Zhang, C., Shen, K.: Reqsminer: Automated discovery of cdn forwarding request inconsistencies and dos attacks with grammar-based fuzzing. In: Proceedings of the 2024 Network and Distributed System Security Symposium (NDSS) (2024)
49. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: 2010 Third international conference on software testing, verification and validation. pp. 421–428. IEEE (2010)
50. Zurowski, S., Lord, G., Baggili, I.: A quantitative analysis of offensive cyber operation (oco) automation tools. In: Proceedings of the 17th international conference on availability, reliability and security. pp. 1–11 (2022)

Appendix

A Path Confusion Analysis of Web Servers

PathFault evaluates discrepancies within web service components, detecting path confusion patterns and percent decoding behavior across seven web servers. Table 7 summarizes the path confusion patterns P_f, P_p, P_c, P_s, including path normalization and percent decoding behaviors. Note that, P_p denotes the normalization of directory sequences `"/../"`.

Table 7. Path Confusion Pattern of Web Service Components

Web Server	Full Subsequent Path Removal	Partial Current Segment Removal	String Substitution	Single Previous Segment Removal	Percent Decoding
Apache HTTP Server	–	–	$P_s(\text{“/./”}, \text{“/”}, s)$ $P_s(\text{“///”}, \text{“/”}, s)$ $P_s(\text{“//”}, \text{“/”}, s)$	$P_p(s)$	True
Apache Tomcat	–	$P_c(\text{“;”}, s)$	$P_s(\text{“/?/”}, \text{“?/”}, s)$ $P_s(\text{“/./”}, \text{“/”}, s)$ $P_s(\text{“///”}, \text{“/”}, s)$ $P_s(\text{“//”}, \text{“/”}, s)$	$P_p(s)$	True
Apache Traffic Server	$P_f(\text{“\#”}, s)$	–	–	–	False
Caddy	–	–	–	–	False
HAProxy	–	–	–	–	False
Nginx	–	–	–	–	False
Traefik	–	–	$P_s(\text{“/./”}, \text{“/”}, s)$ $P_s(\text{“///”}, \text{“/”}, s)$ $P_s(\text{“//”}, \text{“/”}, s)$	$P_p(s)$	False

B Standard Discrepancy Detection Algorithm Details

This section presents the algorithms for detecting discrepancies in URI parsing as defined in RFC3986 [8], focusing on dot-segment normalization and percent decoding.

B.1 Detection algorithm for single previous segment removal

Algorithm 3 identifies whether it performs dot-segment normalization by validating that a path such as `"/tmp1/../tmp2"` resolves to `"/tmp2"`.

B.2 Detection algorithm for percent decoding

Algorithm 4 identifies whether it performs percent decoding by validating that a path such as `"/%21"` decodes to `"/!"`.

Algorithm 3 Detection algorithm for single previous segment removal

```
function DecideP_p
    inpath ← "/tmp1/../tmp2"
    outpath ← getParsingResult(inpath)
    if "/tmp2" ⊂ outpath and "/tmp1" ⊄ outpath then
        return True
    else
        return False
    end if
end function
```

Algorithm 4 Detection algorithm for percent decoding

```
function DecidePercentDecoding
    inpath ← "/%21"
    outpath ← getParsingResult(inpath)
    if outpath = "/!" then
        return True
    else
        return False
    end if
end function
```

C Selective Transformation Combination Algorithm Details

The Selective Transformation Combination Algorithm (STCA) generates combinations of P and dot-normalization using $\mathcal{S}_{pen}$ and $\mathcal{W}_{pen}$ to make candidates for SMT Solving.

Algorithm 5 Selective transformation combination algorithm

function SMTCandidates($\mathcal{S}_{pen}$, $maxTrans$, $maxNorm$)
  $combList \leftarrow [\mathcal{T}_{\mathcal{W}} \mid \mathcal{W}_{pen} \in \mathcal{S}_{pen}]$ *where*
  $\mathcal{T}_{\mathcal{W}}$ is subsets of $\mathcal{W}.Transformations$ ($\leq maxTrans$)
  $transComb \leftarrow \prod_{i=1}^{n} \mathcal{T}_{\mathcal{W}_i}$
  $normPerm \leftarrow \bigcup_{k=0}^{\text{maxNorm}} \text{Perm}_k(\mathcal{S}_{pen})$
  $\mathcal{S}_{pen}^{act}List \leftarrow transComb \times normPerm$
  return $\mathcal{S}_{pen}^{act}List$
end function

Algorithm 5 reduces the SMT solver's search space by restricting transformations (up to $maxTrans$) per $\mathcal{W}_{pen}$ and normalization components (up to $maxNorm$) in $\mathcal{S}_{pen}$. For each $\mathcal{W}_{pen}$, the algorithm computes a Cartesian product on subsets $\mathcal{T}_{\mathcal{W}} \subseteq \mathcal{W}_{pen}.Transformations$ with up to $maxTrans$ transformations, optimizing the constraint structure for SMT solving and generating transformation combinations stored as *combList*. That is, all possible transformation combinations for each $\mathcal{W}_{pen}$ within $\mathcal{S}_{pen}$ are stored in *combList*. Subse-

quently, it calculates the potential permutations of the $\mathcal{W}_{pen}$ to which component performs normalization and ultimately amalgamates them to generate the $\mathcal{S}_{pen}^{act}$ combination that will serve as input to the SMT Solver.

D Enhancing Exploit Payload Algorithm Details

Exploit Payload Enhancement (EPE), as shown in Algorithm 6, enhances the exploit payload to improve detection range for path confusion vulnerabilities at the system level.

Algorithm 6 Exploit payload enhancement

```
1: function EnhancePayload(successS_pen^act List)
2:     enhancedPayloads ← []
3:     for each (S_pen^act, payload) ∈ successS_pen^act List do
4:         normPayloads ← NormExpand(S_pen^act, payload)
5:         finalPayloads ← CondExpand(S_pen^act, normPayloads)
6:         enhancedPayloads.extend(finalPayloads)
7:     end for
8:     return enhancedPayloads
9: end function
```

By applying dot-segment normalization and percent decoding to the normalizable strings of $\mathcal{W}_{pen}$, candidate normalization strings are generated based on whether these operations are performed. Then, Strings enabling normalization from prior $\mathcal{W}_{pen}$ are removed from the candidate set to ensure that normalization is performed in the target $\mathcal{W}_{pen}$. Additionally, string combinations satisfying C_c under transformation conditions are optimized through percent-encoding, thereby enhancing the exploitable payload.

The next section explains the two primary expansion methods in detail: *Normalization Expansion* and *Contains-Condition Expansion*.

D.1 Normalization expansion algorithm

Web service components apply normalization techniques, such as dot-segment normalization and percent-decoding, to produce diverse path representations. However, normalization outcomes depend on inter-component transformations, and overlooking these interactions may lead to inaccurate vulnerability detection. To address this, it is essential to identify all strings that enable triggering normalization in the target component ($\mathcal{W}_{pen}$).

Algorithm 7 first identifies $\mathcal{W}_{pen}$ requiring normalization within $\mathcal{S}_{pen}$. It then generates candidate strings by applying percent-encoding to `"/../"` and incorporating string substitutions where applicable (Lines 6-8). These candidates are

Algorithm 7 Normalization expansion

```
1: function NormExpand(S^act_pen, payload)
2:     normRes ← []
3:     for W^act_pen ∈ S^act_pen do
4:         if W^act_pen.normalize then
5:             W_pen ← GetW(W^act_pen)
6:             currNormSet ← NormEncReplExpand(W_pen)
7:             for prevW ∈ GetPrevWs(W_pen) do
8:                 prevNormSet ← NormEncReplExpand(prevW)
9:                 currNormSet ← MergeNorm(currNormSet, prevNormSet, prevW)
10:            end for
11:            normRes ← ApplyExpansion(payload, currNormSet)
12:        end if
13:    end for
14:    return normRes
15: end function
```

merged with strings from prior $\mathcal{W}_{pen}$ components to capture cumulative normalization effects across the chain (Line 9). The resulting expanded strings are integrated into the payload to generate varied exploit variants (Line 11). This process ensures that the target $\mathcal{W}_{pen}$ performs normalization, enhancing the coverage of vulnerability detection. The next section details NormEncReplExpand and MergeNorm.

Percent-encoding and replacement expansion algorithm. Dot-segment normalization is triggered by `"/../"`, which can be percent-encoded as `"%2F..%2F"`, depending on $\mathcal{W}$'s percent-encoding support. Moreover, certain $\mathcal{W}$ components apply string substitutions via P_s, limiting the effectiveness of percent-encoding alone in generating all normalization inputs. Thus, combining percent-encoding with P_s-based string expansion is essential for generating diverse payloads that trigger normalization behaviors.

Algorithm 8 Percent encoding and replacement expansion

```
1: function NormEncReplExpand(W_pen)
2:     encodeSet ← NormEncodeExpand(W_pen)
3:     replaceSet ← ∅
4:     for each normStr ∈ encodeSet do
5:         for each trans ∈ W_pen.transformations do
6:             if trans ∈ P_s ∧ normStr.Contains(trans.r) then
7:                 replaceSet ← replaceSet ∪ ReplaceExpand(trans)
8:             end if
9:         end for
10:    end for
11:    return encodeSet ∪ replaceSet
12: end function
```

Algorithm 8 generates strings triggering normalization for a web service component $\mathcal{W}_{pen}$. It first applies NormEncodeExpand to produce percent-encoded strings, such as `"/../"` to `"%2F..%2F"`, storing them in an encode set. Next, for each encoded string, string substitutions from P_s are applied, replacing any

matching substring r with a target string t via ReplaceExpand. The resulting strings are combined with the encode set to yield an exhaustive set of percent-encoded and P_s-based normalization strings for $\mathcal{W}_{pen}$.

Path normalization merge algorithm. Algorithm 9 integrates the current

Algorithm 9 Path normalization merge

```
1: function MergeNorm(currNormSet, prevNormSet, prevW)
2:     if prevW.isNormalize then
3:         mrgSet ← currNormSet − prevNormSet
4:         if prevW.isDecode then
5:             mrgSet ← mrgSet ∪ EncodePct25(currNormSet)
6:         end if
7:     else if prevW.isDecode then
8:         mrgSet ← currNormSet ∪ EncodePct25(currNormSet)
9:     else
10:        mrgSet ← currNormSet
11:    end if
12:    return mrgSet
13: end function
```

normalization set ($currNormSet$) with the normalization set of a previous web service component ($prevNormSet$) based on the properties of the previous component ($prevW$). Initially, if $prevW$ performs normalization (Line 2), strings normalizable by $prevW$ are filtered out from $currNormSet$ to retain only those strings that the target component normalizes (Line 3). If $prevW$ also performs percent-decoding (Line 4), the function EncodePct25 encodes `"%"` as `"%25"` in $currNormSet$, and the resulting strings are added to the merged set (Line 5). This step ensures evasion of prior decoding effects. Alternatively, if $prevW$ does not perform the normalization but performs the percent-decoding (Line 7), EncodePct25 is applied to $currNormSet$, and the encoded strings are included in the merged set (Line 8). If neither normalization nor decoding is performed by $prevW$ (Line 9), the merged set retains $currNormSet$ without modification (Line 10). The resulting merged set is returned. This process ensures precise normalization candidate generation, enhancing the detection of path confusion vulnerabilities.

D.2 Contains-condition expansion algorithm

To optimize exploit payload generation, the target $\mathcal{W}_{pen}.Transformations$ must be executed while avoiding interference from other $\mathcal{W}_{pen}.Transformations$. However, target $\mathcal{W}_{pen}.Transformations$ may be blocked by a prior $\mathcal{W}_{pen}$'s P or conditions. This study addresses this by leveraging the $C_c(t, s)$ condition.

Algorithm 10 examines each $\mathcal{W}^{act}_{pen}$ in $\mathcal{S}^{act}_{pen}$ to identify P satisfying the condition $C_c(t, s)$, which requires the presence of the target string t. Expansion is performed only for unprocessed target strings to prevent redundant P.

Algorithm 10 Contains-condition expansion

```
1: function CondExpand(S^act_pen, normPayloads)
2:     seen ← ∅, condSet ← ∅
3:     for W^act_pen ∈ S^act_pen do
4:         for tf ∈ W^act_pen.transformations do
5:             for cond ∈ tf.conditions where cond ∈ C_c do
6:                 if cond.t ∉ seen then
7:                     seen ← seen ∪ {cond.t}
8:                     decNum ← GetPreDecodeWNum(W^act_pen)
9:                     W_pen ← GetW(W^act_pen)
10:                    cands ← EncodeCandidates(cond.t, decNum, W_pen)
11:                    condSet ← condSet ∪ cands
12:                end if
13:            end for
14:        end for
15:    end for
16:    condRes ← ApplyExpansion(normPayloads, condSet)
17:    return condRes
18: end function
```

To perform the expansion, Algorithm 10 counts the components performing percent-decoding before the current $\mathcal{W}^{act}_{pen}$ using GetPreDecodeWNum. It then retrieves the associated $\mathcal{W}_{pen}$ via GetW and generates string candidates for the target string t through percent-encoding with EncodeCandidates, as detailed in Algorithm 11. These candidates form the basis for extending the exploit payload, either satisfying or bypassing the $C_c(t, s)$ condition, and are aggregated in *condSet*. The augmented string set is combined with the normalization-based payload set (*normPayloads*) to produce the exploitable payload set (*condRes*) using ApplyExpansion, ensuring the target $\mathcal{W}_{pen}$ applies the intended transformation.

Encoding candidate generation algorithm. Algorithm 11 generates percent-encoded strings from an input string s, tailored to the percent-decoding behavior of a target web service component $\mathcal{W}_{pen}$ and the number of prior decoding counts ($decNum$). It initializes an empty set *encSet* to store encoded strings (Line 2). If $\mathcal{W}_{pen}$ performs percent-decoding, PctEncode applies a single level of percent-encoding to s, adding the results to *encSet* (Lines 3-5). These strings are then encoded $decNum$ times to account for prior components' decoding (Lines 6-10). If $\mathcal{W}_{pen}$ does not perform percent-decoding but prior components do, s is encoded $decNum$ times and added to *encSet* (Lines 11-13). Finally, the original string s is included in *encSet*, and the candidate set is returned (Lines 14-15).

E Full Performance of PathFault by Configuration for Web Cache Deception: ChatGPT Account Takeover

This presents the performance results of PathFault for various configurations in Web Cache Deception experiments, focusing on ChatGPT account takeover scenarios. The detailed metrics are provided in Table 8.

Algorithm 11 Percent encoded candidate generation

```
function EncodeCandidates(s, decNum, W_pen)
    encSet ← ∅
    if W_pen.isDecode then
        candSet ← PctEncode(s, 1)
        encSet ← encSet ∪ candSet
        if decNum > 0 then
            for c ∈ candSet do
                encSet ← encSet ∪ PctEncode(c, decNum)
            end for
        end if
    else if decNum > 0 then
        encSet ← encSet ∪ PctEncode(s, decNum)
    end if
    encSet ← encSet ∪ {s}
    return encSet
end function
```

Table 8. Full result of performance of PathFault for ChatGPT account takeover

Θ: apachehttpserver ϒ: apachetomcat Φ: apachetrafficserver Ω: nginx

Web Service	Success Rate	# of Payloads	Known Payload	Average Time(s)			Memory (MB)	
				Generation	Validation	Total	Generation	Validation
Θ → ϒ	1.0	525	✗	80.22	44.69	124.91	598.29	110.36
Θ → Φ	0.0	0	✗	5.51	0.00	5.51	204.24	109.83
Θ → Ω	0.0	0	✗	0.25	0.00	0.25	110.27	109.88
ϒ → Θ	1.0	479	✗	91.70	50.28	141.98	634.93	110.43
ϒ → Φ	0.0	0	✗	7.69	0.00	7.69	126.59	109.90
ϒ → Ω	0.0	0	✗	0.39	0.00	0.39	110.58	109.83
Φ → Θ	1.0	270	✓	20.42	39.98	60.40	176.85	110.27
Φ → ϒ	1.0	270	✓	31.12	43.17	74.30	381.83	110.38
Φ → Ω	0.0	0	✗	0.11	0.00	0.11	110.14	109.96
Ω → Θ	1.0	135	✓	10.23	24.92	35.15	174.78	110.30
Ω → ϒ	1.0	135	✓	11.87	25.36	37.23	292.48	110.35
Ω → Φ	0.0	0	✗	0.11	0.00	0.11	110.26	109.88