HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations

Kaiwen Shen*, Jianyu Lu‡, Yaru Yang*, Jianjun Chen*✉,
Mingming Zhang*, Haixin Duan*‡, Jia Zhang*†✉, Xiaofeng Zheng*‡,
*Tsinghua University, ‡QI-ANXIN Technology Research Institute
†Beijing National Research Center for Information Science and Technology
{skw17, zmm18, zxf19}@mails.tsinghua.edu.cn, zhangjia@cernet.edu.cn,
{jianjun, duanhx}@tsinghua.edu.cn, {zeddyu.lu, yangyr17}@gmail.com,

***Abstract*—The Internet has become a complex distributed network with numerous middle-boxes, where an end-to-end HTTP request is often processed by multiple intermediate servers before it reaches its destination. However, a general problem in this distributed network is the *semantic gap attack*, which is defined as inconsistent semantic interpretations in the processing chain. While some studies have found individual semantic gap attacks, most of them are based on ad-hoc manual analysis, which is inadequate for fundamentally enhancing the security assurance of a system as complex as the HTTP network.**

In this work, we propose HDiff, a novel semi-automatic detecting framework, systematically exploring semantic gap attacks in HTTP implementations. We designed a documentation analyzer that employs natural language processing techniques to extract rules from specifications, and utilized differential testing to discover semantic gap attacks. We implemented and evaluated it to find three kinds of semantic gap attacks in 10 popular HTTP implementations. In total, HDiff found 14 vulnerabilities and 29 affected server pairs covering all three types of attacks. In particular, HDiff also discovered three new types of attack vectors. We have already duly reported all identified vulnerabilities to the involved HTTP software vendors and obtained 7 new CVEs from well-known HTTP software, including Apache, Tomcat, Weblogic, and Microsoft IIS Server.

***Index Terms*—Web Application Security, Semantic Gap Attack, Differential Testing, Documentation Analysis**

I. Introduction

The past few decades have witnessed the rapid popularity and deployment of middleboxes [6], such as cache servers, proxy servers, web application firewalls, and content delivery networks (CDNs). They are widely deployed on the Internet to improve security and performance. As a result, a typical end-to-end HTTP request is now processed by multiple intermediate servers before it reaches its destination.

However, HTTP communications can be exposed to *Semantic Gap Attacks* [14] if multiple middleboxes exist in the path. The key reason is that when a malicious client sends an HTTP message with ambiguous fields, different implementations in the HTTP processing chain may interpret it differently. Such semantic inconsistencies can lead to severe security consequences, such as cache poisoning, security policy bypass, or denial-of-services attack. In recent years, there has been a lot of work revealing security issues caused by such semantic gap attacks, such as Host of Troubles [15], HTTP Request Smuggling [3], and Cache-Poisoned Denial-of-Service Attack [36], illustrating such issues have become a serious threat to the Internet.

Though have discovered individual semantic gap attacks [3], [15], [36], most previous studies are based on ad-hoc manual analysis or only analyze one type of semantic gap attack. They require significant human effort in analyzing documents and testing implementation, which is not scalable and error-prone. Thus, a more systematic, generalizable, and extendable methodology is needed to explore and discover previously unknown venues for exploitation.

To address this research gap, we analyzed and summarized the root causes of semantic gap attacks based on previous works [15], [20], [35], [36], [42]. We found inconsistencies in different implementations often occur due to two reasons: First, some implementations do not follow RFC requirements, either due to intended relaxation or programming mistakes. Second, RFC defines optional requirements allowing developers to use their discretion. For example, RFC documents use keywords like MAY, SHOULD, or SHOULD NOT to specify rules. However, developers may prefer different preferences in implementing these rules. When different implementations are connected together, semantic gap bugs may be revealed and lead to security attacks.

Therefore, a systematic solution to discover semantic gap attacks relies on an in-depth understanding of HTTP protocol specifications. It is meaningful to explore a method to automatically extract semantic information in RFC documents and guide the generation and mutation of test cases.

Challenges. Systematically discovering semantic gap attacks with RFC documents is non-trivial, and several challenges need to be addressed. First, RFC specifications are written in natural language that might be unstructured and include implicit requirements. It is difficult to extract such informal descriptions from RFC documents and convert them to formal invariants. Besides, some HTTP RFC specifications are lengthy (e.g., RFC 7230 documenting one part of HTTP specification includes 89 pages). Manually extracting all HTTP documents needs significant human efforts and is error-prone. Second, most semantic gap bugs do not display any explicitly erroneous behavior like crashes or memory corruption bugs and thus are hard to detect.

HDiff. In this paper, we propose a novel detecting framework, HDiff, to address the aforementioned problems and systematically find semantic gap attacks in HTTP implementations.

For the first challenge, we design *Documentation Analyzer* which uses natural language processing techniques to extract rules from RFC documents automatically. We focus on extracting two types of rules: 1) Specification Requirements (SR). SRs are informal descriptions to define HTTP semantic actions, like actions that client/proxy/server should follow when sending or receiving a specific request. A key observation is that all SR sentences tend to use strong sentimental words (e.g., MUST, ought to, not allowed) in emphasizing the importance of a constraint, particularly those security-critical constraints. Thus, HDiff utilizes a sentiment-based SR finder to extract these sentences with potential SRs. After that, HDiff transforms SRs into formal rules through dependency parsing analysis and text entailment techniques. 2) ABNF rules, which are standardized formal grammar and describe the syntax of parsable structures in HTTP protocol. HDiff implements an ABNF filter based on the format features to automatically extract ABNF grammar from RFC, including character cleaning, regular extraction, and separating prose rules. Based on those extracted semantic and syntax rules, HDiff further generates a large number of test cases through the ABNF generator and SR translator.

To overcome the second challenge, HDiff utilizes *Differential Testing* to discover semantic gap attacks. Previous work has proved that differential testing is a promising approach for discovering these attacks [19], [20], [29], [39]. HDiff uses different programs of the same functionality as cross-referencing oracles, comparing their outputs across many inputs. Any discrepancy in the programs' behaviors on the same input is marked as a potential bug. HDiff employs this differential fuzzing strategy, first testing each target implementation in isolation and then comparing logs and responses to identify the pairs that behave differently, signaling a vulnerability. Besides, HDiff can further determine whether a discrepancy conforms with RFC requirements since it has extracted formal rules by *Documentation Analyzer*, and thus can quickly locate the root cause of discovered bugs.

Experiments and Findings. We implemented the HDiff framework and evaluated it by detecting three kinds of semantic gap attacks: Host of Troubles (HoT) attack, Cache-Poisoned Denial-of-Service (CPDoS) attack, HTTP Request Smuggling (HRS) attack. In the experiment, HDiff first analyzed the core documents of HTTP 1.1 (i.e., RFC 7230-7235) [22]–[24], [30], [43], [44], which include 172,088 words and 5,995 valid sentences. It extracted 117 specification requirements (SRs) and 269 ABNF grammar rules. Based on that, HDiff generated 8,427 test cases using the SR translator and 92,658 test cases using the ABNF generator. Last, we tested the generated test cases against 10 popular HTTP implementations through difference analysis.

In total, we found 14 vulnerabilities covering all three types of attacks in 10 popular HTTP implementations (including Tomcat, IIS, Weblogic). Specifically, we found: 1) all HTTP proxies could be affected by our 11 exploits for CPDoS attacks; 2) Nine different servers pairs (e.g., Varnish-IIS, Nginx-Weblogic) are vulnerable to HoT attacks; 3) Eight HTTP implementations do not fully follow HTTP specifications, which could be potentially exploited for HRS attacks. We reported all identified vulnerabilities to the involved HTTP software vendors, and 7 new CVEs were assigned to these vulnerabilities.

In particular, we also discovered three new types of attack vectors that have not been discussed in previous work. We found that incorrect HTTP-version can be exploited to launch HRS (lower or higher HTTP-version with chunked encoding) and CPDoS (malformed HTTP-version, e.g., HTTP/0.9, 1.1/HTTP) attacks. Besides, the inconsistent processing of `Expect` header by different implementations can also lead to HRS or CPDoS attacks. More details are described in Section IV-B.

Contributions. Our contributions are outlined as follows:

- *New Detecting Framework.* We introduced HDiff, a novel detecting framework to discover semantic gap attacks that threaten the Internet. We utilized a documentation analyzer to generate a number of test cases from RFC documents and employed differential testing to find semantic gaps among different implementations.
- *New Implementation and Findings.* We implemented our design and evaluated it to discover three kinds of HTTP semantic gap attacks: HTTP Request Smuggling, Host-of-Trouble, and Cache-Poisoned Denial-of-Service attack. We tested it on 10 popular HTTP implementations and found new vulnerabilities that affect well-known HTTP software, including Apache, Tomcat, Weblogic, and Microsoft IIS Server. We release HDiff[1] through Github for researchers to further study vulnerabilities of HTTP implementations in the future.
- *Responsible Disclosure.* We reported our findings to affected vendors and received positive feedback. Among the discovered vulnerabilities, 7 new CVEs have been assigned to the immediately exploitable ones.

II. Background

A. HTTP Standards

Hypertext Transfer Protocol (HTTP) [44] is a text-based and ASCII-encoded protocol for fetching resources on the Web, which has been the foundation of web data transmission. HTTP standards are commonly defined in Request For Comments (RFCs), which describe the semantic information through natural language and the message format through ABNF rules. As stated in RFC 7230 [44], HTTP protocol

[1]HDiff : https://github.com/mo-xiaoxi/HDiff

conformance includes both the semantics and syntax of protocol elements. This work expects to analyze RFC documents extracting these elements for differential testing in HTTP protocol implementations.

Specification Requirement (SR). An RFC specification contains a series of sentences to express requirements on protocol implementations, such as “A server MUST reject any received request message that contains whitespace between a header field-name and colon with a response code of 400 (Bad Request)”. They are used to guide developers to implement the protocol correctly and ensure security. For convenience, we define this type of sentence as Specification Requirements (SRs). A key observation is that all SRs tend to follow a specific semantic structure, including *a message description* specifying the request format, and *a role action* indicating the action in terms of receiving or sending this message. Any semantic bug that causes deviations from these specification requirements might render the protocol insecure.

ABNF Rule. Augmented Backus-Naur Form (ABNF [21]) is a standardized formal grammar notation for context-free grammars, being used to describe the syntax of parsable structures in communication protocols. Most RFCs use ABNF to describe formal specifications, like the example in Figure 1. An ABNF rule is a set of derivation rules, such as `HTTP-version = HTTP-name "/" DIGIT "." DIGIT`. The left value is a rule name and the rule definition is set on the right side. Additionally, ABNF rules in one RFC can also reference ABNF rules in other RFCs in prose-val format (e.g., as shown in Figure 1 line 6). Since inconsistent parsing may lead to security vulnerabilities, such as HTTP request smuggling [3], it is essential to ensure that the parse of formal specification obeys ABNF rules.

```
1 HTTP-message = start-line *( header-field CRLF ) CRLF [ message-body]
2 HTTP-name = %x48.54.54.50 ; HTTP
3 HTTP-version = HTTP-name "/" DIGIT "." DIGIT
4 ...
5 Host = uri-host [ ":" port ]
6 uri-host = <host, see [RFC3986], Section 3.2.2>
7 Transfer-Encoding = *( "," OWS ) transfer-coding *( OWS "," [ OWS transfer-coding ] )
8 transfer-coding = "chunked" / "compress" / "deflate" / "gzip" / transfer-extension
```

Fig. 1: ABNF rules defining HTTP grammar from RFC 7230.

B. Natural Language Processing

Sentiment Analysis. Sentiment analysis [38], also known as opinion mining, is a technique using NLP and text analysis to identify, extract and study the emotional tone behind a body of text. One of its main tasks is sentiment classification, which aims to classify the polarity of the text into positive or negative opinions. A key observation is that all SRs presented in the specifications are characterized by a strong sentiment to stress the constraints. Thus, HDiff utilizes a sentiment classifier based on stanza [41] to find sentences with SR in RFCs.

Dependency Parsing. Dependency parsing [31] is an NLP technique to extract the dependency tree of a sentence. The dependency tree presents its grammatical structure and defines grammatical relations between the linguistic units (words) in one sentence. The state-of-the-art dependency parser (i.e., spaCy RoBERTa parser [1]) can achieve a 95.1% accuracy in grammatical relation discovery from a sentence. In our study, we leverage the spaCy parser to generate dependency trees in the Text2Rule converter module, which is particularly useful for extracting keywords and clause divisions of sentences.

Textual Entailment (TE). Textual entailment (TE [8]) is a directional relation between two natural-language texts, which predicts whether the facts in one of them (called a premise) necessarily imply the facts in the other (called a hypothesis). If a premise entails a hypothesis, then a human believing the premise would typically be able to conclude that the hypothesis is most likely true [10]. It is designed to identify that the same meaning is expressed by or can be inferred from, various language expressions, such as grammatical variations (e.g., passive tense), synonyms, and other semantic preserving transformations. Thus, it is particularly useful for determining the implication of specification requirement hypotheses. In our research, we utilize the AllenNLP library [33], which integrates state-of-the-art textual entailment NLP models for converting a specification requirement to a formal expression used for the SR translator.

C. Semantic Gap Attack

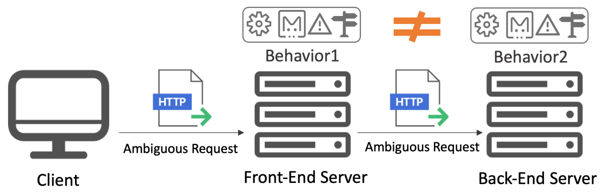


Fig. 2: Semantic Gap Attack: different behavior when parsing an ambiguous HTTP message.

As shown in Figure 2, the semantic gap attack leverages different behaviors between a front-end server (e.g., cache, proxy, firewall) and a back-end server (e.g., load balancing, proxy, origin server) when parsing one ambiguous HTTP request. The inconsistency in message processing can lead to different perceptions of an HTTP message’s syntactic validity or caching behavior. It will further lead to serious security vulnerabilities, such as cache poisoning, security policy bypassing, and denial-of-service attacks. In this study, we take three representative semantic gap attacks as examples to validate our approach.

HTTP Request Smuggling (HRS) Attack. An HTTP Request Smuggling (HRS [3]) attack exploits a semantic gap in parsing more than one `Content-Length` or `Transfer-Encoding` header fields to smuggle a hidden request through an intermediary. With this technique, a malicious client can provoke a web cache poisoning if two intermediaries pick different `Content-Length/ Transfer-Encoding` header fields and therefore read different amounts of the payload. Besides, this semantic gap bug can also be applied to hide malicious requests from security

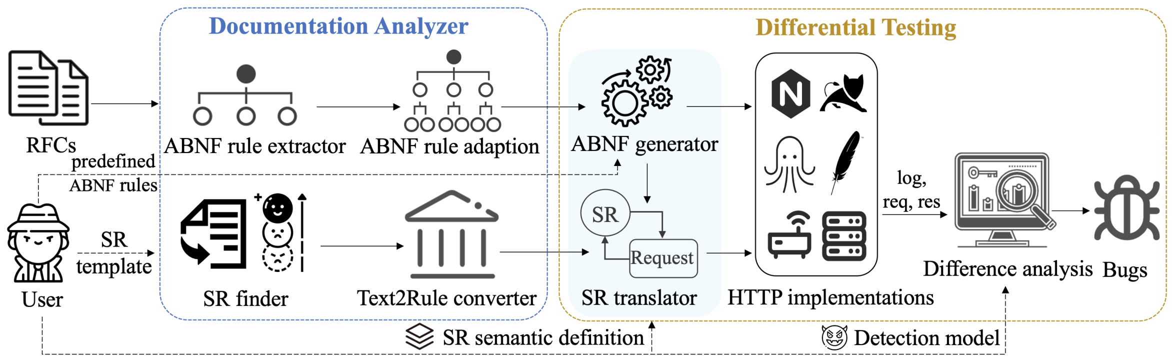


Fig. 3: The Architecture of HDiff.

intermediaries such as WAFs, Intrusion Detection Systems (IDS), and access control mechanisms.

Host of Troubles (HoT) Attack. An HoT attack leverages ambiguous interpretations of HTTP host headers to enable cache poisoning attacks and security policy bypasses [15]. Unlike HRS attacks, this attack may arise when an attacker can generate an HTTP request that contains multiple, ambiguous mechanisms to define the target host, such as multiple `Host` headers or a `Host` header combined with an `absolute-URI` in the request-line. Two different HTTP implementations may accept and understand an ambiguous request differently, which results in an exploitable semantic inconsistency.

Cache-Poisoned Denial-of-Service (CPDoS) Attack. This attack exploits semantic gaps to poison web caches with server-generated error pages, showing that the service is unavailable. Previous work [36] has introduced three CPDoS attack variants, HTTP Header Oversize (HHO), HTTP Meta Character (HMC), and HTTP Method Override (HMO), which exploit the mismatch of header size limits, meta character handling, and the method override header respectively.

III. HDiff: Design and Implementation

In this section, we first show the key observations before our design. And we further introduce the architecture of HDiff, followed by a specific example showing how it works. Then, we delve into the detailed techniques of all modules, including *Documentation Analyzer* and *Differential Testing*.

A. Key Observations

First, some HTTP implementations do not follow RFC requirements. Due to the robustness principle [40], many implementations tend to accept requests that violate RFC requirements. Besides, HTTP RFC specifications are often lengthy (e.g., RFC 7230 includes 89 pages in total). It is difficult for developers to fully understand all RFC specifications and develop a defect-free protocol processing code. As network protocols have become more complex, message processing code becomes more error-prone.

Second, RFC defines optional requirements allowing developers to use their discretion. For example, RFC documents often use keywords like MAY, SHOULD, or SHOULD NOT to specify rules. This flexibility may result in different preferences in implementing rules, which leads to varying HTTP implementations. The different preference appears harmless when examining each implementation independently, but may introduce semantic gap bugs when combining multiple implementations and lead to drastic security attacks.

Therefore, automatically extracting rules from RFC documents would be an effective method to find semantic gap attacks in different protocol implementations. Our research shows that this process can be semi-automated, with a detection model and an SR seed template as the input.

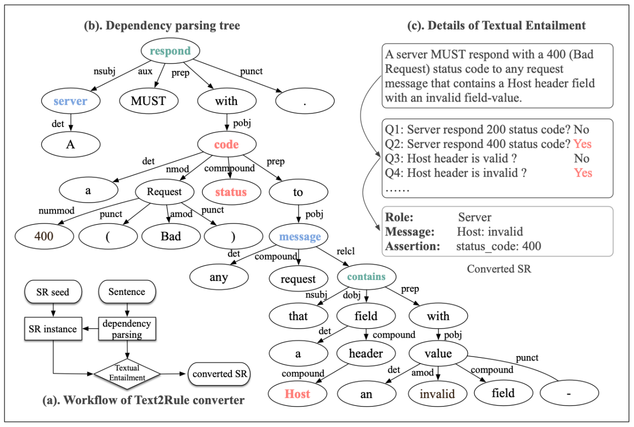
B. HDiff Overview

Architecture. Figure 3 illustrates the architecture of HDiff, including *Documentation Analyzer* and *Differential Testing*.

Documentation analyzer extracts both syntax rules from RFC documents, and semantic information based on a series of NLP-related technologies. More specifically, it extracts a valid and self-contained ABNF ruleset from RFCs through regular pattern matching. Meanwhile, it extracts specification requirements using an SR finder based on sentiment analysis and transforms SRs to formal expressions via a Text2Rule converter. This converter is designed based on dependency parsing and textual entailment techniques. Leveraging these rules, HDiff would generate test cases through an ABNF generator and an SR translator. To trigger as many discrepancies among multiple HTTP servers as possible, HDiff also applies mutations on valid requests generated from the previous steps.

Then, HDiff utilizes *differential testing* to discover semantic gap attacks. It first tests each software independently, and then compares logs, requests, and responses to identify the software pairs that behave differently, which indicate potential vulnerabilities. We introduce the concept of $\overrightarrow{HMetrics}$, which summarizes the observed asymmetries between the behavior of multiple HTTP implementations. Under different detection models, users can define detection rules based on $\overrightarrow{HMetrics}$ to discover semantic gap attacks.

HDiff is a semi-automatic framework as it requires four manual tasks for initialization. As shown in Figure 3, a user needs to provide: 1) SR template sets for Text2Rule converter; 2) SR semantic definition for SR translator; 3) detection

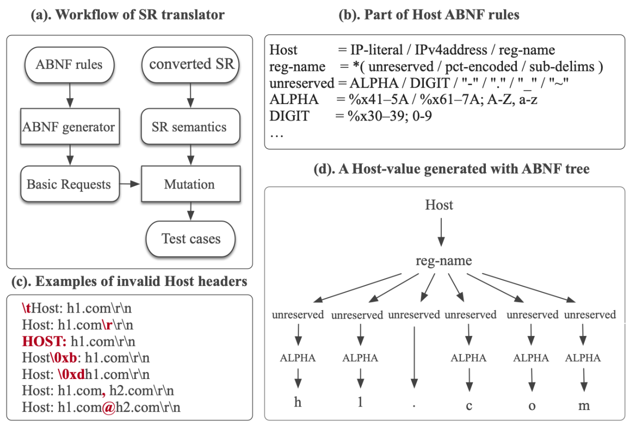


A server MUST respond with a 400 (Bad Request) status code to any request message that contains a Host header field with an invalid field-value.

Q1: Server respond 200 status code? No
Q2: Server respond 400 status code? Yes
Q3: Host header is valid ? No
Q4: Host header is invalid ? Yes
……

Role: Server
Message: Host: invalid
Assertion: status_code: 400

Fig. 4: An example showing details of Text2Rule Converter.



```
Host        = IP-literal / IPv4address / reg-name
reg-name    = *( unreserved / pct-encoded / sub-delims )
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA       = %x41–5A / %x61–7A; A-Z, a-z
DIGIT       = %x30–39; 0-9
…
```

```
\tHost: h1.com\r\n
Host: h1.com\r\r\n
HOST: h1.com\r\n
Host\0xb: h1.com\r\n
Host: \0xdh1.com\r\n
Host: h1.com, h2.com\r\n
Host: h1.com@h2.com\r\n
```

Fig. 5: An example showing details of SR Translator.

models for difference analysis and 4) predefined ABNF rules to optimize the ABNF generator.

Example. Below, we give an example of detecting HoT vulnerability to demonstrate how our approach works. As mentioned earlier, all SRs tend to express a strong sentiment in emphasizing the importance of a constraint. Thus, HDiff first selects sentences with a strong sentiment from RFC documents based on the SR finder, for instance, the sentence shown in Figure 4c. Then, the Text2Rule converter transforms it into a dependency tree (Figure 4b). In this tree, the converter can identify both the target role (`server`) through the relation `nsubj`, and the HTTP fields (`status code` and `Host`) that belong to the field dictionary parsed through ABNF rules. Then, HDiff analyzes the textual entailment of sentences based on HTTP fields and SR seed templates. For this sentence, it tries to infer the following semantics: (1) the `status code` is `200/400`, and (2) the `Host` header is valid/invalid/repeat. Finally, HDiff can get a converted SR (Figure 4c).

Next, the SR translator (Figure 5a) uses the converted SR to guide test case generation. Note that HDiff can generate basic HTTP requests with key-value pairs using ABNF rules. For example, Figure 5b shows some ABNF rules of the `Host` header, and Figure 5d illustrates a simple host-value case with an ABNF tree. The SR translator would mutate the basic requests based on converted SR semantics, such as case conversion, special character insertion, and field repetition.

Last, HDiff employs a differential fuzzing strategy to find semantic gap bugs in different HTTP implementations. In this example, HDiff can discover inconsistent interpretations of `Host` fields among different implementations, which may lead to an HoT attack.

C. Documentation Analyzer

The module consists of four major components: sentiment-based SR finder, Text2Rule converter, ABNF rule extractor, and ABNF rule adaptor. Below, we introduce the basic ideas and considerations for these components.

Sentiment-based SR Finder. Traditional regular templates or keyword-based approaches, as proposed by previous studies [34], [37], do not work well when handling RFC documents. First, RFC documents are described in natural language rather than formal language, in which the sentences are complex and flexible in expression. Second, different document authors may prefer different writing conventions, so it is hard to find common templates that cover the most SRs.

Despite the diversity of writing styles, we find all SRs tend to express a strong sentiment in emphasizing the importance of a constraint, particularly those security-critical constraints. The more important the SRs are, the more forceful the descriptions would be. For example, RFC 7230 states "the server MUST respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header field". The word "MUST" reflects a constraint with the strong sentiment.

Thus, to capture the sentences with potential SRs, HDiff implements a sentiment-based SR finder based on stanza [41]. This method is better than directly filtering SRs with RFC-defined keywords [13] (e.g., MUST, SHALL, SHOULD NOT), since there are still SRs that do not use these keywords, such as "chunked message is not allowed", "cannot contain a message body", and "ought to be handled as an error". Our sentiment analysis module can automatically identify such strong sentiment sentences with potential SRs.

Text2Rule Converter. Natural language expression is flexible, and the same semantics can be expressed in multiple forms, including synonym substitution and grammatical variations (e.g., passive tense). Therefore, converting an SR to a formal expression is not easy. To address this problem, HDiff employs dependency tree analysis and textual entailment (TE) techniques to identify grammatical variations, which can determine the implication of SR hypotheses.

Figure 4a shows the workflow of this module. First, a user needs to provide a message description (e.g., "[`field-name`] header is [represent/valid/invalid/multiple]") or a role action (e.g., "[`role`] respond [200/302/400] status code") as an SR template hypothesis. Among them, the field-name would automatically adapt to the header name defined in ABNF (i.e., the left value in the ABNF expressions). In general, the specification document would describe the common roles of the protocol. In this study, we collect the common 10 role names from RFC 7230 Section 2.5, such as sender, proxy, server, intermediary, and cache. Next, we introduce the dependency tree analysis to identify key messages in a sentence, such as a target role and HTTP-related fields. Then, HDiff can fill the seed template to get an SR instance. Last, HDiff performs a textual entailment analysis to classify sentences into the seed hypothesis. The detailed textual entailment is shown in Figure 4c. It works like an intelligent question answering system, which takes the target sentence with potential SR as the premise and asks whether the sentence implies the hypothesis (i.e., the SR seed instance provided by the user).

In addition, we also need to overcome other challenges in this module. First, sentences in RFC are often long and complex (e.g., with multiple parallel `or/and` clauses). The example shown in Figure 4 is simplified from a complex sentence (with more than 50 words and three co-ordinated clauses). The accuracy rate cannot be guaranteed for these sentences by directly applying textual entailment technology. The parallel semantic information of multiple clauses may also be lost. To solve this challenge, we first split a sentence into multiple short clauses based on dependency tree analysis. Then we perform textual entailment analysis on multiple clauses respectively, which makes the information derivation complete and effective. Specifically, we identify the relationship between different clauses based on Part-of-speech tagging [47]. For example, multiple clauses often are split by words with parallel lexical properties (i.e., `cc`, `conj`). Then, we can locate the dependency tree contextual relationship and split a sentence into multiple short clauses based on dependency tree analysis. Finally, we perform textual entailment analysis on multiple clauses respectively, which makes the information derivation complete and effective.

Second, the textual entailment of cross-sentence structures needs to be considered. Some phrases in RFCs have referential relationships between multiple sentences, such as "this message", "such request", "such URI". HDiff needs to identify its implicit references for these phrases to recover the original semantics between multiple sentences. To address this challenge, we try to eliminate anaphora using existing tools, such as AllenNLP [25] and NeuralCoref [7]. Unfortunately, none of such techniques can address subtle implicit references. We observe that these cross-sentence referential cases are much simpler in RFC documents than in the case considered in natural language. A referent phrase (e.g, "such request") can often find its referred clause in adjacent sentences (e.g., "a request is ..."). And no iterative referential relationships have been found to exist, i.e., the referred sentences include more referential relationships. Therefore, we implemented a simple forward search algorithm based on keyword fuzzy matching to search forward (up to 5) sentences to find the referred clause. Then, HDiff would merge the two sentences into a complex multi-clause sentence for textual entailment analysis when the referred clause is found.

ABNF Rule Extractor. Most RFCs use ABNF to describe formal specifications, which can describe the syntax of parsable structures. Therefore, we extract valid ABNF grammar from RFC to generate test cases and run differential testing. We use the following two steps to automatically export ABNF rules. First, HDiff collects all relevant RFC documents (RFC 7230-7235) through a datatracker tool [27]. Second, we implemented an ABNF filter based on format features to heuristically extract ABNF grammar in RFCs, including character cleaning, regular extraction, case escaping, and separating prose rules.

ABNF Rule Adaption. ABNF rules from different documents require certain automatic adaptation to obtain the final complete and error-free grammar set. To optimize ABNF rules, we use the techniques including replacing rule names with case–insensitive rule names, replacing invalid rule definitions with customized rules, and namespacing transformations for rules with the same name in different RFCs. More specifically, HDiff will use the most recent RFCs for repeated rule names. One challenge is that some missing rules are referenced but not defined. For example, rule A references rules B and C, but rule C is not defined. This could happen if a rule is defined as prose, defined in a referenced document, or in the surrounding textual description. If a rule is referenced from another RFC, HDiff will expand the ABNF rules of the referenced RFC. For example, RFC 7230 [44] contains an angle-bracket notation for prose descriptions, "`<host, see [RFC3986], Section 3.2.2>`". When encountering such rules, the program would automatically expand the ABNF rule set from the new RFC 3986 [11] document.

D. Differential Testing

HDiff utilizes an ABNF generator and a BS translator to generate test cases that are prone to semantic gap attacks. Then, it conducts a difference analysis on target HTTP implementations to discover vulnerable HTTP implementations.

ABNF Generator. HDiff includes a test case generator based on Python code, which is designed to generate numerous test cases directly from ABNF grammar. The approach is to recognize that ABNF defines a tree with seven types of nodes (e.g., alternation, option, concatenation, literal) and that each node represents an operation that can guide a depth-first traversal of the tree.

Specifically, HDiff first parses the ABNF grammar into an ABNF syntax tree. Then, HDiff would locate the target node (e.g., `HTTP-message`, `HTTP-version`) as the start node, and traverse the ABNF syntax tree recursively downwards. Among them, the leaf node (e.g., string literals, num literals) is the termination node.

The test cases generated directly based on the original ABNF syntax tree are often too distorted and easy to be directly rejected by the target server. For example, a valid ABNF Host header (`Host:\t!VAA2.:='i:22`). Due to its high degree of distortion, it is often easily rejected by the HTTP server. We address this problem in two steps. First, the ABNF grammar has variable repetition rules (e.g., `n*nRule`), which can theoretically generate an infinite depth of traversal subtrees. However, exhaustive depth traversal for this type of grammar is meaningless. Therefore, we limit the recursion depth of ABNF syntax tree traversal (e.g., maximum 7). Second, we loaded some predefined rules to reduce the generation of invalid strings, which can specify a certain value of leaf nodes for our empirical experiment purposes. For example, the `Host` header can consist of `IPv4address`. HDiff does not need to test all IPv4 addresses, only representative ones, such as `127.0.0.1` and `8.8.8.8`. In this way, we can generate basic HTTP requests that are fully RFC compliant and not be rejected by the server. These requests would be used as seeds for the SR translator.

To trigger possible processing discrepancies between different HTTP servers, HDiff also introduces common mutations on the valid requests, such as header repeating, inserting Unicode characters, header encoding, and case variation. Through these ways, HDiff can explore the processing of the corner case. We only apply several rounds of mutations to each test case so that the changes make a small impact on the format. It can avoid mutated requests to the degree that they are unrecognizable by the servers.

SR Translator. The SR translator would translate the SR previously extracted in the documentation analyzer module into test cases with assertions. If the protocol implementation violates the assertion in the testing phase, we believe that the target implementation violates the specification.

At this stage, we need to manually input SR semantic definitions to help translate SR into test cases. We defined a series of message descriptions (e.g., valid, invalid, repeat, empty, too long) and role actions (e.g., close connection, report error, respond 200 status code, not forward request). The former is used for automatically generating test cases, and the latter is for determining subsequent difference tests.

For example, when we extract an SR with a message format described as "including an invalid `Host` header", HDiff will first generate a series of host headers that match the ABNF rules and then mutate the original ABNF syntax tree to generate malformed host data.

While these tasks require manual effort, the key observation is that these semantics are limited and enumerable. The description of the message format has only limited relationships (e.g., field order, field values, field restrictions), and the server behavior is also limited (e.g., accept/reject/forward/rewrite requests, close/hold connections). Therefore, this manual work is worth the effort, considering the positive implications for subsequent vulnerabilities discovered.

Difference Analysis. For difference analysis, our key idea is that simultaneously testing multiple HTTP implementations on the same input offers a wide range of information that can be used to compare the tested programs' behaviors relative to each other. Such examples include error messages, debug logs, rendered outputs, return values, observed execution paths of each HTTP implementation, etc. Semantic gap bugs across different HTTP implementations are more likely for the inputs that cause relative features like the above across multiple test implementations.

Semantic Metrics. To analyze and evaluate testing results conveniently, we define an n-dimension vector $\overrightarrow{HMetrics}$ for the server behavior of each request:

$\overrightarrow{HMetrics} = \langle uuid, status\_code, host, data, ...\rangle$

Here, the uuid is a unique number for each request, and the status_code is the response status code from HTTP implementation. The host represents the parsing result of the `Host` field in the request, and the data is the request body. Users can also define much other semantic information (e.g., HTTP-version, method) related to HTTP protocol for discovering semantic gap bugs. This semantic information can be observed in various ways, including response data, error messages, and system logs.

Detecting Bugs. Under different detection models, users can define different detection rules based on $\overrightarrow{HMetrics}$ to discover semantic gap attacks. For example, for the HoT attack, the middleboxes need to forward ambiguous requests (although the middleboxes may modify the request to some extent). In addition, the Host value interpreted by the middleboxes is different from the backend server. At this point, HDiff would output the test case as a potential exploit together with the description of the vulnerability discovered. Then, we further run these potential exploits to complete verification in a real environment.

IV. Experiments and Findings

A. Experiment Setup

Tested HTTP Implementation. To evaluate HDiff, we systematically analyze 10 popular web servers and proxies, which are high in market shares and deployment rates [4], [5]. We believe their security issues can expose a wide range of common users to threats. Table I shows the tested HTTP implementations and their versions.

Experiment Platform Setup. We conducted the experiments on two kinds of virtual machines, Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-62-generic x86_64) and Microsoft Windows Server 2019 x64. Both are configured with one core of an Intel Core i7-4790 CPU (3.60GHz) and 8 GB RAM. Except for the Internet Information Services (IIS) service tested on Windows Server, other services are tested on Ubuntu. For accuracy, each version of HTTP implementation is tested in a different virtual machine.

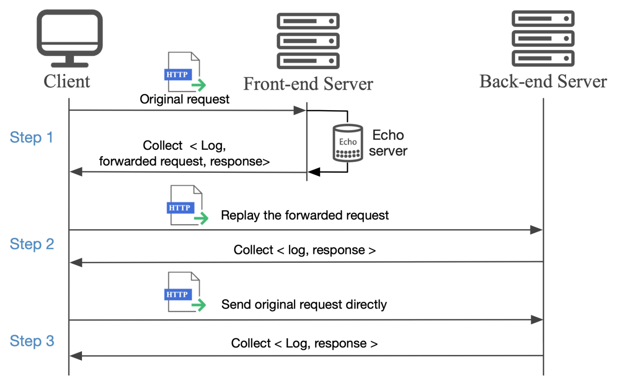

Client
Front-end Server
Back-end Server
Original request
Step 1
Collect < Log, forwarded request, response>
Echo server
Replay the forwarded request
Step 2
Collect < log, response >
Send original request directly
Step 3
Collect < Log, response >

Fig. 6: The test workflow.

TABLE I: Tested HTTP implementations and vulnerability.

Product	Version	Working Mode		Vulnerability		
		Server	Proxy	HRS	HoT	CPDoS
IIS[1]	10	Yes		✓	✓	–
Tomcat	9.0.29	Yes		✓	✓	–
Weblogic	12.2.1.4.0	Yes		✓	✓	–
Lighttpd	1.4.58	Yes		✓		–
Apache	2.4.47	Yes	Yes			✓
Nginx	1.21.0	Yes	Yes		✓	✓
Varnish	6.5.1		Yes	✓	✓	✓
Squid	5.0.6		Yes	✓		✓
Haproxy	2.4.0		Yes	✓	✓	✓
ATS[2]	8.0.5		Yes	✓		✓

[1] The abbreviation IIS stands for the Internet Information Service.
[2] The abbreviation ATS stands for the Apache Traffic Server.
[3] ✓: The target product is vulnerable.
[4] – : We do not consider the vulnerability to CPDoS in server mode.

TABLE II: Examples of semantic gap attacks found by HDiff.

HTTP Field	Description	Example	Vulnerability
Request-Line	Invalid HTTP-version	**1.1/HTTP**; HTTP/**3-1**; **h**TTP/1.1;	CPDoS
	lower/higher HTTP-version	HTTP/**0.9**;**1.0 with chunked**; HTTP/**2.0**	HRS, CPDoS
	Bad absolute-URI vs Host	**test://**h2.com/?a=1; **h1@h2.com**;	HoT
	Fat HEAD/GET request	HEAD/GET with **message-body**	HRS, CPDoS
Header-field	Invalid CL/TE header	Content-Length: **+6**\r\n Content-Length: **6,9**\r\n Content-Length: **[sc]**9\r\n **[sc]**Transfer-Encoding: chunked\r\n Transfer-Encoding: chunked**\r**\r\n Transfer-Encoding**[sc]**: chunked\r\r\n	HRS

	Multiple CL/TE headers	Content-Length: 10\r\n
		Content-Length: **0xff**\r\n;
		Content-Length: 10\r\n
		Transfer-Encoding**[sc]**: chunked\r\n;
			HRS

	Invalid Host header	Host: h1.com**@**h2.com\r\n Host: h1.com**,** h2.com\r\n Host: h1.com**/../../test?**\r\n Host:**[sc]** h1.com\r\n	HoT, CPDoS
	Multiple Host headers	**[sc]**Host: h1.com\r\n Host: h2.com\r\n	HoT
	Hop-by-Hop headers	Connection: close,**Host**\r\n Connection: **Cookie**\r\n	CPDoS
	Expect header	**Expect:** 100-continuce\r\n	HRS, CPDoS
	Obs-fold header	Host: h1.com**\t\n**h2.com\r\n	HoT
	Obsoleted header or value	Transfer-encoding: chunked,**identity**\r\n	HRS, CPDoS

Message-body	Bad chunk-size value	**[bignumber]**\r\nabc\r\n0\r\n;
		0xfgh\r\nabc\r\n9\r\n;
			HRS

	NULL in chunk-data	3\r\n**\x00**abc\r\b0\r\n	HRS

1. The symbol **[sc]** represents special characters, including common spaces(e.g., ,\t,\0xb,\0xd), grammatical characters(e.g., {,},<,>,@,',"$) and unicode characters (e.g., \u0000,\uffff,\u202e).

Test Workflow. The test environment includes one client, one echo server, six proxy (front-end) servers, and six back-end servers. We show a simple schematic diagram in Figure 6.

HDiff automatically generates HTTP requests via the SR translator and the ABNF generator and associates each request with a UUID. The client uses multiple processes to send these test cases to target HTTP implementations (i.e., proxy servers and back-end servers). We directly perform low-level network programming (e.g., raw socket) to reduce confusion during packet parsing from our test tool. All proxies run in reverse-proxy mode; they receive requests and forward them to our echo server. Echo servers would record the forwarded requests for subsequent replay analysis. Besides, each back-end server would feedback on its interpretation of HTTP requests through application scripting languages, such as PHP, and ASPX. Besides, in the combined analysis, all proxies are configured to cache any returned response, such as the response of a non-200 status code, the HTTP version is smaller than 1.1.

Specifically, we collect three types of data for difference analysis: (1). Proxy logs that consist of status_code, host, uri, and other information parsed by the proxy; (2). Echo server logs that record the requests forwarded by proxies; (3). Echo information and logs of back-end servers, which show the parsing results from the end servers.

Since semantic gap attacks are caused by inconsistent behaviors between the actual application logic and the intermediaries, we adopt a behavior-oriented methodology to detect HTTP message processing chains. The workflow can be divided into three steps.

Step 1: The client sends the test case to the proxy, and the proxy automatically forwards the request to the echo server. We can learn how the proxy handles requests through this step, such as forwarding a malformed header or multiple Host headers.

Step 2: HDiff replays the forwarded request to each back-end server. Note that HDiff would reduce the number of replay tests through a series of rules and heuristics to improve fuzzing efficiency. For example, it only replays the forwarded request of which the proxy processing status code is 200 and that contains ambiguous data. Through this step, we can simulate the actual environment of multiple agents and servers in series without building many test environments, which significantly reduces the test workload.

Step 3: HDiff also sends test samples directly to the back-end server to test its understanding and behavior.

By analyzing steps 1 and 3, HDiff can determine whether the tested proxy and server follow RFC specifications, otherwise, they can cause security issues like semantic gap attacks. By combining steps 1 and 2, HDiff can perform a front-end and back-end combination analysis to find exploitable server pairs.

Ethical Considerations. We take almost care to prevent ethical problems in our experiments. First, this study was conducted within a local experimental environment, and no real users or external servers were affected by our experiment. Second, we followed the established coordinated disclosure best practices. Vulnerabilities found in this work have already been reported to all relevant HTTP software vendors.

B. Findings

In our experiment, we mainly analyzed the core specifications of HTTP 1.1 (RFC 7230-7235) [22]–[24], [30], [43], [44], including 172,088 words and 5,995 valid sentences. In total, HDiff reported 117 specification requirements (SRs) and 269 ABNF grammar rules and generated 8,427 test cases with assertions based on the SR translator and 92,658 test cases based on the ABNF generator.

Then, HDiff further found a number of (more than 100) violations of SRs and discrepancies in different HTTP implementations. Finally, we verify the exploitability of the results and confirm that suspected vulnerable HTTP implementation pairs are behind each other. Table I shows the vulnerability of each implementation. All three attacks (HoT, HRS, and CPDoS) have successful attack payloads. We briefly show some of the discovered vulnerabilities in Table II. Among these vulnerabilities, we also discovered three new attack vectors that have not been discussed in previous work.

In addition, we also performed a front-end and back-end combination analysis that we believe has some real-world deployment. Figure 7 summarizes our results, showing the affected server pairs we found. Note that some vulnerabilities may not be applied in these server pairs, but it does not rule out they are not exploitable in the real world. These vulnerabilities may lead to exploitable attacks when chained with other HTTP implementations, such as using CDN as a front-end server.

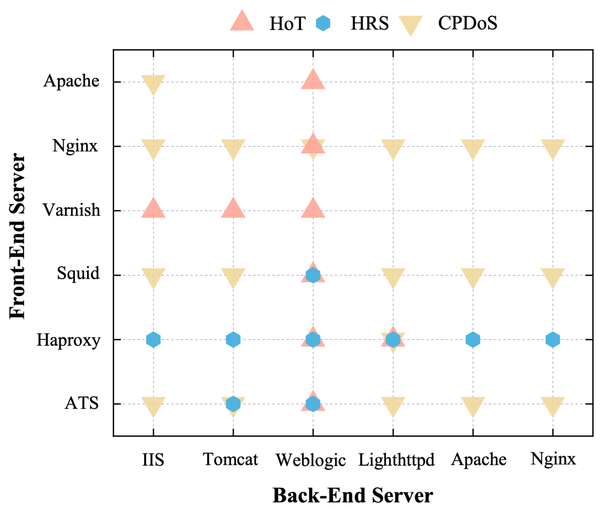


Fig. 7: Server pairs affected by three types of attacks.

HTTP Request Smuggling (HRS) Attack. We have discovered eight different types of HRS attack payloads. In this paper, we only introduce four representative examples. For convenience, we use the abbreviation CL for `Content-Length` and TE for `Transfer-Encoding`.

Invalid CL/TE header. Some HTTP implementations (e.g., IIS, ATS, Weblogic) are compatible and accept requests that violate the RFC definition for robustness considerations. RFC 7230 [44] clearly states that the whitespace between field-name and colon is not allowed, such as `Content-length[whitespace]: 10`. However, the IIS server is compatible with this request type and parses the body data. This improper behavior can cause inconsistencies between proxies and servers, leading to the HRS attack.

Multiple CL/TE headers. According to RFC 7230 [44], the recipient must reject the message with multiple CL/TE or replace the duplicated field-values with a single valid value. In our experiments, the traditional multiple headers have been rejected by the HTTP implementations. However, many HTTP implementations became vulnerable when HDiff made a slight mutation (e.g., special character insertion) based on the ABNF generator on the CL/TE headers. A typical example is that Tomcat will accept requests with both CL and TE headers, where the TE header is malformed data (i.e., `Transfer-Encoding:\x0bchunked`). This type of request may be interpreted inconsistently in other HTTP implementations, further leading to the HRS attack.

HTTP Version 1.0 with TE chunked. The TE header was first introduced in HTTP version 1.1 [44]. So a request with version 1.0 and a TE header should be accepted, but the TE header should be ignored and not interpreted. In the experiments, Tomcat does not support chunked encoding in HTTP version 1.0, while other HTTP implementations support it. This inconsistent behavior can also lead to HRS vulnerabilities.

Bad chunk-size value. Almost all proxies would attempt to recover a usable protocol element from an invalid construct. However, the proper error handling mechanism is not easy. Attackers can abuse improper error handling mechanisms to launch attacks. The attacker can construct an illegal chunk-data to deceive the message correction mechanism and make the repaired data still contain semantically ambiguous data. For example, attackers can make chunk-size mismatches with chunk-body, a big number in chunk-size, or control/Unicode characters in chunk-body. In our experiment, two proxies (i.e., Haproxy, Squid) would try to repair the request with a malformed chunk-data, such as `[big number]\r\nabc\r\n0\r\n`. The correct behavior should repair a big number to `3` (the length of 'abc'), but they repair to an illegal number `a` (10 in decimal), which may be due to integer overflow issues. This incorrect repair behavior can cause an HRS attack.

Host of Trouble (HoT) Attack. Many HTTP implementations have fixed the attacks before [15]. However, we generated more test cases through the ABNF mutation generator. Below, we show two types of effective attack vectors.

Bad absolute-URI vs Host header. HTTP allows a client to send `absolute-URI` as request-target, which contains a host component. The inconsistent interpreting between `absolute-URI` and `Host` is easy to cause the HoT attack. To mitigate this attack, many proxies rewrite the absolute-URI to its path and add a `Host` header when forwarding. However, we found varnish does not rewrite the Host header if the absolute-URI is started with a non HTTP schema. It recognizes the host from the `Host` header and forwards such requests transparently. When IIS and Tomcat receive such requests, they recognize the host from `absolute-URI`. This inconsistent understanding of `host` between the proxy and server would lead to the HoT attack. In addition, Haproxy would transparently forward a request with HTTP schema absolute-URI and no `Host` header. This may also lead to the HoT attack.

Invalid Host header. Based on ABNF mutation generator, we can generate a large number of hostnames with slight distortion, such as '`h1.com@h2.com`', '`h1.com,h2.com`', '`h1.com/../h2.com`'. When a proxy transparently forwards such an ambiguous hostname, it is also easy to cause the HoT attack. The front-end proxy might recognize the h1.com, while the back-end server recognizes the h2.com. Three proxies (i.e., varnish, haproxy, squid) would forward such requests without modification.

Cache-Poisoned Denial-of-Service (CPDoS) Attack. The CPDoS attack is the most easily triggered semantic gap attack. In this work, we found 11 different kinds of CPDoS attack payloads (some examples are shown in Table II).

Invalid HTTP-version. As mentioned earlier, proxies implement the message correction mechanism. Incorrect message correction can also be abused to launch a CPDoS attack. Three proxies (i.e., Nginx, Squid, ATS) would try to repair the request with invalid version (e.g., `1.1/HTTP`, `HTTP/3-2`). They do not delete the old illegal HTTP version but directly add their own HTTP version in the request line. This makes the forwarded request line become `GET /?a=b 1.1/HTTP HTTP/1.0`, which also may cause errors on the backend server and further lead to a CPDoS attack.

Blindly forwarding lower/higher HTTP-version. RFC 7230 [44] requires that all intermediaries (other than acting as tunnels) must forward their own HTTP version instead of blindly forwarding requests. However, Haproxy would transparently forward the `HTTP/0.9` message with request headers, resulting in a CPDoS attack. Only the Weblogic server can handle this message and respond with a 200 status code, while the rest servers report errors.

Blindly forwarding Expect header in GET request. The `Expect` header field in a PUT request indicates a certain set of behaviors (expectations) that need to be supported by the server to handle this request properly. However, it may cause a CPDoS or HRS attack when employed in a GET request. In our experiment, ATS would transparently forward such requests. And Lighttpd would direct reject such a message. As a result, it will cause a CPDoS attack if chained together.

Fat HEAD/GET request. In this scenario, the body is used in the GET request, and the length of the body is indicated by a CL header. This request is automatically generated based on the ABNF grammar. Since the RFC does not strictly explain how the server should handle it, most HTTP implementations loosely handle the GET request with body data. There will also be some implementations that directly consider this type of request to be illegal. As a result, different HTTP implementations would have an inconsistent semantic understanding of such requests, which may also cause HRS and CPDoS attacks.

V. Discussion

Limitations. We only tested HTTP implementations in their default configurations in the experiments and may not find all those bugs because configurations may vary in the wild. For example, some researchers [2] exploited the CPDoS or HRS attack with the forwarded header (e.g., X-Original-URL, X-Forwarded-Host, X-Forwarded-For). But our experimental environment does not support these headers, and we cannot find such attack vectors. In addition, there are numerous HTTP implementation combinations in real-world deployment (e.g., different types of products, versions, configurations). Generally, the front server can be a cache, proxy, firewall, or CDN, while the end server can be another proxy, CDN, or server. Our experiment only tested the scenario where the proxy is the front-end server, and the HTTP server is the back-end server. Therefore, some combinations that cause potential semantic gap attacks may not be covered in our environment. Besides, HDiff is a semi-automatic framework, which still requires four human tasks for building HDiff, i.e., a series of SR template sets, SR semantic definitions, detection models, and predefined ABNF rules. These limitations render our system less effective in discovering new semantic gap attacks. Additionally, when migrating this approach to other protocols, these initial investments also need to be completed as a cost.

Cost and Benefit. HDiff is a semi-automatic framework involving some manual tasks. As described in Section III, these manual tasks are limited, enumerable, and acceptable (e.g., less than 8h throughout our work). In addition, these manual tasks are one-time jobs, which can be reused for testing multiple versions of the HTTP specifications. By automatically extracting rules from HTTP specifications, HDiff can check and determine whether an implementation conformed with or violated specific RFC rules. This helps fix bugs in HTTP implementations. Besides, once the tool is developed, we can reuse the test cases for discovering vulnerabilities in more implementations. And the tool can be run periodically to prevent new vulnerabilities introduced by software updates. Thus, in the long run, we believe the tool will have a good return on investment.

Future Research. More automated and intelligent security test based on specification analysis is worth exploring. Our study on HDiff has made the first step toward this end, but its application just scratched the surface of a large problem space such a technique can make inroads into. HDiff still requires a part of manual tasks to assist in testing, and more automated differential testing techniques are worth exploring.

RFC specifications standardize many communication protocols (e.g., SIP, SMTP, TLS). A good way is to extend our methodology to different protocols and systematically discover semantic gap attacks. Chen et al. [16] conducted an empirical security analysis to find semantic gap attacks bypassing the email security mechanisms. Practical exploration is to apply our framework to the email domain to find this kind of semantic gap attack. Besides, semantic inconsistency between multi-version protocols is an interesting issue. It has been shown, e.g., that a client can cause various types of denial-of-service attacks in cases where an intermediary supports HTTP/2 while the webserver uses HTTP/1.1 [28]. Thus, it is also valuable to expand our work to the HTTP 2.0 version.

VI. Disclosure and Response

All vulnerabilities have been reported to related HTTP software vendors before this paper was submitted. And all vendors would have about six months to implement mitigation techniques. Some vendors quickly confirmed the vulnerabilities and have indeed fixed the vulnerabilities, such as Tomcat, and Weblogic. 7 new CVEs have been assigned to the immediately exploitable ones among the vulnerabilities discovered. Our contact results are summarized as follows.

Tomcat: They acknowledged our report and particularly thanked us for reporting the issue of HRS attacks. And two CVEs (CVE-2019-17569, CVE-2020-1935) for publicly known information-security vulnerabilities and exposures.

Internet Information Services (IIS): They accepted our report and confirmed the vulnerability (CVE-2020-0645). They contacted us and had an in-depth discussion with us about the specifications. They mention that for robustness, proxies may not follow strict RFC guidance when processing malformed requests. Now they strictly process the headers related to HRS attacks for security reasons.

Weblogic: They discussed with us the details of the attacks and their potential consequences. They viewed it as indeed a problem for the HTTP software vendors and actively contacted us to discuss how to defend against it. They also allocated three CVEs (CVE-2020-2867, CVE-2020-14588, CVE-2020-14589) for these vulnerabilities we found.

Apache Traffic Server (ATS): They evaluated the issue as a critical vulnerability (CVE-2020-1944). At first, they believe that the transparent forwarding of repeated headers should not cause security problems. They have now recognized the risk of transparently forwarding repeated `Transfer-Encoding` headers and fixed the vulnerabilities.

Haproxy: They appreciated our work and discussed the vulnerabilities with us in detail. They mentioned that most proxies or gateways deal with HTTP requests in non-RFC ways. They recognize that semantic gap attacks are a severe threat and have done some work to defend against such attacks, such as not cached if the HTTP version is smaller than 1.1 or the response status code is not `200`.

Others: We have contacted other relevant HTTP software vendors and are looking forward to receiving their feedback.

VII. Related Work

Semantic Gap Attack. In recent years, there have been several works about semantic gap attacks, such as evasion attacks against security software [9], bypassing email security mechanisms [16], [45], inconsistent interpretations between different SSL/TLS [42], [46] or HTTP [28], [32], [36] implementations. Gil et al. [26] introduced the Web Cache Deception (WCD) attack, which aims to disclose sensitive information with the help of caches. These previous works demonstrated that semantic gap attacks had been a big threat to the Internet, yet most vulnerabilities are still found by manual analysis, which heavily relies on human experiences and is not scalable. The latest work, T-Reqs [29], uses differential testing to discover HRS attacks and generates test cases based on manually defined context-free grammar (CFG). Unlike his work, our work applies NLP-based techniques to extract RFC rules and constructs test cases to reduce human efforts. Using this approach, we explored three types of semantic attacks instead of a single HRS attack.

Differential Testing Previous works [18], [29], [48] have proved that differential testing is an effective way to discover semantic gap attacks. Among them, Nezha [39] uses differential testing to find semantic gap bugs in SSL/TLS implementations. However, a discrepancy in traditional differential testing is neither why it occurs nor which implementations go wrong. By contrast, HDiff can determine whether a discrepancy conforms with RFC and quickly locate the root causes. In addition, traditional differential testing requires at least two HTTP implementations. Otherwise, it cannot find any discrepancy. HDiff can test a single implementation by checking whether $\overrightarrow{HMetrics}$ matches the assertion from SRs.

Document Analysis for Security. Numerous studies leveraging text analysis techniques to automatically discover various kinds of bugs in diverse domains like SSL/TLS implementations [20], cellular networks [17], API Misuse Detection [34]. Chen et al. [17] implemented a framework that automatically discovers vulnerabilities using the guidance from the LTE documentation. Blasi et al. [12] generate the test oracle from documentation to dynamically find the inconsistency between documentation and code implementation. Different from previous works, our research provides a systematic framework to discover a wide variety of semantic gap attacks in HTTP protocol, including HTTP Request Smuggling, Host-of-Trouble, and Cache-Poisoned Denial-of-Service attack.

VIII. Conclusion

Various semantic gap attacks have been discovered in recent years, which have become a severe threat in web-based layered software systems with multiple intermediaries. However, most of these attacks have been found through ad-hoc manual analysis, which is inadequate for fundamentally enhancing the security assurance of a system as complex as the HTTP network. Our research developed HDiff, a novel detecting framework to discover semantic gap attacks in HTTP implementations from RFC specifications. HDiff employs a suite of NLP primitives to extract the syntax and semantic information in RFCs. It further constructs test cases based on ABNF and SR rules, and runs differential testing to systematically discover semantic gap attacks.

Running HDiff on the HTTP 1.1 core specifications, we discovered 14 vulnerabilities covering all three types of attacks, including three new attacks vectors never reported before in previous work. All of them have been duly reported to the involved HTTTP software vendors. And 7 new CVEs have been assigned to the immediately exploitable ones. With its efficacy demonstrated on HTTP protocols under three attack models, we believe that the HDiff framework has great potential to be applied to other protocols for detecting semantic gap attacks. We hope this work can inspire the community to discover and eliminate semantic gap attacks in other areas.

Acknowledgement

We sincerely thank our shepherd Shuai Hao and all the anonymous reviewers for their valuable reviews and comments to improve this paper. This work is supported in part by the National Natural Science Foundation of China (Grant No. U1836213 and U19B2034), the BNRist Network, and Software Security Research Program.

References

[1] "Facts & Figures · spaCy Usage Documentation." [Online]. Available: https://spacy.io/usage/facts-figures/

[2] "Smuggling HTTP headers through reverse proxies." [Online]. Available: https://github.security.telekom.com/2020/05/smuggling-http-headers-through-reverse-proxies.html

[3] "Http request smuggling," Aug. 2015. [Online]. Available: https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

[4] "12 best web servers as of 2021," May 2021. [Online]. Available: https://www.tecmint.com/best-open-source-web-servers/

[5] "8 top open source reverse proxy servers," May 2021. [Online]. Available: https://www.tecmint.com/open-source-reverse-proxy-servers-for-linux/

[6] "Middlebox," Aug. 2021, page Version ID: 1040594944. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Middlebox&oldid=1040594944

[7] "Neuralcoref 4.0: Coreference resolution in spacy with neural networks." May 2021. [Online]. Available: https://github.com/huggingface/neuralcoref

[8] "Textual entailment," May 2021, page Version ID: 1023387199. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Textual_entailment&oldid=1023387199

[9] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1690–1701.

[10] R. Bar-Haim, I. Dagan, and I. Szpektor, "Benchmarking applied semantic inference: The PASCAL recognising textual entailment challenges," in *Language, Culture, Computation. Computing - Theory and Technology - Essays Dedicated to Yaacov Choueka on the Occasion of His 75th Birthday, Part I*, ser. Lecture Notes in Computer Science, N. Dershowitz and E. Nissan, Eds., vol. 8001. Springer, 2014, pp. 409–424. [Online]. Available: https://doi.org/10.1007/978-3-642-45321-2_19

[11] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform resource identifier (URI): generic syntax," *RFC*, vol. 3986, pp. 1–61, 2005. [Online]. Available: https://doi.org/10.17487/RFC3986

[12] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[13] S. Bradner, "Rfc2119: Key words for use in rfcs to indicate requirement levels," 1997.

[14] A. Büttner, H. V. Nguyen, N. Gruschka, and L. L. Iacono, "Less is often more: Header whitelisting as semantic gap mitigation in http-based software systems," in *ICT Systems Security and Privacy Protection - 36th IFIP TC 11 International Conference, SEC 2021, Oslo, Norway, June 22-24, 2021, Proceedings*, ser. IFIP Advances in Information and Communication Technology, A. Jøsang, L. Futcher, and J. M. Hagen, Eds., vol. 625. Springer, 2021, pp. 332–347. [Online]. Available: https://doi.org/10.1007/978-3-030-78120-0_22

[15] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of Troubles: Multiple Host Ambiguities in HTTP Implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 1516–1527. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978394

[16] J. Chen, V. Paxson, and J. Jiang, "Composition kills: A case study of email sender authentication," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2183–2199.

[17] Y. Chen, Y. Yao, X. Wang, D. Xu, C. Yue, X. Liu, K. Chen, H. Tang, and B. Liu, "Bookworm game: Automatic discovery of LTE vulnerabilities through documentation analysis," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1197–1214. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00104

[18] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 85–99. [Online]. Available: https://doi.org/10.1145/2908080.2908095

[19] Y. Chen and Z. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo Italy: ACM, Aug. 2015, pp. 793–804. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786835

[20] Chu Chen and Cong Tian and Zhenhua Duan and Liang Zhao, "Rfc-directed differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 859–870. [Online]. Available: https://doi.org/10.1145/3180155.3180226

[21] D. Crocker and P. Overell, "Augmented BNF for syntax specifications: ABNF," *RFC*, vol. 5234, pp. 1–16, 2008. [Online]. Available: https://doi.org/10.17487/RFC5234

[22] R. T. Fielding, Y. Lafon, and J. F. Reschke, "Hypertext transfer protocol (HTTP/1.1): range requests," *RFC*, vol. 7233, pp. 1–25, 2014. [Online]. Available: https://doi.org/10.17487/RFC7233

[23] R. T. Fielding, M. Nottingham, and J. F. Reschke, "Hypertext transfer protocol (HTTP/1.1): caching," *RFC*, vol. 7234, pp. 1–43, 2014. [Online]. Available: https://doi.org/10.17487/RFC7234

[24] R. T. Fielding and J. F. Reschke, "Hypertext transfer protocol (HTTP/1.1): authentication," *RFC*, vol. 7235, pp. 1–19, 2014. [Online]. Available: https://doi.org/10.17487/RFC7235

[25] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. Peters, M. Schmitz, and L. S. Zettlemoyer, "Allennlp: A deep semantic natural language processing platform," 2017.

[26] O. Gil, "Web cache deception attack," *Black Hat USA*, vol. 2017, 2017. [Online]. Available: https://www.blackhat.com/docs/us-17/wednesday/us-17-Gil-Web-Cache-Deception-Attack-wp.pdf

[27] S. Ginoza, M. Cotton, and A. Morris, "Datatracker extensions to include IANA and RFC editor processing information," *RFC*, vol. 6359, pp. 1–18, 2011. [Online]. Available: https://doi.org/10.17487/RFC6359

[28] R. Guo, W. Li, B. Liu, S. Hao, J. Zhang, H. Duan, K. Sheng, J. Chen, and Y. Liu, "Cdn judo: Breaking the cdn dos protection with itself." in *NDSS*, 2020.

[29] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event Republic of Korea: ACM, Nov. 2021, pp. 1805–1820. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3485384

[30] R. a. Julian F. Reschke, "Hypertext transfer protocol (HTTP/1.1): semantics and content," *RFC*, vol. 7231, pp. 1–101, 2014. [Online]. Available: https://doi.org/10.17487/RFC7231

[31] S. Kübler, R. McDonald, and J. Nivre, "Dependency parsing," *Synthesis lectures on human language technologies*, vol. 1, no. 1, pp. 1–127, 2009.

[32] W. Li, K. Shen, R. Guo, B. Liu, J. Zhang, H. Duan, S. Hao, X. Chen, and Y. Wang, "Cdn backfired: Amplification attacks based on http range requests," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 14–25.

[33] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *ArXiv*, vol. abs/1907.11692, 2019.

[34] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for API misuse detection," in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds. ACM, 2020, pp. 1837–1852. [Online]. Available: https://doi.org/10.1145/3372297.3423360

[35] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, "Cached and confused: Web cache deception in the wild," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 665–682. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/mirheidari

[36] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your cache has fallen: Cache-poisoned denial-of-service attack," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 1915–1936. [Online]. Available: https://doi.org/10.1145/3319535.3354215

[37] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. M. Paradkar, "Inferring method specifications from natural language API descriptions," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 815–825. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227137

[38] B. Pang and L. Lee, "Opinion Mining and Sentiment Analysis," *Foundations and Trends® in Information Retrieval*, vol. 2, no. 1–2, pp. 1–135, 2008. [Online]. Available: http://www.nowpublishers.com/article/Details/INR-011

[39] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 615–632. [Online]. Available: https://doi.org/10.1109/SP.2017.27

[40] J. Postel, "Transmission control protocol," *RFC*, vol. 793, pp. 1–91, 1981. [Online]. Available: https://doi.org/10.17487/RFC0793

[41] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, and C. D. Manning, "Stanza: A Python natural language processing toolkit for many human languages," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2020. [Online]. Available: https://nlp.stanford.edu/pubs/qi2020stanza.pdf

[42] L. Quan, Q. Guo, H. Chen, X. Xie, X. Li, Y. Liu, and J. Hu, "SADT: syntax-aware differential testing of certificate validation in SSL/TLS implementations," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 524–535. [Online]. Available: https://doi.org/10.1145/3324884.3416552

[43] Roy T. Fielding and Julian F. Reschke, "Hypertext transfer protocol (HTTP/1.1): conditional requests," *RFC*, vol. 7232, pp. 1–28, 2014. [Online]. Available: https://doi.org/10.17487/RFC7232

[44] Roy T. Fielding and Julian F. Reschke, "Hypertext transfer protocol (HTTP/1.1): message syntax and routing," *RFC*, vol. 7230, pp. 1–89, 2014. [Online]. Available: https://doi.org/10.17487/RFC7230

[45] K. Shen, C. Wang, M. Guo, X. Zheng, C. Lu, B. Liu, Y. Zhao, S. Hao, H. Duan, Q. Pan *et al.*, "Weak links in authentication chains: A large-scale analysis of email sender spoofing attacks," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[46] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 521–538.

[47] A. Voutilainen, "Part-of-speech tagging," *The Oxford handbook of computational linguistics*, pp. 219–232, 2003.

[48] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532