T-Reqs: HTTP Request Smuggling with Differential Fuzzing

Bahruz Jabiyev
Northeastern University
Boston, MA, USA

Steven Sprecher
Northeastern University
Boston, MA, USA

Kaan Onarlioglu
Akamai Technologies
Cambridge, MA, USA

Engin Kirda
Northeastern University
Boston, MA, USA

ABSTRACT

HTTP Request Smuggling (HRS) is an attack that exploits the HTTP processing discrepancies between two servers deployed in a proxy-origin configuration, allowing attackers to smuggle hidden requests through the proxy. While this idea is not new, HRS is soaring in popularity due to recently revealed novel exploitation techniques and real-life abuse scenarios.

In this work, we step back from the highly-specific exploits hogging the spotlight, and present the first work that systematically explores HRS within a scientific framework. We design an experiment infrastructure powered by a novel grammar-based differential fuzzer, test 10 popular server/proxy/CDN technologies in combinations, identify pairs that result in processing discrepancies, and discover exploits that lead to HRS. Our experiment reveals previously unknown ways to manipulate HTTP requests for exploitation, and for the first time documents the server pairs prone to HRS.

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

HTTP Request Smuggling; HTTP Desync Attacks

ACM Reference Format:
Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. 2021. T-Reqs: HTTP Request Smuggling with Differential Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3485384

1 INTRODUCTION

Due to the continuing proliferation of web caches, proxies, cloud services, and Content Delivery Networks (CDNs) that deploy massively-distributed networks made up of these technologies, a typical HTTP request is often processed by multiple intermediate servers before it reaches its destination. ***HTTP Request Smuggling (HRS)*** is an attack that exploits the discrepancies between HTTP processing semantics of these different servers to confuse them about message boundaries, and consequently smuggles unintended requests into the connection inside the request body.

HRS was first documented by Linhart et al. in 2005 [24]. However, the technique took off only recently when researchers proposed novel variants and demonstrated attacks on high-profile targets (e.g., [6, 15, 19, 23]). Ultimately, smuggling was shown to be a serious threat leading to response queue and cache poisoning, which can then be exploited for myriad nefarious purposes such as personal data leakage, credential theft, session hijacking, denial of service, and security control bypass attacks, resulting in thousands of dollars in bug bounties (e.g., [5, 16, 17]).

While these same researchers also released tools (e.g., [7, 36]) that partially automate the detection of HRS, these are largely intended for assisting website owners and penetration testers in probing specific targets for vulnerabilities. These tools are also narrowly scoped, primarily testing for exploits that involve the manipulation of two particular HTTP headers, `Content-Length` and `Transfer-Encoding`, which govern how servers determine HTTP message bounds.

To date, HRS has not been studied in a systematic manner; the disclosed vulnerabilities were instead driven by case studies targeting popular websites. In particular, previous work on HRS leaves two important gaps in our understanding of HRS attacks.

First and foremost, HRS is a system interaction problem, involving at least two HTTP processors on the traffic path. These processors may not necessarily be individually buggy; but when used together, they disagree on the parsing or semantics of a given HTTP request, which leads to a vulnerability. This key aspect of HRS has not been explored in previous work. Next, previous attacks focus on malicious manipulation of the two aforementioned HTTP headers. Whether the remaining HTTP headers, or the rest of an HTTP request, could be tampered with to induce similar processing discrepancies remains uncharted territory.

In this paper, we present the first study that investigates HRS in a scientific framework, and we tackle the above research questions. Namely, we present a novel experiment setup with 10 popular web servers and proxies: Apache, NGINX, Tomcat, Apache Traffic Server (ATS), HAProxy, Squid, Varnish, Akamai, Cloudflare, and CloudFront. We study these technologies in pairs, investigating which combinations are vulnerable to HRS. To that end, we propose a grammar-based fuzzer called T-Reqs that incorporates string and tree mutations targeting a large variety of HTTP headers, the request line, and the request body. T-Reqs employs a *differential* fuzzing strategy, first testing each target technology in isolation,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485384

and then comparing responses to identify the pairs that behave differently, signaling a potential vulnerability.

Once we identify the combinations that exhibit discrepancies and the payloads that trigger them, we deploy every server pair in a proxy-to-origin formation for further experimentation and verification of our findings. We examine the conditions which cause the potential HRS attacks to succeed or fail in this setup, and finally demonstrate a range of exploits we discover.

Our results show that attacks can indeed be induced by manipulating every part of a request, and highlights that HRS is a complex system interaction problem that can crop up as a result of seemingly innocuous processing discrepancies between pairs of web technologies that are otherwise shown to be secure in isolation.

We summarize our contributions as follows:

- We present the most comprehensive study of HRS to date, and examine attacks within a scientific framework for the first time in literature.
- We propose a novel approach and experiment setup that identify the HTTP processing discrepancies between 10 popular web servers and proxy services often used together.
- We develop a grammar-based differential HTTP fuzzer called T-Reqs, and make it open source.
- We discover novel HRS payloads made possible by manipulating HTTP request parts beyond the `Content-Length` and `Transfer-Encoding` headers.
- We systematically examine the practical conditions that determine the success of HRS.

Availability. T-Reqs is open-source and publicly available on the authors' websites.

Ethical Considerations. This study was conducted within a controlled experiment setup, and no attacks were launched against any external entities. We followed the established coordinated-disclosure best practices; we notified all tested technology vendors of our findings, provided them with a copy of this paper, and made our data and team available for further assistance.

2 BACKGROUND & RELATED WORK

In this section we explain the basic terminology we use in the rest of this text, and summarize how HRS attacks work.

2.1 HTTP Requests & Chunked Encoding

Listing 1 illustrates the structure of a typical HTTP request, made up of the following three components.

(1) **Request Line.** Line 1 is the request line for this request, which specifies the HTTP method (`POST`), the requested URI (`/search`), and the protocol version (`HTTP/1.1`).

(2) **Header Block.** This section follows the request line, listing header fields and values that define various parameters of the communication. On lines 2-3, the `Host` header specifies the endpoint the request should be dispatched to, and `Content-Length` indicates the length of the message body.

(3) **Request Body.** Separated from the header block by a blank line containing a carriage return and a line feed (often indicated by `CRLF` or `\r\n`), the request body starts on line 5 and contains the message payload. Here, the body consists of a parameter and its value, `query=funny+cats`.

Table 1: Breakdown of the chunked body.

`6\r\n`	Chunk size
`query=\r\n`	Chunk data
`a;foo=bar\r\n`	Chunk size & chunk extension
`funny+cats\r\n`	Chunk data
`0\r\n`	Last chunk
`X-Header:value\r\n`	Trailer part
`\r\n`	Terminating CRLF

Chunked transfer encoding is an alternative encoding scheme available in HTTP/1.1, where the message body is split into multiple chunks transferred independently. This mechanism is useful for streaming applications, when the size of the data to be transferred is not known a priori [10].

Listing 2 shows the same request as before, this time using chunked encoding. Every data chunk is preceded by its size, specified in hexadecimal. Both the size and the data are terminated by `CRLF`. Optionally, a *chunk extension* may immediately follow the size and contain metadata (e.g., a hash of the chunk data). The last chunk is a regular but empty chunk with a size of zero. Again, optionally, the last chunk can include a trailer which is treated similar to message headers, used for sending additional information to the receiver. Table 1 presents a breakdown of these chunk components.

Note that the `Transfer-Encoding: chunked` header in Listing 2 indicates to the receiver that chunked encoding is in effect. When using chunked encoding, sending the `Content-Length` header is not meaningful, and in fact, according to RFC 7230, this is prohibited: *"A sender MUST NOT send a Content-Length header field in any message that contains a Transfer-Encoding header field."* [10]

2.2 HTTP Request Smuggling (HRS)

HRS stems from a discrepancy between the HTTP processing behaviors of two servers that process the same request on the traffic path. These servers could be any technology that intercepts, parses, interprets, or forwards the request, including CDNs, stand-alone proxies, web caches, load balancers, or security products. In this text, we call the first server receiving the request the *entrypoint*, and the next one the *exitpoint*. While this abstraction is sufficient for our discussion, note that a typical request may be processed by more than two such entities, and a hazardous combination of any two could lead to attacks.

HRS involves a maliciously-crafted request such that the entry and exitpoints disagree on the bounds of the message. All documented attacks we previously discussed in Section 1 achieve this by including both the `Content-Length` and `Transfer-Encoding`:

```
1  POST /search HTTP/1.1
2  Host: example.com
3  Content-Length: 16
4
5  query=funny+cats
6
7
8
9
10
```

Listing 1: Regular body.

```
1  POST /search HTTP/1.1
2  Host: example.com
3  Transfer-Encoding: chunked
4
5  6
6  query=
7  a;foo=bar
8  funny+cats
9  0
10 X-Header:value
```

Listing 2: Chunked body.

```
1 POST /search HTTP/1.1
2 Host: example.com
3 Content-Length: 33
4 Transfer-Encoding: ;chunked
5
6 0
7
8 GET /img/i.jpg HTTP/1.1
9 X:X
```

Listing 3: Malicious smuggler request.

```
1 GET /js/j.js HTTP/1.1
2 Host: example.com
3
4
5
```

Listing 4: Incoming request.

```
1 GET /img/i.jpg HTTP/1.1
2 X:XGET /js/j.js HTTP/1.1
3 Host: example.com
4
5
```

Listing 5: Smuggled request.

chunked headers in the request – if the entrypoint honors one header and the exitpoint the other, they parse the message body differently, and an HRS attack is possible. Even when both servers are strictly RFC compliant so that they reject or normalize messages containing both headers, and attacker can still abuse header *parsing* discrepancies (e.g., by introducing invisible characters or minor syntax errors into headers) and prevent one of the servers from recognizing an invalid header combination.

Let's illustrate the attack through an example.

(1) The attacker crafts a *smuggler request* which includes a second hidden request inside the message body. Listing 3 shows such a request including `Content-Length: 33` and `Transfer-Encoding: ;chunked` together. Note the extra semicolon in the latter, which will serve to confuse the message parser in the next step.

(2) The entrypoint receives the request, but cannot correctly parse `Transfer-Encoding: ;chunked` due to the semicolon. The server disregards chunked encoding and instead parses the message body according to the `Content-Length: 33` header. As a result, the entrypoint forwards all 33 bytes shown between lines 5-9 to the next hop.

(3) The exitpoint receives the same request, correctly parses `Transfer-Encoding: ;chunked` thanks to its lenient parser ignoring errors, and processes the body in chunks. Consequently, the exitpoint treats lines 6-7 as the terminating empty chunk, and ignores lines 8-9.

(4) The unprocessed data shown on lines 8-9 remain in the request buffer of the exitpoint. When eventually another request arrives through the same connection (Listing 4), it is appended to this unprocessed data, making up a brand new request (Listing 5). This new request is only seen and processed by the exitpoint; the attacker has successfully *smuggled* it through the entrypoint.

In this example, assuming that the entrypoint is a web cache and the exitpoint a web application server, the attacker uses HRS to launch a cache poisoning-based denial-of-service attack. Specifically, the web cache expects a JavaScript file in response (see Listing 4), but instead receives an image from the application (see Listing 5) and erroneously caches that, likely breaking the application

```
<start> ::= <request>
<request> ::= <line><headers><newline><body>
<line> ::= "POST /search HTTP/1.1\r\n" | "PUT / HTTP/1.1\r\n"
<headers> ::= <host><content-length> | <host>
<host> ::= "Host: example.com\r\n"
<content-length> ::= "Content-Length: 16\r\n"
<newline> ::= "\r\n"
<body> ::= "query=funny+cats" | "query=carrots"
```

Listing 6: Example CFG for a simple HTTP request.

until the cache expires. This is but one example, and researchers have shown that HRS can be utilized for general classes of attacks such as cache poisoning, cache deception, session hijacking, circumvention of security controls, and response queue poisoning, as well as abusing application specific design flaws [6, 15, 19, 23, 24, 35].

2.3 Differential Fuzzing

Fuzzing is a well-established software testing approach with many applications in systems security [14, 38]. Of particular interest for our purposes is *differential fuzzing*, based on the idea of differential testing [27], where the focus is to identify differing behavior between applications when given the same input. To name recent examples, this method was used to detect side-channel attacks [32], to expose vulnerabilities in parsers and applications [33], and to find RFC violations in TLS libraries [37].

To apply this technique to the HTTP protocol, we construct our fuzzer using a custom *context-free grammar (CFG)*. Context-free grammars are sets of rules that allow for a formal definition of a structure, e.g., an HTTP request, and values that correspond to that structure. From this grammar, we are able to generate valid inputs to our system, and make our fuzzing mutations based off of them. An example CFG that produces an HTTP request is shown in Listing 6. Grammar-based fuzzers have also been previously used for software bug hunting (e.g., [1]).

A CFG has four components: a start symbol, non-terminal symbols, terminal symbols, and production rules. The start symbol is where the expansion of a CFG starts from. In Listing 6, the start symbol is denoted by `<start>`. Symbols surrounded by <> are non-terminals, meaning they are expanded before the input is fully generated. For example, `<request>` is expanded to a sequence of other non-terminal symbols, whereas, `<line>` can be expanded into multiple terminal strings. Finally, production rules define how symbols are expanded. Each line in Listing 6 is a production rule. When this CFG is fully expanded, one of the possible results is the request shown in Listing 1.

2.4 Other Related Work

An emerging line of research is the application of HRS to higher HTTP protocol versions; in particular, Emil Lerner and James Kettle independently presented attacks on HTTP/2 [18, 22]. These utilize the same techniques as before, but exploit flaws in the protocol downgrade mechanisms when an entrypoint converts HTTP/2 to HTTP/1.1 before forwarding requests to the exitpoint. Our work does not explore this area.

Beyond the presentations, proof-of-concept exploits, and white papers we discussed so far, there is no academic literature on HRS as

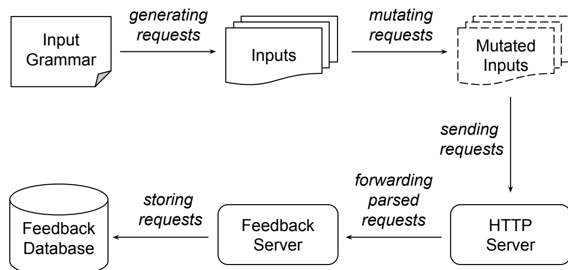


Figure 1: Inputs are generated from a grammar, mutated, and sent to the tested server. The feedback server collects feedback from the requests forwarded by the tested server and stores it for analysis.

of this writing. However, while this paper is the first work exploring HRS within a scientific framework, there exists studies that propose other ways to abuse HTTP processing discrepancies.

Omer Gil presented a novel cache poisoning attack called Web Cache Deception (WCD), which exploits an object cacheability disagreement between a web server and a cache, resulting in data leaks in public caches [12, 13]. Mirheidari et al. generalized WCD as a path confusion problem caused by a discrepancy in the interpretation of a requested URL [28], and conducted a large-scale measurement to identify vulnerable sites in the wild [29].

Nguyen et al. presented a different take on cache poisoning, crafting HTTP requests that are considered valid by a web cache while triggering an error at the origin server [31]. As a result, the error response is erroneously cached, resulting in a denial-of-service attack. Similarly, Chen et al. exploited HTTP servers that respond differently to ambiguities in the `Host` header values, which once again leads to cache poisoning [4].

3 RESEARCH QUESTIONS & METHOD

Previous work on HRS presents valuable concepts behind the attack, but does not explore the issue in depth or breadth, instead demonstrating impact through specific case studies. Our work is motivated by this knowledge gap. Below, we detail our guiding research questions, and explain our methods to answer them.

(Q1) Can we systematically test for HRS at scale?
(Q2) What parts of a request can induce processing discrepancies?
(Q3) What escalates a processing discrepancy to HRS?
(Q4) What technology stacks are at risk?

(Q1) Can we systematically test for HRS at scale? Previous work relies on a combination of manual testing and basic tools designed to target specific controlled environments (e.g., [7, 36]) for attack discovery. In contrast, we aim to design a fully-automated, generalizable, and extendable methodology that can explore HRS at scale and discover previously unknown venues for exploitation.

To address (Q1), we design a multi-stage experiment powered by a novel CFG-based differential fuzzer, T-Reqs. This is an automated process, eliminating the manual labor and narrow scope hindering previous work. This methodology and infrastructure to explore HRS systematically equips us to answer the remaining research questions. Below, we briefly describe the 3 stages of our experiment.

Stage 1. We first point T-Reqs to a set of popular HTTP servers for testing, and send identical requests to each. We record the processing behavior of each individual server, and combine the results to identify servers that process the same request differently.

At this stage, we experiment with each server in isolation to analyze their individual behavior. Each server runs in a reverse-proxy mode, where they receive requests and forward them on to our *feedback server.* The feedback server gleans information about the processing behavior of the tested server by analyzing the forwarded request. This information is stored in a database for later analysis. We specifically look for mismatches between parsed message body lengths, and label those as discrepancies.

Figure 1 depicts this whole process, where the top row is internal to T-Reqs, and the bottom row is the rest of the experiment infrastructure. Note that, in order to avoid adding a confounding layer of parsing in our own tools, we use low-level network programming.

Stage 2. We then reduce our set of discrepancies found in Stage 1 based on rules and heuristics detailed in Section 5. Essentially, we associate mutation sets with the server pairs they cause a discrepancy for, minimize them down to a representative group, and finally, manually classify these groups based on their mutation patterns. We stress that this manual classification is not mandatory; we merely include this step to simplify the presentation for our readers by attaching intuitive labels to similar discrepancy types.

Stage 3. Finally, we verify the exploitability of the results from Stage 2. To achieve this, we layer and deploy suspected vulnerable HTTP server pairs behind each other. We use a testing method inspired by prior work to check whether a given mutated request can really be used for HRS. We present the details of this method in Section 5.3.

(Q2) What parts of a request can induce processing discrepancies? Previous work has explored the parsing discrepancies involving `Content-Length` and `Transfer-Encoding` headers. Whether the remaining request components can be abused to similar effect remains an open question.

We address (Q2) by considerably expanding that scope. Not only do we allow T-Reqs to mutate additional headers, but we also investigate whether abusing the request line and the message body can also induce discrepancies, opening up novel attack vectors. We run three separate experiments, one for each HTTP request component listed above, each following the same stages we designed for addressing (Q1). In each experiment, we only allow T-Reqs to mutate the part under focus, while keeping the other two request components unmutated. This makes it feasible to pinpoint and reason about the exploitable discrepancies in isolation.

(Q3) What escalates a processing discrepancy to HRS? The presence of a discrepancy is a red flag, but not all discrepancies necessarily lead to HRS. In particular, while exploits involving `Content-Length` and `Transfer-Encoding` headers are intuitive (i.e., they directly affect the body parsing behavior, which is a prerequisite for HRS), why the discrepancies in other request components may lead to an attack is not obvious.

To explore (Q3), while we verify our findings in Stage 3 of the experiment, we analyze the conditions affecting exploitability. We document the novel and successful exploit mechanisms we identify, and also the failures that hinder attacks in practice.

(Q4) What technology stacks are at risk? HRS is a system interaction problem involving two HTTP processors, which may not be flawed when operating in isolation; but stacked together they

lead to a vulnerability. Previous work on HRS has made no attempt to measure what server combinations are prone to smuggling.

While we cannot feasibly test all technology combinations in existence, we make the first systematic attempt to answer (Q4) by designing an experiment that measures and documents the hazardous interactions between 10 HTTP processors.

Specifically, we pick popular web server, proxy, and CDN technologies in use today that make up a large portion of the Internet: Apache, NGINX, Tomcat, Apache Traffic Server (ATS), HAProxy, Squid, Varnish, Akamai, Cloudflare, and CloudFront. For specific versions, please see Appendix A. To test CDN vendors, we subscribe to their free or trial tier services. We configure each technology to run as a reverse-proxy fronting our feedback server (except Tomcat, which has no reverse-proxy mode, so we run a Java servlet on it that echoes back the received requests). We use default configurations, save for turning off buffering in NGINX to speed up testing, and disabling caching for clean experiment runs.

4 T-REQS SYSTEM DESIGN

We now detail the design of T-Reqs, our grammar-based differential HTTP fuzzer. T-Reqs is capable of generating HTTP requests as inputs from a grammar, manipulating them with string and tree mutations, and sending them to multiple HTTP servers in parallel for testing.

4.1 Input Generation

To ensure that we test all relevant components of an HTTP request, and their applicable values, T-Reqs uses a context-free grammar (CFG) to generate inputs. Each generated input is a *valid* HTTP request constructed by following one of the paths provided by the CFG, chosen randomly to ensure uniform testing. We record each random seed as the input ID to aid in reproducibility.

When building our input from the included grammar, we adopt a tree structure. The start symbol becomes the root, and each non-terminal is a non-terminated node in the tree. The leaves of the tree, once fully expanded, are made up of the terminal symbols (i.e., string literals), and when combined form our HTTP request. We present the specific CFGs used for our experiments in Section 5.

4.2 Mutating Inputs

In order to exercise the parsers of, and consequently trigger processing discrepancies between, different HTTP servers, T-Reqs makes mutations on the valid requests generated in the previous step.

Symbols, each corresponding to an HTTP element, can be marked in one of two ways: string mutable or tree mutable. If a symbol is not marked, it is assumed to be immutable. While string mutations (e.g., character insertion, deletion) make small changes to parts of an input, tree mutations lead to structural changes (e.g., repeated method specification, missing protocol version). This allows T-Reqs to test both trivial and major changes to an input. Mutation operations are formally defined in Appendix B.

In each iteration, T-Reqs randomly applies up to 2 mutations on each input. This upper bound makes the impact analysis of specific mutations feasible, as well as helping us avoid changing requests to the degree that they are unrecognizable by the servers.

```
1 PORT //search HTTP/1.1
2 Host: example.com
3 Content-Length: 13
4
5 query=bananas
```

Listing 7: String mutations.

```
1 HTTP /search /search HTTP/1.1
2 Host: example.com
3 Content-Length: 13
4
5 query=bananas
```

Listing 8: Tree mutations.

4.3 String Mutations

If a symbol is string mutable, then a random character can be deleted, replaced, or inserted at a random position inside that symbol. To add or replace characters, an external character pool can be defined. T-Reqs uses the ASCII character set (codes 0-127) as the character pool suitable for HTTP requests.

Listing 7 shows an example. The last character in the protocol version (1) is deleted, a letter in the method name (S) is replaced with R, and a forward slash is inserted at the start of the URI.

4.4 Tree mutations

If a symbol is tree mutable, then a random symbol can be deleted, replaced, or inserted at a random position. To add or replace symbols, an external pool of elements can be defined. T-Reqs uses the list of all symbols marked mutable as the external symbol pool.

For example, the request line is represented by `<request-line>` and has several sub-elements including `<method>`, `<URI>`, `<proto>`, and `<version>`. In Listing 8, it is assumed that `<request-line>` is tree mutable, and the following tree mutations are applied: 1) `<method>` is replaced by `<proto>`, 2) an extra `<URI>` is inserted after the current URI, and 3) the existing `<proto>` is deleted.

5 EXPERIMENT DETAILS AND RESULTS

In this section, we provide details and discuss results from the experiments listed in Section 3. We first run three separate experiments on each part of the HTTP request utilizing T-Reqs to expose discrepancies in message body parsing behavior. Next, we reduce, minimize, and categorize the sets of mutations that cause these discrepancies to understand what leads servers to disagree on message boundaries. We then verify the HRS potential of these categories, and explore reasons why they succeed or fail.

5.1 Stage 1 - Finding Discrepancies

For this stage, we run three separate experiments on each part of the HTTP request: the request line, request headers, and request body. Table 2 shows the duration of each experiment, and the number of requests generated and tested. We found that mutations in the request line experiment caused more errors (e.g., `400 Bad Request`), even when bounding the total number of mutations to two. We see more mutations an hour in the request line experiment because these errors are significantly faster for HTTP servers to handle than valid requests.

To make T-Reqs more efficient, we supply different grammars and mutable symbols for each experiment as detailed below.

5.1.1 Request Line Experiment Details. Listing 9 shows the grammar for the request line experiment. We test the standard HTTP methods as defined by their RFCs [9, 11]. Note that we do not

Table 2: General information about experiments.

Name	Duration	# Inputs
Request line	70 hours	8,857K
Request headers	94 hours	3,096K
Request body	72 hours	2,051K

Table 3: Mutability of request line symbols.

String Mutable	Tree Mutable
<method-name> <space> <protocol> <separator> <version> <newline>	<request-line>

Table 4: Mutability of request body symbols.

String Mutable	Tree Mutable
<zero> <trailer-part> <chunk-data><newline> <chunk-size> <chunk-extension>	<chunked-body> <chunk> <last-chunk>

Table 5: Experiment success values.

Experiment Name	# Inputs	# Successful
Request line	8,857K	5K
Request headers	3,096K	1K
Request body	2,051K	595K

test the HTTP/2 or HTTP/3 protocols, but our generated requests merely appear to use them. As we show, the protocol values still trigger unexpected parsing behaviors nonetheless.

Table 3 details what symbols of the grammar are marked as string or tree mutable. In order to test mutations solely on the request line, we mark only those symbols as mutable.

5.1.2 Request Headers Experiment Details. Listing 10 details the grammar for the request headers experiment. We used all standard HTTP request headers as detailed in the "Message Headers" registry of IANA [20], and valid values from their corresponding RFC documents. For a full listing of all 67 headers and values used in this experiment, refer to Section C in Appendix.

Mutable symbols for this experiment consist of the 67 headers and their sub-elements depicted in the full grammar definition. String literals in the grammar are marked as string mutable, while all other symbols are marked as tree mutable.

5.1.3 Request Body Experiment Details. Listing 11 depicts the grammar for the request body experiment. This experiment focuses on chunked bodies, as they have a complex structure with the highest potential for parsing discrepancies. We include all chunked body components in the input grammar, namely, the chunk size, chunk extension, chunk data, trailer and last chunk. We also add the `Trailer` header to the grammar, since it is required to include additional fields at the end of chunked messages. Unlike other experiments, we fix the method to POST and the version to 1.1.

The grammar defines a symbol called `<padding>` which adds 200 `D` characters after the last chunk. This symbol lets us determine whether the experiment server used `Transfer-Encoding` or `Content-Length` when parsing the body. If the server uses `Content-Length`, the output will include our padding values; otherwise, the padding will be omitted from the output, since this is the expected behavior for chunked bodies.

```
<start> ::= <request>
<request> ::= <request-line><base><the-rest>
<request-line> ::= <method-name><space><uri><space>
↪ <protocol><separator><version><newline>
<method-name> ::= "GET" | "HEAD" | "POST" | "PUT" | "DELETE" |
↪ "CONNECT" | "OPTIONS" | "TRACE" | "PATCH"
<space> ::= " "
<uri> ::= "/_URI_"
<protocol> ::= "HTTP"
<separator> ::= "/"
<version> ::= "0.9" | "1.0" | "1.1" | "2.0" | "3.0"
<newline> ::= "\r\n"
<base> ::= "Host: _HOST_\r\nConnection:close\r\nX-Request-ID:
↪ _REQUEST_ID_\r\n"
<the-rest> ::= "Content-Length: 5\r\n\r\nBBBBB"
```

Listing 9: CFG for request line experiment.

```
<start> ::= <request>
<request> ::= <method-name><request-uri><http-version><base>
↪ <entity-size-header><some-header><some-header><body>
<request-uri> ::= " /_URI_ "
<http-version> ::= "HTTP/0.9" | "HTTP/1.0" | "HTTP/1.1"
<method-name> ::= "GET" | "HEAD" | "POST" | "PUT" | "DELETE" |
↪ "CONNECT" | "OPTIONS" | "TRACE"
<base> ::= "\r\nHost: _HOST_\r\nConnection:close\r\nX-Request-ID:
↪ _REQUEST_ID_\r\n"
<entity-size-header> ::= <content-length> |
↪ <chunked-transfer-encoding> |
↪ <content-length><chunked-transfer-encoding> |
↪ <chunked-transfer-encoding><content-length>
<some-header> ::= <accept> | <accept-charset> | (truncated) |
↪ <user-agent> | <via>
<body> ::= "\r\nA\r\nBBBBBBBBBB\r\n0\r\n\r\nBBBBB(truncated)"

(truncated)
```

Listing 10: CFG for request headers experiment.

```
<start> ::= <request>
<request> ::=
↪ <base><entity-size-headers><trailer><chunked-body><padding>
<base> ::= "POST /_URI_ HTTP/1.1\r\nHost:
↪ _HOST_\r\nConnection:close\r\nX-Request-ID: _REQUEST_ID_\r\n"
<entity-size-headers> ::= <content-length><transfer-encoding> |
↪ <transfer-encoding>
<content-length> ::= "Content-Length: 200\r\n"
↪ <transfer-encoding> ::= "Transfer-Encoding: chunked\r\n"
<trailer> ::= "Trailer: Content-Length\r\n\r\n" | "Trailer:
↪ Transfer-Encoding\r\n\r\n" | "Trailer: Foo\r\n\r\n" | "\r\n"
<chunked-body> ::= <chunk><last-chunk><newline> |
↪ <chunk><last-chunk><trailer-part><newline>
<chunk> ::=
↪ <chunk-size><chunk-extension><newline><chunk-data><newline> |
↪ <chunk-size><newline><chunk-data><newline>
<chunk-size> ::= "4"
<chunk-extension> ::= ";foo=bar"
<chunk-data> ::= "BBBB"
↪ <last-chunk> ::= <zero><chunk-extension><newline> |
↪ <zero><newline>
↪ <zero> ::= "0"
<trailer-part> ::= "Transfer-Encoding: chunked\r\n" |
↪ "Transfer-Encoding: identity\r\n" | "Content-Length: 180\r\n" |
↪ "Bar: Foo\r\n"
<newline> ::= "\r\n"
<padding> ::= "DDDDDDDDDD(truncated)"
```

Listing 11: CFG for the request body experiment.

Table 4 shows that three chunked body symbols are marked as tree mutable, while the other symbols are string mutable. The rest of the request remains immutable.

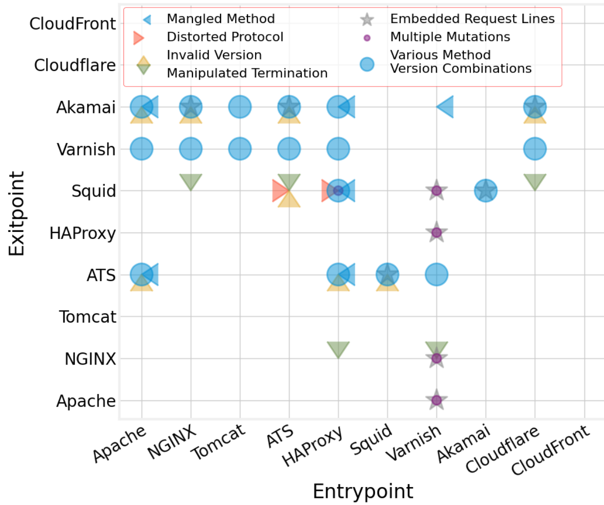


Figure 2: Request line mutation categories affecting server pairs.

5.2 Stage 2 - Discrepancy Reduction and Classification

We now detail the process for determining mutation success, and present our classifications of mutation sets that cause discrepancies.

Successful Mutation Sets. To determine if a mutation from the previous stage has HRS potential, we first need to define what a successful mutation is.

A successful mutation set causes a discrepancy in the body parsing behavior in at least one server pair, where the absence of the mutation set does not. Essentially, if a mutation set causes a discrepancy that the base unmutated request does not, we consider that a successful mutation set.

Table 5 shows the number of successful mutations for each experiment. To gain deeper insights into the causes and potential for HRS, we set out to reduce this set further. We reduce mutation sets based on the following definition.

A mutation set M_2 is reducible to M_1 iff. $M_1 \subseteq M_2$ and $s(M_2) \subseteq s(M_1)$ where $s(M)$ is the set of server pairs which disagree about parsing on an input mutated by a mutation set M.

Classification of Mutation Sets. We classify mutation sets based on their mutation pattern beyond the specifics of how the mutations are carried out. For example, all mutations deleting, replacing, or inserting a character in the method of a request line follow the same pattern: *Distorting Method.* We explore these categories and the server pairs they affect for each experiment below.

5.2.1 Request Line Experiment. Figure 2 lists all mutation categories affecting the request line, and the server pairs that disagree on body parsing for each category. Table 6 shows examples for each mutation category and the impacted server pairs.

Mangled Method. This class comprises mutation sets which modify the method name. Mutations can change the case of a letter, replace the entire method name, or modify it in another way. The first row in a Table 6 shows an example where a single mutation causes a discrepancy between 4 different server pairs. We observe that the entrypoint (Apache or HAProxy) parses and forwards the body of the request, while the exitpoint (Akamai or ATS) ignores it. The remaining servers return an error because of the mutation. For brevity, in the remainder of this section an error should be assumed if an experiment server is not mentioned explicitly.

Table 6: Examples for each request line mutation category.

Category	Request Line	Entrypoint-Exitpoint
mangled method	`hEAD / HTTP/1.1\r\n`	Apache-Akamai Apache-ATS HAProxy-Akamai HAProxy-ATS
distorted protocol	`GET / HhTTP/1.1\r\n`	ATS-Squid
invalid version	`GET / HTTP/`~~`1`~~`.19\r\n`	ATS-Akamai ATS-Squid
manipulated termination	`CONNECT / HTTP/1.0 \r\n\r\n`	Varnish-NGINX HAProxy-NGINX
embedded request lines	`OPTIONS / HTTP/OPTIONS / HTTP/0.9\r\n1.1\r\n`	Akamai-Squid
multiple mutations	`GET / HT`~~`T`~~`P//1.1\r\n`	HAProxy-Squid
various method version combinations	`TRACE / HTTP/1.0\r\n`	Apache-ATS HAProxy-ATS Squid-ATS Varnish-ATS

Distorted Protocol. This category consists of mutation sets that replace one character in the protocol name with another, usually adding the character h to the beginning of the protocol, or changing the case of an existing letter. Table 6 shows a mutation that causes a discrepancy between ATS and Squid. ATS handles the mutation and parses the message body whereas Squid ignores the body.

Invalid Version. Mutation sets with this classification add a digit, replace a digit with another, or remove a digit from the beginning and adds one to the end. This category of mutations primarily involve digits, keeping the versions numeric, yet invalid.

Manipulated Termination. These mutations primarily add a space, tab, or CRLF before the CRLF that terminates the request line. The example in Table 6 triggers a discrepancy when Varnish or HAProxy is the entrypoint, and NGINX is the exitpoint. The entrypoint determines that the body is what directly follows from the double CRLF, whereas the exitpoint ignores the body completely.

Embedded Request Lines. This category includes mutation sets that insert a whole request line into the existing request line at various positions, including after the method and the protocol. The example in Table 6 has a request line inserted after the protocol.

Multiple Mutations. We classify mutation sets into this category when the individual mutations alone do not trigger a discrepancy, but together they do. These mutation sets mainly have two forms: 1) the method name is mutated while another CRLF is added next to the terminating CRLF, or 2) a character is deleted from the protocol name and a second slash is added after the protocol name.

Various Method Version Combinations. Sometimes inconsistent behavior is triggered by bringing various methods and versions together with no need for mutations. In the example in Table 6, ATS ignores the message body for TRACE requests, whereas Apache, HAProxy, Squid, and Varnish do not.

5.2.2 Request Headers Experiment. Figure 3 lists all server pairs and mutation categories affecting request headers. Table 7 presents examples as before; however, each example shows both the method

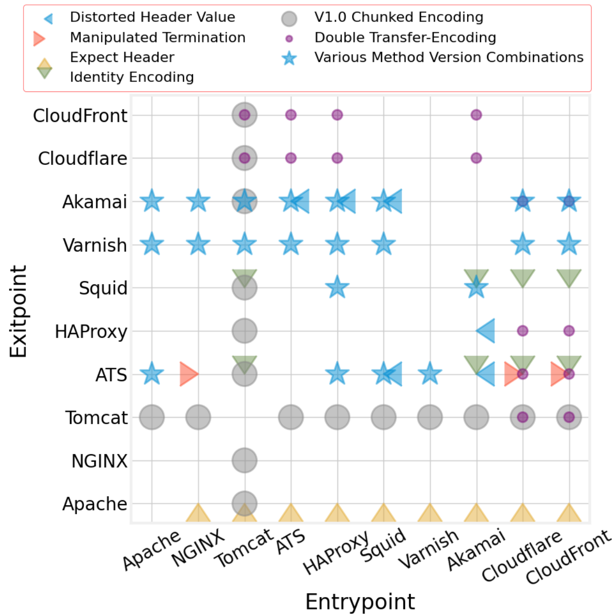


Figure 3: Request header mutation categories affecting server pairs.

name and the mutation, since successful mutations in this experiment vary from method to method.

Distorted Header Value. This category includes mutation sets that add specific characters, such as a vertical tab, new page, space, plus, and comma to the beginning and end of specific header values. The headers in this category are `Transfer-Encoding: chunked` and `Content-Length: LENGTH`. In Table 7, a comma is appended to the header value. Akamai ignores the request body, while Tomcat, HAProxy and ATS parse and forward it.

Manipulated Termination. Mutations in this category mainly insert a space or tab after the header-terminating `CRLF`, resulting in parsing discrepancies. In Table 7, a space is added after the `CRLF` following the first header value. As a result, ATS ignores the request body, whereas NGINX, Cloudflare and CloudFront do not.

Expect Header. We find that the `Expect` header is interpreted differently by Apache. When Apache receives a request with this header and its `100-continue` value, it ignores the body in the request as opposed to every other server we experimented with.

Identity Encoding. When a request has a `Transfer-Encoding` header with the `identity` value, Squid and ATS ignore the message body. Tomcat, Akamai, Cloudflare and CloudFront parse and forward the body.

V1.0 Chunked Encoding. Tomcat does not support chunked encoding in HTTP version 1.0. Thus, this causes an inconsistency between Tomcat and all other servers we experimented with. When a request has both `Transfer-Encoding` and `Content-Length` headers, all servers prefer the former, whereas Tomcat prefers the latter.

Double Transfer-Encoding. We observe an interesting behavior when a request has two Transfer-Encoding headers. When the first header has the value `identity` and the second `chunked`, Cloudflare and CloudFront use the `Content-Length` header, while Tomcat, HAProxy, and Akamai use `Transfer-Encoding` to dictate message body parsing.

Table 7: Examples for each request header mutation category.

Category	Method ; Request Header	Entrypoint-Exitpoint
distorted header value	`GET;Transfer-Encoding: chunked,\r\n`	Tomcat-Akamai ATS-Akamai HAProxy-Akamai
manipulated termination	`GET;Transfer-Encoding: chunked\r\n␣{Header}:{Value}\r\n`	NGINX-ATS Cloudflare-ATS CloudFront-ATS
expect header	`POST;Expect: 100-continue\r\n`	NGINX-Apache (truncated)
identity encoding	`POST;Transfer-Encoding: identity\r\n`	Cloudflare-Squid CloudFront-Squid (truncated)
v1.0 chunked encoding	`POST;Transfer-Encoding: chunked\r\n`	Apache-Tomcat (truncated)
double transfer-encoding	`POST;Transfer-Encoding: identity\r\n Transfer-Encoding: chunked\r\n`	Cloudflare-Tomcat CloudFront-Tomcat Cloudflare-Akamai (truncated)
various method version combinations	`OPTIONS-0.9; Content-Length:5\r\n`	HAProxy-Squid Akamai-Squid

Various Method Version Combinations. Similar to the request line experiment, various options for methods and versions defined in the input grammar are combined in different ways to generate our input requests. In this experiment, these combinations are also combined with various headers including `Transfer-Encoding`, and can cause discrepancies without a mutation.

5.2.3 Request Body Experiment. Figure 4 lists the server pairs that have discrepancies with mutation categories affecting the request body, and Table 8 shows examples.

Chunk-Size Chunk-Data Mismatch. These mutations add to or remove a character from chunk data to make its size different from what is claimed in the chunk size. This causes Akamai to process the request body using `Content-Length` and ignore `Transfer-Encoding`, while every other server (except Tomcat and Apache that give an error) continues to use `Transfer-Encoding`.

Manipulated Chunk-Size Termination. Mutations in this category modify the `CRLF` terminating the chunk size, and typically add a character such as a new page, semicolon, or space. This causes Akamai to make a different preference between `Transfer-Encoding` and `Content-Length` headers compared to the other servers.

Manipulated Chunk-Extension Termination. In this category, mutation sets remove a part of the newline which terminates the chunk extension. Usually, the carriage return character is deleted. Again, this causes Akamai to use the `Content-Length` header instead of the `Transfer-Encoding` header.

Manipulated Chunk-Data Termination. These mutations remove the terminating `CRLF`, partially or wholly from the chunk data part of the request. In the example shown in Table 8, the `CRLF` is completely removed at the end of the first chunk data.

Mangled Last-Chunk. Mutations in this category include removing one `CRLF` before the last chunk, inserting digits next to the chunk size in the last chunk (as seen in Table 8), or removing the entire last chunk itself. Unlike HAProxy, Squid, and CloudFront, Akamai does not treat the request body as chunked-encoded.

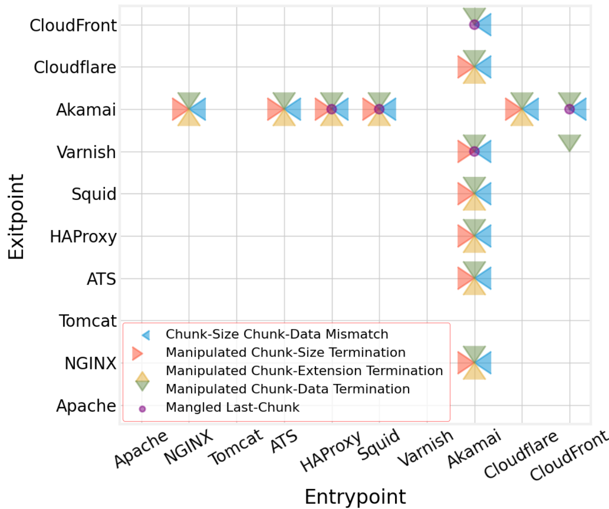


Figure 4: Request body mutation categories affecting server pairs.

5.3 Stage 3+ - Determining Discrepancy HRS Potential

Stage 2 yielded classified mutation sets that are reduced to representative examples, and the server pairs that have parsing discrepancies on said sets. We now determine if these parsing discrepancies can be used for HRS. For every unique server pair that appears in the results from Stage 2, we set up our lab to position them as entrypoint and exitpoint on path. We ensure that the connection between the two servers is persistent, as HRS requires this.

To understand if our mutated requests lend themselves to HRS, we craft a smuggler request as shown in Listing 12 for every mutation (Listing 12 shows a *Mangled Method* mutation). Immediately after we send the smuggler request, we send a benign request like Listing 13 on the same connection. If our smuggler request was successful, our payload (shown in Listing 14) will invalidate the valid request, and the exitpoint will return a `400 Bad Request` error message in a response to the benign request.

To ensure the error message did not come from the entrypoint, we note that each server has their own unique fingerprint in the HTML error page returned, and verify the exitpoint's fingerprint.

For each experiment, we now detail the categories of mutations that can be used for HRS, and discuss why they succeeded while the others failed. We note that for all but two specific cases, the payload format allowed with these HRS attack vectors is unrestricted. For the following, please reference Figure 5 for the server combinations and mutation categories that successfully carried an HRS payload, and Figure 6 for a breakdown of the server combinations and reasons mutations failed for them.

```
1 hEAD / HTTP/1.1
2 Host: example.com
3 Content-Length: 2
4
5 A␣
```

Listing 12: Smuggler request.

```
1 POST / HTTP/1.1
2 Host: example.com
3 Content-Length:5
4
5 AAAAA
```

Listing 13: Benign request.

```
1 A␣POST / HTTP/1.1
2 Host: example.com
3 Content-Length:5
4
5 AAAAA
```

Listing 14: Poisoned request.

Table 8: Examples for each request body mutation category.

Category	Request Body	Entrypoint-Exitpoint
chunk-size chunk-data mismatch	4\r\n~~B~~BBB \r\n0\r\n\r\n	Akamai-NGINX Akamai-Varnish (truncated)
manipulated chunk-size termination	4\t\nBBBB \r\n0\r\n\r\n	Cloudflare-Akamai Squid-Akamai (truncated)
manipulated chunk-extension termination	4;foo=bar~~\r~~\nBBBB \r\n0\r\n\r\n	Akamai-Cloudflare Akamai-ATS (truncated)
manipulated chunk-data termination	4\r\nBBBB~~\r\n~~4 \r\nBBBB \r\n0\r\n\r\n	CloudFront-Varnish Akamai-Varnish (truncated)
mangled last-chunk	4\r\nBBBB \r\n20\r\n\r\n	HAProxy-Akamai Squid-Akamai (truncated)

5.3.1 Request Line Mutations. Among the request line mutations, only two categories failed to carry out HRS, *Distorted Protocol*, and *Multiple Mutations*. We found that some servers normalized parts of the request line before forwarding them to the next server when they encountered our mutations, or just flat out closed the connection. We observed servers being particularly sensitive to invalid requests, which is very common in these two mutation categories. The successful categories contained less invalidating mutations, and thus proved more fruitful.

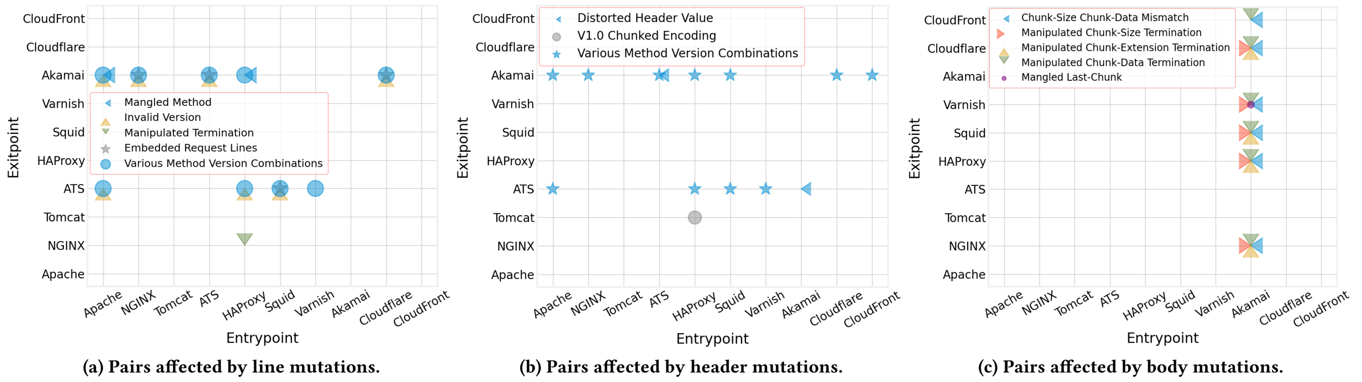
In one case, there was a restriction on the format of the smuggled content. Using a specific method version combination with Squid-Akamai, the entrypoint expected the request to be in chunked encoding. Therefore, the smuggled content had to follow the chunked format.

Unlike others, Varnish cleans its connection, preventing HRS. When Varnish receives a GET request with a body, even though it ignores the body, it does not leave the body in the connection.

5.3.2 Header Mutations. We observe that the categories *Manipulated Termination*, *Expect Header*, *Identity Encoding*, and *Double Transfer-Encoding* failed to work in any server combination. Similar to the request line mutations, mutations to the `Transfer-Encoding` and `Content-Length` headers often were not preserved. Servers typically re-wrote their own headers in place of our mutations based off of what they parsed, effectively stopping all HRS. For the `Expect Header` failure, that category only affected Apache, and Apache closes the connection after receiving these requests.

Similar to the request line experiment, only one case restricted the format of the smuggled content. When a request with a distorted header value was sent to ATS-Akamai, ATS required the smuggled content to be in chunked encoding.

5.3.3 Body Mutations. For failed HRS attempts using body mutations, the entrypoint re-wrote the mutated chunked body and therefore did not preserve the mutation. In all cases where Akamai was an entrypoint, HRS attempts succeeded except for Akamai-ATS.



(a) Pairs affected by line mutations. (b) Pairs affected by header mutations. (c) Pairs affected by body mutations.

Figure 5: Server pairs affected by request smuggling.

ATS ignored the message body part that follows the last chunk as expected, yet it did not leave that part in the connection.

5.3.4 Reasoning about Discrepancies. Server developers typically base their design decisions on official documents like HTTP RFCs. Unfortunately, RFCs cannot accommodate information about how to interpret every single iteration of a valid HTTP request. When confronted with gray areas, developers have to make implementation decisions that conform to the RFC, but are not explicit. HRS arises from this gray area. We illustrate this by providing examples from each part of the HTTP request in our experiments that successfully lead to HRS.

Request Line Mutations. In Listing 15, the version in the request line is mutated. Despite this mutation, ATS still forwarded the message body to an exitpoint. In fact, we observed that ATS forwarded the message body for all GET requests with any decimal version number (i.e., 99.99). Conversely, this mutation caused Squid to ignore the message body, presumably because it could not decide what the version is. Squid's body parsing behavior is dependent on the version, as it ignores request bodies in version 0.9, yet accepts them in newer versions.

Header Mutations. Listing 16 shows a request which uses chunked encoding with HTTP version 1.0, even though the chunked encoding was introduced to the protocol with version 1.1. Despite this fact, HAProxy supports chunked encoding in HTTP version 1.0. Tomcat ignores this header, presumably because it assumes that requests with version 1.0 cannot use chunked encoding. As a result, the body of the request shown in Listing 16 is ignored by Tomcat, while it is parsed by HAProxy.

Body Mutations. Listing 17 terminates the chunk extension with an LF rather than a CRLF. NGINX treats the LF the same as a CRLF and parses the message body as chunked. However, Akamai handles this error by defaulting to Content-Length instead of Transfer-Encoding to parse the message body.

```
1 GET / HTTP/.11
2 Host: example.com
3 Content-Length: 5
4
5 AAAAA
```

Listing 15: Line mutation.

```
1 POST / HTTP/1.0
2 Host: example.com
3 Transfer-Encoding: chunked
4
5 4\r\nBBBB\r\n0\r\n\r\n
```

Listing 16: Header and version.

```
1 POST / HTTP/1.1
2 Host: example.com
3 Content-Length: 5
4 Transfer-Encoding: chunked
5
6 4;foo=bar\r\nBBBB\r\n0\r\n\r\n
```

Listing 17: Body mutation.

6 IMPACT ASSESSMENT

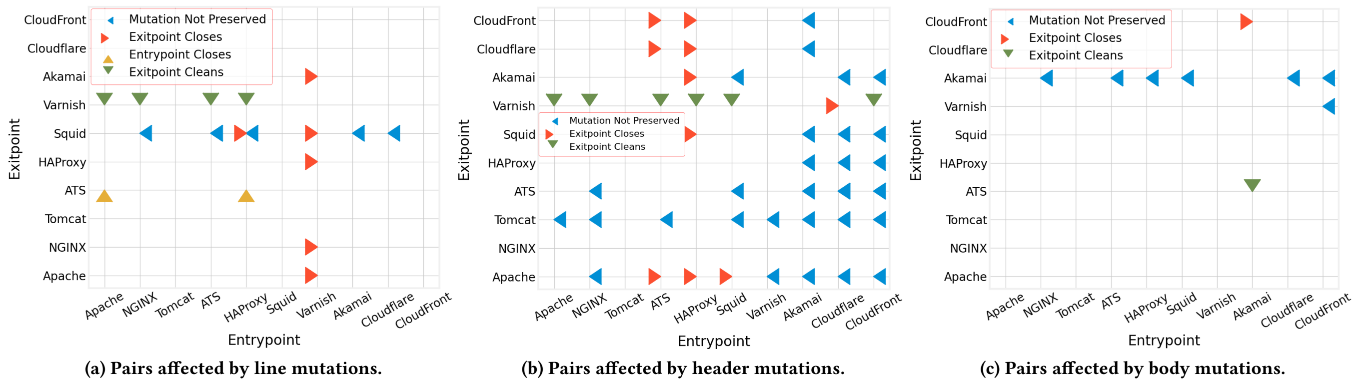
So far we have answered our core research questions and systematically confirmed that HTTP processing discrepancies lead to novel HRS vulnerabilities. Next, we present a set of empirical experiments to reaffirm that these vulnerabilities in fact have practical impact, and compare our work to existing HRS testing tools.

6.1 Demonstrating Possible Attacks

The damage caused by an HRS attack depends on the web application and data exposed by the vulnerable server pair. In this paper, we do not quantify such damage. Instead, we explore the discrepancies between HTTP processors and quantify the HRS attack surface independent of the outcome of any particular exploitation scenario.

Regardless, to demonstrate end-to-end attacks in a proof-of-concept, we set up an environment with a vulnerable application behind a server pair with actionable discrepancies. Specifically, we abused a chunked body parsing discrepancy between Akamai-NGINX. We configured NGINX to serve OWASP Mutillidae [8], a deliberately vulnerable web application for security training.

We tested three scenarios using HRS: 1) bypassing header rewriting, 2) hijacking requests, and 3) delivering attack payloads. In (1), we smuggled a request with an arbitrary X-Forwarded-For value, evading re-writing by the entrypoint. This is critical, since this header is often used in authentication and authorization schemes [26]. In (2) our smuggler request payload constructed a poisoned request to an attacker-controlled destination, leaking a random user's request content including session cookies. Finally, in (3) we smuggled a request which exploits a reflected XSS vulnerability at the destination to have the XSS response delivered to a random user. Videos of these attacks in action are available on the authors' websites.



(a) Pairs affected by line mutations. **(b) Pairs affected by header mutations.** **(c) Pairs affected by body mutations.**

Figure 6: Failed request smuggling reasons for each server pair.

6.2 Estimating Server Combinations

At a first glance, some of the server pairs we test may seem unrealistic for a real-life deployment scenario. However, the Internet has become a complex ecosystem (and patchwork) of middle-boxes and cloud services, where any given request may be processed by not two, but many servers. CDN deployments are prevalent [2], and multi-CDN chaining is practical [21]. These services themselves may depend on popular proxies, web caches, and web servers (e.g., Fastly uses Varnish [25], Cloudflare uses NGINX [30]).

To illustrate our point, we conducted an experiment with the top 10K sites of the Tranco list [34], exploring what server technologies are deployed in the wild. Namely, we visited the homepage of each site and collected the HTTP response headers. We simultaneously ran route traces for IP addresses seen on path, and performed WHOIS lookups for each. We then searched through this data for known header & value combinations that fingerprint the technologies, and for explicit service identifiers inside HTTP responses, WHOIS data, and email domains. This process resulted in a set of potential server technologies used for each site.

This methodology has limitations. There is no known way to reliably detect proxy services via traffic analysis, particularly because many services allow operators to strip the identifying headers to prevent fingerprinting. Furthermore, different endpoints on a site may use distinct proxy technologies, requiring a comprehensive crawl of each site for an accurate analysis. Finally, a blackbox detection methodology cannot determine the *placement order* of the servers, but only the fact that they are used in some combination. These are non-trivial challenges that we do not tackle in this work.

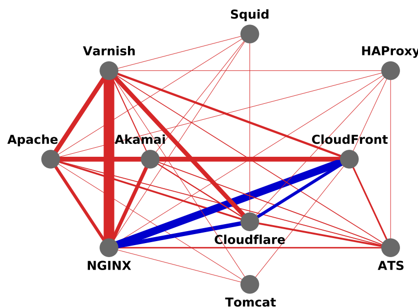


Figure 7: HTTP processors paired in the wild. This is an unordered graph, showing pairwise combinations. Red edges indicate pairs that exhibit processing discrepancies, blue edges represent pairs that do not. Edge thickness corresponds to the incidence of pairs.

Figure 7 summarizes our results, showing pairwise server combinations we observed, where the edge weights represent the incidence. We find that approximately **17%** of the sites among the top 10K use technologies that we have identified discrepancies between. We observe an average of 2.8 technologies, with a median of 3, and a maximum of 5 per site.

Given the aforementioned limitations, these results represent a loose lower bound on the incidence of server pairings. Yet, they show two important points. First, out of the 45 possible combinations of the 10 servers in our setup, **36** are used in the wild. Second, seemingly unrealistic combinations are viable, and chained CDNs are more frequent than other combinations. We conclude that making presumptions about what server combinations are viable in the wild is counterproductive when exploring HRS and similar systems-level hazards. Processing discrepancies can crop up on any technology, and therefore, all combinations are worth investigating.

6.3 Comparing T-Reqs to Existing Tools

James Kettle's Burp Suite extension *HTTP Request Smuggler* [36] and Evan Custodio's Python script *smuggler* [7] are the state-of-the-art tools used when testing sites for HRS.

Foremost, both of these operate on fundamentally different targets and serve a different purpose than T-Reqs. In particular, these tools are designed for penetration testing of a given *target site*, treating the entire web deployment as a *blackbox*, and testing it for a set of *known* `Content-Length` and `Transfer-Encoding` header manipulation attacks presented in the authors' respective works.

In contrast, T-Reqs is not designed to test live sites against known exploits. T-Reqs tests *pairwise combinations of HTTP processors* in a lab environment, and exercises each component individually, in a *greybox* manner. It is designed to *discover novel HRS vectors*, as opposed to testing a real-life deployment for known attacks.

Therefore, these tools are not substitutes for each other. T-Reqs serves to discover novel HRS payloads that the others are bound to miss given their limited scope. That does not diminish the value

of previous work. In fact, these tools are suited to work in tandem, where T-Reqs finds novel exploits, which can then be added to Burp or smuggler to automate their use in penetration testing.

With that in mind, we next describe how these existing tools work, and present an empirical study demonstrating T-Reqs's ability to create new knowledge for HRS research.

HRS Detection in Existing Tools. Existing tools use the detection methodology proposed by Kettle [15]. First, the request in Listing 18 is sent to a target to determine if it is affected by a CL.TE discrepancy, meaning the entrypoint processes the `Content-Length` header while the exitpoint prefers `Transfer-Encoding`. If the target is vulnerable, the entrypoint will forward the first four bytes (i.e., chunk size `1` and chunk data `Z`), and the exitpoint will timeout waiting for the next chunk which will never arrive. This timeout delay flags the site as vulnerable.

If the CL.TE test fails, the request in Listing 19 is sent to check for a TE.CL discrepancy. If there is a vulnerability, a similar delay can be observed: The entrypoint forwards the body without the byte `X` that comes after the last chunk, and the exitpoint receives less content than what `Content-Length` indicates, therefore timing out while waiting for one additional byte.

As also emphasized by Kettle, the order of the above two checks is important. The TE.CL test should only be performed after confirming the absence of a CL.TE discrepancy. Otherwise the TE.CL request could poison the connection with the byte `X` in CL.TE-impacted targets, launching an attack on arbitrary Internet users.

All in all, both tools iterate through numerous mutations in the `Transfer-Encoding` header of the requests in Listings 18 and 19, and check the target for CL.TE and TE.CL discrepancies using the above methodology.

Safety of the Detection Methodology. Unfortunately, we have empirically confirmed in our tests that neither tool is currently safe to run on real-world targets.

Firstly, Custodio's smuggler does not follow the above order of requests which is critical for preventing inadvertent attacks. More interestingly, even though Burp follows the protocol, we have determined that a false assumption made in the detection methodology makes it hazardous for running real-world experiments. Namely, the methodology assumes that, when the target is affected by a TE.CL discrepancy, the entrypoint will treat the byte `Q` in Listing 18 as an invalid chunk size and return an error. That will prevent accidentally poisoning the connection during the CL.TE check. This assumption does not hold; Akamai servers do not return an error and forward the request as if `Q` was a proper chunk size.

```
1 POST / HTTP/1.1
2 Host: example.com
3 Transfer-Encoding: chunked
4 Content-Length: 4
5
6 1
7 Z
8 Q
```

Listing 18: CL.TE test request.

```
1 POST / HTTP/1.1
2 Host: example.com
3 Transfer-Encoding: chunked
4 Content-Length: 6
5
6 0
7
8 X
```

Listing 19: TE.CL test request.

While this could be an oversight in design, it is also possible that Akamai's behavior has changed in the two years since the publication of Kettle's work. We conclude that testing the existing tools in a large-scale experiment is not safe without explicit penetration-testing agreements. Instead, we conducted our comparative study in the same test environment used for the experiments with T-Reqs.

Empirical Comparison. We have already presented T-Reqs's output in Section 5. To compare those results with the detections from the two existing tools, we ran them in the same experimental setup. However, there were two cases we could not test. It is not possible to run Tomcat in a reverse-proxy mode, and therefore we did not test pairs having Tomcat as the entrypoint. Additionally, we were unable to set up a Cloudflare-CloudFront pair, because the `Host` header rewriting capability necessary for that deployment is only available for Cloudflare's Enterprise plan customers [3]. This limitation was not a factor during our core experiments with T-Reqs; we had found no discrepancies for this CDN pair, and therefore we did not need to attempt a deployment for exploitability testing.

Unsurprisingly, T-Reqs was the only tool that found the request line and request body related attacks described in Sections 5.2.1 and 5.2.3. The others missed this category of attacks entirely, because they are only designed to test for the Content-Length and `Transfer-Encoding` header manipulation attacks. One exception was that Burp found an exploitable Mangled Last-Chunk discrepancy on Akamai-ATS. We manually verified that this was an *accidental* true positive, as the request template Burp uses unintentionally had the trigger for this discrepancy built into the chunked body – no mutations were necessary for this finding. T-Reqs also detected the same vulnerability through body mutations.

An unanticipated outcome was neither smuggler nor Burp flagged any request header attacks either, even though they are designed to test for those. We reviewed the source code for both tools and verified that the header modifications they use[1,2] indeed do not lead to any exploitable HRS vulnerabilities today on the 10 technologies in our setup. We attribute this to the fact that these tools repeat known exploits, and the server vendors have already had two years to implement mitigations since their disclosure.

In summary, Burp detected one HRS vulnerability and smuggler detected none, whereas T-Reqs yielded all the findings we presented in Section 5. We conclude that T-Reqs indeed fulfills its role of finding novel HRS vectors left out of scope in previous work.

6.4 Testing HRS in the Wild

Due to the aforementioned safety issues, the state-of-the-art HRS detection methodology should not be used outside of specific targets that explicitly allow external testing of their sites. Designing a safe detection scheme likely requires whitebox analysis of different HTTP processors to ensure that request queues are not inadvertently poisoned. Conducting a large-scale HRS measurement in the wild safely is a promising future research direction.

Here, we instead present a preliminary experiment testing real-world deployments that has Akamai as entrypoint, and only with a specific HRS payload. This is not an arbitrary choice. Prior to

[1] https://github.com/PortSwigger/http-request-smuggler/blob/master/src/burp/DesyncBox.java

[2] https://github.com/defparam/smuggler/blob/master/configs/exhaustive.py

running the experiment, we carefully studied Akamai's behavior and crafted an HRS detection payload, based on Kettle's approach, that is guaranteed to be safe for this particular experiment.

Specifically, we used a body mutation in the *Chunk-Size Chunk-Data Mismatch* category. Recall from Section 5.2.3 and Figure 4 that this category impacts server pairs that has Akamai as entrypoint, and always makes Akamai prefer the `Content-Length` header. That enables us to test sites for this novel HRS vulnerability while actively avoiding the unsafe situation, where an Akamai server prefers the `Transfer-Encoding` but allows an invalid chunk size through.

To make sure that Akamai is the entrypoint in the target, we sent a TRACE request with the `Max-Forwards:0` header to force the request to stop at the first HTTP server on path even if it does not support the TRACE method. Out of 861 Akamai customers from Tranco Top 10K identified previously in Section 6.2, we were able to confirm 367 had Akamai as the entrypoint.

We tested these 367 domains by sending our new HRS detection request, and flagged sites as vulnerable if they did not respond within 5 seconds (i.e., the default threshold used in existing tools). Out of the 367 domains tested, we found **23** to be vulnerable. These included a high-profile financial institution, online retailers, and other technology, news, and entertainment sites.

This experiment is decidedly narrow in scope. However, it successfully demonstrates that real-world deployments are exposed to HRS vulnerabilities we discovered with T-Reqs, despite the many hidden layers of complexity present in the wild that we could not account for in a lab environment. Designing a generalizable detection methodology and enabling a full-fledged measurement study is the logical next step for characterizing the impact of HRS.

7 DISCUSSION

As we conclude, we underscore considerations for the correct interpretation of our results.

Limitations. While this paper represents the most holistic investigation into HRS to date, it is by no means exhaustive. For example, we leave non-standard HTTP headers out of scope. There are further restrictions we impose on our methodology to make the experimentation and analysis feasible, such as limiting the maximum number of mutations for an input, and mutating request components in isolation in their respective experiment runs.

Nonetheless, our approach provides sufficient evidence to address our research questions, showing that there are indeed vast and unexplored opportunities for crafting HRS attacks – and that the security community must stay alert. We make T-Reqs available in the hopes that our fellow researchers will improve on it and make even more exciting discoveries.

Real-World Considerations. In our experiments we test all servers in their default configurations. While all of the exploits we find are real and practical, configurations will vary considerably in the wild. What is more, servers that we flag as impacted in this experiment may be deployed behind other proxies (e.g., a web application firewall or load balancer between the entry and exitpoints), which intentionally or inadvertently strip out exploit payloads. On the flip side, non-default configurations may also expose dangerous discrepancies that we were not able to catch in our study.

We stress that our findings should not be taken at face value. This work is not intended to be a prescribed list of vulnerabilities and their mitigations. We provide strong indicators for hazardous server combinations and demonstrate the severity of the issue, so that system owners can vet their environments.

Blame Nobody. We reiterate that HRS is a system interaction problem. Individual components of the system are not necessarily flawed, but their hazardous combination results in a vulnerability that is not trivial to detect or mitigate. This implies that technology vendors are not always in a position to correct these flaws on their own; an ideal HTTP processor that is strictly RFC compliant, using a formally-verified parser, and implemented by the best developers on the planet may still get caught in an HRS attack when combined with a different technology that interprets a request differently. Unfortunately, the reality is even more complicated, where RFCs are ambiguous, bugs are inevitable, and powerful mechanisms to rewrite HTTP requests are desirable and necessary features. In this complex ecosystem, predicting, detecting, mitigating, or fixing HRS is a non-trivial, open research problem.

While the results we present in this paper may appear to show that some technologies and vendors are better than others, that is an incorrect interpretation of our results. Our findings do not represent a meaningful security comparison between the tested servers, and they should not be taken out of context to pit one technology against another. Once again, this work presents a scientific, systematic methodology to identify HRS, and uncover previously unexplored venues for attacks, so that the developers and users of these technologies are better equipped to understand the implications of the issue, and vet their own systems.

8 CONCLUSION

This paper is the first systematic exploration of HRS attacks. Revisiting our research questions from Section 3, we proposed an experiment infrastructure and methodology for efficient discovery of attacks (Q1), developed a novel grammar-based differential fuzzer to test all components of an HTTP request for viable exploits (Q2), provided insights into previously unknown success (and failure) modes enabled by our exploits (Q3), and finally documented hazardous combinations of popular servers (Q4). Our findings collectively show that HRS may yet evolve into an even more complex attack, and it is paramount that the security community tackle the open research questions in the areas of detection and defense.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant CNS-1703454 and by Secure Business Austria.

REFERENCES

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *The Network and Distributed System Security Symposium.*

[2] BuiltWith. [n.d.]. BuiltWith Technology Lookup. https://trends.builtwith.com/CDN/Content-Delivery-Network.

[3] Cloudflare Help Center. 2021. Using Page Rules to Re-Write Host Headers. https://support.cloudflare.com/hc/en-us/articles/206652947-Using-Page-Rules-to-Re-Write-Host-Headers.

[4] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *ACM Conference on Computer and Communications Security.*

[5] Evan Custodio. 2019. Mass account takeovers using HTTP Request Smuggling on https://slackb.com/ to steal session cookies. https://hackerone.com/reports/737140.
[6] Evan Custodio. 2020. Practical Attacks Using HTTP Request Smuggling by @defparam. NahamCon. https://www.youtube.com/watch?v=3tpnuzFLU8g.
[7] Evan Custodio. 2020. Smuggler. https://github.com/defparam/smuggler.
[8] Jeremy Druin. 2021. OWASP Mutillidae II. https://github.com/webpwnized/mutillidae.
[9] Roy Fielding, James Gettys, Jeff Mogul, Henrik Frystyk, and Tim Berners-Lee. 1997. Hypertext Transfer Protocol – HTTP/1.1. https://tools.ietf.org/html/rfc2068.
[10] Roy Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. https://tools.ietf.org/html/rfc7230.
[11] Roy Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231.
[12] Omer Gil. 2017. Web Cache Deception Attack. Black Hat USA. https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.
[13] Omer Gil. 2017. Web Cache Deception Attack. https://omergil.blogspot.com/2017/02/web-cache-deception-attack.html.
[14] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (2020).
[15] James Kettle. 2019. HTTP Desync Attacks: Request Smuggling Reborn. PortSwigger Web Security Blog. https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.
[16] James Kettle. 2019. Password theft login.newrelic.com via Request Smuggling. HackerOne. https://hackerone.com/reports/498052.
[17] James Kettle. 2019. Stored XSS on https://paypal.com/signin via cache poisoning. HackerOne. https://hackerone.com/reports/488147.
[18] James Kettle. 2021. HTTP/2: The Sequel is Always Worse. Black Hat USA. https://www.blackhat.com/us-21/briefings/schedule/#http2-the-sequel-is-always-worse-22668.
[19] Amit Klein. 2020. HTTP Request Smuggling in 2020 – New Variants, New Defenses and New Challenge. Black Hat USA. https://www.blackhat.com/us-20/briefings/schedule/#http-request-smuggling-in---new-variants-new-defenses-and-new-challenges-20019.
[20] Graham Klyne. 2021. Message Headers. https://www.iana.org/assignments/message-headers/message-headers.xhtml.
[21] Dima Kumets. 2019. 8 best practices for multi-CDN implementations. https://www.fastly.com/blog/best-practices-multi-cdn-implementations.
[22] Emil Lerner. 2021. http2smugl. https://github.com/neex/http2smugl.
[23] Regis Leroy. 2016. Hiding Wookiees in HTTP: HTTP smuggling. DEF CON. https://www.youtube.com/watch?v=dVU9i5PsMPY.
[24] Chaim Linhart, Amit Klein, Ronen Heled, and Steve Orrin. 2005. HTTP Request Smuggling. Watchfire. https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.
[25] Anna MacLachlan. 2015. The benefits of using Varnish. https://www.fastly.com/blog/benefits-using-varnish.
[26] Lori MacVittie. 2017. Security Rule Zero: A Warning about X-Forwarded-For. https://www.f5.com/company/blog/security-rule-zero-a-warning-about-x-forwarded-for.
[27] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998).
[28] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. 2020. Cached and Confused: Web Cache Deception in the Wild. In *USENIX Security Symposium*.
[29] Seyed Ali Mirheidari, Matteo Golinelli, Kaan Onarlioglu, Engin Kirda, and Bruno Crispo. 2022. Web Cache Deception Escalates!. In *USENIX Security Symposium*.
[30] NGINX. [n.d.]. Cloudflare boosts performance and stability for its millions of websites with NGINX. https://www.nginx.com/success-stories/cloudflare-boosts-performance-stability-millions-websites-with-nginx/.
[31] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2019. Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack. In *ACM Conference on Computer and Communications Security*.
[32] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DifFuzz: Differential Fuzzing for Side-channel Analysis. In *IEEE/ACM International Conference on Software Engineering*.
[33] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. Nezha: Efficient Domain-Independent Differential Testing. In *IEEE Security & Privacy*.
[34] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2021. Tranco – A Research-Oriented Top Sites Ranking Hardened Against Manipulation. https://tranco-list.eu/.
[35] PortSwigger. [n.d.]. Exploiting HTTP request smuggling vulnerabilities. https://portswigger.net/web-security/request-smuggling/exploiting.
[36] PortSwigger. 2019. HTTP Request Smuggler. https://github.com/PortSwigger/http-request-smuggler.

Table 9: Tested HTTP servers and versions.

HTTP Server	Tested Version
Apache	2.4.46
NGINX	1.18.0
Tomcat	9.0.40
Apache Traffic Server (ATS)	8.1.1
HAProxy	2.3.1
Squid	4.13
Varnish	6.0.7
Akamai	N/A
Cloudflare	N/A
CloudFront	N/A

[37] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. 2017. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *IEEE Security & Privacy*.
[38] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/.

A TESTED HTTP PROCESSORS

We experiment with 10 popular HTTP processors in this work, using the latest stable versions available at the time of writing. Table 9 shows specific versions of each technology, with the exception of CDNs which do not have public release labels.

B FORMAL MUTATION DEFINITIONS

T-Reqs uses string and tree mutations to generate HTTP requests. Here, we provide formal definitions for mutation operations.

String mutation operations.
Given:

- N is a set of all non-terminal symbols in a CFG.
- T is a set of all terminal symbols in a CFG.
- B is a predefined character pool.

Let s be a string mutable symbol represented by a CFG as

$$s ::= t_1|t_2|...|t_k$$

where $s \in N$, and $t_i \in T$.

Given the expansion of $s \rightarrow t_1$, where t_1 is a sequence of n characters $c_1c_2...c_n$, a string mutator is represented as a function $f(t_1, op, j, b)$ where $op \in \{$*delete-char, replace-char, insert-char*$\}$, $1 \leq j \leq n$, and $b \in B$:

$$f(c_1...c_n, op, j, b) = \begin{cases} c_1...c_{j-1}c_{j+1}...c_n, & \text{if } op = \textit{delete-char} \\ c_1...c_{j-1}bc_{j+1}...c_n, & \text{if } op = \textit{replace-char} \\ c_1...c_jbc_{j+1}...c_n, & \text{if } op = \textit{insert-char} \end{cases}$$

Tree mutation operations.
Given:

- N is a set of all non-terminal symbols in a CFG.
- T is a set of all terminal symbols in a CFG.
- H is a predefined symbol pool.

Let s be a tree mutable symbol which is represented by a CFG as

$$s ::= \langle n_1 \rangle ... \langle n_k \rangle \mid \langle n_l \rangle ... \langle n_m \rangle \mid ...$$

where $s \in N$, $n_i \in N$ for any $1 \leq i \leq k$ and $l \leq i \leq m$. Given the expansion of $s \rightarrow \langle n_1 \rangle ... \langle n_k \rangle$, a tree mutator is represented as a

function of the symbol s, an operation op, a sequence position j, and a symbol $<h>$:

$$g(<n_1>...<n_k>, op, j, <h>) = \begin{cases} <n_1>...<n_{j-1}><n_{j+1}>...<n_k>, & \\ \qquad\qquad \text{if } op = \mathit{delete\text{-}elem} & \\ <n_1>...<n_{j-1}><h><n_{j+1}>...<n_k>, & \\ \qquad\qquad \text{if } op = \mathit{replace\text{-}elem} & \\ <n_1>...<n_j><h><n_{j+1}>...<n_k>, & \\ \qquad\qquad \text{if } op = \mathit{insert\text{-}elem} & \end{cases}$$

where op is the operation type, $1 \leq j \leq k$, and $h \in H$.

C FULL GRAMMAR FOR THE REQUEST HEADERS EXPERIMENT

The request headers experiment tests all standard HTTP request headers together with numerous valid values. Listing 20 shows the full grammar where an expansion for every 67 header is defined. In addition, we present expansions for sub-elements of each header.

```
<start> ::= <request>
<request> ::= <method-name><request-uri><http-version><base><entity-size-header>
↪  <some-header><some-header><body>
<request-uri> ::= " /_URI_ "
<http-version> ::= "HTTP/0.9" | "HTTP/1.0" | "HTTP/1.1"
<method-name> ::= "GET" | "HEAD" | "POST" | "PUT" | "DELETE" | "CONNECT" |
↪  "OPTIONS" | "TRACE"
<base> ::= "\r\nHost: _HOST_\r\nConnection:close\r\nX-Request-ID:
↪  _REQUEST_ID_\r\n"
<entity-size-header> ::= <content-length> | <chunked-transfer-encoding> |
↪  <content-length><chunked-transfer-encoding> |
↪  <chunked-transfer-encoding><content-length>
<some-header> ::= <accept> | <accept-charset> | <accept-encoding> |
↪  <accept-language> | <accept-ranges> | <allow> | <alpn> | <alt-used> |
↪  <authorization> | <cache-control> | <caldav-timezones> | <cdn-loop> |
↪  <content-encoding> | <content-language> | <content-length> |
↪  <content-location> | <cookie> | <date> | <depth> | <destination> |
↪  <early-data> | <expect> | <expires> | <forwarded> | <from> | <http2-settings>
↪  | <if> | <if-match> | <if-modified-since> | <if-none-match> | <if-range> |
↪  <if-schedule-tag-match> | <if-unmodified-since> | <link> | <max-forwards> |
↪  <mime-version> | <odata-isolation> | <odata-maxversion> | <odata-version> |
↪  <ordering-type> | <origin> | <oscore> | <overwrite> | <position> | <pragma> |
↪  <prefer> | <proxy-authorization> | <range> | <referer> | <schedule-reply> |
↪  <sec-token-binding> | <sec-websocket-accept> | <sec-websocket-extensions> |
↪  <sec-websocket-key> | <sec-websocket-protocol> | <sec-websocket-version> |
↪  <slug> | <te> | <timeout> | <topic> | <trailer> | <transfer-encoding> | <ttl>
↪  | <urgency> | <upgrade> | <user-agent> | <via>
<newline> ::= "\r\n"
<body> ::=
↪  "\r\nA\r\nBBBBBBBBBB\r\n0\r\n\r\nBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
↪  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
<comma> ::= ","
<colon> ::= ":"
<semicolon> ::= ";"
<space> ::= " "
<start-tag> ::= "<"
<end-tag> ::= ">"
<start-parenthesis> ::= "("
<end-parenthesis> ::= ")"
<equals> ::= "="
<boolean> ::= "T" | "F"
<quality> ::= "q=1.0" | "q=0.0"
<chunked-transfer-encoding> ::=
↪  <transfer-encoding-header-name><colon><chunked-encoding><newline>
<chunked-encoding> : "chunked"
<accept> ::= <accept-header-name><colon><accept-types><newline>
<accept-header-name> ::= "Accept"
<accept-types> ::= <accept-type> | <accept-type><comma><accept-type>
<accept-type> ::= <mime-type-subtype><semicolon><quality>
<mime-type-subtype> ::= "*/*" | "application/octet-stream" | "application/pdf" |
↪  "application/pkcs8" | "application/zip" | "audio/mpeg" | "audio/vorbis" |
↪  "audio/example" | "font/woff" | "font/ttf" | "font/otf" | "image/jpeg" |
↪  "image/png" | "image/svg+xml" | "model/3mf" | "text/html" | "video/mp4"
<accept-charset> ::= <accept-charset-header-name><colon><charsets><newline>
<accept-charset-header-name> ::= "Accept-Charset"
<charsets> ::= <charset> | <charset><comma><charset>
<charset> ::= <charset-name><semicolon><quality>
<charset-name> ::= "utf-16" | "utf-16BE" | "utf-32" | "utf-32BE" | "us-ascii" |
↪  "iso-8859-1" | "utf-7" | "utf-8"
<accept-encoding> ::= <accept-encoding-header-name><colon><encodings><newline>
<accept-encoding-header-name> ::= "Accept-Encoding"
<encodings> ::= <encoding> | <encoding><comma><encoding>
<encoding> ::= <encoding-name><semicolon><quality>
<encoding-name> ::= "gzip" | "compress" | "deflate" | "br" | "identity" |
↪  "chunked"
<accept-language> ::= <accept-language-header-name><colon><languages><newline>
<accept-language-header-name> ::= "Accept-Language"
<languages> ::= <language> | <language><comma><language>
<language> ::= <language-name><semicolon><quality>
<language-name> ::= "fr" | "en" | "de"
<accept-ranges> ::= <accept-ranges-header-name><colon><range-unit><newline>
<accept-ranges-header-name> ::= "Accept-Ranges"
<range-unit> ::= "bytes" | "none"
<allow> ::= <allow-header-name><colon><method-names><newline>
<allow-header-name> ::= "Allow"
<method-names> ::= <method-name> | <method-name><comma><method-name>
<alpn> ::= <alpn-header-name><colon><protocol-ids><newline>
<alpn-header-name> ::= "ALPN"
<protocol-ids> ::= <protocol-id> | <protocol-id><comma><protocol-id>
<protocol-id> ::= "http%2F1.1" | "h2"
<alt-used> ::= <alt-used-header-name><colon><alt-svc><newline>
<alt-used-header-name> ::= "Alt-Used"
<alt-svc> : "alternate.example.net"
<authorization> ::=
↪  <authorization-header-name><colon><auth-scheme><space><creds><newline>
<authorization-header-name> ::= "Authorization"
<auth-scheme> : "Basic" | "Bearer","Digest","HOBA","Mutual","Negotiate","OAuth",
↪  "SCRAM-SHA-1","SCRAM-SHA-256","vapid"
<creds> ::= "123456" | "YWxhZGRpbjpvcGVuc2VzYW1l"
<cache-control> ::=
↪  <cache-control-header-name><colon><cache-directives><newline>
<cache-control-header-name> ::= "Cache-Control"
<cache-directives> ::= <cache-directive> |
↪  <cache-directive><comma><cache-directive>
<cache-directive> : "max-age=5" | "max-stale=5" | "min-fresh=5" | "no-cache" |
↪  "no-store" | "no-transform" | "only-if-cached"
<caldav-timezones> ::= <caldav-timezones-header-name><colon><boolean><newline>
<caldav-timezones-header-name> ::= "CalDav-Timezones"
<cdn-loop> ::= <cdn-loop-header-name><colon><cdn-infos><newline>
<cdn-loop-header-name> ::= "CDN-Loop"
<cdn-infos> ::= <cdn-info> | <cdn-info><comma><cdn-info>
<cdn-info> : "foo123.foocdn.example" | "barcdn.example; trace='abcdef'" |
↪  "AnotherCDN; abc=123; def='456'"
<content-encoding> ::=
↪  <content-encoding-header-name><colon><transfer-encodings><newline>
<content-encoding-header-name> ::= "Content-Encoding"
<content-language> ::= <content-language-header-name><colon><languages><newline>
<content-language-header-name> ::= "Content-Language"
<content-length> ::=
↪  <content-length-header-name><colon><content-length-value><newline>
<content-length-header-name> ::= "Content-Length"
<content-length-value> ::= "40" | "60" | "80"
<content-location> ::=
↪  <content-location-header-name><colon><content-location-value><newline>
<content-location-header-name> ::= "Content-Location"
<content-location-value> ::= <absolute-uri> | <relative-uri>
<absolute-uri> : "http://example.com/example"
<relative-uri> ::= "/example"
<cookie> ::= <cookie-header-name><colon><cookie-value><newline>
<cookie-header-name> ::= "Cookie"
<cookie-value> : "SID=31d4d96e407aad42" | "PHPSESSID=298zf09hf012fh2;
↪  csrftoken=u32t4o3tb3gg43; _gat=1"
<date> ::= <date-header-name><colon><date-value><newline>
<date-header-name> ::= "Date"
<date-value> : "Sun, 06 Nov 1994 08:49:37 GMT" | "Sun, 06 Nov 2094 08:49:37 GMT"
<depth> ::= <depth-header-name><colon><depth-value><newline>
<depth-header-name> ::= "Depth"
<depth-value> : "0" | "1" | "infinity"
<destination> ::= <destination-header-name><colon><absolute-uri><newline>
<destination-header-name> ::= "Destination"
<early-data> ::= <early-data-header-name><colon><early-data-value><newline>
<early-data-header-name> ::= "Early-Data"
<early-data-value> : "1"
<expect> ::= <expect-header-name><colon><expect-value><newline>
<expect-header-name> ::= "Expect"
<expect-value> : "100-continue"
<expires> ::= <expires-header-name><colon><date-value><newline>
<expires-header-name> ::= "Expires"
<forwarded> ::= <forwarded-header-name><colon><by><space><absolute-uri><newline>
<forwarded-header-name> ::= "Forwarded"
<by> ::= "by"
<from> ::= <from-header-name><colon><mailbox><newline>
<from-header-name> ::= "From"
<mailbox> ::= "webmaster@w3.org"
```

```
<http2-settings> ::= <http2-settings-header-name><colon><setting><newline>
<http2-settings-header-name> ::= "HTTP2-Settings"
<setting> ::= "AAMAAABkAARAAAAAAAIAAAAA"
<if> ::= <if-header-name><colon><tag-list><newline>
<if-header-name> ::= "If"
<urn-tag> ::= <start-tag><urn-value><end-tag>
<urn-value> ::= "urn:uuid:181d4fae-7d8c-11d0-a765-00a0c91e6bf2" |
↪  "urn:uuid:58f202ac-22cf-11d1-b12d-002035b29092"
<tag-list> : <start-parenthesis><urn-tag><end-parenthesis>
<if-match> ::= <if-match-header-name><colon><entity-tags><newline>
<if-match-header-name> ::= "If-Match"
<entity-tags> ::= <entity-tag> | <entity-tag><comma><entity-tag>
<entity-tag> : "*" | "'xyzzy'"
<if-modified-since> ::=
↪  <if-modified-since-header-name><colon><date-value><newline>
<if-modified-since-header-name> ::= "If-Modified-Since"
<if-none-match> ::= <if-none-match-header-name><colon><entity-tags><newline>
<if-none-match-header-name> ::= "If-None-Match"
<if-range> ::= <if-range-header-name><colon><if-range-value><newline>
<if-range-header-name> ::= "If-Range"
<if-range-value> ::= <entity-tag> | <date-value>
<if-schedule-tag-match> ::=
↪  <if-schedule-tag-match-header-name><colon><entity-tag><newline>
<if-schedule-tag-match-header-name> ::= "If-Schedule-Tag-Match"
<if-unmodified-since> ::=
↪  <if-unmodified-since-header-name><colon><date-value><newline>
<if-unmodified-since-header-name> ::= "If-Unmodified-Since"
<link> ::= <link-header-name><colon><link-value><newline>
<link-header-name> ::= "Link"
<link-value> ::= <start-tag><absolute-uri><end-tag>
<max-forwards> ::=
↪  <max-forwards-header-name><colon><max-forwards-value><newline>
<max-forwards-header-name> ::= "Max-Forwards"
<max-forwards-value> : "0" | "1"
<mime-version> ::=
↪  <mime-version-header-name><colon><mime-version-value><newline>
<mime-version-header-name> ::= "MIME-Version"
<mime-version-value> : "1.0" | "1.1"
<odata-isolation> ::=
↪  <odata-isolation-header-name><colon><odata-isolation-value><newline>
<odata-isolation-header-name> ::= "OData-Isolation"
<odata-isolation-value> : "snapshot"
<odata-maxversion> ::=
↪  <odata-maxversion-header-name><colon><odata-version-value><newline>
<odata-maxversion-header-name> ::= "OData-MaxVersion"
<odata-version-value> : "4.0"
<odata-version> ::=
↪  <odata-version-header-name><colon><odata-version-value><newline>
<odata-version-header-name> ::= "OData-Version"
<ordering-type> ::=
↪  <ordering-type-header-name><colon><ordering-type-value><newline>
<ordering-type-header-name> ::= "Ordering-Type"
<ordering-type-value> ::= "DAV:unordered" | "DAV:custom" |
↪  "http://example.org/example.html"
<origin> ::= <origin-header-name><colon><origin-value><newline>
<origin-header-name> ::= "Origin"
<origin-value> ::= "http://example.com" | "null"
<oscore> ::= <oscore-header-name><colon><oscore-value><newline>
<oscore-header-name> ::= "OSCORE"
<oscore-value> ::= "CSU" | "AA"
<overwrite> ::= <overwrite-header-name><colon><boolean><newline>
<overwrite-header-name> ::= "Overwrite"
<position> ::= <position-header-name><colon><position-value><newline>
<position-header-name> ::= "Position"
<position-value> : "first" | "last" | "after example.html"
<pragma> ::= <pragma-header-name><colon><pragma-directive><newline>
<pragma-header-name> ::= "Pragma"
<pragma-directive> : "no-cache"
<prefer> ::= <prefer-header-name><colon><preferences><newline>
<prefer-header-name> ::= "Prefer"
<preferences> ::= <preference> | <preference><comma><preference>
<preference> : "respond-async" | "wait=100" | "handling=lenient"
<proxy-authorization> ::=
↪  <proxy-authorization-header-name><colon><auth-scheme><space><creds><newline>
<proxy-authorization-header-name> ::= "Proxy-Authorization"
<range> ::= <range-header-name><colon><range-unit><equals><range-value><newline>
<range-header-name> ::= "Range"
<range-value> ::= "5-8" | "5-"
<referer> ::= <referer-header-name><colon><absolute-uri><newline>
<referer-header-name> ::= "Referer"
<schedule-reply> ::= <schedule-reply-header-name><colon><boolean><newline>
<schedule-reply-header-name> ::= "Schedule-Reply"
<sec-token-binding> ::=
↪  <sec-token-binding-header-name><colon><sec-token-binding-value><newline>
<sec-token-binding-header-name> ::= "Sec-Token-Binding"
<sec-token-binding-value> ::= "AIkAAgBBQLgtRpWFPN66kxhxGrtaKrzcMtHw7HV8"
<sec-websocket-accept> ::=
↪  <sec-websocket-accept-header-name><colon><sec-websocket-accept-value><newline>
<sec-websocket-accept-header-name> ::= "Sec-Websocket-Accept"
<sec-websocket-accept-value> ::= "s3pPLMBiTxaQ9kYGzzhZRbK+xOo="
<sec-websocket-extensions> ::= <sec-websocket-extensions-header-name><colon>
↪  <sec-websocket-extensions-values><newline>
<sec-websocket-extensions-header-name> ::= "Sec-Websocket-Extensions"
<sec-websocket-extensions-values> ::= <sec-websocket-extensions-value> |
↪  <sec-websocket-extensions-value><comma><sec-websocket-extensions-value>
<sec-websocket-extensions-value> ::= "deflate-stream" | "mux" | "max-channels:4;
↪  flow-control"
<sec-websocket-key> ::=
↪  <sec-websocket-key-header-name><colon><sec-websocket-key-value><newline>
<sec-websocket-key-header-name> ::= "Sec-Websocket-Key"
<sec-websocket-key-value> ::= "dGhlIHNhbXBsZSBub25jZQ=="
<sec-websocket-protocol> ::=
↪  <sec-websocket-protocol-header-name><colon><sec-websocket-protocol-values><newline>
<sec-websocket-protocol-header-name> ::= "Sec-Websocket-Protocol"
<sec-websocket-protocol-values> ::= <sec-websocket-protocol-value> |
↪  <sec-websocket-protocol-value><comma><sec-websocket-protocol-value>
<sec-websocket-protocol-value> ::= "chat" | "superchat"
<sec-websocket-version> ::=
↪  <sec-websocket-version-header-name><colon><sec-websocket-version-value><newline>
<sec-websocket-version-header-name> ::= "Sec-Websocket-Version"
<sec-websocket-version-value> ::= "13"
<slug> ::= <slug-header-name><colon><slug-value><newline>
<slug-header-name> ::= "Slug"
<slug-value> ::= "The Beach at S%C3%A8te"
<te> ::= <te-header-name><colon><te-encodings><newline>
<te-header-name> ::= "TE"
<te-encodings> ::= <te-encoding> | <te-encoding><comma><te-encoding>
<te-encoding> ::= <te-encoding-name><semicolon><quality>
<te-encoding-name> ::= "gzip" | "compress" | "deflate" | "br" | "identity" |
↪  "chunked" | "trailers"
<timeout> ::= <timeout-header-name><colon><timeout-values><newline>
<timeout-header-name> ::= "Timeout"
<timeout-values> ::= <timeout-value> | <timeout-value><comma><timeout-value>
<timeout-value> ::= "Infinite" | "Second-4100000000"
<topic> ::= <topic-header-name><colon><topic-value><newline>
<topic-header-name> ::= "Topic"
<topic-value> ::= "upd"
<trailer> ::= <trailer-header-name><colon><trailer-value><newline>
<trailer-header-name> ::= "Trailer"
<trailer-value> ::= "Expires"
<transfer-encoding> ::=
↪  <transfer-encoding-header-name><colon><transfer-encodings><newline>
<transfer-encoding-header-name> ::= "Transfer-Encoding"
<transfer-encodings> ::= <encoding-name> | <encoding-name><comma><encoding-name>
<ttl> ::= <ttl-header-name><colon><ttl-value><newline>
<ttl-header-name> ::= "TTL"
<ttl-value> ::= "0" | "1"
<urgency> ::= <urgency-header-name><colon><urgency-value><newline>
<urgency-header-name> ::= "Urgency"
<urgency-value> ::= "very-low" | "low" | "normal" | "high"
<upgrade> ::= <upgrade-header-name><colon><upgrade-values><newline>
<upgrade-header-name> ::= "Upgrade"
<upgrade-values> ::= <upgrade-value> | <upgrade-value><comma><upgrade-value>
<upgrade-value> ::= "websocket" | "HTTP/2.0" | "SHTTP/1.3" | "IRC/6.9" |
↪  "RTA/x11"
<user-agent> ::= <user-agent-header-name><colon><user-agent-value><newline>
<user-agent-header-name> ::= "User-Agent"
<user-agent-value> ::= "curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3"
<via> ::= <via-header-name><colon><via-values><newline>
<via-header-name> ::= "Via"
<via-values> ::= <via-value> | <via-value><comma><via-value>
<via-value> ::= "1.0 fred" | "1.1 p.example.net"
```

Listing 20: The full grammar for the request headers experiment.