H2FUZZ: Guided, Black-box, Differential Fuzzing for HTTP/2-to-HTTP/1 Conversion Anomalies

Anthony Gavazzi
Northeastern University
Boston, MA
gavazzi.a@northeastern.edu

Weixin Kong
Northeastern University
Boston, MA
kong.weix@northeastern.edu

Engin Kirda
Northeastern University
Boston, MA
e.kirda@northeastern.edu

***Abstract*—HTTP/2 is by far the most popular HTTP version, yet in practice, HTTP connections rarely occur over end-to-end HTTP/2. This is due in large part to the fact that reverse proxies such as Content Delivery Networks (CDNs) between the client and server universally support HTTP/2 on the client side of the connection, but rarely on the server side. Proxies must therefore dynamically convert between HTTP/2 and HTTP/1, and anomalies in this conversion process can lead to critical vulnerabilities.**

Prior work proposed generational fuzzing techniques to discover these anomalies. However, such an approach lacks meaningful feedback, limiting the expressiveness of the generated requests and the number of anomalies it can induce. We, therefore, propose H2FUZZ, a black-box differential fuzzer for HTTP/2 which uses a comprehensive mutator and novel feedback system to drive a set of reverse proxies to increasingly divergent behavior, uncovering conversion anomalies in the process. We fuzz a set of 11 standalone reverse proxies and 5 CDNs with H2FUZZ, and find that it induces 50% more conversion anomalies than the state-of-the-art, many of which have immediate security implications.

I. INTRODUCTION

The HTTP/2 protocol has seen an explosion in popularity since its introduction in 2015. A measurement [24] by the HTTP Archive found that 85% of all HTTP requests in 2024 were served over HTTP/2, up from 64% in 2020 [12]. However, these numbers reflect the percentage of requests initiated *directly* from clients. In practice, clients and origin servers rarely communicate over end-to-end HTTP/2.

This is due in large part to the fact that content delivery networks (CDNs) and other standalone reverse proxies intercept and process the messages exchanged between clients and upstream servers. While these systems near-universally support HTTP/2 on the client end of the connection, they rarely do so on the server end, and must therefore dynamically convert between HTTP/2 and HTTP/1[1] as they forward requests and responses in either direction.

HTTP is a complex protocol, and contemporary research has demonstrated how discrepancies in the way two applications interpret the same HTTP request can make systems vulnerable to an increasing number of semantic gap attacks such as HTTP Request Smuggling, Cache-Poisoned Denial-of-Service, and Host of Troubles. Such attacks are practical, and have been weaponized against high-profile targets to, for example, serve malicious JavaScript by poisoning PayPal's web cache [1] and steal HTTP responses destined for arbitrary users of Atlassian Jira [19].

HTTP/2-to-HTTP/1 conversion adds even more complexity to this ecosystem, and indeed broadens the attack surface for this class of vulnerabilities. The first (and thus far only) scientific analysis of HTTP/2-to-HTTP/1 conversion anomalies was performed by Jabiyev et al. [17]. In their work, the authors developed FRAMESHIFTER, a fuzzer which generates HTTP/2 requests from a predefined grammar. By sending these HTTP/2 requests to a set of 12 popular reverse proxy technologies and CDNs, then capturing the HTTP/1 requests forwarded by them, they studied the anomalies that arise during the conversion to HTTP/1, the causes thereof, and the attacks that could be crafted using them.

Though their study uncovered a number of conversion anomalies leading to critical security vulnerabilities, it had several major limitations.

Limited mutable fields. HTTP/2 is a highly-structured binary protocol with a significant number of semantic rules that must be followed to construct a valid request. When building a grammar by which to generate HTTP/2 requests, it is essential for the grammar to output requests that are unusual enough to trigger anomalous behavior, yet not so unusual as to break the protocol entirely, causing requests to be rejected immediately by the systems under test.

To strike such a balance, the grammar used by FRAMESHIFTER maintains a limited set of strictly-valid values for all fields of each frame type, and can only make byte-level mutations to a small, curated set of important headers. Though this may improve test throughput, it inhibits the expressiveness of the requests generated by the grammar, and has no way to iteratively modify fields to observe how they impact the behavior of the systems under test.

Limited stacked mutations. Similarly, even with a carefully-crafted and expressive input grammar, applying too many random mutations to a valid request will invariably break structural and semantic rules, and lead to it being rejected by the systems being fuzzed. FRAMESHIFTER, therefore, makes a maximum of four total mutations to a minimal, semantically valid HTTP/2 stream. With so few mutations, the expressiveness of the requests generated by FRAMESHIFTER is inherently limited.

[1]For brevity, we refer to all HTTP/1.* protocol versions as HTTP/1.

No feedback. As a purely generational fuzzer, FRAMESHIFTER has no visibility into how any one request or set of mutations performs on the proxies being tested at runtime. Without such visibility, and without a specific objective, it has no way to minimize redundancy or prioritize further mutations to requests that have resulted in interesting behavior.

As a result, FRAMESHIFTER continually generates requests blindly until the program is stopped, dumping millions of streams comprising many GB of data to the file system to be processed in post. A human analyst must then use manual analysis and ad-hoc scripting to find anomalies and discern a finite set of input classes that trigger them. Due to the scale of data created by FRAMESHIFTER, the authors resorted to sampling the requests in its output to analyze for input patterns, leaving the question of *completeness* of their analysis open.

In light of these limitations, we aim to answer the following research questions:

(Q1) Can a feedback-driven fuzzing approach that still operates in a black-box manner effectively trigger HTTP/2-to-HTTP/1 conversion anomalies?
(Q2) How does such an approach compare against the state-of-the-art generational fuzzer at triggering such anomalies?
(Q3) What manipulation patterns cause these anomalies, and what are their security implications?

To answer these questions, we develop H2FUZZ, an HTTP/2 fuzzer designed to trigger HTTP/2-to-HTTP/1 conversion anomalies. Key to H2FUZZ's functionality is a novel, lightweight feedback mechanism that parses, hashes, and normalizes HTTP/1 requests to detect *new differences* among the requests forwarded by multiple proxies while they process the same input HTTP/2 stream. The feedback mechanism operates in a fully black-box manner, enabling H2FUZZ to gracefully integrate CDNs, which are otherwise challenging to fuzz due to being live systems that cannot be instrumented or run in a sandbox.

Additionally, we describe a methodology to automatically determine the minimal set of features an HTTP/2 stream must have to induce a given reverse proxy to forward a request with a given anomaly. Our technique learns binary decision tree classifiers on a custom set of 716 features extracted from HTTP/2 streams, and allows for a comprehensive, interpretable analysis of the entire fuzzer output to find all input categories that trigger all anomalies.

We fuzz 16 popular proxy technologies, comprising 11 standalone reverse proxies and five CDNs, with H2FUZZ and FRAMESHIFTER over three 72-hour experiments each, and find that H2FUZZ exposes 50% more anomalies than FRAMESHIFTER uncovers in both of its running modes combined.

In summary, this paper makes the following contributions:

- We present H2FUZZ, a guided, black-box, differential fuzzer for HTTP/2-to-HTTP/1 conversion anomalies.
- We implement a novel feature set on HTTP/2 streams and describe a methodology to learn the minimal input patterns that cause anomalies in HTTP/1 requests.
- We discover new conversion anomalies and provide insights into their causes and security implications.

Availability. H2FUZZ is open source alongside all code needed to run our experiments and analysis. It is available at https://github.com/Gavazzi1/h2fuzz.

Responsible Disclosure. We have notified all of the tested proxies' vendors of our findings, and a summary of their responses is included in Section VI-C.

II. BACKGROUND AND RELATED WORK

In this section, we provide an overview of relevant aspects of the HTTP/2 protocol, conversion anomalies, and semantic gap attacks, then discuss fuzzing techniques for network applications and past research on detecting semantic gap attacks for HTTP systems.

A. The HTTP/2 Protocol

Unlike HTTP/1, HTTP/2 is a binary protocol with discrete bit-ranges dedicated to specific fields in a given request. The atomic unit exchanged between two HTTP/2 endpoints is the *frame*, which itself is part of a *stream*. Each stream is given a unique identifier, and frames from different streams can be interleaved over a single TCP connection.

HTTP/2 supports 10 different frame types, which all share a common nine-octet header

The format of the frame's payload depends on the type of frame. In a `DATA` frame, which carries the body of an HTTP request or response, for example, the payload consists of the body itself and, if the `PADDED` flag is set in the frame header, an arbitrary amount of padding bytes along with a field indicating the number of padding bytes.

B. Conversion Anomalies and Semantic Gap Attacks

Although HTTP/2 and HTTP/1 are capable of expressing the same basic semantics of request and response, they are fundamentally different protocols, with different data structures and execution loops. Intermediaries converting from HTTP/2 to HTTP/1 must be able to extract the information necessary to build an HTTP/1 request while still supporting all HTTP/2 features.

Ambiguous, overlooked, or in any way unexpected patterns in a stream can cause an intermediary to forward a request that is *anomalous* – that is, it contains some semantic or structural pattern that is likely to cause an upstream server to interpret critical aspects of the HTTP/1 request differently than the intermediary.

Discrepancies in the way that an intermediary and upstream server process the same request are central to a number of novel attacks on web systems. One such example is HTTP Request Smuggling (HRS) [22]. In HRS, a discrepancy arises in the way the intermediary and server determine the bounds of a request, such that the intermediary sees one request while the server sees two. This second request seen by the server

is considered "smuggled" through the connection, and can lead to firewall bypasses, cache poisoning, and response queue poisoning.

In a Host of Troubles (HoT) attack [10], a discrepancy arises between the intermediary and server's interpretation of the `Host` header, causing them to disagree over which authority, and thus which resources, to serve in a request, leading to cache poisoning and security policy bypassing.

Another attack caused by such discrepancies is a Cache-Poisoned Denial-of-Service (CPDoS) attack [25], which occurs when a server returns a cacheable error response to a request that the intermediary does *not* believe to be an error, and which caches under the same key as a valid request for the same resource. When another user tries to access the same resource, they will be unable to view it, instead receiving the cached error response.

C. Fuzzing for Network Applications

Fuzzing is the process of iteratively providing unusual inputs to a system under test (SUT), which is then monitored for aberrant behavior such as a crash or illegal memory access. Fuzzing approaches are categorized as either black-box or gray-box, and several approaches in both classes have been applied to network applications.

Black-box fuzzers operate without any explicit program feedback, and determine the success of a given input based only on program outputs. Some black-box fuzzers [2], [3] are purely generational, where each input is constructed independently from any other. Black-box fuzzers *can* incorporate feedback, however, as long as the feedback depends only on program input and output. Such novel feedback methods may be based on, for example, exchanged packet sequences [23] or inferred protocol state coverage [13], in order to determine when one input has exercised new behavior, and should therefore be mutated further to explore more of the program.

Gray-box fuzzers, by contrast, incorporate explicit program feedback, typically in the form of code-coverage [27], in order to guide the fuzzing process. If new code blocks are executed when the SUT processes an input, then that input is prioritized for further mutation in order to exercise, iteratively, more of the program. Gray-box fuzzers for network applications often incorporate a means to inferring protocol state coverage from the code itself [7], [28], and use that as program feedback on top of standard code coverage.

D. Fuzzing for Semantic Gap Attacks in HTTP Systems

A handful of recent works have used both black-box and gray-box fuzzing techniques to find novel semantic gap attacks in HTTP systems. Jabiyev et al. [18] developed T-REQS, a black-box grammar-based fuzzer for HTTP/1, which they used to discover numerous mutation patterns that enable HRS between popular proxies and end servers.

Shen et al. [33] took a broader approach to detecting semantic gap attacks and developed HDiff, a tool which generates HTTP/1 requests based on rules it automatically extracts from the HTTP RFC. They then sent this suite of requests to a set of 10 proxies and servers, discovering a number of HoT, CPDoS, and HRS vulnerabilities.

Moving beyond black-box approaches, Jabiyev et al. [16] developed GUDIFU, a gray-box differential fuzzer which uses the combined code coverage of multiple proxies being fuzzed to guide input selection and mutation. They fuzz six popular reverse proxies with GUDIFU to find a set of generic "parsing discrepancies," which they systematize and use to discover HRS, CPDoS, and access control bypass attacks.

Outside of fuzzers designed for HTTP/1, and most relevant to this work, Jabiyev et al. [17] developed FRAMESHIFTER, a black-box grammar-based fuzzer for HTTP/2, which they used to find HTTP/2-to-HTTP/1 conversion anomalies in 12 popular proxy technologies. They then organized the HTTP/2 streams that caused conversion anomalies into a discrete set of input categories and described a set of novel attacks that could be crafted using the conversion anomalies.

III. H2FUZZ

In this section, we describe the architecture and functionality of H2FUZZ, our guided, black-box, differential fuzzer. Figure 1 shows the individual components of the fuzzer and the data flow among them. We provide a brief overview of the fuzzer's workflow here, and then discuss each component in more depth.

In one fuzzer iteration, each of a configurable number of processes in the fuzzer core first selects one HTTP/2 stream from the input corpus. It then parses the stream into a structured representation and passes it to the Structure-Aware Mutator.

The Structure-Aware Mutator performs a random sequence of field-level and frame-level mutations on the stream, then passes the stream to a collection of HTTP/2 clients. Each client corresponds to one SUT, and is responsible for updating any `:authority` or `host` headers for the given SUT. The client then serializes the structured representation back into a binary stream and sends it over a TCP connection to a transparent proxy positioned between the fuzzer and the SUT.

The role of the transparent proxy is to manage the SUT, serving as a synchronization point to stop the fuzzer processes from interacting with the SUT and restart it if it becomes unresponsive. It also delivers the serialized HTTP/2 stream to the SUT.

Each SUT receives the same HTTP/2 stream and either returns an error back to the HTTP/2 client or forwards an HTTP/1 request to an upstream echo server. The echo server captures the entire forwarded request and returns it back in the body of a `200 OK` response. This response is transferred back down the chain of components to the HTTP/2 client.

The HTTP/2 client parses the HTTP/1 request into a structural representation, then extracts key fields such as the method, body, and important header values. These values are then hashed and mean-normalized before being passed back again to the fuzzer core, which uses the resulting set of hash values to detect whether a new difference in the forwarded HTTP/1 requests was found. If so, it adds the mutated HTTP/2

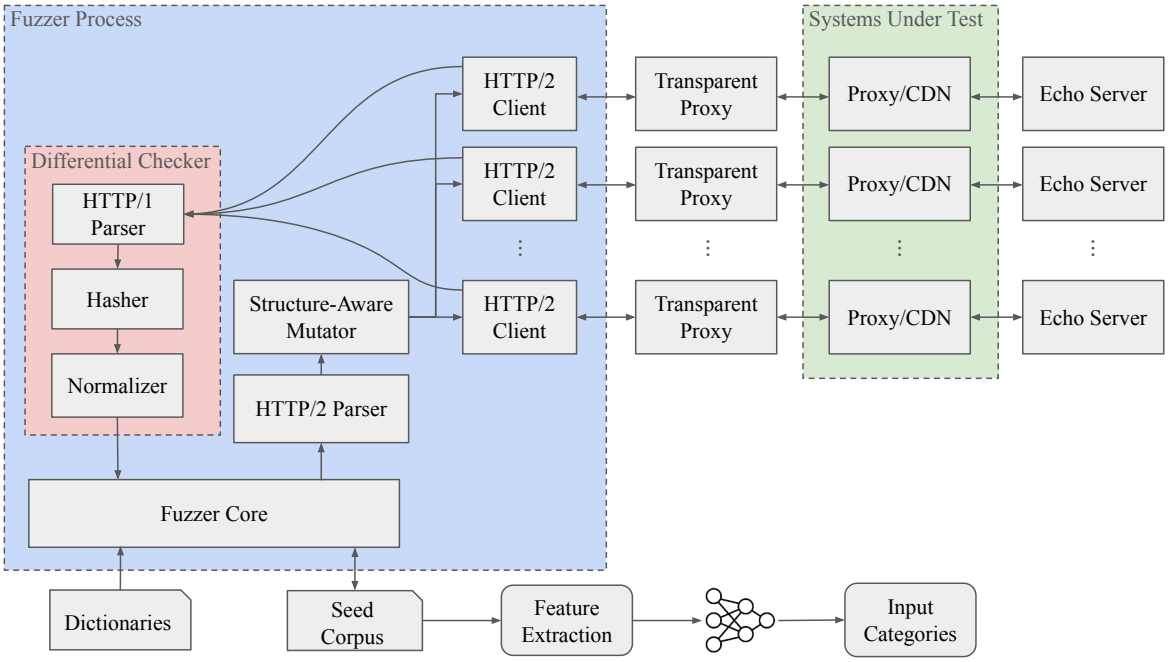

Fig. 1. System Architecture of H2FUZZ.

stream to the input corpus, and the process repeats until the fuzzer is stopped.

A. Input Corpus

The input corpus is a set of structurally-valid HTTP/2 streams shared by all fuzzer processes, from which inputs are selected and to which any inputs that trigger new differences among the requests forwarded by the SUTs are added. All HTTP/2 streams in the corpus must be structurally correct according to the frame definitions in the RFC in order for the mutator to parse the stream into a structured representation.

B. Fuzzer Core

The fuzzer core launches and monitors a configurable number of fuzzer processes, each of which executes the same fundamental loop described at the beginning of this section. One of the primary bottlenecks to throughput when fuzzing network systems is waiting until a SUT has finished processing a request, and much research has been directed at addressing this [6], [21], [28], [32]. In a black-box environment, however, hard-coded wait times are unavoidable. Thus, significant parallelism is employed to maintain fuzzer productivity.

Each fuzzer process operates independently and handles a different HTTP/2 stream from the other processes, but all share the same input corpus. Each fuzzer process is also separately responsible for maintaining a set of all unique differences seen among the HTTP/1 requests forwarded by the SUTs since the beginning of the fuzzing session.

The fuzzer core is built on top of NEZHA [26], a differential fuzzing engine developed by Petsios et al. and based on libFuzzer [4]. NEZHA is designed to detect differences in the execution of a set of SUTs and drive them to increasingly divergent behavior. Because each SUT processes the same request and converts between the same two protocols, differences among the forwarded requests are the most likely indicators that an anomaly exists in one of the forwarded requests, which may be indicative of a vulnerability. We discuss our custom differential checker and how we integrate it with NEZHA in more detail in Section III-H.

C. HTTP/2 Parser

We implemented a serializer and deserializer for every HTTP/2 frame type. The binary stream from the input corpus is deserialized to a structured representation with which any individual field in the stream can be modified at will. The structured representation can then be serialized back into a binary stream to be sent over a network connection to a SUT.

D. Structure-Aware Mutator

LibFuzzer, and by extension NEZHA, is by default capable of a number of random byte-level mutation operators that can be applied to test inputs. However, HTTP/2 is a highly structured protocol for which exact byte ranges and values must be respected in order for an upstream proxy to parse a request.

For such data types, structure-aware and grammar-based mutations are much more effective than byte-level mutations at exercising diverse behavior in the SUTs, because most byte-level mutations run a risk of breaking the structural validity of the stream being mutated, resulting in the fuzzer spending

most of its time exercising error states of the SUTs. Thus, we implement a custom HTTP/2 mutator capable of three categories of mutations: frame-level mutations, random byte-level mutations, and smart byte-level mutations.

1) Frame-level Mutations: Frame-level mutations are mutations applied at the granularity of the frames within a stream. Such mutations include swapping two frames in a stream, duplicating a frame in a stream, deleting a frame, and inserting a frame from another stream selected from the input corpus.

2) Random Byte-level Mutations: Both random and smart byte-level mutations are applied to individual fields within any type of frame, but are guaranteed to preserve the structural validity of the overlying stream itself. Random byte-level mutations include standard, general-purpose fuzzer mutation operations such as as bit flips, inserting random bytes, and deleting random consecutive bytes from a single field in a frame. When the field being mutated must remain a constant size, such as the flags in a frame's header, the random byte mutations are guaranteed to mutate the field in-place.

3) Smart Byte-level Mutations: We additionally implement a set of smart mutations that modify the stream based on known protocol details so that H2FUZZ can more quickly exercise meaningful semantics in HTTP/2 streams.

One category of these mutations is dictionary mutations. We maintain a set of well-known HTTP keywords in an external dictionary. The mutator may, at random, select values from this dictionary and insert them into random fields of the stream. Additionally, when performing a mutation on a header value, the mutator may check the associated header's name and add a known meaningful value for that header. For example, we maintain a set of HTTP methods to be used when mutating a `:method` header.

Additionally, container frames such as `HEADERS` and `SETTINGS` frames support a "split" operation which splits their payloads at a random index into two consecutive frames of the same type.

The last type of smart mutation is a "fix flags" operation which, if chosen by the mutator, iterates over the entire stream and adjusts each frame's flags such that only the final frame that contains headers in the stream has the `END HEADERS` flag, and only the final `HEADERS` or `DATA` frame in the stream has the `END STREAM` flag. This is designed to improve the validity of each stream so that fewer requests are rejected outright from the SUTs for having invalid flags.

4) Additional Mutation Rules: To maximize the expressiveness of the requests generated by H2FUZZ, we enable mutations on every field of every frame, with a few key exceptions. First, we never delete a frame if it is the only frame left in the stream. The fuzzer must be able to send something to the SUTs.

Next, we never modify a frame's length field, as modifying it directly impacts the receiving endpoint's ability to determine the beginning and end of the frame, breaking the structural validity of the stream.

We also never modify the stream identifier field, as this creates the possibility of sending multiple streams at once to the SUT. In such cases, a race condition occurs where responses to each unique stream arrive in different orders for different proxies, preventing the fuzzer from comparing the requests forwarded by the SUTs when they process the same *single* HTTP/2 stream.

Last, when mutating a `SETTINGS` frame, we never modify the value of a `SETTINGS_INITIAL_WINDOW_SIZE` parameter. When an endpoint in an HTTP/2 connection sends this parameter, it advertises the maximum number of bytes it is prepared to receive in the stream's `DATA` frames. If the mutator sets this to a value smaller than the size of the HTTP/1 request forwarded by the SUT, then the SUT will respond with up to that many bytes of data, cutting the forwarded request off before the end. In such cases, H2FUZZ can no longer meaningfully process the request, and so mutating this parameter is considered out of scope for our experiments.

E. HTTP/2 Clients

Once the structural representation of the HTTP/2 stream has been mutated, a unique copy of it is given to one HTTP/2 client for each SUT. The HTTP/2 client's responsibility is to preprocess the request for one specific SUT, transmit the request over the network, and process the SUT's response.

Preprocessing comprises two steps. First, modifying the value of any `:authority` and `host` headers to match the values expected by the SUT. This is essential for CDNs, as each corresponds to a unique domain and cannot be configured to accept requests on localhost in the same way that locally-running proxies can.

The second step is simply to serialize the mutated, preprocessed request into a binary stream and transmit it over a network connection to the transparent proxy positioned between the fuzzer and the SUT.

When a response arrives, either as an error directly from the SUT or as a response containing the HTTP/1 request forwarded by the SUT, the HTTP/2 client deserializes the HTTP/2 stream and extracts the HTTP/1 request from the `DATA` frames in the stream. Or, in the case of an error from the SUT, the HTTP/2 client simply sets a flag indicating that the SUT did not forward a request.

F. Transparent Proxies

As the fuzzer core spawns a number of fuzzer processes, a substantial number of requests are sent in parallel to each SUT. As independent, standalone processes, they still require some means to synchronization if an SUT becomes unresponsive and needs to be restarted. To enable this kind of synchronization and control over the SUT during runtime, each SUT has one transparent proxy that sits between it and H2FUZZ.

The transparent proxies are simple TCP proxies that forward all requests, unmodified, to and from the SUT. If an SUT ever becomes unresponsive, the transparent proxy maintains a queue of all outbound requests and either restarts the proxy (in the case of locally-running proxies) or waits for a predetermined number of seconds (in the case of CDNs), after which point it will send all of the requests in its queue.

(A)

```
POST / HTTP/1.1
Host: a.com
Content-Length: 5

ABCDE
```

(B)

```
POST / HTTP/1.1
Host: a.com:xyz
Content-Length: 5

ABCDE
```

(C)

```
POST / HTTP/1.1
x-req-id: 0x123
content-length: 5
host: a.com

ABCDE
```

(D)

```
POST /abc HTTP/1.0
Content-Length: 10
Host: a.com

ABCDEFGHIJ
```

(E)

```
POST /abc HTTP/1.0
Host: a.com:xyz
Content-Length: 10

ABCDEFGHIJ
```

Fig. 2. Sample HTTP/1 requests that H2FUZZ must differentiate appropriately.

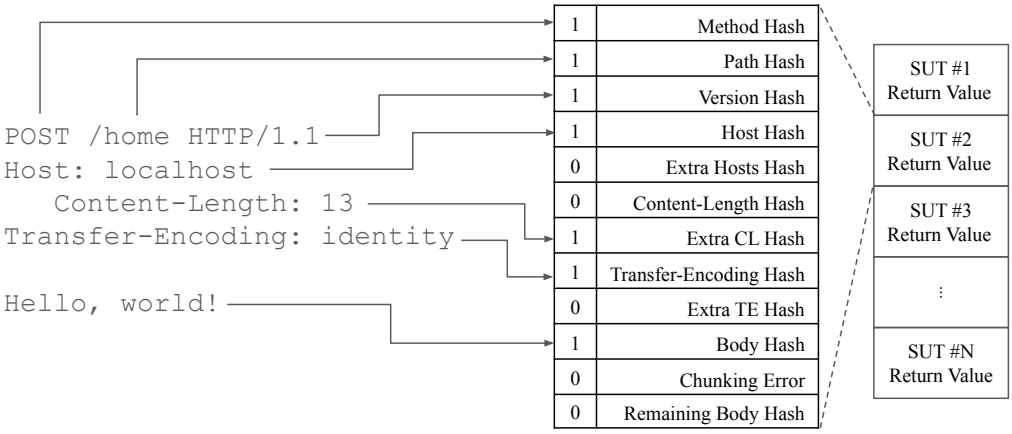

Fig. 3. Hash-based differential checker for HTTP/1 requests.

G. Echo Servers

Echo servers are the upstream servers which each SUT is configured to forward their HTTP/1 requests to. The role of an echo server is to transmit the HTTP/1 request back to the fuzzer core for processing, so as soon as the SUT has finished transmitting data, the echo server sends all of the data it has received back in the body of a `200 OK` response.

H. Differential Checker

The objective of H2FUZZ is to trigger and detect anomalies in the HTTP/1 requests forwarded by each SUT. By detecting meaningful differences in the forwarded requests and prioritizing further mutations to those that trigger new or especially interesting differences, H2FUZZ aims to drive the SUTs to increasingly divergent behavior, discovering more substantial anomalies that lead to vulnerabilities.

Once each SUT's forwarded request (or error response) has been received in the HTTP/2 clients, the next necessary step is therefore to identify meaningful differences in the requests and determine whether they reflect newly-exercised behavior.

The role of the differential checker is determine exactly this, but detecting general differences across HTTP/1 requests is nontrivial. HTTP/1 requests are complex strings, and determining whether two SUTs forward the "same" request cannot be achieved by a simple string comparison. Body encoding, header name capitalization, header ordering, and the presence of substantial irrelevant and/or user-controlled data are all aspects that must be handled carefully in order to avoid a high false positive rate in detecting differences.

Additionally, in order to make effective use of NEZHA, which we selected as our fuzzer core in order to make use of its guided differential engine, each HTTP/1 request must be flattened into a single integer value to act as the SUT's "return value."

Motivating Examples. To motivate our solution, we will consider several cases where two SUTs forward pairs of HTTP requests from Figure 2. First, imagine that the SUTs forward requests (A) and (B). The requests are identical except for the `Host` header, as request (B) has a non-numeric port value after the domain. This is a notable difference in an important HTTP header that should be captured by the differential checker.

Next, imagine that the SUTs forward requests (A) and (C). Though these requests are more different than (A) and (B) on a character-by-character basis, they are semantically the same, with the same request line, `Host` and `Content-Length` headers, and body. The header names in (C) are lowercase, but HTTP header names are case-insensitive, and the `x-req-id` header is likely added by the SUT to every request, with no impact to how an upstream server would process the request. Also, although the `Host` and `Content-Length` headers are in a different order, these headers have no dependencies on one another, and so the order will not affect any upstream processing. The differential checker should not

consider these requests to be different.

Last, imagine that the SUTs forward requests (D) and (E). Like requests (A) and (B), these requests differ only in the `Host` header, for which request (E) has a non-numeric port. The differential checker should consider these requests different, but it should consider them different in the *same* way that it considered (A) and (B) to be different. Thus, even though (A) and (B)'s paths, versions, `Content-Length` values, and bodies are different than (D) and (E)'s, if the fuzzer observes both pairs of forwarded requests during execution, it should only consider *one* of them to be an "interesting" difference, as whatever HTTP/2 stream induced the SUTs to forward the other pair has not exercised any new divergent behavior.

The differential checker addresses each of these cases. At a high level, the differential checker parses the HTTP/1 requests, extracts the values of particular fields within them, hashes these fields, mean-normalizes the hashes to capture only the byte-level differences between respective fields across requests, flattens the set of field-hashes for each request to a single integer, and then passes a vector of all these integers to NEZHA.

Figure 3 depicts the translation of one HTTP/1 request into the hash vector, and we discuss each step of this translation in more detail below.

1) HTTP/1 Parsing: The HTTP/1 requests forwarded by each SUT are unstructured string data, often containing a substantial amount of extraneous, random data in the form of custom headers and identifiers. Additionally, variations in header cases and ordering vary substantially without changing the semantics or validity of the request. Simply comparing the strings themselves would therefore result in excessive false positives, and so relevant information must be parsed out of the requests first.

Similarly to the HTTP/2 parser, we parse each HTTP/1 request into a structural representation according to the protocol ABNF defined in RFC 7230 [11]. It may seem odd, given that the objective of H2FUZZ is to identify anomalies in HTTP/1 requests, to assume that the forwarded request will be well-formed. However, a benefit of studying protocol conversions is that the forwarded request must be *constructed* based on information extracted from the HTTP/2 stream. As a result, in our experiments, it was always possible to parse the HTTP/1 requests according to the RFC, with anomalies tending to occur in more subtle ways, such as in value mismatches, illegal characters, and missing or duplicate headers.

While parsing the request headers, if a `transfer-encoding` header is observed with a value of `chunked`, the HTTP/1 parser will attempt to parse the request body as if it were in a chunked encoding. If any errors occur during this parsing process, the type of error (e.g., a chunk size is invalid, a terminating CRLF is missing, etc.) is recorded, and any data in the body past the point of error is stored separately from the body parsed up to that point.

Algorithm 1 Normalization algorithm used in the differential checker.

```
1:  procedure NORMALIZE(FieldNames, ReqHashes)
2:      for each Field ∈ FieldNames do
3:          mean ← 0
4:          for each Hashes ∈ FwdRequests do
5:              if Hashes.Field ≠ 0 then
6:                  mean ← mean + Hashes.Field
7:              end if
8:          end for
9:          for each Hashes ∈ FwdRequests do
10:             if Hashes.Field ≠ 0 then
11:                 Hashes.Field ← Hashes.Field − mean
12:             end if
13:         end for
14:     end for
15: end procedure
```

2) Request Hashing: Given the structured representation of the HTTP/1 request, we next extract the information most relevant to identifying security-relevant anomalies. Based on the anomalies and vulnerabilities described in prior work, we choose to extract the request method, path, version, body, error encountered when parsing the body, and values of the `Content-Length`, `Transfer-Encoding`, `Host`, `Connection`, and `Expect` headers. If multiple of any of these headers are encountered, or if the value of the header is a comma-separated list, we add each unique value after the first to a separate list of values for that header.

When searching for these important headers, we look for case-insensitive matches from the first non-whitespace character in the header name up to the last non-whitespace character before the colon separator. If whitespace characters are encountered before or after the header name, the header value is considered a "special match" and added to the same list as duplicate header values. Whitespace-surrounded header names are a pattern that have led to many vulnerabilities involving HTTP parsing in the past [10], [18], [33], and so capturing them is essential to detecting security-critical anomalies.

Next, each of these parsed fields is hashed using a checksum function. Every 8-bit character in the string is summed into a single 32-bit integer representation of the value. For the lists of header values, each entry in the list is summed together into a single 32-bit value. When a field is not present in the HTTP/1 request (for example, if the request does not have a body) then the integer value for that field is set to zero.

3) Normalization: The final step is to compute the differences among corresponding fields in each of the forwarded requests. To do this, we mean-normalize the hashes of each field following the procedure in Algorithm 1. Specifically, for each field extracted and hashed in the previous step, we first compute the mean of all nonzero hash values of that field across all forwarded requests. Then, we subtract that mean from each nonzero hash value for that field.

After this step, any nonzero value for a field's hash indicates

that there was a difference among the forwarded requests' values for that field, and the particular byte-level difference is reflected in the unique hash values after performing this mean-normalization.

Additionally, to differentiate a request where a field was normalized to zero from a request that did not have a value for that field, each field's hash is prepended with a single "field-presence bit," which is set to one if the request had any value for that field, and is zero otherwise.

Next, for a single request, each normalized field hash (with the field-presence bit) is added to a vector, which itself is hashed to a single integer which acts as the return value for that SUT. If an SUT returned an error response, then its return value is simply zero.

Finally, every SUT's return value is added in a deterministic and consistent order to a single vector, which is passed to NEZHA. NEZHA compares this vector against the vectors obtained for every other HTTP/2 stream in the fuzzing session, and if the vector is unique, then the HTTP/s stream that induced it is added to the input corpus for further mutation.

Referring back to our motivating examples, this normalization step is what enables H2FUZZ to infer that the difference between (A) and (B) is the same as that between (D) and (E). For both pairs of requests, every field that is equivalent between requests is normalized to zero, and only the difference between the `Host` headers is preserved.

I. Fuzzer Output

Any time a new difference is observed by the NEZHA core, the HTTP/2 request that triggered it is written to the input corpus, and the HTTP/1 requests themselves are written altogether to a file in a separate directory, tagged such that it can be directly linked to the HTTP/2 file for post-processing.

IV. EXPERIMENTS

To evaluate the effectiveness of H2FUZZ at finding conversion anomalies, we run a series of experiments in which we fuzz a number of popular reverse proxy technologies and CDNs. We develop a large testbed of anomaly-detectors based on the findings of prior work and use it to detect anomalies in the HTTP/1 requests saved during these fuzzing sessions. We then describe our novel method of reducing this data to a minimal set of input categories that trigger the anomalies. Last, to understand how H2FUZZ compares against state-of-the-art tools, we fuzz the same set of proxies with FRAMESHIFTER, run our same analysis on its output, and compare the tools' outputs.

A. Subjects Studied

We fuzz 5 CDNs and 11 popular open-source reverse proxies that support HTTP/2. Table I shows the names of our subjects along with the tested versions, where applicable, which were the latest versions of each at the time of experimentation. Reverse proxies run as standalone Docker images on a local machine, while the CDNs are live systems running externally.

TABLE I
NAMES, TECHNOLOGIES, AND VERSIONS TESTED.

Name	Technology	Version
NGINX	Reverse Proxy	1.25.4
Caddy	Reverse Proxy	2.7.6
Apache	Reverse Proxy	2.4.58
Envoy	Reverse Proxy	1.21.6
HAProxy	Reverse Proxy	3.0
Traefik	Reverse Proxy	3.0
Varnish	Reverse Proxy	7.5.0
H2O	Reverse Proxy	2.2.6
Apache Traffic Server	Reverse Proxy	9.2.4
nghttp2	Reverse Proxy	1.61.0
OpenLiteSpeed	Reverse Proxy	1.7.19
Akamai	CDN	N/A
Cloudflare	CDN	N/A
CloudFront	CDN	N/A
Fastly	CDN	N/A
Azure	CDN	N/A

We run each proxy in as close to its default configuration as possible, making only the changes necessary to act as a reverse proxy to a specified domain. This configuration is in line with the experimental setup of prior work [17], [18], and to date there is no evidence that this impacts the forwarding behavior of the proxies being studied.

B. Ethical Concerns

When fuzzing live systems such as CDNs, we take measures to ensure that our experiments do not harm other Internet users or website owners. We create an account with each CDN and configure it to proxy our own machine running upstream from it. CDNs isolate requests destined for a specific domain, and we ensure that every request is directed only to a domain under our control.

We also configured each CDN not to cache requests or use connection pooling [5], ensuring that any dangerous requests generated while fuzzing will not impact any other users who are sending requests to our echo servers, if such users exist at all. Any anomalous behavior triggered by the tools we study will at worst only impact the throughput of the fuzzers themselves.

C. Corpus Creation

H2FUZZ requires a pre-populated intial corpus of structurally-valid HTTP/2 requests in order to run. To create this corpus, and to enable a fair comparison against FRAMESHIFTER, we populate the initial corpus with the same six base streams that FRAMESHIFTER's grammar uses when it builds streams.

These base streams, however, only contain `HEADERS`, `CONTINUATION`, and `DATA` frames. H2FUZZ is only capable of mutating frames already in its corpus, so to populate the corpus with other frame types for H2FUZZ to pull from and mutate, we generate 64 unique streams using FRAMESHIFTER's grammar, which we found experimentally was enough to ensure that at least two of every type of frame were present in the initial corpus. For each experiment run using H2FUZZ, we begin from this same set of 70 HTTP/2 streams.

D. Experiment Parameters

Modern fuzzing guidelines suggest that fuzzing experiments last at least 24 hours [20], [31] to allow fair comparison between tools. Given that execution throughput in black-box fuzzing of network applications is extremely poor [6], [32], however, we fuzz all of our systems under test for 72 hours in each experiment to maximize the behavior exercised and ensure fair comparison between tools. Additionally, to reduce the effect of randomness on the results, we run three full fuzzing sessions, and compare tools based on average results across all experiments.

E. Evaluation Metrics

The most common metrics by which fuzzers are compared are the amount of code covered and the number of crashes triggered in a given time span. In a black-box environment such as the one H2FUZZ assumes, however, code coverage is not available. Also, because the objective of H2FUZZ is not to crash the systems, but rather to have them forward anomalies, the number of crashes triggered by each fuzzer is also not a suitable evaluation metric.

Instead, for our evaluation, we choose to measure and analyze the number of unique ⟨anomaly, proxy⟩ pairs (referred to as "anomaly pairs"), representing a single anomaly pattern present in any HTTP/1 request forwarded by a specific proxy, found in the output of the fuzzer. To measure this, we first compile a set of 39 HTTP/1 request patterns from prior work on semantic gap attacks in HTTP applications [10], [17], [29], [33].

Every pattern described in these works led to at least one vulnerability between a proxy-server pair in the respective paper's analysis, so we refer to these patterns as "anomalies." The full list of anomalies in our evaluation, along with a short description of each, the papers that studied them, and the attacks that arose from them is reported in Table III in the Appendix.

We note that a proxy simply forwarding a request containing an anomaly is not necessarily an indicator of a vulnerability, or even a deviation from the protocol specification. Some, such as the "data after last chunk" anomaly, *are* obvious indicators of vulnerabilities, as any request with this anomaly contains data after the point that almost any upstream server would consider the end of the request, leading to HRS.

The "NULL in chunk-data" anomaly reported in [33], on the other hand, is not an inherently dangerous pattern at all. Chunk data in a chunked transfer coding is composed of arbitrary octets, and so a null byte is perfectly permissible.

Vulnerabilities arising from semantic gaps in HTTP applications are particular to the interaction of two specific systems. An input that causes a critical security vulnerability between a proxy and one upstream server may be completely benign with a different upstream server. The anomalies we report in Section V should thus not be understood as direct indicators of vulnerabilities in the tested systems. At minimum, they reflect the *expressiveness* of the requests generated by a fuzzer, and we separately discuss the security implications of particular anomalies when we evaluate input categories in Section V-B.

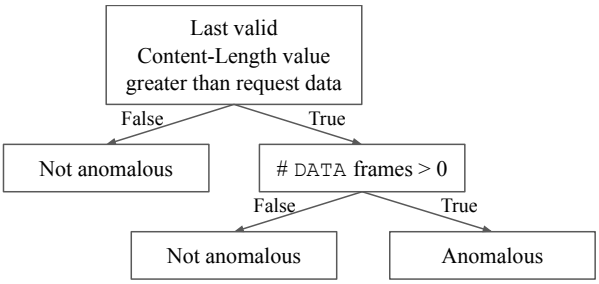

Fig. 4. Decision tree describing the request category that induces Envoy to forward a request with an Incomplete Content-Length With Body anomaly.

F. Input Categorization

While fuzzing, H2FUZZ outputs many thousands of HTTP/2 streams that trigger unique differences in the outputs of the SUTs. Each HTTP/2 stream is mapped to a separate file containing the HTTP/1 requests that were forwarded by the systems when processing that stream, and the anomaly checkers are run on these HTTP/1 requests. This step often finds thousands of instances of each type of anomaly, which is far too much for a human analysis to process themselves.

In their analysis of FRAMESHIFTER's output, Jabiyev et al. took random samples of the HTTP/2 streams that triggered anomalies and organized them into discrete input classes using ad-hoc scripting and manual analysis. For a complete analysis of the experiment and to enable a more effective investigation of the security implications of each anomaly, it must be possible to determine the minimal input categories discovered by the fuzzer that triggered each anomaly.

To accomplish this, in our analysis, we curate a set of 716 features that can be extracted from HTTP/2 streams. These features capture both structural information, such as the number of `DATA` frames in a stream, as well as semantic information, such as whether the value of the first `Content-Length` header matches the amount of data in all `DATA` frames before the `END_STREAM` flag.

For a given anomaly pair, we extract all HTTP/2 streams for which the proxy in the pair forwarded *any* request, and compute the features from these streams only. This comprises the training dataset. We then train a binary decision tree classifier on the dataset, where the training label represents whether the given anomaly was present in the HTTP/1 request forwarded by the proxy.

Because the objective is to learn patterns across the entire fuzzer output, we train our classifiers on the entire dataset. After model fitting completes, we evaluate the model on the entire dataset, and output a visualization of the final decision tree for a human analyst to examine. Figure 4 shows an example of one such decision tree. We choose to train decision tree classifiers precisely for this easy interpretability.

TABLE II
NUMBER OF UNIQUE ANOMALY PAIRS EXPOSED BY EACH EXPERIMENT.

Tool	Run 1	Run 2	Run 3	Total
H2FUZZ	57	54	51	60
FRAMESHIFTER	40	40	40	40
ONLY-SEQ	30	30	30	30
SEQ-CONN	26	26	26	26

In addition, the system evaluates the model's performance on the entire dataset. If the model's accuracy is not completely perfect, then it outputs a textual representation of any misclassified HTTP/2 streams for manual analysis and further feature engineering. After engineering additional features, the process repeats until the model has perfect accuracy.

Then, we construct minimal HTTP/2 streams based on the feature values indicated in the decision tree, send the streams to the target proxy, and capture its forwarded requests. If the anomaly is not present in any forwarded request, then we add the respective HTTP/2 stream to the dataset as a counterexample and repeat training from the start. Otherwise, we record the decision tree and stop.

G. FRAMESHIFTER *Configuration*

As the only other fuzzer designed to trigger HTTP/2-to-HTTP/1 conversion anomalies, the single system we compare H2FUZZ against is FRAMESHIFTER. FRAMESHIFTER's expressiveness is entirely dependent on its input grammar, which is completely customizable. To compare as closely as we can to the results described in the paper, we run FRAMESHIFTER using the exact grammar available in its open-source implementation at the time of experimentation [15].

FRAMESHIFTER has two operating modes used in the original work. In its `ONLY-SEQ` mode, the only mutations made are sequence mutations, in which frames are generated from the grammar and either inserted into a base stream or spliced over an existing frame in the stream. Its other mode is `SEQ-CONN`, in which both sequence mutations are applied to the base streams as well as content mutations, which insert random characters at the edges of certain specified headers.

Because each of FRAMESHIFTER's modes may discover different anomalies, we fuzz in both modes and report the anomalies triggered by the requests generated by them separately. We fuzz all 16 proxies with FRAMESHIFTER in both modes for 72 hours, and repeat the experiment three times, as with H2FUZZ.

Only one fuzzer runs at a time to reduce network load on the CDNs, but we cycle the fuzzer being run in each experiment in a round-robin fashion to limit the possible effects of a CDN's functionality changing before one tool could fuzz it, ensuring that each tool fuzzes as close to the exact same systems as possible.

V. RESULTS

A. Anomalies Discovered

Across all nine experiments, we discovered 66 unique anomaly pairs. Table II displays the total unique anomaly pairs observed in each of the nine experiments. Additionally, Figure 5 plots every unique anomaly pairs observed across all of the experiments, along with which tool discovered which pairs.

H2FUZZ found a total of 60 anomaly pairs across all three experiments, and found on average 54 pairs per experiment. Of these, 28 were not present in the output of any FRAMESHIFTER experiment. The majority (27) of these unique anomaly pairs pertained to the `Host` header of the forwarded request, a pattern largely attributable to the fact that FRAMESHIFTER's grammar is tailored towards triggering anomalies related to parsing the body of the forwarded request, and leaves minimal room for modifying the `:authority` header.

Every experiment running FRAMESHIFTER in a given mode found the exact same anomaly pairs. `ONLY-SEQ` found 30 anomaly pairs while `SEQ-CONN` found 26, with 40 unique anomaly pairs across both modes' experiments combined. All six anomaly pairs not found by H2FUZZ were found in the `SEQ-CONN` experiments, and of these, four were anomalies related to specific special characters being present in the `Content-Length` and `Transfer-Encoding` headers. The content mutations performed in the `SEQ-CONN` mode insert special characters into exactly these headers, so it is not surprising that it was able to trigger these anomalies.

Given that every individual H2FUZZ experiment found more anomaly pairs than both of FRAMESHIFTER's modes combined, and across all of its experiments found over twice as many anomaly pairs as the either of FRAMESHIFTER's modes running individually, it is clear that the expressiveness of the requests generated by H2FUZZ far exceeds that of FRAMESHIFTER, and results in substantially more anomalous behavior in the systems under test.

B. Input Categories and Security Implications

Due to the large number of anomaly pairs found by both tools, we cannot discuss each decision tree individually. Instead, we summarize broader input categories of HTTP/2 requests that caused conversion anomalies.

1) Missing `END STREAM`: Inputs that do not have an `END STREAM` flag tend to cause proxies to forward incomplete requests to the upstream server. When proxies receive a sequence of frames that does not contain an `END STREAM` flag, they often convert any frames received so far to HTTP/1 and forward them to the upstream. If the client never sends the remaining frames containing an `END STREAM` flag, then the data received at the upstream will remain incomplete.

This category was involved in 25 anomaly pairs across all experiments, and is one example of an input category that was identified by Jabiyev et al. in [17], but has evidently remained unpatched in every system they studied except for Varnish and Fastly.

2) No Mutation Filter: This is another input category identified by Jabiyev et al. in [17], and results in 30 of the 66 anomaly pairs in our dataset. Inputs of this category simply

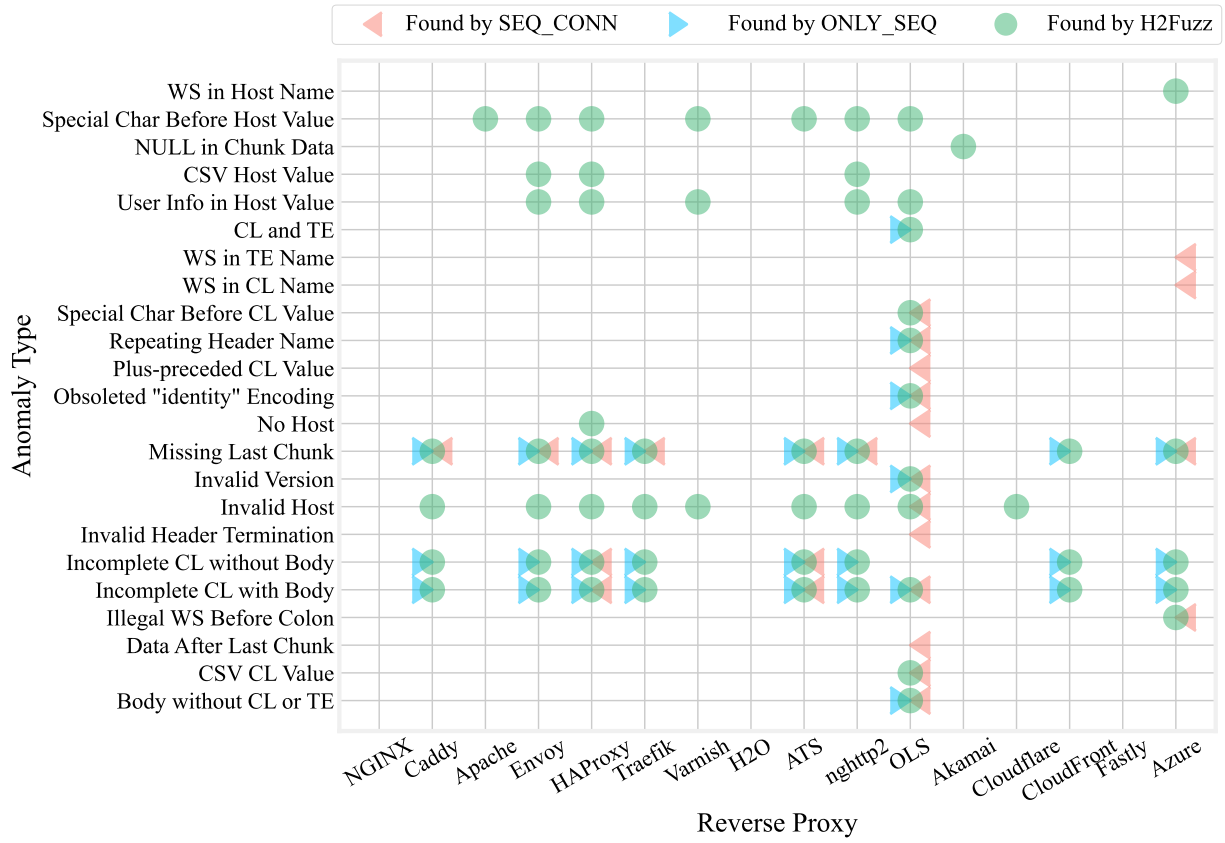

Fig. 5. Anomalies Discovered by H2Fuzz and Frameshifter. For brevity, OLS=OpenLiteSpeed and ATS=Apache Traffic Server.

have a mutation in a header field which the reverse proxy does not reject or remove from the request before forwarding it.

For example, when the value of the `:authority` header includes a non-numeric port value, Caddy, Traefik, and Cloudflare preserve the non-numeric port, despite RFC 3986 specifying that a port may only be a sequence of digits [8]. Such request patterns were identified by [33] to cause Host-of-Troubles attacks, as it may cause upstream servers to interpret the `Host` header's value incorrectly.

As another example, Azure is the only system we tested that forwards requests with whitespace in header names. RFC 9113 states that a header name "MUST NOT contain characters in the ranges 0x00-0x20, 0x41-0x5a, or 0x7f-0xff" [34]. However, Azure does not reject such requests and in fact preserves the whitespace in the forwarded HTTP/1 request.

Space-surrounded headers are a paradigm of semantic gap attacks on HTTP systems, as early HTTP RFCs did not have explicit text for whether to accept or reject headers with certain types of surrounding whitespace [10]. Adding whitespace to a header also bypasses Azure's check for duplicate header names, causing it to forward requests with multiple `Content-Length`, `Transfer-Encoding`, and `Host` headers with spaces surrounding the names. Such ambiguous requests are highly likely to result in HRS and Host-of-Troubles attacks.

3) No Mismatch Check: Inputs of the "no mismatch check" category have semantically-valid values for all fields, but where two fields that *together* affect the parsing of the HTTP/1 request have mismatched values, and the SUT forwards a request without detecting or correcting the values. Such inputs effect three anomaly pairs in our experiments.

For example, the "body without CL/TE" anomaly occurs in requests forwarded by OpenLiteSpeed when an input stream contains at least one non-empty `DATA` frame before the `END_STREAM` flag, but has no `Content-Length` or `Transfer-Encoding` header. Such streams are converted directly to HTTP/1 without detecting the missing header, resulting in requests with bodies but no headers that would indicate the presence of a body.

Requests with this anomaly lead to HRS with any upstream server, as without a `Content-Length` or `Transfer-Encoding` header in the request, the server will stop processing the request at the end of the headers block. However, OpenLiteSpeed will still have sent the request body, which the upstream server will read and process as if it were another incoming HTTP/1 request.

4) Mixed :authority and Host: Inputs of this category contain both an :authority and a Host header with differing values. RFC 9113 states that the :authority field "conveys the authority portion ... of the target URI," and that "an intermediary that needs to generate a Host header field (which might be necessary to construct an HTTP/1.1 request) MUST use the value from the ':authority' pseudo-header field." The RFC is also clear that "an intermediary that forwards a request over HTTP/2 MAY retain any Host header field" but has no clear language on how to process explicit Host header fields when converting to HTTP/1.

When a stream contains both an :authority and a Host header, most proxies reject the request, but Envoy forwards a request where both headers' values are combined as a comma-separated list in a single Host header value, resulting in one unique anomaly pair in our results. Upstream servers receiving such requests may accept either value as the authority portion of the target URI, and this ambiguity can lead to a Host-of-Troubles attack, as demonstrated in [33].

Among the proxies we study specifically, we find that when HAProxy or OpenLiteSpeed is configured as the upstream from Envoy and receive the anomalous HTTP/1 request, they take the value from the Host header as the authority.

5) Flawed Duplicate Validation: Another input category, which results in six anomalies with OpenLiteSpeed, involves what we call a "flawed duplicate validation." Input streams in this category have two or more headers with the same name, and have particular character mutations in any of the duplicate headers except for the first in the stream.

For example, when a stream contains two or more content-length headers, OpenLiteSpeed does not reject the request and also does not apply validation to any of the headers except for the first one. The mutated content-length headers are included as-is in the forwarded HTTP/1 request, enabling requests with content-length values that do not match the body, comma-separated content-length values, and even entirely non-numeric content-length values.

Additionally, when a stream has a second :method header and the value of the header is a valid HTTP method followed by any amount of whitespace, various copies of the :method header name and its value overwrite data elsewhere in the HTTP/1 request. The exact location of the copies depends on the amount of whitespace following the method. If a copy appears in the request line, it can result in an invalid method or version. If it appears in the headers block, it can overwrite other headers, resulting in requests that lack a Host header.

6) Pad Length Overflow: A final category we discuss was the only category for which our automated input categorization methodology could find no pattern, as the anomaly is entirely nondeterministic. The input category, which we call "pad length overflow," affects Akamai, and occurs when any DATA frames in the input stream have a payload of at least one byte and have the PADDED flag set in the frame header.

In such cases, we observe that each DATA frame payload is converted into one chunk in the chunk-encoded body of the forwarded HTTP/1 request, but the first byte of the chunk is set to the value of the pad length field of the DATA frame. When this anomaly occurs, it occurs in every chunk in the forwarded request that was derived from a padded DATA frame.

If the input stream contains a sequence of padded DATA frames, each of which carries a single byte of data, then the entire request body in the forwarded request is derived from the pad length fields rather than the data itself. If any kind of transparent middlebox between the client and Akamai would inspect the stream and block it based on the contents of the body (for example, a censor looking for specific keywords or a web application firewall looking for attack payloads), then the request would pass unimpeded, and the payload would then be reconstructed from the pad length fields in the upstream HTTP/1 request.

VI. Discussion

A. Comparison with State-of-the-Art

In this paper, we developed a comprehensive HTTP/2 stream mutator and a novel black-box differential checker for HTTP/1 requests that can be used as feedback to a fuzzer to drive a set of HTTP/2 reverse proxies to increasingly divergent, anomalous behavior. The purpose of this was to identify HTTP/2-to-HTTP/1 conversion anomalies that could not be discovered by the state-of-the-art fuzzer for such anomalies, FRAMESHIFTER.

Based on our metrics alone, we certainly succeeded in this effort. Across all experiments, H2FUZZ causes 50% more conversion anomalies than FRAMESHIFTER's ONLY-SEQ and SEQ-CONN modes combined, and each individual H2FUZZ experiment causes on average 35% more anomalies.

It is worth noting that nearly all of the anomalies unique to H2FUZZ pertained to the Host header in the forwarded HTTP/1 requests. While these anomaly categories are important and security-relevant, mutating the :authority header was considered out of scope for FRAMESHIFTER's investigation, and so comparing the tools based purely on the total anomalies discovered across all experiments is not necessarily sufficient to prove one tool's *methodological* improvement over another. However, in light of a deeper analysis of our results and FRAMESHIFTER's functionality, we argue that H2FUZZ's performance over FRAMESHIFTER *is* significant, and a byproduct of the fuzzing methodology itself.

First, simply modifying FRAMESHIFTER to make mutations to the :authority header would not be enough to expose the same anomalies as H2FUZZ. The anomalies found by H2FUZZ rely on input patterns that exercise user data in Host values, invalid port values, mismatches between the :authority header and :path URI, and the inclusion of both :authority and Host headers to the input stream. Modifying FRAMESHIFTER to be capable of generating such requests would require an overhaul of its entire mutation system, as it would need to be capable of substring mutations as opposed to simple byte-insertion mutations, as well as integrate an external dictionary of host-relevant values to apply to specific headers.

Additionally, both the ONLY-SEQ and SEQ-CONN experiments found unique anomalies from one another. ONLY-SEQ finds 13 anomalies that SEQ-CONN does not, and SEQ-CONN finds 11 that ONLY-SEQ does not. Yet combined, only eight anomalies, all of which are from the SEQ-CONN experiments, are not found by H2FUZZ. H2FUZZ therefore demonstrates the expressiveness of both modes in one, finding more anomalies in the same time span on the same initial corpus, all while making fewer assumptions about what fields should be mutated and how.

B. Limitations and Future Work

Though the novel feedback mechanism introduced in H2FUZZ results in finding substantially more anomalies than FRAMESHIFTER, it is still far from perfect. One major limitation of H2FUZZ stems from the fact that the fuzzer core, NEZHA, is built on libFuzzer. LibFuzzer is an unsophisticated fuzzer engine which uses uniform random selection to both draw inputs from the input corpus and choose which mutation operator to apply to the input.

More sophisticated input selection algorithms can substantially improve a fuzzer's performance [30], and smart mutators such as the one implemented in SGFuzz [7] can prioritize mutations on specific regions of the input that were observed to lead to new behavior in prior execution cycles. Given the extremely poor execution throughput when fuzzing network applications, spending more local execution time deciding what to mutate, and how, could help save the time wasted executing useless, slow inputs.

Additionally, at runtime, H2FUZZ reasons about the behavior of the tested systems in terms of general differences in their forwarded requests, and we later process these requests to look for anomalies. Systems that can detect *specific* aberrant behavior at runtime are better able to drive the search for that behavior and provide concise, easily-analyzable outputs. Geneva [9], for example, is a tool designed to find censorship evasion strategies via mutations to TCP packets. Geneva learns minimal mutation strategies that, when applied to an outgoing request, enable the request to bypass a censor. Prior work saw great success applying Geneva to HTTP [14], so directing Geneva at a reverse proxy and modifying it to detect specific anomalies in the forwarded HTTP/1 requests could allow for finding minimal input categories at runtime, avoiding the need for any offline processing.

Finally, in the still-nascent field of research into semantic gap attacks for HTTP systems, all published approaches use some form of fuzzing followed by offline processing to detect attack vectors. These approaches work well, given the need for black-box systems to study CDNs, but notoriously struggle to find specific edge-cases or complex patterns in requests, and also provide no guarantees of the completeness of the analysis. Methods rooted in formal verification or symbolic execution of the proxies could be a promising direction to address both aforementioned issues with fuzzing.

C. Vendor Disclosure and Responses

We reported all immediately security-relevant anomalies discussed in Section V-B to their respective vendors. Akamai acknowledged our findings and patched the pad-length overflow bug. OpenLiteSpeed thanked us for our report and has thus far fixed all but one anomaly. We are working with them to address the remaining one. Azure likewise acknowledged our findings and we are working with them to complete a patch. Envoy and Cloudflare patched their own anomalies in the time between our experiments and us contacting them. For the remaining anomalies, we have not received a response from the vendors but look forward to hearing from them.

VII. CONCLUSION

In this paper, we presented H2FUZZ, a guided, black-box differential fuzzer designed to trigger HTTP/2-to-HTTP/1 conversion anomalies. We fuzzed 16 popular reverse proxy technologies with H2FUZZ and the state-of-the-art fuzzer for such anomalies, FRAMESHIFTER, and found that H2FUZZ discovers 50% more anomalies than FRAMESHIFTER, many of which have immediate security implications.

VIII. ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their thorough reviews and insightful comments. We especially thank our shepherd for their guidance and timely feedback. We would also like to thank Bahruz Jabiyev and Ryan Williams for their help with early brainstorming. This work has been supported by NSF grant 2329540.

REFERENCES

[1] Stored xss on https://paypal.com/signin via cache poisoning. https://hackerone.com/reports/488147, 2019.

[2] Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/, 2021.

[3] boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz, 2025.

[4] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2025.

[5] Persistent connections: Edge to origin. https://techdocs.akamai.com/property-mgr/docs/persistent-connections-edge-to-origin, 2025.

[6] Anastasios Andronidis and Cristian Cadar. Snapfuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pages 340–351, 2022.

[7] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3255–3272, 2022.

[8] Tim Berners-Lee, Roy T. Fielding, and Larry M. Masinter. Uniform resource identifier (uri): Generic syntax. https://datatracker.ietf.org/doc/html/rfc3986, 2005.

[9] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. Geneva: Evolving censorship evasion strategies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2199–2214, 2019.

[10] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1516–1527, 2016.

[11] Roy T. Fielding and Julian F. Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing. https://datatracker.ietf.org/doc/html/rfc7230, 2014.

[12] Andrew Galloni, Robin Marx, and Mike Bishop. Web almanac. https://almanac.httparchive.org/en/2020/http, 2020.

[13] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*, pages 330–347. Springer, 2015.

[14] Michael Harrity, Kevin Bock, Frederick Sell, and Dave Levin. GET /out: Automated discovery of Application-Layer censorship evasion strategies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 465–483, 2022.

[15] Bahruz Jabiyev. Frameshifter. https://github.com/bahruzjabiyev/frameshifter/tree/f9ebb13e46789e127225a4723784bf74c6b5fcc0, 2022.

[16] Bahruz Jabiyev, Anthony Gavazzi, Kaan Onarlioglu, and Engin Kirda. Gudifu: Guided differential fuzzing for http request parsing discrepancies. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 235–247, 2024.

[17] Bahruz Jabiyev, Steven Sprecher, Anthony Gavazzi, Tommaso Innocenti, Kaan Onarlioglu, and Engin Kirda. FRAMESHIFTER: Security implications of HTTP/2-to-HTTP/1 conversion anomalies. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1061–1075, 2022.

[18] Bahruz Jabiyev, Steven Sprecher, Kaan Onarlioglu, and Engin Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, 2021.

[19] James Kettle. Http/2: The sequel is always worse. https://portswigger.net/research/http2, 2021.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[21] Johannes Krupp, Ilya Grishchenko, and Christian Rossow. AmpFuzz: Fuzzing for amplification DDoS vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1043–1060, 2022.

[22] Chaim Linhart, Amit Klein, Ronen Heled, and Steven Orrin. Http request smuggling. https://cgisecurity.com/lib/HTTP-Request-Smuggling.pdf, 2005.

[23] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, 2023.

[24] Robin Marx. Web almanac. https://almanac.httparchive.org/en/2024/http, 2024.

[25] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. Your cache has fallen: Cache-poisoned denial-of-service attack. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1915–1936, 2019.

[26] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[27] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[28] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–26, 2023.

[29] Jannis Rautenstrauch and Ben Stock. Who's breaking the rules? studying conformance to the http specifications and its security impact. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 843–855, 2024.

[30] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[31] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024.

[32] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 166–180, 2022.

[33] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13. IEEE, 2022.

[34] Martin Thomson and Cory Benfield. Http/2. https://www.rfc-editor.org/rfc/rfc9113, 2022.

APPENDIX

TABLE III
ANOMALIES EXAMINED IN OUR EVALUATION, THE PRIOR WORKS THAT STUDIED THEM, AND THE ATTACKS THEY HAVE CAUSED.

Anomaly Name	Description	Studied By				Attacks Caused
		[17]	[29]	[33]	[10]	
Incomplete CL without Body	Request has a nonzero Content-Length value but no body	✓				DoS
Incomplete CL with Body	Request has a body and a nonzero Content-Length value greater than the body size	✓				DoS
Missing Last Chunk	Chunk-encoded body is missing the terminal zero-length chunk	✓				DoS
Missing Chunk Termination	Chunk-encoded body is missing a terminating CRLF after chunk data	✓				DoS
Missing Chunk Size Termination	Chunk-encoded body is missing a terminating CRLF after a chunk size	✓				DoS
Missing Chunk Data	Chunk-encoded body has a size, but no data follows it	✓				DoS
Invalid CL Value	Content-Length value is non-numeric	✓				HRS
Invalid Header Termination	Any header is terminated with a single LF rather than a CRLF	✓		✓		HRS
Repeating Header Name	Request has two or more Content-Length headers	✓		✓		HRS
Repeating Header Value	Transfer-Encoding value contains two “chunked” values separated by a comma	✓				CPDoS
Multiple Forwarded Requests	Multiple HTTP requests are forwarded for a single HTTP/2 stream	✓				HRS
Data After Last Chunk	There is data after the termination of the last chunk in a chunk-encoded body	✓				HRS
Body without CL or TE	Request has a body but neither a Content-Length nor Transfer-Encoding header	✓				HRS
No Host	Request has no Host header		✓			HoT
Multiple Host	Request has more than one Host header field line		✓		✓	HoT
Invalid Host	Any Host value is not a valid authority per RFC 3986		✓			HoT
Illegal Characters in Header	Any request header contains a NULL or newline character		✓			HRS
Illegal WS After Request Line	The line immediately after the request line begins with whitespace		✓			HoT
Illegal WS Before Colon	A space or horizontal tab exists between any header name and the separating colon		✓	✓		HoT
TE Without HTTP/1.1	Request as a Transfer-Encoding header with a version less than HTTP/1.1		✓	✓		HRS, CPDoS
WS in Host Name	A Host header name is surrounded by spaces and/or horizontal tab characters			✓	✓	HoT
Bad Absolute-URI vs Host	Request target is an absolute URI with a different Host component than the value of the Host header			✓	✓	HoT
Invalid Version	The protocol version in the request line is either unparsable or not 0.9, 1.0, or 1.1			✓		CPDoS
Plus-preceded CL Value	The first non-whitespace character in a Content-Length value is a plus (+) character			✓		HRS
CSV CL Value	The value of a Content-Length header is a comma-separated list			✓		HRS
Special Char Before CL Value	The first non-whitespace character in the value of any Content-Length header is not an alphanumeric character			✓		HRS
WS in TE Name	A Transfer-Encoding header name is surrounded by spaces and/or horizontal tab characters			✓		HRS
WS in CL Name	A Content-Length header name is surrounded by spaces and/or horizontal tab characters			✓		HRS
User Info in Host Value	A Host header’s value is preceded by user information. e.g., data@example.com			✓		HoT, CPDoS
CSV Host Value	The value of a Host header is a comma-separated list			✓		HoT, CPDoS
Special Char Before Host Value	The first non-whitespace character in the value of any Host header is not an alphanumeric character			✓		HoT, CPDoS
Expect 100-continue with GET	The request contains a “Expect: 100-continue” header with a GET method			✓		HRS, CPDoS
Connection Host	There exists a Connection header with value “Host”			✓		CPDoS
Connection Cookie	There exists a Connection header with value “Cookie”			✓		CPDoS
Obsoleted “identity” Encoding	There exists a Transfer-Encoding header with value “identity”			✓		HRS, CPDoS
NULL in Chunk Data	A null byte exists in a chunk-encoded body			✓		HRS
Fat HEAD or GET	Request has a body and the method is HEAD or GET			✓		HRS
Bad Chunk Size	Any chunk size in a chunk-encoded body is not a valid hexadecimal number			✓		HRS
CL and TE	Request has a both a Content-Length and Transfer-Encoding header			✓		HRS