Untangle: Multi-Layer Web Server Fingerprinting

Cem Topcuoglu*, Kaan Onarlioglu†*, Bahruz Jabiyev*‡, and Engin Kirda*
Northeastern University*, Akamai Technologies†, Dartmouth College‡

***Abstract*—Web server fingerprinting is a common activity in vulnerability management and security testing, with network scanners offering the capability for over two decades. All known fingerprinting techniques are designed for probing a single, isolated web server. However, the modern Internet is made up of complex layered architectures, where chains of CDNs, reverse proxies, and cloud services front origin servers. That renders existing fingerprinting tools and techniques utterly ineffective.**

We present the first methodology that can fingerprint servers in a multi-layer architecture, by leveraging the HTTP processing discrepancies between layers. This technique is capable of detecting both the server technologies involved and their correct ordering. It is theoretically extendable to any number of layers, any server technology, deployed in any order, but of course within practical constraints. We then address those practical considerations and present a concrete implementation of the scheme in a tool called `Untangle`, empirically demonstrating its ability to fingerprint 3-layer architectures with high accuracy.

I. Introduction

Web server fingerprinting is a standard exercise in security testing, vulnerability management, and network observability. Accordingly, popular network scanners have been equipped with this capability, and both tool makers and academics have explored the domain for more than two decades (e.g., [8], [18], [30], [32]). The fingerprinting techniques in literature inspect HTTP responses for well-known strings and server quirks, with no breakthrough technique presented since the early 2000s.

This approach is no longer sufficient in the modern Internet, where web applications are typically deployed behind multiple layers of proxy servers–load balancers, caches, firewalls, anomaly detection systems, and similar middle HTTP processors–that intercept the traffic for performance and security services. In particular, Content Delivery Networks (CDNs) have become critical infrastructure for scalable applications; according to the data presented by BuiltWith as of June 2023, 64% of the top 10K sites are fronted by a CDN service [3].

Existing fingerprinting techniques are not capable of tackling such complex infrastructures of layered servers. Specifically, HTTP request probes generated by a fingerprinting tool, and the resulting response, may be processed and transformed by any combination of servers on the traffic path. This renders the established server behavior analysis methods ineffective, even at correctly identifying the client-facing server, let alone the infrastructure hidden behind it.

This is particularly problematic in the face of the surging systems-centric web application attacks, including HTTP request smuggling, web cache poisoning, web cache deception, and HTTP2 protocol downgrade exploits (e.g., [7], [12], [13], [15]). These issues fundamentally result from *HTTP processing discrepancies* between different servers, as opposed to bugs that can be squished by a single technology vendor. The research community has built on this work and demonstrated that automated discovery of novel vulnerabilities impacting any given server combination is viable–and very effective [9], [10], [22]–[24]. Thus, it is crucial that security professionals adapt to this new threat model, and employ technologies that can accurately identify and test layered servers.

In this work, we propose the first web server fingerprinting technique that addresses this problem. Our methodology is rooted in the same observation that enables the aforementioned discrepancy attacks: The modern web is a patchwork of HTTP processors, each with distinctive behavior. Given an understanding of this behavior differential, it may be possible to devise a strategy where the target infrastructure is probed with carefully crafted requests, each leaking information about a particular server layer, until all layers are successfully identified. The effectiveness of this methodology is correlated with the prevalence of processing discrepancies between the server technologies in scope, and our ability to discover them; but in theory, the process is extendable to any server technology deployed in any layered configuration.

We implement this methodology in a prototype we call `Untangle`, and evaluate it over all practicable 3-layer permutations of 13 popular proxy technologies: Akamai, Cloudflare, CloudFront, Fastly, NGINX, Varnish, HAProxy, Apache, Caddy, Envoy, ATS, Squid, and Tomcat. `Untangle` leverages HTTP fuzzing techniques to automatically exercise each server under investigation and create a behavior repository, and then uses this information to craft the appropriate requests that fingerprint each layer. In 756 experiments, `Untangle` was able to correctly identify **all** servers in the first layer, **90.3%** of the second layer, and **50.7%** of the final layer, demonstrating that our novel methodology is viable.

In summary, we make the following contributions:

- We propose the first methodology in literature that can fingerprint multi-layer web servers by leveraging the HTTP processing discrepancies between them. Our methodology is generic, extendable to any layering of servers and any number of layers.
- We implement `Untangle`, a prototype that leverages HTTP fuzzing to discover processing discrepancies and our methodology to perform the fingerprinting.

†This work was performed solely at Northeastern University.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24497
www.ndss-symposium.org

- We evaluate `Untangle` with 3-layer permutations of 13 popular proxy and server technologies, experimentally demonstrating that the approach is viable and effective.

Availability. `Untangle` is open-source and publicly available on the authors' websites.

II. Background and Research Overview

A. HTTP Proxies and Discrepancy Attacks

Reverse proxies that provide performance and security services such as caching, load balancing, and traffic filtering are mainstays of modern web application architecture design. In particular, due to the proliferation of Content Delivery Networks (CDNs) and public cloud services with their massively distributed Internet overlay networks, and complex infrastructures that combine all of the aforementioned technologies together, a client request often traverses multiple HTTP processors on its way to its ultimate destination, the origin server. Note that all reverse proxies are necessarily HTTP servers themselves, and therefore we will refer to both the middle processors and the origin technology as *servers* for brevity.

While this increasing architectural complexity is essential for scalable and performant web services, it has also led to a steady stream of security issues over recent years. The attacks follow a common pattern: When there are two servers on the traffic path that process the same HTTP message in different ways, this discrepancy could be abused to harm the Internet users and/or the application owners. We refer to all issues that fall under this category as *discrepancy attacks.*

For instance, Omer Gil coined the term *Web Cache Deception* for an attack that involves exploiting path confusion vectors between a caching server and an origin, which results in confidential data leaking into a public cache [7]. Mirheidari et al. followed up on this work with two large-scale Internet measurements and new path confusion vectors, demonstrating that the issue is widespread [22], [23].

Cache poisoning attacks instead exploit processing discrepancies to cache a malicious payload, later to be served to subsequent clients, with a plethora of techniques documented online; for instance, see the popular works by James Kettle [12], [14]. In academia, Chen et al. presented poisoning attacks that result from inconsistent processing of the `Host` header in a request [4]. Nguyen et al. explored the issue from a different angle, and showed that discrepancies could be abused to cache an error response in place of the requested object, in effect causing a denial-of-service on the victim site [24].

HTTP Request Smuggling (HRS) is another damaging attack, originally documented in 2005, but now resurging due to the Internet's increasing complexity [13], [16], [19]. HRS abuses discrepancies that lead to confusion around HTTP message boundaries, allowing attackers to smuggle hidden requests through a proxy. This has been shown to facilitate cache poisoning and a wide array of application specific attacks. Jabiyev et al. later explored HRS within a scientific framework, exercising server pairs via differential fuzzing to automatically discover discrepancies, and showed that HRS vectors are ubiquitous, affecting every technology under test [10].

Finally, James Kettle tackled HTTP2-to-HTTP1 protocol conversions performed by virtually all middle processors, leading to yet another opportunity for eliciting discrepancies [15]. Jabiyev et al. followed up and investigated the same at a larger scale, again demonstrating that the problem is far-reaching [9].

Initial attempts at mitigating discrepancy attacks involve loose heuristics, only suitable for specific attack scenarios, and without scientific evaluation [1], [5]. To date, there is no comprehensive defense against discrepancy attacks documented in literature. Analyzing whether a deployment is exposed to discrepancy attacks is a complex, open research problem. This is exacerbated by the fact that there could be more than two servers involved in traffic delivery, and a discrepancy between any pair involved could expose a vulnerability.

Consequently, the referenced prior works emphasize that discrepancy attacks are *safety* problems resulting from hazardous interactions between technologies that may otherwise be flawless when analyzed in isolation–individual technology vendors cannot address this problem by hardening their products. System owners must establish an asset management strategy that accurately tracks all servers involved in their deployments. Security teams must utilize this information and check for hazardous server interactions in their testing.

B. Web Server Fingerprinting

Web server fingerprinting is a routine exercise in security management, often utilized for tasks such as automated asset discovery in an enterprise network, or for vulnerability tracking via matching published CVEs to servers. Penetration testers also utilize fingerprinting for reconnaissance and attack surface discovery. As such, fingerprinting has long become a standard capability of popular network and vulnerability scanners such as Nmap, Nessus, httprint, and httprecon [8], [21], [30], [32].

The techniques utilized by existing fingerprinting tools follow a common pattern: The tool probes the target with a set of valid and invalid requests, analyzing the responses for server characteristics such as a specific ordering of the response headers or unique errors. Banner grabbing is also commonplace, in particular, examining the `Server` response header. Lee et al. presented one of the earliest works that documented these techniques in academic literature, and implemented a tool called HMAP that performs lexical, syntactic, and semantic analyses of HTTP responses [18].

Fingerprinting has a similar utility for miscreants, for example, making it possible to scan the Internet for exposed vulnerable servers. The HTTP specification, in particular, RFC 9110, Section 10.2.4 calls out this possibility, and recommends that "An origin server SHOULD NOT generate a Server header field containing needlessly fine-grained detail[...]. Overly long and detailed Server field values [...] potentially reveal internal implementation details that might make it (slightly) easier for attackers to find and exploit known security holes." [6]. Today, it is common security practice to remove such headers, or to place the web server behind a hardened proxy to hamper fingerprinting [25]. Kar et al. investigated the effectiveness of server masking against 8 fingerprinting tools run on 4 popular HTTP servers, and confirmed that simple obfuscation techniques could indeed deter accurate detection [11].

C. Fuzzing

Fuzzing is a well-established software testing technique that involves automatically generating inputs using a grammar or static corpus, mutating them, and injecting these into a system to reveal defects. A variation on this idea, differential fuzzing, aims to find bugs by sending identical inputs to a set of systems, and observing the discrepancies in their behavior.

Researchers have also applied differential fuzzing to the security domain. For instance, by using differential fuzzing, Bernhard et al. [2] discovered JavaScript engine bugs, Reen and Rossow [29] developed DPIFuzz to detect techniques for evading deep packet inspection, Petsios et al. [26] developed a generic differential testing tool NEZHA to find semantic bugs in a wide set of applications.

Most relevant to our work, recent research has applied differential fuzzing to find HTTP parsing discrepancies. Jabiyev et al. developed an open-source grammar-based fuzzer called T-Reqs, which they used to search for parsing discrepancies in HTTP request bodies that could lead to HTTP Request Smuggling [10]. Shen et al. developed HDiff to search for discrepancies that cause `Host` header confusion and cache poisoning, in addition to smuggling [31].

D. Research Statement

In this work, we draw from the above, seemingly disjointed research domains to tackle a major limitation of all existing web server fingerprinting tools and techniques: **State-of-the-art fingerprinting techniques were not designed for the modern Internet where web applications are deployed behind multiple layers of HTTP processors, and therefore, they cannot accurately identify such infrastructure.**

Foremost, existing techniques have no concept of layered servers; they report a single detection result, fundamentally incapable of capturing the complexity of a multi-layer architecture. Interestingly, existing techniques cannot reliably fingerprint the client-facing server (i.e., the first layer) either. This is due to all the servers on the traffic path transforming and rewriting the request in unpredictable ways, rendering the traditional banner-grabbing and response analysis techniques that focus on a single server's characteristics ineffective, presumably in the same way response obfuscation hinders detection. We empirically demonstrate this point in Section VI.

This current state puts security practitioners in a difficult position, crippling the automated discovery, observability, and testing tasks that have been the backbone of security management. The challenge has now come to the forefront with the surging discrepancy attacks targeting layered architectures, which calls for excellent situational awareness as emphasized by the referenced works.

In our pursuit to address this problem, we leverage the same observation that forms the basis of discrepancy attacks, but turn it on its head. We hypothesize, **if web servers commonly exhibit request processing discrepancies with measurable outcomes, then it should be possible to craft requests that elicit distinct responses for different combinations of those servers.** This hypothesis does not imply that there should exist an all-powerful request that can uniquely identify an arbitrary multi-layer server architecture, but instead, that we should be able to find multiple requests that gradually leak information about the servers and their ordering. Hence, we can iteratively fingerprint the entire system. In summary, we explore one overarching research question: **Is it possible to detect multi-layer web servers by utilizing HTTP parsing discrepancies?**

In Section III, we first present a generic and extendable methodology to perform the task for any server technology, layered in any order, and for any number of layers. At this stage, we assume a hypothetical, comprehensive behavior repository that captures all processing discrepancies displayed by all servers. Of course, this is not feasible in practice, and the real-world performance of a tool based on our methodology would be correlated to the accuracy and completeness of the behavior repository available to us. Hence, in Sections IV and V, we further describe a concrete implementation of this methodology and how fuzzing can be employed to automatically build an incomplete, yet viable, behavior repository.

III. MULTI-LAYER FINGERPRINTING METHODOLOGY

Our fingerprinting strategy involves probing *each individual layer* with a specially crafted request and analyzing the response. Error responses are particularly useful for this task–they significantly differ between server implementations, with various status codes, error reasons, non-standard headers, unique bodies–and therefore this is what we exclusively focus on. However, in practice we can only directly interact with the client-facing server of the target. The novel challenge then lies in correctly determining *which layer* in the multi-layer target responds to our probes with the error we observe, so that we can attribute the result to the correct layer.

For the rest of this discussion, we let:

Servers be the set of all possible server technologies.
Behavior(request, server) $\rightarrow$ **{*Error, Pass, Other*}**, be a function indicating how a given server behaves when probed with a request, where ***Error*** means the server responds with an error, and ***Pass*** means the server forwards the request without any modifications. ***Other*** covers all cases where requests are modified before passed forward, or processed by the terminal origin server with a success response.
Match(response) $\rightarrow$ **{*server* $\in$ *Servers, Unknown*}** be a function that maps a given error response to a unique server.
Ordered be an ordered list of servers identified with the correct layer ordering.
Unordered be an unordered set of servers, where the servers are identified, but the layer order has not been decided.

As a precondition for implementing our methodology, we assume the availability of a behavior repository that provides us with the two functions *Behavior* and *Match* defined above. We discuss how such a repository can be automatically generated for our prototype implementation `Untangle` in the later sections; however, for this abstract description of the methodology, those details are not pertinent.

The fingerprinting process follows three phases:

1) We iteratively fingerprint each layer in order, via probe requests that guarantee eliciting an error response from the first undetected layer. In the ideal scenario where the behavior repository is *complete*, this concludes the fingerprinting process, where *Ordered* contains the results.

2) Any time during Phase 1, if we cannot find a request that meets the criterion of eliciting an error response from the immediate next layer, we relax this condition and use probes that can trigger an error response from *any* subsequent layer. This identifies the servers in the subsequent layers, but without specifying their order. This yields the *Unordered* set.
3) Finally, by combining the findings from Phases 1 and 2, we perform a refinement pass over *Unordered* to determine their correct ordering.

We next describe each of these phases in more detail. The full methodology is available in Algorithm 1.

A. Phase 1: Fingerprint In Order

The basic scheme is based on the following high-level idea: Given a target multi-layer architecture and a universal set of N possible server types making that up, if we probe the target with a request for which all N servers are known to return an error response, we are guaranteed to observe the error response returned by layer 1, the client-facing server. We can then match this error response to a specific server technology, completing the fingerprinting for the first layer. Next, we repeat this process, instead picking a probe request that gets forwarded by the detected layer intact, but returns an error response from all remaining $N-1$ servers, guaranteeing us an error response from layer 2, which we can again match to a known server technology.

That concludes the multi-layer fingerprinting for a 2-layer architecture, but the scheme extends to any number of layers. We depict the general scheme in Algorithm 1 lines 1 through 18, which is a near one-to-one mapping of the above explanation to our terminology. We start with the $Ordered$ list empty. For each layer, if a request exists where for all s in $Servers \backslash Ordered$, $Behavior(request, s) \rightarrow Error$, and for all s in $Ordered$, $Behavior(request, s) \rightarrow Pass$, we send this request to the target infrastructure and expect an error response from the next immediate undetected layer. We analyze and match this error response to one of the known server technologies, or an *Unknown* server if the analysis fails to find a match. We append the fingerprinted server to $Ordered$ and repeat until all layers are detected.

Three conditions will terminate the fingerprinting: 1) We are unable to find a request that meets the conditions for eliciting an error response from the right set of servers, 2) we detect that the next server is unknown, or 3) we receive a success response, not an error. The first two conditions are tied to the completeness of the behavior repository.

In the ideal case where the behavior repository contains sufficient discrepancy and identifying error response information, Phase 1 successfully completes the fingerprinting. $Ordered$ contains an accurate list of all layers; Phase 2 and 3 are not necessary. We visually demonstrate an end-to-end 3-layer instance of how this scenario plays out in Figure 1–this is a real `[Cloudflare, Squid, Tomcat]` fingerprinting example we select from our experiments.

B. Phase 2: Fingerprint Without Order

In many practical scenarios, the behavior repository will be incomplete, and Phase 1 may fail to complete the task due to a lack of probe requests that meet our requirements. This leaves us with a number of layers correctly identified in $Ordered$, and Phase 2 aims to fingerprint the servers in the remaining layers. Specifically, we now relax the requirements for selecting probes, with the trade-off being that we can only detect the server technologies present, but not their ordering.

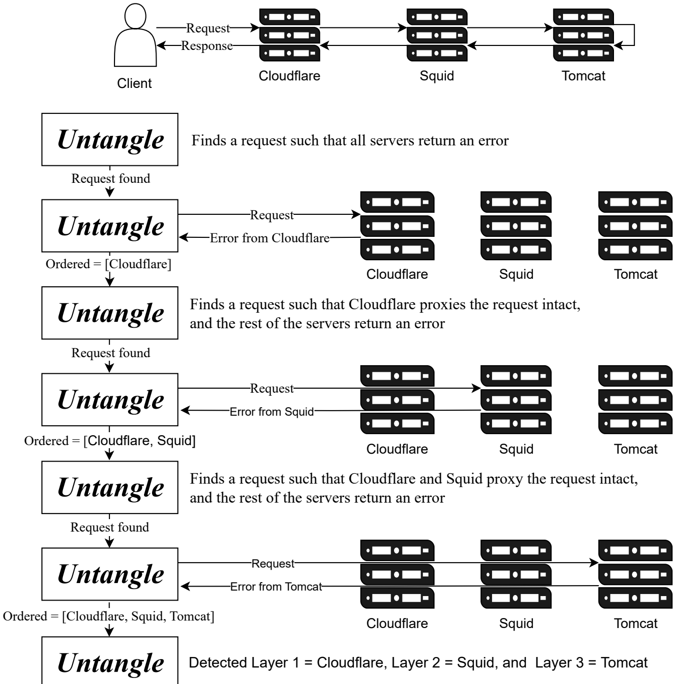

Fig. 1: Fingerprinting [Cloudflare, Squid, Tomcat] in Phase 1.

We depict Phase 2 in Algorithm 1, lines 20 through 31. We begin with the $Ordered$ list built in Phase 1, and the $Unordered$ set is empty. We probe the target with all requests where i) $Behavior(request, s) \in \{Error, Pass\}$ for all s in $Servers \backslash Ordered$, **and** ii) $Behavior(request, s) \rightarrow Pass$ for all s in $Ordered$. This triggers error responses from the subsequent unindentified servers, without any information regarding their layer placement, which we can then match to a known server technology and add to the $Unordered$ set. Note that it is again possible that our response analysis fails to find a match, in which case we add the symbol $Unknown$ to the set. $Unknown$ does not have to be a single server; it represents any number of servers that we fail to fingerprint.

Figure 2 demonstrates how Phases 1 and 2 interact through a concrete case study drawn from our experiments, involving the layers [CloudFront, HAProxy, Apache]. Steps 1 and 2 demonstrate a run of Phase 1, leading to the successful detection of layer 1, CloudFront. However, in Step 3, we fail to find the necessary probe, i.e., a request that only CloudFront proxies intact, but other servers respond to with an error. Hence, we enter Phase 2 with Step 4. For brevity, Step 5 depicts the entire process of probing the target with all qualifying requests. We receive a set of error and success responses, and we match the error responses to HAProxy and Apache. As a result, we set $Unordered$ to HAProxy and Apache, and conclude Phase 2.

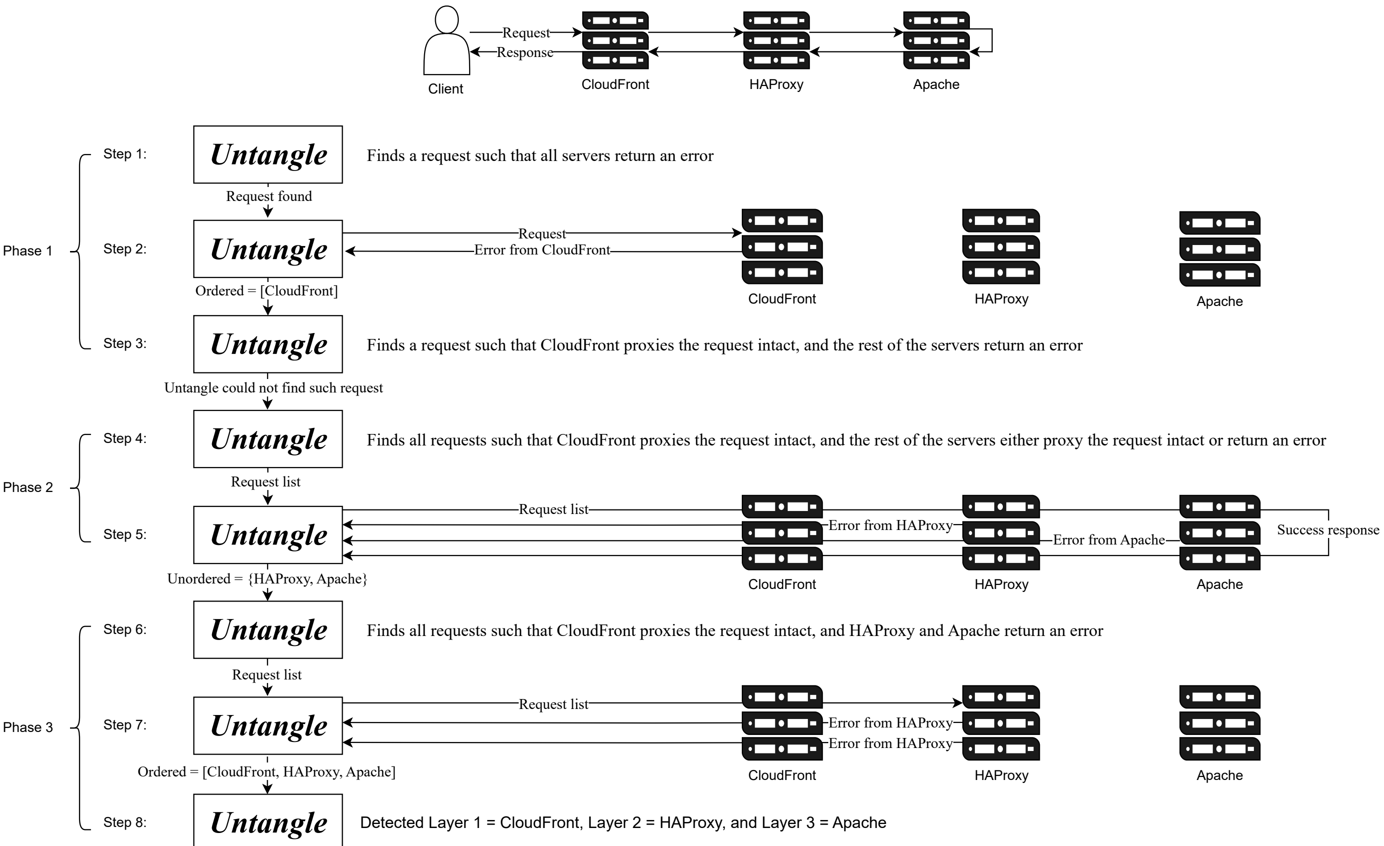

Fig. 2: Fingerprinting case study with [CloudFront, HAProxy, Apache] where Phase 1 of the methodology cannot complete the task due to an incomplete behavior repository.

C. Phase 3: Refining The Ordering

In this final phase, we aim to determine the ordering of the servers that we found in the previous step, i.e., move the entries in $Unordered$ to $Ordered$ according to their correct placement.

We show the process in Algorithm 1, lines 33 to 65. Intuitively, we probe the target with all requests where i) $Behavior(requests, s) \rightarrow Pass$ for all s in $Ordered$, **and** ii) $Behavior(request, s) \rightarrow Error$ for all s in $Unordered$. This guarantees an error response from the server that is positioned closest to the client among $Unordered$, which we can analyze and match to a known server technology as usual. We move the fingerprinted server to $Ordered$ and repeat the process.

Three conditions terminate this final phase: 1) $Unordered$ contains $Unknown$. Since we do not know how the unknown server behaves, we cannot enter Phase 3 at all. 2) We are unable to find a probe request that meets the necessary conditions. 3) There is only one server in $Unordered$; the order is implicit and there is no need to continue Phase 3. We move this server to the end of $Ordered$ and stop.

In the former two conditions, we return $Ordered$ and $Unordered$ separately as the final result. We call this a *partial fingerprint*, where, for an N-layer target, we are able to detect both the technology and order for the first m layers; the remaining $N - m$ layers are still correctly identified technologies, but without a specific order. While this is not the ideal outcome, it is still useful and actionable information that no prior fingerprinting technique can provide.

The latter, third halting condition represents the best case where we are able to correctly identify both the server technology and ordering for all N layers. We call this a *full fingerprint*.

Note that both the full and partial fingerprints above can still be *incorrect* if the methodology returns wrong results due to a bad behavior repository. We will explore these cases when evaluating our implementation later in the paper.

Revisiting the case study in Figure 2, Steps 6, 7, and 8 demonstrate the application of Phase 3. Previously, we found that HAProxy and Apache exist, however, we could not decide on their ordering. We now probe the target with all requests that CloudFront proxies, but HAProxy and Apache both respond to with an error. We receive an error response that, upon analysis, matches HAProxy. In other words, we now know that HAProxy is the server that comes after CloudFront. We could then repeat the process, however, since we only have Apache left in the $Unordered$ set, we meet our termination condition, and append Apache to the $Ordered$ list. Ultimately, we correctly identify that CloudFront stands in layer 1, HAProxy in layer 2, and Apache in layer 3.

So far, we have demonstrated that it is possible to fingerprint multi-layer web servers by using a generic and ex-

Algorithm 1 Multi-Layer Fingerprinting Methodology

1: /* Phase 1 Start */
2: $Ordered = []$, $Unordered = \emptyset$
3: **while** $\exists req \mid$
4: $\forall s \in (Servers \setminus Ordered), Behavior(req, s) \rightarrow Error \wedge$
5: $\forall s \in Ordered, Behavior(req, s) \rightarrow Pass$
6: **do**
7: $resp = send(req, target)$
8: **if** $isError(resp)$ **then**
9: $detected = Match(req)$
10: **if** $detected = Unknown$ **then**
11: **return** $Ordered$
12: **else**
13: $Ordered.append(detected)$
14: **end if**
15: **else**
16: **return** $Ordered$
17: **end if**
18: **end while**
19:
20: /* Phase 2 Start */
21: **for each** $req \in \{requests \mid$
22: $\forall s \in Servers \setminus Ordered, (Behavior(req, s) \rightarrow Error \vee$
23: $Behavior(req, s) \rightarrow Pass \quad) \wedge$
24: $\forall s \in Ordered, Behavior(req, s) \rightarrow Pass\}$
25: **do**
26: $resp = send(req, target)$
27: **if** $isError(resp)$ **then**
28: $detected = match(resp)$
29: $Unordered.add(detected)$
30: **end if**
31: **end for**
32:
33: /* Phase 3 Start */
34: **if** $Unknown \in Unordered$ **then**
35: **return** $Ordered, Unordered$
36: **else**
37: **while** $|Unordered| > 1$ **do**
38: $detected_list = []$
39: **for each** $req \in \{requests \mid$
40: $\forall s \in Ordered, Behavior(req, s) = Pass \wedge$
41: $\forall s \in Unordered, Behavior(req, s) = Error\}$
42: **do**
43: $resp = send(req, target)$
44: **if** $isError(resp)$ **then**
45: $detected = match(resp)$
46: $detected_list.append(detected)$
47: **end if**
48: **end for**
49: **if** $|detected_list| > 0$ **then**
50: $detected = \text{mode}(detected_list)$
51: **if** $detected = Unknown$ **then**
52: **return** $Ordered, Unordered$
53: **else**
54: $Unordered = Unordered \setminus detected$
55: $Ordered.append(\text{mode}(detected_list))$
56: **end if**
57: **else**
58: **return** $Ordered, Unordered$
59: **end if**
60: **end while**
61: **if** $|Unordered| = 1$ **then**
62: $Ordered.append(Unordered)$
63: **end if**
64: **end if**
65: **return** $Ordered$

tendable methodology, employing requests that leverage the discrepancies among servers, and error responses that uniquely identify servers. In the rest of the paper, we fill in the blanks regarding implementation details, limitations, and other practical considerations.

IV. Building A Behavior Repository via Fuzzing

In this section we venture into the practical considerations for implementing our fingerprinting methodology in a prototype called `Untangle`. So far we have assumed the existence of a behavior repository that captures two critical pieces of information: 1) What requests result in the $Error$ and $Pass$ behaviors for each server in scope, and 2) what distinct error responses each server returns. In our prototype, we leverage differential fuzzing to automatically build this repository.

TABLE I: Server technologies in scope for `Untangle`.

Server Name	Version
Akamai	N/A
Cloudflare	N/A
CloudFront	N/A
Fastly	N/A
Apache httpd	2.4.54
Apache Traffic Server (ATS)	9.1.2
Caddy	2.5.1
Envoy	1.21.1
HAProxy	2.6.0
NGINX	1.22.0
Squid	5.4
Tomcat	10.0.22
Varnish	6.0.10

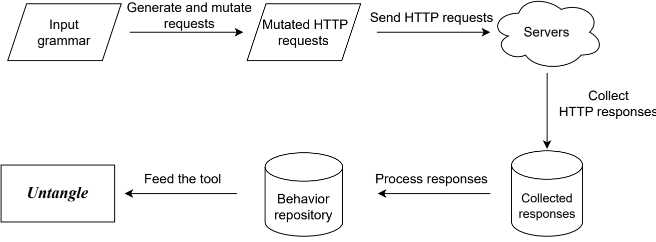

Fig. 3: Fuzzer setup overview.

We use the differential HTTP fuzzer *T-Reqs* by Jabiyev et al. as the basis for our own fuzzer, extending it to suit our needs [10]. We emphasize that we do not recycle any research results from Jabiyev et al.'s work, or claim contributions to the fuzzing domain–we merely utilize T-Reqs as an open-source fuzzing framework. We then select 13 popular technologies, 4 CDNs and 9 standalone servers, and exercise them in our lab to reveal processing discrepancies and collect the error responses we need. See Table I for a full list of servers in scope for our work, and Figure 3 for an overview of the experiment setup.

Note that this implies a minor deviation from how we described our methodology. `Untangle` does not need to dynamically generate and test probe requests for the correct behavior (i.e., via the $Behavior(request, server)$ function). Instead, all probe requests of interest with their corresponding behaviors for each server are pre-computed as the output of our fuzzing experiment. `Untangle` simply searches the behavior repository for a probe suitable for the circumstances.

A. Generating And Mutating HTTP Requests

T-Reqs is a grammar-based fuzzer that generates valid HTTP requests and then applies character mutations on it. The grammar also includes a symbol pool of non-standard headers that the fuzzer can insert into the generated request.

To mutate HTTP requests, T-Reqs employs character insertion, removal, and substitution, applied at random. We define a pool of 74 random ASCII characters for this purpose. The

```
_METHOD_ _URI_ HTTP/1.1 # mutations
Host: _HOST_
Connection: close
Content-Length: 8 | 0

\r\ndata\r\n\r\n | \r\n
```

Listing 1: Request line mutations experiment.

```
_METHOD_ / HTTP/1.1
Host: _HOST_
Connection: close
Content-Length: 8 | 0
_STANDARD HEADERS_ # mutations
_NON-STANDARD HEADERS_ # mutations

'data' | ''
```

Listing 2: Header mutations experiment.

```
POST / HTTP/1.1
Host: _HOST_
Connection: close
_ENTITY SIZE HEADERS_ # mutations
_TRAILER_ # mutations

_BODY_ # mutations
```

Listing 3: Body mutations experiment.

fuzzer also allows us to define an HTTP symbol pool where we declare the request components to which we wish to apply the mutations. T-Reqs then randomly selects from the pool of symbols and characters, mutating the request accordingly.

We design three fuzzing experiments focusing on the main parts of an HTTP request: the request line, header, and body. We modify the grammar and the mutation capabilities of T-Reqs, and run the experiments as follows.

1) Request Line Mutations: This experiment targets request line components by selecting from the pool of *method name*, *URI*, and *protocol* symbols and applying character mutations on them. During our experiments, we configure the fuzzer to apply up to two character mutations. To further increase the uniqueness of our requests, we diversify the `Content-Length` header value and request body.

Listing 1 presents the generic HTTP request that we use in this experiment. The color blue represents the parts to which we applied character mutations. Also, parts that we randomly choose from the grammar are depicted with the variable name that starts and ends with the '_' character (e.g., **_METHOD_**). For example, Listing 1 shows that we select the method name from a pool of method name values and make character mutations on it. The color brown depicts the particular values that we use in the requests. For instance, we only use '8' or '0' for the Content-Length header value. In total, we use 40 method names that include both common and uncommon ones and 8 different request URIs.

2) Header Mutations: Here, we randomly insert headers into the HTTP request we generate, using a pool of 66 standard and 1142 non-standard header names, compiled from header list of PortSwigger's Burp Suite extension param-miner [28]. For the non-standard headers, we use 'test' and '123' as the values. Again, using the character pool, we make character mutations on these headers. During our experiments, we insert one to two headers, with zero to two character mutations. As in the previous experiment, we also diversify the `Content-Length` header value and body. Finally, we randomly generate the *method name* from the pool of 40 method names. Listing 2 presents a summary of the requests we create in this experiment.

3) Body Mutations: Unlike the previous experiments, here we set the method to 'POST' and the request URI to '/'. We focus on the headers that change the body parsing behavior (i.e., entity size headers), such as `Content-Length` and `Transfer-Encoding`, and apply their valid and invalid combinations. We also utilize the `Trailer` header with the following values: 'Transfer-Encoding', 'Content-Length', and a non-standard header name, 'Foo'. We use both chunked and regular bodies. Listing 3 demonstrates the requests we generate for this experiment.

All in all, we generated 2,029,592 unique requests in the request line mutation experiment, 1,753,955 in the header mutation experiment, and 497,292 in the body mutation experiment, for a total of 4,280,839.

B. Discovering Discrepancies And Errors

We set up the 13 servers in scope for our experiments in our lab environment, and exercise them with the requests generated by T-Reqs as above. Since we are not only interested in collecting their responses but also the forwarded requests, we set up each server in proxy mode, in front of an origin that acts as a feedback server. This feedback server replies back to us with the requests proxied by the tested server. The sole exception to this experiment mode is Tomcat, which does not have a proxy mode and therefore cannot proxy requests.

Upon collecting the responses and requests proxied by the 13 servers, we analyze each and categorize them into three behavior buckets: 1) The server returns an error response, indicating that there is a problem in processing the request. This bucket corresponds to the $Error$ behavior previously defined in our fingerprinting methodology. 2) The request is successfully processed and the server proxies it without making any changes to the mutated sections. This is our $Pass$ behavior. Our focus on discrepancy triggering requests that only get forwarded by the proxies intact, with the mutations preserved, is a necessary design choice. It serves to avoid unexpected interactions due to request transformations between layers during the fingerprinting process. 3) All other behaviors are irrelevant for fingerprinting, and therefore we group them under the $Other$ category. These include requests proxied without preserving the applied mutations, request timeouts, HTTP 0.9 responses without headers but only a body, and zero-byte responses.

The behavior repository is now ready. For every request, we have the corresponding behaviors for all 13 servers. We also have each server's respective error responses.

TABLE II: Unique behaviors where servers in each cell proxy the request (i.e., $Pass$, or P), and servers not shown return an error (i.e., $Error$, E). The numbers represent the number of requests that trigger the corresponding unique behavior.

Akamai: {Akamai}, Cloudflare: {Cloudflare}, CloudFront: {CloudFront}, Fastly: {Fastly}, Apache: {Apache}, ATS: {ATS}, Caddy: {Caddy}, Envoy: {Envoy}, HAProxy: {HAProxy}, NGINX: {NGINX}, Squid: {Squid}, Varnish: {Varnish}

P=0, E=13	P=1, E=12	P=2, E=11	P=3, E=10	P=4, E=9	P=5, E=8	P=6, E=7	P=7, E=6	P=8, E=5
∅ : 1803699	{Varnish}: 24833	{HAProxy}{Varnish}: 45	{Akamai}{Cloudflare}{CloudFront}: 957	{ATS}{HAProxy}{NGINX}{Squid}: 33	{Fastly}{Envoy}{HAProxy}{Squid}{Varnish}: 1	{Fastly}{ATS}{Envoy}{HAProxy}{Squid}{Varnish}: 5	{Fastly}{Apache}{ATS}{Caddy}{HAProxy}{Squid}{Varnish}: 12893	{Fastly}{Apache}{ATS}{Caddy}{HAProxy}{Envoy}{Squid}{Varnish}: 374
	{HAProxy}: 2099	{HAProxy}{Squid}: 381	{Caddy}{HAProxy}{Varnish}: 632	{Cloudflare}{ATS}{HAProxy}{NGINX}: 324	{Fastly}{Apache}{ATS}{Caddy}{Varnish}: 2	{Cloudflare}{Fastly}{ATS}{HAProxy}{Squid}{Varnish}: 12	{Fastly}{Apache}{Caddy}{Envoy}{HAProxy}{Squid}{Varnish}: 217	{Akamai}{Cloudflare}{Fastly}{ATS}{Envoy}{HAProxy}{Squid}{Varnish}: 3
	{Squid}: 115	{ATS}{NGINX}: 4	{Cloudflare}{CloudFront}{HAProxy}: 24	{Akamai}{Cloudflare}{CloudFront}{Squid}: 135	{Cloudflare}{ATS}{HAProxy}{NGINX}{Squid}: 22003	{Cloudflare}{Fastly}{Envoy}{HAProxy}{Squid}{Varnish}: 3	{Cloudflare}{Fastly}{ATS}{Envoy}{HAProxy}{Squid}{Varnish}: 30	
	{ATS}: 14449	{Cloudflare}{Squid}: 1	{Cloudflare}{CloudFront}{Squid}: 62	{Akamai}{Cloudflare}{CloudFront}{HAProxy}: 14	{Fastly}{ATS}{HAProxy}{Squid}{Varnish}: 3		{Akamai}{Fastly}{ATS}{Envoy}{HAProxy}{Squid}{Varnish}: 1	
	{Caddy}: 17		{Cloudflare}{Fastly}{ATS}: 2					
	{Cloudflare}: 11		{Cloudflare}{ATS}{NGINX}: 2866					
	{Akamai}: 1		{ATS}{HAProxy}{Squid}: 1177					
			{ATS}{HAProxy}{NGINX}: 2					

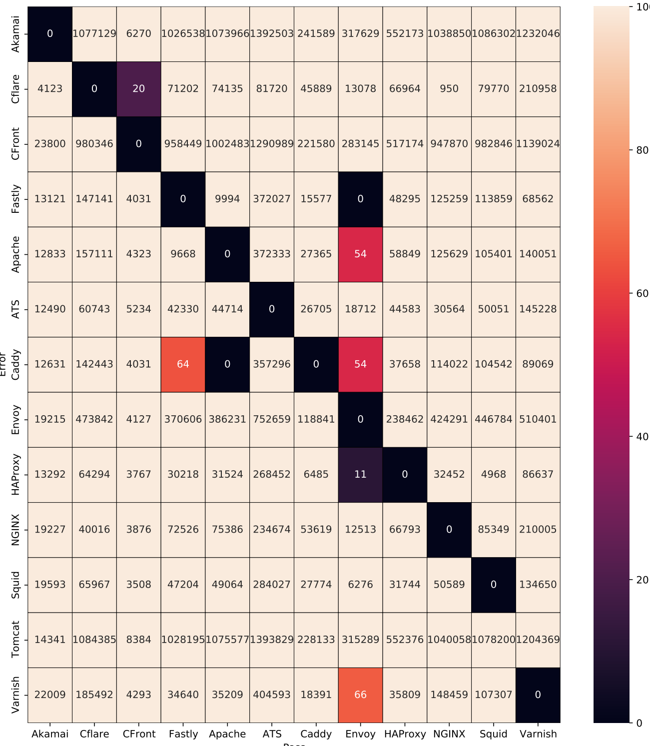

Fig. 4: Number of requests that trigger discrepancies for server pairs.

C. Viability Of Fuzzing To Build The Behavior Repository

Building the behavior repository via fuzzing is an implementation choice, and other implementations may choose different approaches. In our exploratory study, we considered two other possibilities: 1) Static analyses of server source code, and 2) manually guided dynamic analyses more focused than fuzzing. The former static analyses are inefficient or intractable for complex server code, but more importantly, they are impossible to carry out for proprietary CDN technologies from our external vantage point. The latter class of dynamic analyses are more promising. However, the easy automation and generally higher code coverage properties of fuzzing are especially desirable for our use case. That is because we anticipate a need to periodically repeat the fuzzing process and update the behavior repository in order to track server updates, which may change the servers' HTTP processing behavior. By settling on fuzzing, we address the problem of keeping `Untangle`'s fingerprinting capabilities up-to-date in an entirely automated manner.

An important question remains: Does fuzzing discover a sufficient number of discrepancies to build a practical fingerprinting tool? We present statistics below demonstrating that fuzzing is not only viable, but also effective at finding a wide variety of discrepancies.

Figure 4 depicts the number of requests that trigger pairwise discrepancies. The X-axis shows the servers that forward requests intact, and the Y-axis shows the servers that return an error. Recall that Tomcat does not have a proxy mode, and therefore it does not appear on the X-axis. In total, among the 144 pairings possible, we found at least 11 requests (and usually drastically more) that trigger HTTP parsing discrepancies in 142 pairs. For only two pairs, where Envoy forwarded as is and Fastly returned an error response, or where Apache forwarded as is and Caddy returned an error response, we could not find any requests that triggered discrepancies.

These pairwise results look promising; however, to understand the full extent of our findings, we explore the *unique* behaviors as well. Table II lists the unique behaviors we observed for all server combinations. We represent each server with a distinct shape and color. Each cell represents a unique behavior where the servers in that cell proxy the request (i.e., $Pass$, or P), and the servers not shown return an error (i.e., $Error$, or E). The number in each cell represents the number of requests that trigger the corresponding unique behavior. Columns represent the number of servers that show $Pass$ behavior for the unique behaviors under that column. Tomcat is again not present in this table since it cannot operate as a proxy, i.e., it only exists among the servers that show the $Error$ behavior. In total, we have 37 unique behaviors. To demonstrate how to read the table, one of the unique behaviors is depicted in the first row of the second column: only Varnish (i.e., {Varnish}) proxies the request, and the remaining 12 servers respond with an error. We found 24833 requests that led to this behavior.

To assess the viability of rebuilding the behavior repository in the face of upstream server updates, we reviewed the release cycles for all technologies in our work, as advertised by the developers and evidenced by their release dates. This analysis excludes the CDNs, which have no publicly discernible releases. We also ignore one-off hotfix patches and consider regular releases only. Our findings show that the most frequently updated server receives new versions on a monthly basis. As a point of reference, building the behavior repository

as described in this section, using a single commodity machine and no parallelization, took us below 6 days. We conclude that the process can be feasibly automated at a reasonable cadence to maintain an up-to-date fingerprinting tool.

These results demonstrate that fuzzing is able to provide us with a rich (albeit imperfect) behavior repository that makes it possible to capture the discrepancies between a great majority of the server pairs, with a plethora of request variations that make the approach robust against future server changes, all the while adopting a fully automated testing approach.

V. Error-to-Server Matching

We have one final implementation detail to complete `Untangle`: Determining an analysis strategy to match error responses to servers. We prefer not to solely rely on rigid signatures (e.g., checking for `Server` header values) for this task to make our detection robust against common server cloaking practices, though using such checks on top of other flexible heuristics to boost detection performance is acceptable.

Unlike most existing tools and their signature databases, we have access to specific server responses for each one of the HTTP requests utilized by `Untangle`. We take advantage of this visibility, and design a simple but effective response similarity metric. Specifically, we transform response messages into a set of tokens, and compute a Jaccard similarity score by analyzing the tokens shared between probe responses observed during fingerprinting and the known responses recorded in the behavior repository. Consequently, we match the response to the most similar entry in the repository, and label the probed layer with the corresponding server technology.

What response components should be considered for this similarity comparison is an implementation choice. We experiment with two options: 1) Computing similarity over the entire response message, and 2) computing three separate similarities over the full response, error code, body data, and combining them for the final score. The latter option is motivated by our exploratory work which shows that higher weighing of error codes and response bodies yields more accurate matches, due to the discrepancies affecting these components the most. We evaluate both approaches in Section VI.

We supplement this score by checking for a small number of fixed strings, once again motivated by our observations during an exploratory study. These are the following five CDN response headers: `CF-Cache-Status` and `CF-RAY` for Cloudflare, `X-Amz-Cf-Pop` and `X-Amz-Cf-Id` for Cloudfront, and `X-Served-By` for Fastly. We avoid relying on the commonly removed `Server` header. Again, we evaluate the impact of this addition in Section VI.

Lastly, we can perform the similarity-based server matching in two ways. One approach selects the best match; this simplifies the methodology by eliminating an $Unknown$ match, as there will always be a match even when the similarity score is low. However, in any real-life use of `Untangle`, we anticipate the possibility of interacting with a server technology that our behavior repository has no history of, making an $Unknown$ match necessary to flag a partial or unsuccessful fingerprinting. Therefore, the second approach uses a similarity threshold for error-to-server matching. In the absence of a score above the threshold, the probed server will be labeled $Unknown$. We experiment with both approaches in Section VI as well.

VI. Evaluation

We evaluate `Untangle` in a test setup that replicates a real-life multi-layer topology under the following conditions:

- The number of layers is set to three.
- CDNs are always placed before stand-alone servers to capture realistic deployments.
- The last layer is never a CDN, since they often cannot act as origin servers.
- Multiple instances of the same server cannot be present.
- When Tomcat is used, it is always in the last layer, because Tomcat does not have a proxy mode.

A. Fingerprinting Accuracy

Our core experiment set aims to measure how well `Untangle` performs in terms of producing accurate fingerprints, while incrementally enabling methodology phases and implementation refinements to observe how they influence the results. Our test driver dynamically deploys all permutations of the 13 servers in scope following the aforementioned constraints, and tests them using `Untangle`, resulting in a total of 756 unique fingerprinting runs per experiment. `Untangle` has no knowledge of the infrastructure beyond a domain name resolving to Layer 1, mimicking an end-to-end use case.

There are three main outcomes for each test:

- **A full fingerprint,** meaning that all 3 layers are correctly identified, both server type and order. This is the ideal outcome.
- **A partial fingerprint,** This is a broad category that captures all cases where we correctly identify the name and order of the first n layers. The remaining $3-n$ layers are either identified without specifying order, or flagged as $Unknown$. Although incomplete, this is still useful and actionable information.
- **Misclassification,** indicating at least one layer was labeled as an incorrect server, regardless of how many other layers were correctly detected. This is our failure case.

We conduct four experiments, each over all 756 server permutations, described next. In Table III we summarize all results, providing a detailed breakdown of partial fingerprinting outcomes. Table IV presents a different view of the same, breaking down the numbers with respect to the layers correctly identified–a correct detection for Layer n implies a correct detection for all layers $< n$.

In the following, cases where `Untangle` cannot produce a full fingerprint are due to the behavior repository not containing a suitable probe request for the tested server permutation. This could either be due to our fuzzing experiment not achieving the desired coverage to identify such probes, or otherwise there may not exist an appropriate discrepancy that could fingerprint that particular permutation. Misclassifications, on the other hand, result from the shortcomings of the response similarity checks we use; we discuss the specifics in each experiment below.

TABLE III: Summary results of fingerprinting accuracy experiments.

	Exp. A	Exp. B	Exp. C	Exp. D
Full fingerprint				
Layer 1, 2, and 3	38 (5.0%)	43 (5.7%)	390 (51.6%)	383 (50.7%)
Partial fingerprint				
Layer 1 only	330 (43.7%)	329 (43.6%)	30 (4.0%)	32 (4.2%)
Layer 1 and 2	339 (44.8%)	377 (49.8%)	293 (38.8%)	300 (39.7%)
Layer 1 and 3	11 (1.5%)	7 (0.9%)	40 (5.2%)	38 (5.0%)
Partial subtotal	680 (90.0%)	713 (94.3%)	363 (48.0%)	370 (48.9%)
Error				
Misclassification	38 (5.0%)	0 (0.0%)	3 (0.4%)	3 (0.4%)
Total runs	756	756	756	756

TABLE IV: Fingerprinting accuracy results broken down by layers.

Experiment	Layer 1	Layer 2	Layer 3
Exp. A	756 (100.0%)	382 (50.5%)	38 (5.0%)
Exp. B	756 (100.0%)	420 (55.5%)	43 (5.7%)
Exp. C	756 (100.0%)	683 (90.3%)	390 (51.6%)
Exp. D	756 (100.0%)	683 (90.3%)	383 (50.7%)
Nmap	450 (59.5%)	0 (0.00%)	0 (0.00%)

Experiment A: Phase 1 Only. We start with the base case, configuring `Untangle` to only use Phase 1 of the methodology without the refinement steps. The error-to-server matching is performed following the basic scheme where the similarity score is computed over the entire response body, and there are no checks for static headers. There is also no similarity threshold defined; we pick the highest similarity response match, and an $Unknown$ classification is not possible.

This restricted configuration can still detect Layer 1 in all runs, but the accuracy drops steeply with the deeper layers. In particular, we achieve a full fingerprint only in 38 (5.0%) cases, with 680 (90.0%) runs stopping after a partial fingerprint. We have a further 38 (5.0%) misclassifications. Manual analysis shows that, although all misclassified responses originated from the expected layers, the similarity metric falls short of classifying them to the right server type. This is due to our missing the subtle differences in the responses; for example, we observed that some misclassified responses differ only by their response bodies, but not the headers.

Experiment B: Phase 1 With Similarity Score Refinements. In light of the findings above, we now enable the alternative similarity comparison approach that combines three separate scores calculated over different response components, and static CDN header checks, both described in Section V. This experiment still runs Phase 1 only.

Our positive detections only improve slightly, but importantly, all misclassifications from the previous experiment are now eliminated. This demonstrates that the similarity score refinements we apply achieve the desired effect, and we incorporate them in all future experiments.

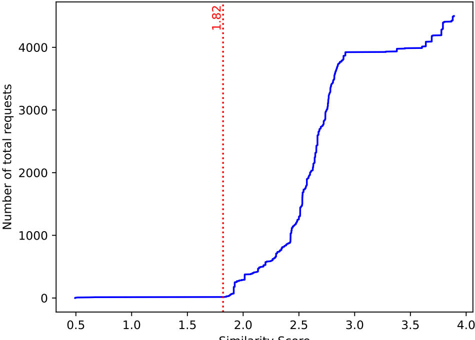

Fig. 5: Similarity score with respect to the accumulated number of requests. The vertical line represents the threshold.

Experiment C: All Phases. We now employ the complete methodology with all three phases, on top of the configuration we used in Experiment B.

The results show that introducing Phases 2 and 3 makes the detections in Layer 2 and Layer 3 surge to 683 (90.3%) and 390 (51.5%) respectively. In other words, we achieve a drastically better full fingerprint rate of 51.5%. The complete methodology also introduces 3 misclassifications; we find all the server names, but mix up Layer 2 and Layer 3's orders.

Despite the small error rate, this experiment empirically shows that the full methodology is effective, and the refinement phases significantly contribute to the overall scheme.

Experiment D: All Phases With Similarity Threshold. In the last experiment, we show that `Untangle` can achieve good accuracy. However, we still have one missing piece necessary for practical use in environments where there may be servers whose behavior is not captured in our repository. Hence, we need to introduce a similarity score threshold to be used during the error-to-server matching step, and enable the $Unknown$ category as described in Section V.

We empirically compute a plausible threshold as follows. We plot the maximum similarity scores observed during Experiment C; Figure 5 depicts the results. The X-axis represents the maximum similarity score x for a specific request, and the Y-axis represents the total number of requests that have a similarity score of x or lower. We observe that the number of requests with similarity scores higher than 1.82 increases drastically, and therefore select 1.82 as our threshold similarity score. That is, if a response has a maximum similarity score of 1.82 or lower, we classify it as $Unknown$.

We then run a final experiment with the similarity threshold check enabled, on top of the configuration we used in Experiment C. The results show that there is no change in Layer 1 and Layer 2 detections compared to Experiment C. In Layer 3, there is a small decrease, bringing our full fingerprint accuracy down to 383 (50.7%). This is expected as the selected threshold necessarily covered some of the successfully matched cases in our data set, but this minor degradation is an acceptable trade-off for enabling $Unknown$ classifications. **In summary, Experiment D represents `Untangle`'s canonical configuration and performance.**

```
HTTP/1.1 400 Bad Request
Connection: close
Content-Length: 11
content-type: text/plain; charset=utf-8
x-served-by: cache-bos4666

Bad Request
```

Listing 4: Misclassified HTTP response by Fastly.

```
HTTP/1.1 400 Bad Request
content-length: 11
content-type: text/plain
date: Fri, 30 Dec 2022 08:29:09 GMT
server: envoy
connection: close

Bad Request
```

Listing 5: Misclassified HTTP response by Envoy Proxy.

B. Accuracy With Unknown Servers

Now that we have the complete methodology fine-tuned with the empirically determined configuration parameters, we perform an additional set of experiments where we simulate how `Untangle` would perform in the presence of an unknown server, completely missing from the behavior repository.

We perform 13 experiments. In each, we make `Untangle` oblivious to one of 13 servers by removing all entries corresponding to that server from the behavior repository, crippling our ability to compute accurate similarity scores for that server's responses. We only permute the cases that include the removed server. As a result, we expect to identify the removed servers as unknown. Overall, for the experiments in which we remove the CDNs, we test 118 permutations. For Tomcat, we test 100 permutations, and for the rest, 212 permutations.

In the experiments where we remove Akamai, CloudFront, Apache, ATS, HAProxy, NGINX, Squid, Varnish, and Tomcat, we had no misclassifications. That is, if `Untangle` detects the layer, it correctly classifies it as $Unknown$.

When we remove Envoy, we erroneously match it to Caddy in 58 cases. When we remove Caddy, we erroneously match it to Envoy in 9 cases, and to both Envoy and $Unknown$ in 4 cases. When we remove Cloudflare, we erroneously match it to NGINX in 8 cases, and to both NGINX and $Unknown$ in 8 cases. When we remove Fastly, we erroneously match it to Envoy in 90 cases, and to Varnish in 8 cases.

As a misclassification example, see how Fastly and Envoy respond to an identical request in Listings 4 and 5. These two responses have a matching body and request line. We manually examine the remaining cases and observe that, likewise, they are all highly similar, which leads to the misclassification.

All together, `Untangle` made 185 misclassifications in 2268 total permutations. This is a reasonable error rate, and an inherent limitation of all fingerprinting techniques working with unrecognized targets. Further fine-tuning the similarity score and threshold is a possible way to minimize the misclassifications that we do not pursue in this work.

TABLE V: Nmap fingerprinting results against 3-layer servers.

	Permutations
Layer 1 detected	450 (59.6%)
Layer 2 detected	45 (5.9%)
Layer 3 detected	34 (4.5%)
Misclassification	73 (9.6%)
No detection	154 (20.4%)
Total	756 (100.0%)

C. `Untangle` Versus Nmap

We compare `Untangle` with the popular network scanning tool Nmap's fingerprinting capabilities [8]. Nmap has no concept of layered servers, and makes no claims of being able to fingerprint such architectures. Our goal in this experiment is two-fold: First, to evaluate how `Untangle`'s Layer 1 detection fares against Nmap, and next, to understand whether Nmap fails to detect a server in a multi-layer architecture when it *can* successfully identify the same server in isolation.

We first test Nmap against all 13 servers in isolation, where it is able to identify 9. These are Akamai, CloudFront, Cloudflare, Apache, ATS, Caddy, NGINX, Squid, and Tomcat. Nmap could not correctly identify Envoy, HAProxy, or Varnish even in isolation. Also, it misclassifies Fastly as Varnish–this is not surprising as Fastly is known to use a version of Varnish on their edge servers [20]. In contrast, `Untangle` detects **all** servers in Layer 1 in all experiments, with no misclassification. Hence we can confidently say `Untangle` outperforms Nmap for single server fingerprinting.

Next, we test Nmap in the same 3-layer setup with 756 server permutations we used for our core experiments. We present the results obtained with Nmap on the last row of Table IV for easy comparison to `Untangle`. The results show that, Nmap correctly detects the Layer 1 server in only 450 permutations out of 756.

Table V presents more details on this experiment, breaking down the cases where Nmap falls short. In 45 and 34 tests, Nmap returns the correct results that match Layer 2 and Layer 3 respectively. We are not able to determine with certainty whether these are due to Layer 2 and 3 server indicators masking the Layer 1 signals that Nmap checks for, or some other coincidental interaction between the layers. Nmap also reports an entirely incorrect detection for 73 permutations, matching none of the layers, and it could not detect anything for another 154 permutations.

Manual analysis of the results confirms that in some cases, Nmap was indeed confused due to the layering. For example, for the [Squid, Caddy, Tomcat] permutation, Nmap reports Caddy as the detected server, while normally, it is able to detect Squid in isolation. In another case, for [Apache, Squid, ATS], Nmap could not detect any servers, whereas it is able to fingerprint Apache in isolation.

These results demonstrate that a well-established tool like Nmap and the traditional fingerprinting techniques it is based on do not always function correctly against the complexity of multi-layer infrastructures. In the end, `Untangle` is able to perform better than Nmap in fingerprinting single servers, and it is the only option for fingerprinting multi-layer servers.

D. Testing In the Wild

Finally, we test Untangle in the wild with real web deployments. We stress that the goal of this experiment is *not* to present a scientific measurement study of server incidence. Unfortunately, without access to ground truth describing proprietary infrastructures, validating Untangle's output is not viable. Therefore, we present the raw results of this experiment as is, and instead focus on analyzing the practical implications of running Untangle on production infrastructures, which we may have missed in our previous lab evaluation.

We seed our experiment with the Tranco top 10K domains[1] generated on 22 October 2023 [17]. An immediate practical consideration with our methodology is that, we require interactions with an ordinary server that responds with a variety of 200, 4xx, and 5xx responses to guide our fingerprinting process. However, we observe in the wild that some domains and their root URLs lead to load balancers that unconditionally issue 3xx redirects (e.g., pointing to the "www" subdomain, or different paths based on geo-location). Therefore, we preprocess the list by visiting each domain and following all redirects until we get a 200 response, remove the duplicates the process may yield, and utilize these as our fingerprinting targets. We also eliminate targets that do not go to a functional server during this step, including DNS errors, server errors, timeouts, and 403 responses. This results in a final list of 7528 sites. The 2472 sites we eliminate are higher than the non-functional sites encountered in the original Tranco research [17], [27]; however, we manually verified with a random sample that the sites we filtered out were indeed not functional at the time of our testing. Also note that, Untangle is necessarily hindered by bot management systems likely to be deployed in front of production servers, and these may have had an impact; bypassing such defenses is outside our scope.

To work around the lack of a ground truth, we run Nmap on the same list, and use the consensus between the two tools to estimate Untangle's accuracy in the wild. Therefore, in the rest of this section, we strictly focus on Layer 1 fingerprints.

Untangle is able to fingerprint 6360 out of 7528 targets[2], mapping them to one of the 13 servers captured in our behavior repository. Starting with the same list, Nmap can map 5877 of the targets to one of these 13 servers. We summarize the results in Figure 6 for a visual comparison, demonstrating that the results are similar. Nmap of course produces fingerprints that correspond to other technologies, not among the 13 servers Untangle is aware of; we are not interested in a breakdown of these, our purpose is not to evaluate Nmap.

Untangle and Nmap's output do not perfectly overlap. Specifically, the tools agree on 4984 cases (66.2%), mapping a given target to the same server. The disagreements that signal issues fall under one of the following categories.

- **Category 1:** Nmap maps 219 targets to one of the 13 servers in scope, while Untangle detects an *Unknown* server.

[1]Available at https://tranco-list.eu/list/997K2

[2]Readers be advised that we discuss many overlapping result sets in this section, or otherwise focus on specific slices of findings. Numbers do not add up to 6360, this is not in error.

TABLE VI: Random sample analysis. Numbers do not add up to 100%, remaining cases are undecidable, or both tools are wrong.

	Nmap Correct	Untangle Correct	Sample Size
Category 1	62 (44.2%)	68 (48.5%)	140 (100%)
Category 2	50 (19.5%)	147 (57.4%)	256 (100%)

- **Category 2:** Untangle maps 1119 targets to one of the 13 servers in scope, while Nmap either identifies a different server among those 13, or an entirely different technology not captured in our behavior repository.

360 cases in Category 2 correspond to the systemic Nmap issue we demonstrated earlier, where Nmap misclassifies Fastly as Varnish. That is, Untangle yields the correct result. Eliminating these, for all remaining cases, we perform a manual analysis over random samples taken from each category to decide which tool is correct. At a 95% confidence level, 5% margin of error, this yields sample sizes of 140 and 256 targets. Authors perform the analysis based on DNS and HTTP traces; this is necessarily a subjective, best-effort approach. We present the results in Table VI.

All outcomes where Untangle is wrong are due to two major reasons: Sites that heavily customize their error responses, and unknown servers that return errors highly similar to those that are in the behavior repository, incorrectly reaching our similarity threshold.

Looking beyond the consensus set, for 705 targets, Untangle fingerprints the server as *Unknown*, while Nmap detects a server not in our behavior repository. Assuming that Nmap is accurate, we consider this the correct outcome for us; Untangle is designed to map foreign technologies as such.

We also glean an interesting insight by comparing the reasons for each tool's failure cases. Nmap is easily confused when the Server field is removed, indicating server cloaking measures. Untangle is resistant against such intentional obfuscation, but instead thwarted by more invasive changes to responses, or by devices that unconditionally return errors or redirects–these are not defensive techniques, but customization and infrastructure features. Each tool and their methodologies have unique properties, and established fingerprinting techniques are not without merit given the right application.

While runtime performance is not critical for fingerprinting, we nevertheless conclude with five-number summaries over data points collected during this experiment. Table VII presents end-to-end fingerprinting runtimes for both tools, and the number of requests issued for Untangle. We must clarify that Nmap performs a more generic fingerprinting function that can identify non-HTTP services as well, and therefore it naturally takes longer to execute; this is not a fair comparison, but a reference point. The extreme low-performance cases are outliers, likely due to slow servers or tarpitting by bot defenses.

In summary, Untangle demonstrably works in practice, despite server version and configuration diversity. To the extent that we can validate the results in the absence of ground truth, the comparison with Nmap gives us assurance that Untangle is effective and can fingerprint cases where Nmap fails.

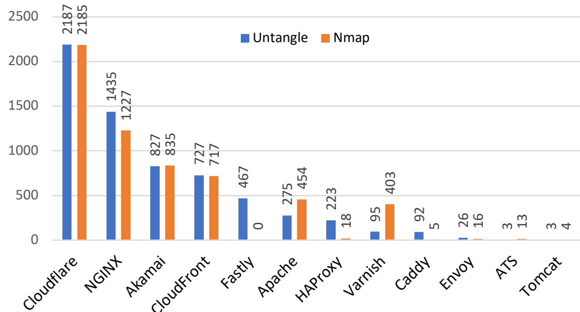

Fig. 6: `Untangle` and Nmap in the wild.

TABLE VII: Performance summaries for `Untangle` and Nmap.

	Nmap	`Untangle`	`Untangle` Requests
Min	0.35 sec	0.01 sec	1
Q_1	14.09 sec	1.72 sec	2
Median	19.74 sec	3.79 sec	4
Q_3	21.58 sec	7.28 sec	7
Max	255.94 sec	253.35 sec	27

VII. Limitations

The methodology we presented, though sufficiently generic in its application to any multi-layer architecture, still has limitations that must be called out inherent to relying on server discrepancies. Foremost, the efficacy of the scheme is correlated with having a rich behavior repository. This implies that the mechanism for discovering discrepancies is a critical consideration. In this work, we do not scientifically evaluate the different ways to build the repository; that remains an open consideration. However, as discussed in Section IV, our choice of differential fuzzing is not arbitrary; it is motivated by the technique's well-documented discrepancy discovery capabilities and ease of automated future updates to the repository.

Another limitation is inherent in the premise: There exists server discrepancies. Our evaluation of pairwise discrepancies in Section IV shows that discrepancy availability is not a given–though we cannot determine whether this was a limitation of our fuzzer or truly a lack of discrepancies, the end result negatively impacts accuracy. Proving or disproving the existence of discrepancies is a non-trivial challenge we do not tackle here. However, evidenced by the increasing complexity of the Internet, long history of formal HTTP specifications not being able to provide prescriptive processing instructions, and the steady stream of discrepancy attacks, we have empirical assurance that a lack of discrepancies will not be the blocker to our approach. On the flip side, in a hypothetical world where all server technologies perform identically and eliminate all discrepancies, the security utility of multi-layer web server fingerprinting diminishes, and that is a good outcome for all.

How robust is our methodology, in the face of regular server updates that may introduce behavior changes? Can server developers or operators intentionally modify their behavior in an attempt to block fingerprinting? These are valid concerns that we do not quantify within a scientific framework here, but we emphasize that our leveraging of fuzzing is motivated by this very consideration. We anticipate that existing discrepancies will disappear and brand new discrepancies will crop up during the course of server development; fuzzing makes it possible for `Untangle` to track these changes by frequent periodic updates. Our experiments in the wild demonstrate that `Untangle` can otherwise endure the expected configuration and infrastructure diversity.

Beyond issues related to the completeness of the behavior repository, we cannot accurately fingerprint a specific case: If there are *identical* servers layered back-to-back, by definition there cannot be discrepancies between them, and our methodology will collapse those servers into a single layer. Technically, this is an incorrect result that does not match the physical deployment, but we argue that the utility of the output does not significantly change except for asset discovery purposes. The resulting fingerprint is useful for reasoning about discrepancy attacks and vulnerability management.

If the behavior repository is not aware of a given server technology, we arrive at a similar outcome, where there may be layers invisible to `Untangle` in the fingerprint. This is analogous to how existing single-server fingerprinting techniques fail; if they are not aware of the server signature, they cannot possibly detect the server. It is important to clarify that, since the repository can never feasibly be aware of the universal set of servers including proprietary technologies, the results should always be considered a relative ordering of the layers. We stress once again that this is still useful and actionable information for security uses of fingerprinting, and in many cases where common technologies are targeted, the result will in fact match the physical deployment.

We acknowledge that our evaluation is limited by the 13 technologies we selected and the scope of our fuzzing experiments. These choices were dictated by our resource constraints; they are not fundamental limitations of the research. We were also unable to run conclusive fingerprinting measurements in the wild, as there is no ground truth. We tested `Untangle` against 4 enterprise infrastructures where we were able to confirm the actual deployment via our personal connections in those companies. We correctly fingerprinted their 2-layer and 3-layer technologies. Sadly, we do not have the means to provide any scientific evidence or perform a larger-scale experiment without external collaborators.

VIII. Conclusion

We have presented a web server fingerprinting methodology unique in its ability to target multi-layer architectures, and determine both the server technology and ordering, setting it apart from all previous works. In doing so, we also showed that the increasing complexity of the Internet and seemingly inevitable processing discrepancies, so far exploited with a surge of novel attacks, can also be leveraged by the security community for inventing creative new techniques.

Our methodology is imperfect. It has limitations inherent to the concept of fingerprinting, and doing so by relying on server discrepancies. Nevertheless, we showed that a practical application is not only viable, but also effective at solving an open problem. Recall our overarching research question: Is it possible to detect multi-layer web servers by utilizing HTTP parsing discrepancies? We answer the question affirmatively, and make `Untangle` available for the community.

Acknowledgment

This project was partially supported by NSF grants 2219921, 2127200, 2031390, and 2329540.

References

[1] Amazon Web Services, "HTTP Desync Guardian," 2020, https://github.com/aws/http-desync-guardian.

[2] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential Fuzzing of JavaScript Engines," in *ACM Conference on Computer and Communications Security*, 2022.

[3] BuiltWith, "BuiltWith Technology Lookup," https://trends.builtwith.com/CDN/Content-Delivery-Network.

[4] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of Troubles: Multiple Host Ambiguities in HTTP Implementations," in *ACM Conference on Computer and Communications Security*, 2016.

[5] Cloudflare, "Cache Deception Armor," 2023, https://developers.cloudflare.com/cache/cache-security/cache-deception-armor/.

[6] R. T. Fielding, M. Nottingham, and J. F. Reschke, "HTTP Semantics," 2022, https://datatracker.ietf.org/doc/html/rfc9110.

[7] O. Gil, "Web Cache Deception Attack," Black Hat USA, 2017, https://www.blackhat.com/us-17/briefings.html#web-cache-deception-attack.

[8] Gordon Lyon, "Service and Version Detection," Nmap Network Scanning, 2008, https://nmap.org/book/man-version-detection.html.

[9] B. Jabiyev, S. Sprecher, A. Gavazzi, T. Innocenti, K. Onarlioglu, and E. Kirda, "FRAMESHIFTER: Security Implications of HTTP/2-to-HTTP/1 Conversion Anomalies," in *USENIX Security Symposium*, 2022.

[10] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: HTTP Request Smuggling with Differential Fuzzing," in *ACM Conference on Computer and Communications Security*, 2021.

[11] A. Kar, A. Natadze, E. Branca, and N. Stakhanova, "HTTPFuzz: Web Server Fingerprinting with HTTP Request Fuzzing," in *International Conference on Security and Cryptography*, 2022.

[12] J. Kettle, "Practical Web Cache Poisoning," PortSwigger Web Security Blog, 2018, https://portswigger.net/blog/practical-web-cache-poisoning.

[13] ——, "HTTP Desync Attacks: Request Smuggling Reborn," PortSwigger Web Security Blog, 2019, https://portswigger.net/blog/http-desync-attacks-request-smuggling-reborn.

[14] ——, "Web Cache Entanglement: Novel Pathways to Poisoning," PortSwigger Research, 2020, https://portswigger.net/research/web-cache-entanglement.

[15] ——, "HTTP/2: The Sequel is Always Worse," PortSwigger Web Security Blog, 2021, https://portswigger.net/research/http2.

[16] A. Klein, "HTTP Request Smuggling in 2020 – New Variants, New Defenses and New Challenge," Black Hat USA, 2020, https://www.blackhat.com/us-20/briefings/schedule/#http-request-smuggling-in---new-variants-new-defenses-and-new-challenges-20019.

[17] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation," in *Annual Network and Distributed System Security Symposium*, 2019.

[18] D. Lee, J. Rowe, C. Ko, and K. Levitt, "Detecting and Defending against Web-Server Fingerprinting," in *Annual Computer Security Applications Conference*, 2002.

[19] C. Linhart, A. Klein, R. Heled, and S. Orrin, "HTTP Request Smuggling," Watchfire, 2005, https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf.

[20] A. MacLachlan, "The benefits of using Varnish," Fastly Blog, 2015, https://www.fastly.com/blog/benefits-using-varnish.

[21] Marc Ruef, "httprecon," 2023, https://www.computec.ch/projekte/httprecon/?s=documentation.

[22] S. A. Mirheidari, S. Arshad, K. Onarlioglu, B. Crispo, E. Kirda, and W. Robertson, "Cached and Confused: Web Cache Deception in the Wild," in *USENIX Security Symposium*, 2020.

[23] S. A. Mirheidari, M. Golinelli, K. Onarlioglu, E. Kirda, and B. Crispo, "Web Cache Deception Escalates!" in *USENIX Security Symposium*, 2022.

[24] H. V. Nguyen, L. L. Iacono, and H. Federrath, "Your Cache Has Fallen: Cache-Poisoned Denial-of-Service Attack," in *ACM Conference on Computer and Communications Security*, 2019.

[25] OWASP, "Fingerprint Web Server," Web Security Testing Guidance, 2020, https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/01-Information_Gathering/02-Fingerprint_Web_Server.

[26] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient Domain-Independent Differential Testing," in *IEEE Symposium on Security and Privacy*, 2017.

[27] V. L. Pochat, T. V. Goethem, and W. Joosen, "Evaluating the Long-term Effects of Parameters on the Characteristics of the Tranco Top Sites Ranking," in *USENIX Workshop on Cyber Security Experimentation and Test*, 2019.

[28] PortSwigger, "param-miner," 2020, https://github.com/PortSwigger/param-miner/blob/master/resources/headers.

[29] G. S. Reen and C. Rossow, "DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies For QUIC," in *Annual Computer Security Applications Conference*, 2020.

[30] Saumil Shah, "httprint–An Introduction to HTTP Fingerprinting," 2004, https://www.net-square.com/httprint_paper.html.

[31] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022.

[32] Tenable Nessus, "HMAP Web Server Fingerprinting," 2023, https://www.tenable.com/plugins/nessus/11919.

Appendix

We present `Untangle`'s fingerprinting capabilities in Table VIII. This is a dense table meant as a detailed reference material; readers can safely skip it.

The table is three-dimensional, where each column represents the server that stands in Layer 1 and each row represents the server that stands in Layer 2. During fingerprinting, if we cannot detect the server in Layer 2, we use ✗ to indicate this. If we detect the server in Layer 2, but cannot find any of the Layer 3 servers, we use ✓ to indicate this. If we detect the server in Layer 2 and at least one server in Layer 3, we represent the list of Layer 3 servers by using symbols shown in the table caption. Note that we detect all Layer 1 servers in all permutations, and therefore do not show Layer 1 information in the table for brevity.

TABLE VIII: `Untangle`'s complete 3-layer detection reference.

Akamai: ⊙, Cloudflare: ■, CloudFront: ★, Fastly: ⊞, Apache: ↑, ATS: →, Caddy: ⊕, Envoy: ▣, HAProxy: ←, NGINX: ●, Squid: ↓, Tomcat: ×, Varnish: ◎

		Layer 1											
		Akamai	Cloudflare	CloudFront	Fastly	Apache	ATS	Caddy	Envoy	HAProxy	NGINX	Squid	Varnish
Layer 2	Akamai	N/A	↑⊕●×	✓	↑⊕●×	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Cloudflare	↑⊕●×	N/A	✗	↑→⊕▣←●↓×	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	CloudFront	↑→⊕▣←●↓×◎	✓	N/A	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Fastly	↑⊕●×	↑→⊕▣←●↓×◎	✓	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Apache	✓	✓	✓	→▣←●↓×	N/A	▣←●↓×	→▣←●↓×	→●×	→▣●×	✓	→▣●×	→▣←●↓×
	ATS	↑⊕●×	↑⊕▣←●↓×◎	✓	↑⊕●×	●×	N/A	▣←●↓×	↑⊕●×	↑⊕▣●↓×◎	✗	↑⊕▣●×◎	↑⊕▣←●↓×
	Caddy	✓	✓	✓	→▣←●↓×	✗	▣←●↓×	N/A	→●×	↑→▣●↓	✓	→▣●×	↑→▣←●↓
	Envoy	↑⊕●×	↑→⊕●×	✓	↑⊕●×	●×	↑⊕●×	→●×	N/A	↑→⊕●×	✓	↑→⊕●×	↑→⊕●×
	HAProxy	↑→⊕▣●↓×◎	↑⊕▣●↓◎	↑→⊕▣●↓×◎	↑⊕▣●×	▣●×	↑⊕▣●↓×◎	✗	✗	N/A	↑⊕▣↓×◎	↑→⊕▣●×◎	↑→⊕▣●↓×
	NGINX	✓	↑⊕▣←↓×◎	✓	✓	✓	↑⊕▣←↓×◎	✓	✓	↑⊕▣↓×◎	N/A	↑⊕▣×◎	✓
	Squid	↑→⊕▣←●×◎	↑→⊕▣←●×◎	↑→⊕▣←●×◎	↑⊕▣●×	▣●×	↑⊕▣●×◎	→▣●×	✗	↑→⊕▣●×◎	↑⊕▣×◎	N/A	↑→⊕▣●×
	Varnish	↑⊕●×	↑→⊕▣●×	✓	↑⊕▣←●↓×	✗	↑⊕▣←●↓×	↑→⊕▣←●↓◎	✗	↑→⊕▣●↓×	✓	↑→⊕▣●×	N/A