Scan Me If You Can: Understanding and Detecting Unwanted Vulnerability Scanning

Xigao Li
Stony Brook University

Babak Amin Azad
Stony Brook University

Amir Rahmati
Stony Brook University

Nick Nikiforakis
Stony Brook University

ABSTRACT

Web vulnerability scanners (WVS) are an indispensable tool for penetration testers and developers of web applications, allowing them to identify and fix low-hanging vulnerabilities before they are discovered by attackers. Unfortunately, malicious actors leverage the very same tools to identify and exploit vulnerabilities in third-party websites. Existing research in the WVS space is largely concerned with how many vulnerabilities these tools can discover, as opposed to trying to identify the tools themselves when they are used illicitly.

In this work, we design a testbed to characterize web vulnerability scanners using browser-based and network-based fingerprinting techniques. We conduct a measurement study over 12 web vulnerability scanners as well as 159 users who were recruited to interact with the same web applications that were targeted by the evaluated WVSs. By contrasting the traffic and behavior of these two groups, we discover tool-specific and type-specific behaviors in WVSs that are absent from regular users. Based on these observations,

we design and build ScannerScope, a machine-learning-based, web vulnerability scanner detection system. ScannerScope consists of a transparent reverse proxy that injects fingerprinting modules on the fly without the assistance (or knowledge) of the protected web applications. Our evaluation results show that ScannerScope can effectively detect WVSs and protect web applications against unwanted vulnerability scanning, with a detection accuracy of over 99% combined with near-zero false positives on human-visitor traffic. Finally, we show that the asynchronous design of ScannerScope results in a negligible impact on server performance and demonstrate that its classifier can resist adversarial ML attacks launched by sophisticated adversaries.

CCS CONCEPTS

• **Security and privacy** → **Web application security**; **Intrusion detection systems**; **Vulnerability scanners**.

KEYWORDS

Web Vulnerability Scanner, Vulnerabilities, Fingerprinting

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WWW '23, April 30–May 04, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9416-1/23/04…$15.00
https://doi.org/10.1145/3543507.3583394

ACM Reference Format:
Xigao Li, Babak Amin Azad, Amir Rahmati, and Nick Nikiforakis. 2023. Scan Me If You Can: Understanding and Detecting Unwanted Vulnerability Scanning. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30–May 04, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543507.3583394

1 INTRODUCTION

As the web continues to become the platform of choice for delivering applications to users, attackers are increasingly targeting web applications to compromise their underlying systems and exfiltrate personal and financial data. Moreover, the popularity of certain web-facing technologies leads to software monocultures where a single high-impact vulnerability discovered in a single piece of software can be weaponized against millions of worldwide deployments of that software. Just in 2021, NIST recorded 18,378 vulnerabilities [26], representing a new record, with many web-related, high-impact vulnerabilities among them, including the recently discovered Log4J vulnerability [10], as well as critical RCE vulnerabilities in the web UIs of VMWare and F5 products [15, 27].

One of the strategies used by developers and system administrators to identify and correct vulnerabilities before they are abused by attackers is the use of Web Vulnerability Scanners (WVSs). WVSs are automated "point-and-click" tools that scan web applications for known and unknown vulnerabilities such as XSS, CSRF, RCE, and exposed private files. WVSs can be used either manually (*e.g.*, as part of a penetration-testing engagement) or incorporated in Continuous Integration/Continous Delivery (CI/CD) pipelines to scan a web application every time developers commit new code to their repositories [32, 34].

Unfortunately, even though WVSs are meant to be used by legitimate administrators and authorized penetration testers, nothing stops attackers from downloading an off-the-shelf WVS, pointing it to a target of interest, and scanning that target. Most WVSs support rate-limiting and changing the default User-agent header, which can be readily abused by attackers to hide their identity when scanning targets. In fact, in their recent work on characterizing the automated browsing activity that websites observe, Li *et al.* reported traces of a specific WVS scanning their deployed web applications, despite the lack of popularity of their honeypot websites [20]. From a research perspective, most prior work on known WVSs has evaluated the ability of these tools to identify vulnerabilities [5, 12, 23, 24, 35, 36, 39], as opposed to trying to identify the tools themselves when they are used for unauthorized scanning. While there exists a rich body of research on detecting Internet bots [6, 9, 16, 18, 20, 21, 31, 37, 40, 43], WVS behavior is significantly different from the generic bots' activities: Bots commonly conduct crawling, indexing, or occasional

probing for specific vulnerabilities, while WVSs are designed for the systematic evaluation or pentesting of websites against a long list of potential vulnerabilities.

In this paper, we approach the problem of unwanted web vulnerability scanning by compiling a list of 12 popular WVSs and characterizing their capabilities and network-level behavior. To this end, we develop a testbed capable of automatically launching these scanners against our own targets while monitoring the WVSs' network-level behavior and extracting fingerprintable attributes using state-of-the-art browser and network-fingerprinting methods. To understand how the traffic that WVSs generate is different from the traffic of benign users, we conduct a separate user study with 159 users conducting typical activities on the same web applications (*e.g.,* reading articles and searching for content).

By contrasting these two datasets, we identify significant differences in the type of traffic these two groups generate and the overall network-level behaviors they exhibit. Among others, we discover that WVSs send large numbers of requests to the applications that they scan (up to hundreds of thousands of requests in a single run), causing a disproportionate number of HTTP errors while scanning (up to 98% HTTP error rate), and exhibiting lacking/partial support of security mechanisms that are universally present in modern web browsers (*e.g.,* the inability to enforce CSP policies, as well as the inappropriate loading of mixed content). We also explore the deterministic nature of WVSs (*i.e.,* will a given WVS produce the same requests when scanning the same web application) and discover the use of randomness in some of the evaluated WVSs.

Based on our observations of the differences between users and WVSs, we propose ScannerScope, a system for detecting unwanted web vulnerability scanning. ScannerScope is deployed at the server-side of web applications, acting as a reverse proxy between incoming HTTP requests and the webserver. Using supervised machine learning, ScannerScope *asynchronously* classifies incoming requests as belonging to users vs. WVSs. We demonstrate that ScannerScope exhibits high detection accuracy (*e.g.,* 99.30% for protecting WordPress applications), which it largely retains even when the protected web application is entirely different from the web application on which it was trained. Moreover, ScannerScope keeps its detection accuracy even when facing unseen WVSs that were not part of its training set. Across both scenarios, ScannerScope has near-zero false positives (*i.e.,* misclassifying human visitors as scanners) and can be combined with additional server-side techniques to ensure that both regular users, as well as benign bots, are not affected.

Finally, we demonstrate that ScannerScope incurs a negligible performance overhead in deployments that already use reverse proxies (*e.g.,* for load-balancing purposes) and quantify how well ScannerScope resists adversarial attacks aimed at confusing its classifier. Overall, this paper makes the following contributions:

- We deploy a testbed for measuring web vulnerability scanners (WVS) and use it to curate a wide range of fingerprints (browser, TLS, and behavioral) from the evaluated tools.
- We characterize a total of 12 popular web vulnerability scanners and 159 user participants, pointing out the differences in the browsing behavior that they exhibit. Through this process, we obtain two ground-truth datasets that can be used in a supervised machine-learning setting to differentiate between users and WVSs. We will be sharing these datasets with other researchers.
- We propose ScannerScope, an ML-based detection system that can detect WVSs in incoming HTTP traffic. We show that ScannerScope can effectively detect unwanted scanning activity without adding significant overhead to the web server while retaining its robustness against attackers who attempt to spoof their identity.

2 BACKGROUND AND THREAT MODEL

Web Vulnerability Scanners (WVSs) are automated tools used to scan web applications for common vulnerabilities. These scanners range from simple tools that request a series of predetermined endpoints from the scanned web application (such as directory brute-forcers), to complicated crawling-driven tools that first map all the endpoints of a web application before attempting a series of attack vectors in search of SQL injections, XSS vulnerabilities, RCEs, etc. WVSs are available as open-source tools (*e.g.,* OWASP ZAP [29], and Arachni [4]), commercial tools, as well as via Scanning-as-a-Service deployments (*e.g.,* the Tenable [2] and Acunetix [1] cloud scanners).

While the intended audience of WVSs are hired penetration testers as well as web application administrators, these tools can be used by attackers to scan arbitrary third-party web applications, without their permission. In fact, Li *et al.*'s recent study on malicious bots [20] reported evidence of WVS activity even on newly-created websites with zero organic traffic. Many bug bounty programs (*e.g.,* Trello [8], United Airlines [38], and Piwik [33]) *explicitly* prohibit the use of automated scanners against their assets, mainly due to the large number of requests that they generate, which will be amplified when multiple researchers try to find vulnerabilities on the same websites.

2.1 WVS Functionality

Prior work has focused on comparing WVSs across dimensions related to their ability to discover vulnerabilities. We focus on the dimensions that are relevant for detecting their unwanted presence in incoming web traffic.

- **Target Dependence:** Target-independent tools send the same requests regardless of the targeted web application. These requests are typically aimed at identifying hidden directories, backup files, and other sensitive content that is not directly linked from a web application. Contrastingly, target-dependent WVSs first crawl the target web application and then launch a series of attacks against the identified endpoints.
- **Use of Browser Engine:** Some WVSs send requests to their target web applications through the use of simple HTTP libraries (API equivalents of `wget` and `curl`). In contrast, more sophisticated WVSs incorporate a full browser engine in their tool. This can be done by proxying all requests through a real browser or by actually embedding a headless version of a browser in their tools.

2.2 WVS Threat Model

Our threat model targets malicious actors abusing off-the-shelf web vulnerability scanners to scan target websites without the permission of their owners. We anticipate that attackers can use the full native capabilities offered to them by these tools, both in terms of attack vectors as well as stealthiness, to find as many vulnerabilities as possible while evading detection. Our goal is to fingerprint the incoming requests generated by these tools, differentiate them from the requests of regular users and benign bots, and enable administrators to apply one or more access-control

policies to the detected WVSs (*e.g.,* blocking their IP address). Even if a web application is secure against the types of vulnerabilities that WVSs are likely to find, we argue that knowing that a specific host or group of hosts are illicitly scanning a web application is of interest to administrators because it reveals an ongoing attack that can be countered early, before it escalates to other tools and attack vectors.

3 DATA COLLECTION

To be able to detect unwanted vulnerability-scanning activity on a web application, we must first understand how popular WVSs operate and analyze the type of traffic they generate. To this end, we developed a testbed consisting of real web applications that we can ask WVSs to scan for vulnerabilities. This testbed adopts state-of-the-art fingerprinting and monitoring techniques to extract as much information as possible about the connecting clients. This information will be later compared against the traffic that real users produce when visiting the same websites to build a supervised machine-learning classifier for differentiating between WVSs and users.

3.1 Web Applications

To understand the extent to which a given WVS's network activity is coupled to the web application that it is scanning, our testbed uses two different types of web applications. Given their popularity, we opted to deploy recent versions of WordPress and Joomla, two Content Management Systems that can be extensively customized and are together estimated of powering more than 40% of online websites [41, 42]. WordPress in particular is so popular that two of the WVSs that we evaluate are custom-made to only attack WordPress web applications.

3.2 Fingerprinting Setup

We follow the fingerprinting regime recently proposed by Li *et al.* [20] to build our fingerprinting capabilities. We augment the deployed web applications with traditional JavaScript-based browser fingerprinting, behavior fingerprinting, and TLS fingerprinting as described below:

Browser fingerprinting. Our testbed first evaluates a client's Java-Script support by invoking standard APIs related to AJAX requests and DOM manipulation. For example, the testbed uses JavaScript to create a new `<img>` tag on the client-side and append that image to the DOM. If the client requests that image, this indicates basic support of JavaScript. Similarly we fingerprint the client's support for security headers such as CSP and framing policies by adding additional resources in the webpage.

Following Li *et al.*'s intuition, a lack of basic security-mechanism support can reveal the presence of a non-standard client (*i.e.,* a WVS), regardless of that client's identity claims. Lastly, we check for the presence of ad-blockers by loading resources that are commonly attributed to advertisement libraries and checking whether the client browser loads and executes such scripts.

Behavioral fingerprinting. Behavioral-fingerprinting techniques analyze a client's browsing patterns such as visited pages, injected parameters, and payloads, server-response codes, and caching. Our testbed records the response code for each request so that we can analyze the response-code distribution (*i.e.,* ratio of successful vs. error HTTP codes) as part of our WVS analysis. Related to the aforementioned fingerprinting of security mechanisms, we also test to what extent WVSs load mixed resources (*e.g.,* loading a remote JavaScript file over HTTP, in an otherwise HTTPS-protected page), in search of behavior divergence compared to what all modern browsers do.

TLS fingerprinting. TLS fingerprinting extracts information from the TLS `Client Hello` message that a client sends to the server when attempting to establish an encrypted communication channel. Prior work has shown that this information can reveal the true nature of the connecting client since modern browsers present different support for TLS versions and ciphersuites, compared to command-line clients and HTTP libraries [19, 20]. We incorporate the FingerprinTLS library [7] in our system to passively collect TLS fingerprints.

3.3 Scanners Data Collection

To obtain a comprehensive view of Web Vulnerability Scanners (WVSs), we selected the top 10 open source WVSs from the list of top OWASP pentesting tools [28, 30], which included all scanners that are non-commercial and publicly available. We augment this list with two academic scanners: *Black Widow* [14] and *Enemy of the State* [11], resulting in 12 scanners. Though we do not expect academic scanners to be used for attacks in the wild, we opted to include two characteristic versions to evaluate the extent to which these scanners behave differently compared to popular WVSs. Table 1 lists the 12 scanners that we analyze in this study along with their corresponding version information. For each tool, we used the latest version available at the time of our analysis. We analyze the scanners and report their characteristics such as the number of requests they send, the crawling behavior, and their browser engine.

Overall, we run each scanner for 10 rounds, against both our WordPress and Joomla web applications. This results in a total of 240 experiment runs ($12\ WVSs \times 2\ webapps \times 10\ rounds$) for which our testbed collected extensive logs of the requests that WVSs sent, the responses these requests elicited, and the fingerprintable attributes of the WVSs during their runs.

3.4 User Data Collection

To identify how the traffic that real users produce when they interact with a web application is different from that of WVSs, we conducted an IRB-approved user study by hiring 159 online participants using the Amazon Mechanical Turk platform [25]. A summary of demographic information is included in Table 5 in the Appendix. Overall, we were able to collect user-browsing data for numerous different browsers and underlying platforms which we contrast against browsing data generated by WVSs.

4 SCANNER BEHAVIOR

In this section, we provide an in-depth analysis of web vulnerability scanners (WVS) through the lens of our collected dataset. Our aim is to understand how these scanners behave, how they are different from each other, and how their traffic can be differentiated from that of regular users browsing the same web applications. Along with the discovered statistics, we also present a series of observations which we later capitalize on, for our supervised ML detection of WVSs.

> ***O1.*** *The majority of scanners send a large number of requests, which can negatively affect the performance of web servers.*

While our testbed websites contain fewer than 20 pages and less than 100 resources (*e.g.,* JavaScript and CSS files), we observe that the

Table 1: List of web vulnerability scanner tools. Results represent the median of 10 runs.

Scanner Name	Version	Number of Requests (Median)	Site-Specific	Deterministic	Invalid URL Ratio	Browser-Based
WPScan(kali)	3.8.13	168	✓	✓	86.25%	✗
Arachni	1.5.1	220,822.5	✓	✓	13.98%	✓(Optional)
OWASP Zap	D-2020-12-21	128,346	✓	✓	4.90%	✓
WMap	1.5.1	29,183	✗	✓	98.67%	✗
Wapiti	3.0.3	50,970.5	✓	✓	5.06%	✗
Nikto	2.1.6	8,651.5	✗	✓	91.09%	✗
W3af	1.6.45	4,698	✓	✗	33.41%	✗
Skipfish (kali)	2.10b	11,464	✓	✗	43.50%	✗
Commix	2.9-stable	18,518	✗	✓	0.00%	✗
Google Tsunami	0.0.5	1,182.5	✓	✗	4.96%	✗
Black Widow	N/A	135,042.5	✓	✗	0.00%	✓
Enemy of the State	N/A	32	✗	✓	0.00%	✗

median number of requests per run is more than 1,000 requests for 10 out of 12 scanners. Looking at the scanners with the highest number of requests, we observe Arachni, Black Widow, and the OWASP ZAP sending out 220,823, 135,043, and 128,346 requests respectively.

Contrastingly, some scanners exhibit a small footprint. Namely, WPScan only sent 168 requests per run. WPScan is specific to WordPress platforms and is equipped with a list of vulnerable plugins and endpoints. Unlike the application-agnostic scanners in our dataset, WPScan does not inject its payloads in the identified fields and inputs of every page, and as a result, we observe fewer requests even on WordPress websites, compared to other WVSs. Enemy of the State (one of the two academic WVSs in our dataset) terminates early on in the scan process. This is most likely due to the tool not having been updated since its release. Nevertheless, we kept this tool in our dataset as its other features can still be used in our classifier for detection.

O2. *Some scanners have distinct exploration and attack phases which change based on the content of target web applications.*

Scanners with a distinct exploration phase initially crawl and map the structure of the target by issuing and modifying requests based on the server's response. Not all scanners, however, perform these two steps sequentially. Scanners like Arachni send their payloads as soon as they discover new entry points in the application.

Moreover, we observed that some scanners incorporate a hardcoded list of endpoints that they request while others dynamically mapped the web applications. To capture this effect into our data models, we categorize the browsing behavior of scanners into two major groups: *Site-specific* and *Deterministic*. Table 1 shows how different WVSs behave across these two categories.

The Site-specific attribute describes whether the WVS behaves differently based on the target web application. We compare the scan results of WordPress and Joomla for each scanner, and if we observe over 70% difference in the request URIs, we mark that scanner as site-specific. We chose the 70% threshold empirically to account for hybrid tools that send requests towards hardcoded endpoints as well as endpoints they discovered during their mapping phase. These hardcoded endpoints correspond to checks for sensitive resources including configuration files and common backup filenames.

On the other hand, WVSs that are not site-specific send out the same request URIs regardless of the target web application. Scanners like WMap and Nikto fall into this category.

O3. *Scanners may only use a subset of their attack vectors during each execution.*

We analyzed the scanner's behavior over multiple runs on the same web application and identified that certain scanners have randomness built into their scans, specifically in the list and order of the scanned files. In Table 1, we marked each scanner as deterministic if more than 70% of the requested files across multiple runs on the same web application are similar.

Unexpectedly, we discovered that a third of the scanners in our dataset (W3af, Skipfish, Google Tsunami, and Black Widow) show non-deterministic behavior. For instance, W3af only incorporates a subset of its payloads in every scan. Similarly, Black Widow has built-in randomization mechanisms to choose the next payload.

Even among deterministic scanners such as Nikto which scans for the same URLs over subsequent scans, we observe the randomization of a subset ($< 25\%$) of its payloads.

Overall, we consider the use of randomization by WVSs as less than ideal for vulnerability-detection purposes. In practice, the use of randomization means that any given vulnerability may remain undetected for long periods of time if it happens to not be selected in any given run. Particularly in the context of Continuous Integration/Continuous Delivery (CI/CD) pipelines, a vulnerability discovered by a non-deterministic scanner may be wrongly associated with the last commit that triggered the scan, sending developers down the wrong path for detecting it and fixing it.

O4. *Scanners focus on different endpoints and produce a large number of invalid requests compared to human visitors.*

This behavior is rooted in the design principles of web vulnerability scanners. We analyzed the scanned URIs by extracting the top terms using the TF-IDF algorithm. The results indicate that scanners place more emphasis on resources within the main pages of web applications, such as JavaScript resources and links.

We looked at the HTTP response-code distribution for each scanner, focusing on those associated with invalid URLs. We define the invalid URL ratio as the ratio of requests with an HTTP 404 response code compared to the total number of requests. Since we did not deliberately include any links to non-existing resources on our testbed websites, we do not expect normal browsing to lead to

Table 2: Request-based features. For HTTP-header-names and TLS fingerprints, we incorporate the information about the order of elements in the form of bigrams and trigrams.

Feature name	Type	N-grams
URI-Word	URI	Unigram
HTTP-header-name	Headers	Unigram,Bigram
HTTP-header-value	Headers	Unigram
TLSFP	TLS fingerprints	Unigram,Bigram,Trigram

any significant number of invalid requests. For non-site-specific scanners that incorporate a static list of potentially vulnerable or sensitive resources, we observe a large ratio of requests for non-existing files. This is reflected in the ratio of HTTP 404 errors produced for each scan (shown in Table 1). For example, A larger portion of requests from Skipfish target potentially sensitive files with extensions such as *.bak*, *.bat*, *.orig*, *.ver*. Nikto scans include keywords such as "passwd", "exe", "dir", and "formmail". Those keywords are part of scanned URLs, which point to sensitive files that may contain passwords and executable files. Overall, we can clearly attribute a higher invalid URL ratio to the probing activities of scanners.

O5. *Browser-based scanners have similar capabilities as human visitors.*

One category of features that is of interest for detection is the various types of browser fingerprints that a server can extract from a connecting client. Some scanners incorporate an HTTP library to generate their HTTP requests while others use instrumented browsers. We refer to the scanners that use instrumented browsers as browser-based in Table 1. Browser-based scanners are specifically harder to detect using traditional browser-fingerprinting techniques. The JavaScript capabilities and support for security mechanisms of these scanners will be similar (if not identical) to the capabilities exhibited by regular users. For example, the "Black Widow" WVS fully honors our CSP rules only requests CSS and images that are allowed by these rules.

Overall, we observe that there exist a number of dimensions where different WVSs exhibit different behaviors, not just from regular users, but also from each other. In the next section, we describe how we can capture these differences in features used to detect the presence of unwanted WVSs.

5 SYSTEM DESIGN OF SCANNERSCOPE

Having observed that users and WVSs exhibit different behaviors across our testbed, we incorporate these differences into an automated detection system. In this section, we introduce ScannerScope, a web-application agnostic, server-side tool for differentiating between WVSs and benign users.

A high-level view of ScannerScope is shown in Figure 1. ScannerScope is placed between the HTTP traffic reaching the server and the webserver(s) receiving and processing that traffic. ScannerScope routes client requests through its reverse proxy and relays them to the destination web server. Upon receiving responses from the webserver, ScannerScope then passes the responses back to clients. ScannerScope transparently augments the outgoing response pages with different fingerprinting modules and extracts fingerprints and statistical information from the requests. This information is then provided to the classifier module over an asynchronous message queue, decoupling the performance of ScannerScope from the overall performance of the protected web application.

5.1 Proxy Setup

The main component of ScannerScope is a reverse proxy. ScannerScope's reverse-proxy architecture allows it to intercept and analyze the incoming traffic regardless of the web applications being used, as well as stop malicious incoming traffic from ever reaching the web servers. Our reverse proxy consists of the following subcomponents: i) Fingerprinting modules, ii) Asynchronous Queue, iii) WVS classifier, and iv) Access-control module. ScannerScope automatically collects the browser and network-level fingerprints by appending fingerprinting resources described in Section 3.2 (*e.g.,* JavaScript fingerprints, CSP support, caching behavior, and TLS fingerprints) to the outgoing HTML pages and headers. When requests arrive, ScannerScope immediately routes them to the webserver. In parallel, it asynchronously sends a copy of each incoming request to the feature-extraction module; Based on the verdicts of our classifier, we can decide to block the requests from scanners using ScannerScope's access-control module.

5.2 Data Modeling

In this section we discuss the details of data modeling (such as, the process of vectorizing features) based on our prior observations, and identify the best performing machine learning models for use in ScannerScope.

5.2.1 Feature Extraction.

Based on our observations in Section 4, we extract features from request header and body, and categorize them in to Request-based and Capability-based features.

For the request-based features, we choose the request URI, HTTP headers and TLS fingerprints based on observations *O1-O4*. For request URIs, we only retain their values, while for HTTP headers and TLS fingerprints, we retain the bigram and trigram relationships to model their order. We use a TF-IDF vectorizer to extract distinctive terms from the request file paths, the request parameters, HTTP header names, and a subset of HTTP header values. To rule out the randomness (*O3*) from dynamic HTTP headers and avoid spoofing, we ignore the value of dynamic headers such as `Host`, `Referer`, `User-Agent` and `Cookies`. Note that 9/10 of the non-academic WVSs we evaluated in this paper, support the spoofing of User-Agent headers. We incorporate the information about the order of HTTP headers and TLS parameters by using bigrams and trigrams in our vectorization as listed in Table 2.

Capability Features describe the browsing environment of clients. Based on *O5*, we look at features extracted from our fingerprinting scripts which report on a client's support of JavaScript, CSP, Framing, mixed-content, and even the presence of ad-blockers. We refer to this set of features as Capability-based features. Unlike the request-based features, these features are tracked over a browsing session across multiple requests. As a result, ScannerScope's accuracy directly benefits from larger number of requests. For example, to determine whether the client supports CSP, we have to wait for the client to have a chance to load the resources on the web pages before we observe potential CSP violations.

5.2.2 Selection of Machine Learning Model.

Our initial dataset contains a total of 240 runs from WVSs, and 159 browsing sessions from human visitors. We split 80% of the dataset for training, and

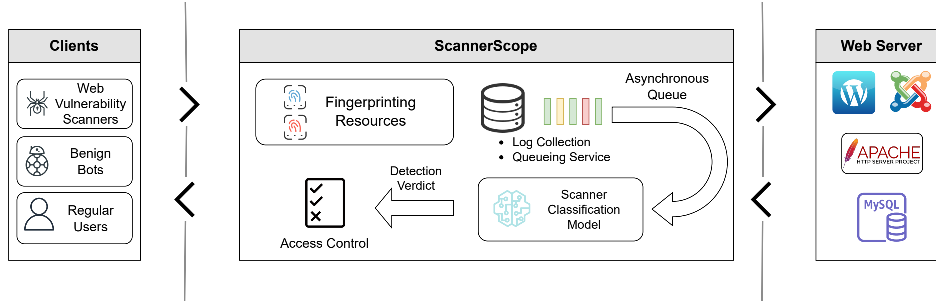

Figure 1: Architecture of ScannerScope. ScannerScope passes incoming HTTP requests to the webserver while making a copy of each request which is placed on its asynchronous queue to be consumed by the classifier. HTTP responses containing HTML content are modified by ScannerScope to include fingerprinting code before returning to clients.

Table 3: Capability-based features. Each feature category is implemented through certain number of tests reported under "Test #" column.

Category	Test #	Explanation
JavaScript	1	Does the client load our JavaScript Library?
	2-4	Does client execute JavaScript for loading images/performing AJAX requests?
	5	Does the client send JavaScript-computed fingerprints?
CSP Support	6-7	Does the client load CSP-allowed resources?
	8-11	Does the client load CSP-forbidden resources?
	12	Does the client send a CSP report when the page has violated CSP rules?
Framing Options Support	13-17	Does the client load iFrames and resources within that frame?
Mixed Content Rules	18-19	Does the client load HTTP content under HTTPS context?
Browser Extension	20-21	Is the client running an ad-blocker?

use the remaining 20% for testing. To capture the values of all features, we process the requests in batches, which we refer to as the *Window of requests.* We discuss the process to identify the optimal window-size in more detail in Section 6.1.1. We use the window-size of 15 for our experiments which provides us with 6,351 training and 1,572 test samples. Each sample in our dataset is composed of feature vectors for a batch of requests including the fingerprints and other request characteristics.

After extracting the aforementioned features, we evaluated multiple machine learning models such as decision trees, K-Nearest Neighbors (KNN), Support Vector Machine (SVM) and a gradient boosting model (LightGBM). After training and testing the different classifiers, we experimentally identified that LightGBM results in the best performance and accuracy numbers among the evaluated models. As such, ScannerScope incorporates LightGBM in its classifier.

6 RESULTS

In this section, we report the accuracy of our ScannerScope's classification module in detecting the traffic from the web vulnerability scanners. We begin by evaluating the effect of batching requests in ScannerScope' performance, and then evaluate ScannerScope in increasingly challenging settings. Lastly, we evaluate ScannerScope's performance overhead on the protected web applications.

6.1 Classifier Performance

The goal of ScannerScope is to differentiate between WVS and non-WVS traffic (i.e., human visitors as well as benign bot traffic). Our

Table 4: Performance of classifier under various training and testing conditions. For the first three rows, the Model is represented by the Training-Testing dataset. ScannerScope generalizes well for unseen scanners while maintaining high accuracy.

Model	Accuracy	Precision	Recall	F1-score
WordPress-WordPress	99.30%	97.79%	99.58%	98.66%
Joomla-Joomla	99.22%	99.17%	99.14%	99.15%
WordPress-Joomla	91.44%	92.52%	91.44%	91.39%
1 Unseen Scanner	98.27%	96.71%	98.53%	97.43%
4 Unseen Scanners	96.20%	93.65%	97.13%	95.19%
6 Unseen Scanners	91.26%	85.50%	94.38%	87.66%

classifier operates on batches of requests, as opposed to individual ones. This batching allows us to determine the value of various fingerprintable properties of a client (such as its support for CSP) and not prematurely ask for a classification decision with incomplete data.

As a result, before analyzing the accuracy of ScannerScope, we need to identify the optimal window size, i.e., the number of consecutive requests from each client that need to be batched together before ScannerScope can provide a high-confidence verdict.

6.1.1 *Finding the Optimal Window-size.*

In order to determine the optimal window size for our classifier, we plot the model performance over a range of sample window sizes depicted in Figure 2. The window size has a unique effect on different features. Most notably, the URL features and Capabilities benefit the most from larger window sizes. In our setup, most capabilities are verified across multiple requests and by observing the presence or absence of requests towards certain resources (*e.g.,* loading images and iframes). Thus, these features benefit the most from larger window sizes, increasing the window size from one request to 20 requests boosts the accuracy of this feature from 50% to 90%.

Conversely, we observe that some features do not necessarily benefit from larger window sizes. For instance, looking at the HTTP headers in Figure 2 for WordPress-Joomla (Red line), the accuracy slightly drops as we increase the window size. We attribute this effect to the over-fitting of our classifier when trained on larger windows. Therefore, we choose 15 requests as the window size for our model to detect WVS. Combining multiple features increases the overall accuracy across all window sizes, and a window size between 15 to 20 requests achieves the best accuracy for all testing scenarios.

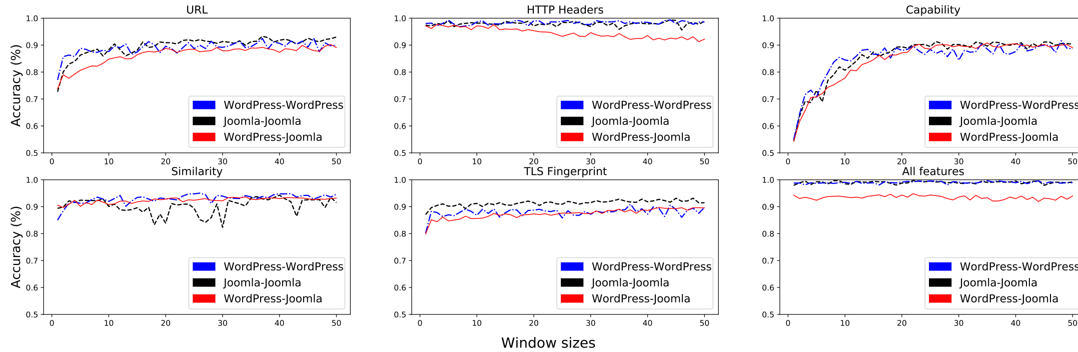

Figure 2: Accuracy of the classifier based on feature groups when trained on various window sizes. The blue line represents WordPress-WordPress test accuracy, the black line represents Joomla-Joomla test accuracy, and the red line represents WordPress-Joomla cross-test accuracy.

6.1.2 Accuracy and Precision Results. Table 4 reports the performance numbers of trained models under various combinations of training and test data. First, we report the performance numbers when we train and test the classifier on the same web application. Under this setup, ScannerScope is able to achieve a high accuracy across both of our testbed web applications (99.30% on WordPress and 99.22% on Joomla).

6.1.3 Time to Detect a WVS. ScannerScope performs its classification for each incoming window of requests. As a result, the time it requires to flag a WVS is 15 requests in our current setup. Given that a typical WVS issues thousands to hundreds of thousands of requests (See Table 1), this means that ScannerScope can block WVSs traffic long before they can report any meaningful findings.

To further put this number into perspective, we crawled the top 1,000 domains from the Majestic Million ranking [22], and measured the number of requests required to load the homepage of each website on the list. On average, loading each web page results in 132 requests. Given that ScannerScope flags WVS with high accuracy in 15 requests, it will block scanners nine times faster than the time it takes for a normal browser to load the home page of a popular website (*i.e.,* typically, in less than a second). Moreover, the computational overhead of ScannerScope adds as little as a 2% performance overhead to the web server; the detailed performance analysis is available in the Appendix.

6.1.4 Feature Group Importance. In this section, we first evaluate the prediction power of individual feature groups and then investigate the importance of individual features. To this end, we train our classifier on each feature group separately. Figure 3 shows the accuracy of ScannerScope when trained on individual feature groups. One can observe that most feature groups can individually provide an accuracy of over 80%. For instance, training a classifier on URLs only leads to 92% accuracy, and combining URLs with HTTP headers improves the accuracy to 98%. We observe a similar trend when combining Capability and Similarity-based features.

Overall, we observe that most of the feature groups provide exhibit strong predictive powers, particularly when combined together. Although a subset of features may already provide high accuracy, we argue that training ScannerScope on a wide range of features allows it to better deal with unseen scanners as well as potential evasion attempts by attackers. We evaluate a sophisticated future attacker's ability to evade ScannerScope in Appendix C.

Next, we evaluate the importance of individual features. Our model contains 26,494 features, with the majority of them being vectorized terms extracted by the TF-IDF algorithm. Our features are made up of 845 HTTP header terms, 25,445 URL terms, 21 capability-based, and 8 similarity-based features. Figure 4 visualizes the importance of features with an impact factor of greater than five. Interestingly, a small subset of terms extracted from the URLs and request parameters have a high impact on the classification results. Overall, vectorized URLs and HTTP headers are among the most impactful features in ScannerScope's model. Nevertheless, capability and similarity-based features also provide significant support.

6.1.5 Dealing with False Positives. In the deployment setting of ScannerScope, false positives are important since they can potentially prevent human users from reaching the protected web applications. In our setup, we reported the accuracy and precision for a one-time detection decision; in actual deployment, the model can be configured to detect multiple times and provide a confidence score to accommodate various scenarios. As mentioned earlier, apart from reconfiguring the model, the access-control module can also be configured to block the IP address for a certain amount of time or incorporate CAPTCHAs to reduce the probability of blocking organic web users. Due to space constraints, the detailed analysis of false positives for bots and human requests is available in Appendix B.

6.2 Classifier Robustness against New Web applications and Unseen WVS

New Web Applications. We measure the robustness of the classifier when trained on WordPress and tested on Joomla. This setup models the generalizability of our classifier and to what extent ScannerScope needs to be trained on the same web application as the one tasked to protect. For certain features such as browser fingerprints and capabilities, the choice of the web application will not affect ScannerScope's accuracy. Conversely, for features that depend on the structure of the web applications (*i.e.,* URIs and Similarity), a change in the structure of the web application may affect the results. As such, we devise this setup to measure the effect of transferring the trained model to other web applications. As listed on the third row of Table 4, training ScannerScope on WordPress and testing on Joomla yields a 91.44% accuracy. This level of accuracy is likely too low to be coupled with the automated blocklisting of clients but could be coupled with less intrusive countermeasures (such as CAPTCHAs) if retraining ScannerScope is not an option in a given deployment.

Unseen WVS. We also measure the performance of ScannerScope's classifier in detecting unseen scanners (shown in the last three rows of Table 4). To this end, we started removing the scanners from our

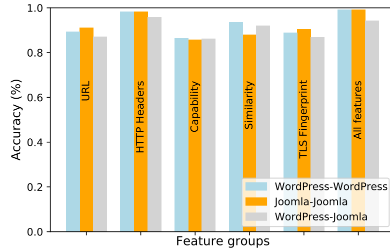

Figure 3: Accuracy of feature groups when training the classifier only on one or a subset of feature groups at a time.

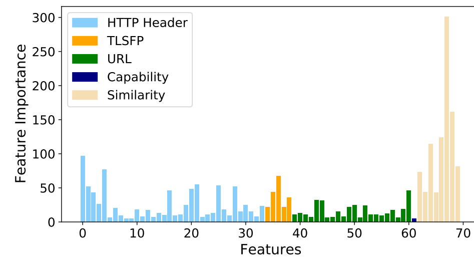

Figure 4: Model Feature Importance for WordPress-WordPress test. Figure shows the importance of individual features from various feature groups for importance value > 5. The X-axis represents individual features, Y-axis represents the feature importance value from the model.

training set, gradually increasing the number of "unseen" scanners. By removing each scanner from the training set, retraining, and testing with the omitted scanner, ScannerScope achieves an average accuracy of 98.27% for this leave-one-out setup. Even if we remove 33% and 50% of all scanners from the training dataset, our classifier still retains a high accuracy of 96.20% and 91.26% respectively. This high accuracy demonstrates the strength of our classifier in extracting generic patterns from the scanners which can be generalized to unseen scanners. The evaluation of ScannerScope's ability to resist adversarial attacks is available in Appendix C.

7 RELATED WORK

Even though web vulnerability scanners have been evaluated extensively in related work, the majority of that work is focused on whether WVSs can identify known vulnerabilities and how one could increase their crawling coverage in order to be able to identify as many vulnerabilities as possible. To the best of our knowledge, this paper is the first one that seeks to evaluate WVSs, not on their vulnerability-detection performance, but on the type of HTTP traffic that they generate and to what extent they are fingerprintable and differentiable from regular users. We briefly review the related work on the performance of WVSs as well as the most relevant work from the closely-related field of bot detection.

Evaluating WVS Performance

Prior work has evaluated the effectiveness of web vulnerability scanners through different metrics, including their ability to crawl the deeper states of web applications and to what extent scanners can identify vulnerabilities on known vulnerable web applications [5, 12, 23, 24, 35, 36, 39]. In general, these works reported that even though there clearly are differences between different tools in terms of their abilities to discover vulnerabilities, no tool is always the best across all possible deployments. Moreover, even the best-performing WVSs have difficulty navigating web applications that make heavy use of JavaScript, such as in modern single-page applications.

In contrast with prior work, this paper's focus is not whether WVSs can discover complicated vulnerabilities but whether they can be detected by web applications in scenarios where attackers deploy WVSs against sites without their permission.

Detecting Bots and WVSs

In recent years, there has been an increased interest in web-bot detection, which is powered by the continuous migration of software to the web and the ever-increasing attacks facilitated by malicious web bots, including credential stuffing, content scraping, and vulnerability exploitation.

Prior work has proposed numerous approaches for detecting web bots using fingerprinting techniques [6, 9, 16], deception [20, 31, 40], and supervised as well as unsupervised machine learning techniques based on the frequency and type of web requests [17, 18, 21, 37, 43]. While the detection component of ScannerScope shares some similarities with prior bot-detection methods, our starting point is that of known web vulnerability scanners (as opposed to unknown bots behaving maliciously) that we first analyze in order to understand their behavior at a network level. In this way, ScannerScope benefits from accurate ground truth (compared to the best-effort labeling of available HTTP traffic followed by prior work) and is not affected by low-and-slow scanners or scanners with spoofed identities. Moreover, because of our user study (where we collected HTTP traces from real users interacting with the protected web applications) we can accurately measure ScannerScope's performance and its ability to avoid false positives (*i.e.,* flagging users as WVSs).

8 CONCLUSION

In this paper, we systematically explored the behavior and capabilities of web vulnerability scanners (WVSs). We developed a testbed that can automatically launch WVSs and collect their behavior and network-level fingerprints. Using this testbed, we identified WVS differences regarding a tool's browsing engine, whether its scans are deterministic, and its distribution of HTTP requests that result in errors.

To understand how the traffic that a typical WVS generates when scanning a web application is different from that of regular users interacting with the same web application, we conducted a user study with 159 participants. By comparing the two datasets, we observed large differences that could be capitalized for differentiating between the two populations. To that end, we proposed ScannerScope, a server-side, application-agnostic tool that can identify unwanted WVS in incoming traffic and apply one or more access-control policies against them. We showed that ScannerScope exhibits high detection accuracy (over 99%), which it mostly retains even in less-than-favorable deployments. Moreover, ScannerScope has the ability to resist future attacker evasions without incurring a noticeable performance overhead on the web applications that it protects and with near-zero false positives on human-visitor traffic samples.

ACKNOWLEDGMENTS

This work was supported by the Office of Naval Research (ONR) under grant N00014-20-1-2720 as well as by the National Science Foundation (NSF) under grants CNS-1813974, CNS-2126654, and CNS-2211575.

Availability: To facilitate and advance research on the topic of scanner detection, we make our datasets available to other researchers at https://scan-me-if-you-can.github.io/.

REFERENCES

[1] 2022. Acunetix online scanner. https://www.acunetix.com/online-vulnerability-scanner/.
[2] 2022. Tenable cloud web scanner. https://www.tenable.com/products/tenable-io.
[3] apachebench 2022. AB - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.
[4] arachni 2022. Arachni Web Application Security Scanner Framework. https://www.arachni-scanner.com/.
[5] Andrew Austin and Laurie Williams. 2011. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*.
[6] Babak Amin Azad, Oleksii Starov, Pierre Laperdrix, and Nick Nikiforakis. 2020. Web runner 2049: Evaluating third-party anti-bot services. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.
[7] Lee Brotherston. 2022. TLS Fingerprinting library. https://github.com/LeeBrotherston/tls-fingerprinting. https://github.com/LeeBrotherston/tls-fingerprinting
[8] BugCrowd. 2022. Trello bug bounty program. https://bugcrowd.com/trello.
[9] Elie Bursztein, Artem Malyshev, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight device class fingerprinting for web clients. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*.
[10] Cybersecurity and Infrastructure Security Agency. 2021. Apache Log4j Vulnerability Guidance. https://www.cisa.gov/uscert/apache-log4j-vulnerability-guidance.
[11] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st {USENIX} Security Symposium ({USENIX} Security 2012)*.
[12] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.
[13] Peter Eckersley. 2010. How unique is your web browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer.
[14] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black Widow: Blackbox Data-driven Web Scanning. *IEEE Symposium on Security and Privacy* (2021).
[15] Alicia Hope. 2021. Massive Cyber Attacks Target F5 BIG-IP Critical Vulnerabilities After Firm Releases Updates. https://www.cpomagazine.com/cyber-security/massive-cyber-attacks-target-f5-big-ip-critical-vulnerabilities-after-firm-releases-updates/.
[16] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *2021 IEEE Symposium on Security and Privacy*.
[17] Gregoire Jacob, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2012. PUBCRAWL: Protecting Users and Businesses from CRAWLers. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 507–522. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jacob
[18] Steve TK Jan, Qingying Hao, Tianrui Hu, Jiameng Pu, Sonal Oswal, Gang Wang, and Bimal Viswanath. 2020. Throwing darts in the dark? detecting bots with limited data using neural data augmentation. In *2020 IEEE Symposium on Security and Privacy*.
[19] Brian Kondracki, Babak Amin Azad, Oleksii Starov, and Nick Nikiforakis. 2021. Catching Transparent Phish: Analyzing and Detecting MITM Phishing Toolkits. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.
[20] Xigao Li, Babak Amin Azad, Amir Rahmati, and Nick Nikiforakis. 2021. Good bot, bad bot: Characterizing automated browsing activity. In *2021 IEEE symposium on security and privacy*.
[21] Anália G Lourenço and Orlando O Belo. 2006. Catching web crawlers in the act. In *Proceedings of the 6th international Conference on Web Engineering*.
[22] majestic 2022. Majestic Million. https://majestic.com/reports/majestic-million.
[23] Yuma Makino and Vitaly Klyuev. 2015. Evaluation of web vulnerability scanners. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*.
[24] Balume Mburano and Weisheng Si. 2018. Evaluation of web vulnerability scanners based on owasp benchmark. In *2018 26th International Conference on Systems Engineering (ICSEng)*. IEEE.
[25] mturk 2022. Amazon Mechanical Turk. https://www.mturk.com/.
[26] NIST. 2021. CVSS Severity Distribution Over Time. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time.
[27] Charlie Osborne. 2021. Critical remote code execution flaw in thousands of VMWare vCenter servers remains unpatched. https://www.zdnet.com/article/critical-remote-code-execution-flaw-in-thousands-of-vmware-vcenter-servers-remains-unpatched/.
[28] owasp 2022. Free for Open Source Application Security Tools. https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools.
[29] owasp 2022. OWASP Zed Attack Proxy (ZAP). https://owasp.org/www-project-zap/.
[30] owasp 2022. WSTG - v4.1, Testing Tools Resource. https://owasp.org/www-project-web-security-testing-guide/v41/6-Appendix/A-Testing_Tools_Resource.
[31] KyoungSoo Park, Vivek S Pai, Kang-Won Lee, and Seraphin B Calo. 2006. Securing Web Service by Automatic Robot Detection.. In *USENIX Annual Technical Conference, General Track*.
[32] Davor Petreski. 2019. Integrating Web Vulnerability Scanners in Continuous Integration: DAST for CI/CD. https://blog.probely.com/integrating-web-vulnerability-scanners-in-continuous-integration-dast-for-ci-cd-7637eaff26bd.
[33] Piwik. 2022. Piwik Pro bug bounty program. https://piwik.pro/security-bug-bounty-programat-piwik-pro/.
[34] PortSwigger. 2019. CI/CD security testing. https://portswigger.net/developers/ci-cd-security.
[35] Sugandh Shah and Babu M Mehtre. 2015. An overview of vulnerability assessment and penetration testing techniques. *Journal of Computer Virology and Hacking Techniques* (2015).
[36] Larry Suto. 2010. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February* (2010).
[37] Pang-Ning Tan and Vipin Kumar. 2004. Discovery of web robot sessions based on their navigational patterns. In *Intelligent Technologies for Information Analysis*. Springer.
[38] UnitedAirlines. 2022. United Airlines bug bounty program. https://www.united.com/ual/en/us/fly/contact/vdppolicy.html.
[39] Tom Van Goethem, Frank Piessens, Wouter Joosen, and Nick Nikiforakis. 2014. Clubbing seals: Exploring the ecosystem of third-party security seals. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.
[40] Nikos Virvilis, Bart Vanautgaerden, and Oscar Serrano Serrano. 2021. Changing the game: The art of deceiving sophisticated attackers. In *2014 6th International Conference On Cyber Conflict (CyCon 2014)*. IEEE.
[41] w3techs 2022. Usage statistics and market share of Joomla. https://w3techs.com/technologies/details/cm-joomla.
[42] w3techs 2022. Usage statistics and market share of WordPress. https://w3techs.com/technologies/details/cm-wordpress.
[43] Guowu Xie, Huy Hang, and Michalis Faloutsos. 2014. Scanner hunter: Understanding http scanning traffic. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*.
[44] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. 2012. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications.. In *NDSS*.

APPENDIX

A DATA COLLECTION DETAILS

Our user data collection is done through Amazon Mechanical Turk (AMT) Platform. On AMT, we deployed two Human Interaction Tasks (HIT) with a total of 100 participants (50 for WordPress and 50 for Joomla).

Each participant was given access to either a WordPress or a Joomla installation (populated with mock content) and was given a list of tasks that they had to complete. These tasks included typical user behavior that one could expected on a Content Management System (CMS), such as reading articles, posting comments, and searching for specific keywords. Each action translates to tens of client-side requests corresponding to user clicks, form submissions, the loading of images, JavaScript, CSS, etc. The participants received a randomized list of tasks to ensure that the order of their requests was different so as to more faithfully mimic the actions that real users would perform on a CMS. An example of the task lists that were given to participants is available in Table 6.

To ensure that our webserver logs did not contain traffic from web bots that discovered our web applications during the period of our user study, each user was provided with a unique token embedded in their URLs. These tokens were removed during post processing and any requests that were lacking these tokens (i.e. they did not originate from our HITs) were discarded. At the end of our study, we observed that we had recorded information for more than 100 participants since some participants started their tasks but

Table 5: Demographic data of AMT user study participants.

Web application	User count	Number of requests (median)	Time to finish task (median)
WordPress	77	592	0:16:19
Joomla	82	823	0:11:56

never finished them. We opted to keep these requests since they still account for valid user-browsing patterns (*e.g.,* reading a single article and then leaving the website).

In total, we recorded 159 user browsing sessions, consisting of 77 WordPress users and 82 Joomla users. On average, each participant spent 15 minutes on our websites. WordPress users required a median of 16 minutes to finish the assigned tasks, compared to 12 minutes for Joomla users. Each participant who completed the task list was paid $0.5. The vast majority of users (91.10%) navigated to our websites using a Google Chrome browser, with the remaining users (8.9%) completing their list of tasks using Mozilla Firefox, Microsoft Edge, Safari, and Opera. A total of 106 (66.7%) participants used Microsoft Windows 10, while the rest of 53 (33.3%) participants used other operating systems including Windows 8.1, Mac OS X, and Android. All of these statistics were extracted from the participants' User-Agent headers (based on the findings of prior browser-fingerprinting studies [13, 44], we assume that AMT users are highly unlikely to be spoofing their User Agents).

B ANALYSIS OF FALSE POSITIVES OF SCANNERSCOPE

Benign bots. Given that ScannerScope is trained on WVS vs. human-user data, we seek to understand whether the traffic originating from benign bots looks more like human-user traffic, as opposed to vulnerability-scanning traffic.

To measure this, we extracted the search-engine bot traces (including Google Bot, Bing bot, as well as other smaller benign bots) from the Good-Bot-Bad-Bot dataset by Li *et al.* [20]. Given that our classifier uses similar web applications and fingerprinting features as those used by Li *et al.*, we were able to successfully extract features from their dataset (we pick one month of traffic at random from Li *et al.*'s dataset and focus on the requests labeled as belonging to well-known benign bots) and pass them to ScannerScope's classifier, as if these requests would have arrived on our own web applications.

Out of 411 search engine bot IP addresses, ScannerScope marked 408 (99.27%) as non-scanners. Upon further analysis, we identified that the Similarity-based features, HTTP headers, and URLs are the determining features that tell search engine bots from vulnerability scanners apart. More specifically, we observed that while scanners often request invalid resources that result in HTTP 404 response codes, search engine bots mostly request valid resources.

False positives on human requests. Although ScannerScope exhibits high accuracy in detecting scanners, false positives are still costly (from a business perspective) when they occur. Looking at the test results of training on WordPress and testing WordPress, we overall observe 238 true negatives, 0 false positives, 11 false negative, and 1,323 true positives. Across the WordPress and Joomla experiments, we observe a false positive rate ranging from 0% to 0.86%.

To identify possible skews in our user study dataset due to the geographic distribution of AMT workers, we verified the distribution of locale-related HTTP headers. From this perspective, we found that all users report "en_US" locale as their preferred language setting under "Accept-Language" HTTP header. At the same time, less than 20.1% of users advertised multiple locales such as "en-GB" or "en-IN". In comparison, in the WVS dataset, we observed that some scanners do not send the "Accept-Language" header, while others advertised "en-US". As a result, language-related HTTP header preferences due to the geographic distribution of AMT users, should not skew the fingerprints.

C ROBUSTNESS AGAINST ADVERSARIAL ATTACKS

In Section 6.1, we demonstrated the high accuracy and precision of ScannerScope in successfully classifying WVS traffic even in challenging deployment scenarios, such as, when considering unseen scanners or when the testing web application is different from the training one. In this section, we evaluate the robustness of our model against adversarial attacks. To that end, we simulate sophisticated attackers who have the ability to modify and spoof certain properties of their scanners beyond the options provided through their configurations. These properties could be spoofed either by changing the source code of the tools or by proxying all connections at the client-side and rewriting fields appropriately. While these attack scenarios are expressly outside our threat model (Section 2), we evaluate them to understand the detection limits and degradation behavior of ScannerScope.

In the first adversarial scenario, we consider attackers that can modify arbitrary HTTP headers from their scans. Even though we have already removed the easily modifiable HTTP headers from our training data (*e.g.,* User-agent and cookies), attackers may still modify other headers (such as encoding, content length, etc.) which are not particularly crucial for web servers, in an effort to evade classifiers that are relying on them. Separate from modified HTTP request headers, we also explore the robustness of our classifier against attackers with modified TLS fingerprints.

For these two experiments, we gradually replace the HTTP headers and TLS fields from scanners with values from human-visitor traffic and measure the drop-off in the accuracy of our ScannerScope's classifier. We refer to the ratio of replaced fields in Figure 5 as the "adversarial rate" which ranges from 0 to 1. The value of one denotes that the WVS features are fully replaced with non-WVS samples.

As expected, gradually replacing the features with the opposite class lowers the accuracy until the majority of scanner samples are classified as non-scanners. As evident in Figure 5, replacing merely 10% of the headers from the requests of scanners can degrade the accuracy down to 61.86% for the classifier only trained on HTTP headers. Similarly, replacing 20% of HTTP headers with human-request samples results in 57.21% accuracy. These numbers quantify the ability of attackers to bypass detection for classifiers that only focus on a subset of easily modifiable scanner properties; Performing the same test on TLS fingerprints yields similar results.

D PERFORMANCE OVERHEAD IN PROXY MODE

ScannerScope is meant to be deployed inline with the web applications that it is protecting. At the same time, through the use of asynchronous queues and ML classifiers, we have designed ScannerScope to have a minimal impact on a web application's performance. In this section, we report on the performance overhead of ScannerScope.

Table 6: Sample list of tasks given to AMT user study participants. The participants must follow the user instructions and provide the responses to survey questions based on the content of our websites.

Question #	User instructions	Applied web applications
1	Click article <article_name>, and type the last word of the first paragraph.	WordPress / Joomla
2	Click the article <article_name>, and type the last word of the article.	WordPress / Joomla
3	Click the article <article_name>, and type the last word in second paragraph.	WordPress / Joomla
4	Click article <article_name>, and type the first word of last paragraph.	WordPress / Joomla
5	Click the article <article_name>, and type the year (4-digit number) appeared in second paragraph.	WordPress / Joomla
6	Click article <article_name>, How many ranks (rows) are there?	WordPress / Joomla
7	Scroll down to the end of main page, then search for word <keyword>, type 'done' in the textbox.	WordPress / Joomla
8	Scroll to article <article_name>, then click the 'Uncategorized' tag, type the first article name (two words).	WordPress
9	Navigate to article <article_name>, then leave today's date as a comment, type 'done' in the textbox.	WordPress

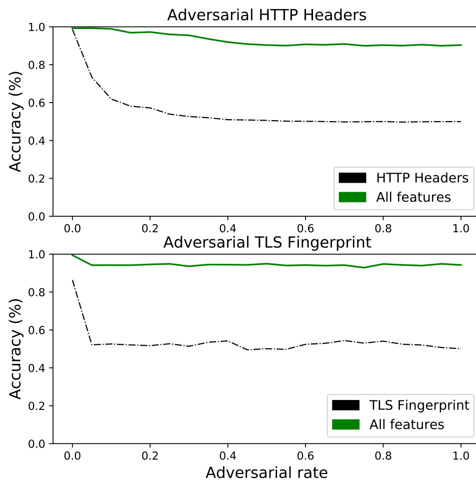

Figure 5: Model robustness when replacing scanner data with user data. Using a combination of different feature groups greatly enhanced the model's robustness.

We use the ApacheBench [3] HTTP benchmarking tool as our client and send 1,500 requests. We repeat the test three times and report the response time in Figure 6 for WordPress and Joomla. When serving WordPress and Joomla using only Apache (*i.e.,* without a reverse proxy), the average response time is 295*ms*, and the median response time is 287*ms* for WordPress. Similarly, when serving a Joomla web application, the average response time is 355*ms*, with a median of 350*ms*.

When we introduce an Nginx reverse proxy that merely relays requests and responses between users and the web server, WordPress tests show an average of 328*ms* response time, while Joomla tests receive the response in 373*ms*. This slight increase in the response time is solely due to the introduction of a reverse proxy into the setup. Lastly, Figure 6 shows the CDF of response times when ScannerScope is introduced and is fully operational. There, we observe an average response time of 357*ms* for WordPress and 389 ms for Joomla.

As one can observe in Figure 6, the CDF of Nginx vs. ScannerScope are virtually indistinguishable, with Nginx alone adding $6 - 11\%$ performance overhead over the Apache-only setup compared to the $4 - 9\%$ overhead of ScannerScope. This means that ScannerScope adds negligible overhead in deployments where a reverse proxy is already present, *i.e.,* when load balancers and other inline devices are already deployed which we argue is the vast majority of the modern web.

E ETHICS

Our user study is approved by the Office of Research Compliance of our institution under the IRB exemption category 45 CFR 46.104. d.3(i)(A).

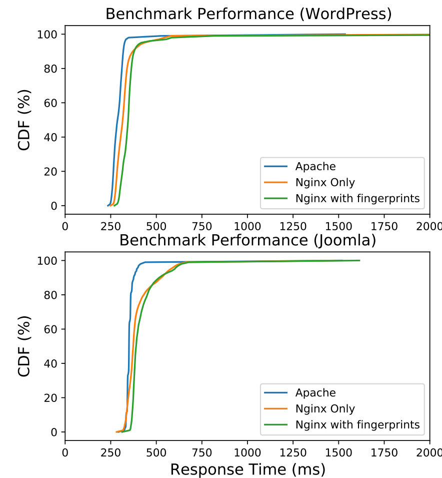

Figure 6: CDF of Response times; The top figure represents WordPress results, Bottom figure represents Joomla results. Deploying ScannerScope has a negligible effect on the overall response time.