SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements

Aleksei Stafeev
CISPA Helmholtz Center for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security

Abstract

Web crawlers are tools widely used in web security measurements whose performance and impact have been limitedly studied so far. In this paper, we bridge this gap. Starting from the past 12 years of the top security, web measurement, and software engineering literature, we categorize and decompose in building blocks crawling techniques and methodologic choices. We then reimplement and patch crawling techniques and integrate them into `Arachnarium`, a framework for comparative evaluations, which we use to run one of the most comprehensive experimental evaluations against nine real and two benchmark web applications and top 10K CrUX websites to assess the performance and adequacy of algorithms across three metrics (code, link, and JavaScript source coverage). Finally, we distill 14 insights and lessons learned. Our results show that despite a lack of clear and homogeneous descriptions hindering reimplementations, proposed and commonly used crawling algorithms offer a lower coverage than randomized ones, indicating room for improvement. Also, our results show a complex relationship between experiment parameters, the study's domain, and the available computing resources, where no single best-performing crawler configuration exists. We hope our results will guide future researchers when setting up their studies.

1 Introduction

Web security measurements have become invaluable tools for empirical investigations exploring various security and privacy-related issues from real distributions, such as the prevalence of in-the-wild vulnerabilities [37,45,56] and the effectiveness of defense mechanisms [59,78] at scale, to name a few. An essential technique enabling such measurements is *web crawling*, which starts from a list of seed websites, e.g., Alexa, Tranco [62], and CrUX [1], and then iteratively browses web pages to discover as many states as possible.

Crawlers' effectiveness in finding new states relies on determining whether a state is new via *page similarity* algorithms, and executing the appropriate actions, using *navigation* algorithms. Both types of algorithms have been challenged by increasingly dynamic pages and complex web interface logic, leading to partial exploration, thus missing potentially vulnerable states, ultimately affecting the precision and accuracy of web measurement studies. Over the past decades, the research community has proposed new crawling techniques to address these shortcomings, often combining novel page similarity and navigation algorithms, e.g., `EotS` [20], `Black Widow` [25], and `FeedEx` [27]. Although web crawlers have become more sophisticated, their impact has been limited as prior works use rather elementary crawling techniques.

Assessing the effectiveness of crawlers and their use in empirical studies is challenging, primarily because prior works have only marginally addressed comparative evaluations of crawling techniques. Most comparative evaluations are often conducted in isolation when assessing a new approach's effectiveness, using heterogeneous experimental parameters, testbeds, and datasets, making results difficult to transfer across papers. Only recently, independent comparative studies [4,80] measured the accuracy and precision of different crawling algorithms. However, either they covered only page similarity from domains like computer vision, leaving out approaches used in practice [80], or evaluated web automation tools like Puppeteer [2] and Selenium [3], which do not implement any crawler algorithm. As a result, we still know little about crawlers' effectiveness as used in security evaluations.

In this paper, we bridge this gap with one of the first systematizations of the current state-of-the-art web crawling techniques—both proposed and used in practice—and an assessment aimed at measuring performance in concrete application scenarios such as testing and data collection in live websites. As a first step, we review the past 12 years of publications from the top venues in security, privacy, and web measurements, i.e., 7,840 papers, identifying 403 conducting a measurement, identifying algorithms, parameters, and methodologies for collecting data through crawling. We then review prior works proposing new crawlers in the top three software engineering conferences (917 papers), identifying additional 27 papers that proposed, in total, 35 crawling techniques proposing the first taxonomy of crawling algorithms. Finally, we reimplement and patch 27 algorithms and variants into our evaluation framework called `Arachnarium` and run them against popular web applications and in-the-wild websites to measure code coverage, link, and JavaScript source code discovery. We also assess the impact of the methodological choices identified in our literature review, estimating the impact of algorithmic and parameter adjustments. Finally, we summarize and put into perspective our findings within the research community. We offer concrete inputs for future large-scale security studies and insights on the current state-of-the-art algorithms in the web crawling domain for security testing.

Overall, we observed that descriptions of methodological parameters and algorithms are often insufficient, incomplete, and not self-contained, making reimplementation challenging. For example, 24 papers in the crawling algorithms survey do not contain sufficient details about the implemented algorithms. We discovered that breadth-first search (BFS) and URL comparison are the two most popular techniques among the ones with adequate descriptions. The empirical evaluation of all techniques shows that randomized algorithms, in particular randomized BFS, offer better performance than standard algorithms, providing a general increase of average coverage, from +8% in LoCs up to +21% for JavaScript source code. Our results also identified time breakpoints after which a crawling algorithm performance increment becomes more appreciable, i.e., 30, 50, and 100 seconds for code, link, and JS source code coverage.

To summarize, we make the following contributions:

- We present a systematization of knowledge of web crawlers as used in security and privacy, and web empirical measurements based on the review and analysis of 403 papers;
- We decompose the identified algorithms and organize them in a taxonomy of proposed and used-in-practice crawling algorithms;
- We propose and release `Arachnarium`, a framework for the comparative evaluation of web crawlers against real-size web applications and live websites;
- We evaluate algorithms both individually and in combination, assessing their performance across metrics extracted from real application scenarios identified in our survey, revisiting the impact of prior methodologic decisions;
- We present 14 insights, lessons learned, and recommendations for our community and researchers developing and using crawlers.

Open Science Statement — We release `Arachnarium` source code[1] and experiment data[2].

[1]https://github.com/pixelindigo/arachnarium/tree/sec24
[2]https://github.com/pixelindigo/state-of-the-krawlers

2 Systematization

The first part of our paper is a systematization of large-scale security and web measurement studies using crawlers as a means of data acquisition (Section 2.1). Then, we systematize prior work proposing and combining new techniques, where we decompose and organize them in a taxonomy for crawling algorithms (Section 2.2).

General methodology — Two researchers executed the survey and systematization of this section. Both defined selection criteria, analysis rules, and executed data collection. One read and processed each paper and the other one validated results via random sampling. We detail the remaining steps of our analysis in the remaining sections.

2.1 Web Measurements

2.1.1 Analysis Methodology

Survey criteria — We selected all published papers from the security and web measurements conferences of the past 12 years (from 2010 to 2022). We selected the USENIX Security Symposium, the IEEE S&P Symposium, the ACM CCS, the NDSS Symposium, the ACM IMC, the ACM WWW and the PET Symposium, covering a total of 7,840 papers, and performed a keyword-based search to identify web measurement papers. We downloaded papers matching any of these keywords: `tranco`, `alexa`, and the combination of `top` and `site`. We did not include the keyword `CrUX` as it began publishing rankings in 2022 [66]. Whenever available, we used the web interface for the full-text keyword-based search, i.e., ACM and IEEE. USENIX Security, NDSS, and PETS do not have a web interface with full-text search support and thus we downloaded all papers and performed the keyword search on our server. In total, the papers matching our keywords are 1,057 papers.

Systematization criteria — We processed the 1,057 papers as follows. First, we located and annotated the text containing a description or instantiation of the paper's methodology, such as paragraphs, sentences, tables, and appendices. Then, we identified and enumerated the relevant features presented in their methodologies. These features include the specific algorithms used, such as the navigation strategy and the page similarity algorithm, and experimental parameters, such as navigation depth, navigation limit, page load limit, and the collected resources. In addition, we manually noted the topics of a paper and the crawling objectives in the paper. For the topics, we compiled a list of topics starting from the top-level topics extracted from the HotCRP paper submission page and removed or merged duplicated terms. For the objectives, we noted the resource being collected and the purpose of the data collection via crawlers.

Collected sources — Out of the 1,057 papers matching our keywords, we discarded 654 papers because they did not em-

Algorithms						**Crawling parameters**							
		Page sim.				*Nav. depth*			*Nav. limit*			*Load*	
Navigation	Tot.	*Spec.*	*None*	*Unspec.*	*N/A*	*Spec.*	*Unspec.*	*N/A*	*Spec.*	*Unspec.*	*N/A*	*Spec.*	*Unspec.*
BFS	27	19	4	4	-	26	1	-	24	3	-	9	18
DFS	2	1	-	1	-	2	-	-	2	-	-	1	1
Random	36	20	14	2	-	30	6	-	34	2	-	23	13
Rule-based	19	4	14	1	-	16	3	-	15	4	-	8	11
Search engine	8	2	6	-	-	8	-	-	7	1	-	3	5
Unspec.	38	12	6	20	-	14	24	-	23	15	-	7	31
None	273	-	-	-	273	-	-	273	-	-	273	156	117
Tot.	403	58	44	28	273	96	34	273	105	25	273	207	196

Table 1: The distribution of page similarity algorithms and crawling parameters over navigation algorithms.

ploy automated crawling. The final number of papers considered in this survey is 403.

2.1.2 Results

Algorithms used in practice — Of the 403 surveyed papers, only 32.3% of them (i.e., 130 papers) navigate websites whereas the remaining 273 papers (i.e., 67.7%) visit a single page only, thus not relying on neither a navigation nor a page similarity algorithm.

Table 1 shows the result of our analysis distributed over the navigation algorithms. We say that the algorithm is *specified* if the paper presents or mentions the name of the algorithm being used. We say *unspecified* if the paper uses an algorithm, but it does not specify which one. We also consider two other categories for algorithms. When an algorithm is not used, we say that the algorithm is *none*. For example, consider the case of a methodology that does not navigate web pages. Finally, when a paper does not use any navigation strategy, we say that the similarity algorithm is *not applicable* (*N/A*).

Specified vs unspecified algorithms — More than half of the papers navigating websites (i.e., 84 papers which is 64.6%) specify precisely both the navigation strategy and the page similarity, including when page similarity is not used. On the other hand, 35.4% of the papers navigating websites do not specify (i) the navigation strategy (i.e., 18 papers), (ii) the page similarity (i.e., eight papers), or (iii) both (i.e., 20 papers).

Navigation strategies — 70.8% of the papers navigating websites (i.e., 92 papers, amounting to 22.8% of surveyed papers) specify the exact navigation strategy. The most common strategy is randomized (36 papers, e.g., [31, 42, 58]). Other 27 papers (e.g., [21, 72]) use the breadth-first search (BFS), whereas 19 papers (e.g., [11, 68]) use custom strategies. Next, eight papers employed search engines to retrieve URLs of popular domains to visit (e.g., [6, 44]). Finally, two papers (i.e., [19, 35]) use the depth-first search (DFS). The remaining papers either use a navigation strategy but do not specify one (i.e., 38 papers) or do not go beyond the first page (i.e., 273 papers).

	Topics							**Navigation**						**Page Sim.**				
	Web Security	Measurements	Privacy	Fingerprinting	System Security	Malware	Networks	*None*	*Unspecified*	Random	BFS	Rule-based	*Others*	URL	None	*Unspecified*	*Others*	**Total**
	×	×						28	3	6	11	2	3	13	10	2	0	53
	×							19	10	3	2	3	0	5	4	8	1	37
	×		×	×				26	3	3	0	0	0	4	0	2	0	32
		×	×					19	1	4	1	5	1	6	6	0	0	31
		×						21	2	4	1	0	1	6	0	1	1	29
	×	×	×					11	1	4	1	1	1	3	4	1	0	19
	×		×					10	4	4	1	0	0	4	2	2	1	19
			×					12	1	0	0	4	0	1	3	1	0	17
					×			8	2	0	1	0	1	1	1	2	0	12
	×					×		7	2	0	0	0	1	0	0	3	0	10
		×					×	10	0	0	0	0	0	0	0	0	0	10
$\star$	21	47	37	28	17	11	4	102	9	8	9	4	2	11	14	6	1	134
$\substack{\star\\\star}$	191	189	155	60	29	21	14	273	38	36	27	19	10	54	44	28	4	403

Table 2: Algorithms over the top 10 most popular topics. Legend: $\star$ = papers in less popular topics; $\substack{\star\\\star}$ = col. totals

Page similarity — In total, 78.5% of the papers navigating websites (i.e., 102 papers) specify the way they handle page deduplication. More than half of them specify the deduplication algorithm, with URL matching being the most used with 54 papers (e.g., [21, 35]). In contrast, only four papers (i.e., [39, 73, 81, 84]) use DOM-based algorithms. The remaining papers navigating websites either do not use page deduplication collecting all pages (i.e., 44 papers) or use one but do not specify which one (i.e., 28 papers).

Topics — When looking at the topics, the most popular three are web security, privacy, and measurement. In total, 339 are at least on one of these three topics, with web security measurements being more represented than privacy measurements (i.e., 84 papers vs 60).

No Crawling — Among the 273 papers that did not navigate websites, we looked for the rationales behind such a decision. We first searched for them in the problem statement, research questions, methodology, and limitations text. Alternatively, we looked for similar papers in the same domain or with a similar problem setting to infer the necessity of not navigating pages. When we could not find similar papers, we determined whether crawling was necessary based on the paper's objectives. Table 3 shows the results of our analysis, where we count the number of papers that do not navigate sites over the objectives and the need to crawl. Overall, 105 papers did not need to navigate webpages as collecting more data by visiting more webpages from the same website is unnecessary as it produces duplicated data points. The remaining 168 papers

Objective	Need navigation *No*	*Yes*	*Tot.*
Predefined URL List	48	-	48
Website Probing	20	-	20
Censorship	13	-	13
Network Routing	13	-	13
Traffic Simulation	4	-	4
Side-Channel	3	-	3
Website Categorization	3	-	3
Website Fingerprinting	-	44	44
Browser Performance	-	25	25
Tracking	-	14	14
Page Resources	-	14	14
In-Browser Network Activity	-	10	10
CSP & Other Headers	-	8	8
Back-end Stack	-	8	8
Breakage	-	7	7
Advertisements	-	6	6
Cookies	-	6	6
Page Content	-	4	4
JavaScript Analysis	1	21	22
Total	105	168	273

Table 3: The distribution of no navigation papers.

did not navigate not because it was unnecessary but as a trade-off for the limited resources or as an arbitrary choice. When aggregating papers by objectives, we observe two distinct clusters and one outlier paper, which we will discuss next.

The first cluster is of papers that do not need to crawl. Among these, to mention a few, we have a line of works (48 papers) that need to analyze specific pages found in URL feeds, e.g., phishing [40], or papers that probe specific endpoints to detect vulnerable or malicious websites (20 papers), e.g., typosquatting domains [75]. We also have works on censorship (e.g., [63, 79]) that measure or circumvent blocking (13 papers) and network routing (13 papers). In all these cases, navigating web pages with a crawler is unnecessary.

The second cluster is dominated by the papers studying website fingerprinting (44 papers), evaluating browser enhancements (25 papers), measuring tracking prevalence (14 papers), and analyzing embedded resources (14 papers). These works could benefit the most from using crawlers. The authors often explicitly discuss or acknowledge the implication of not crawling. For example, the authors limit the scope of the study to a lower-bound analysis (e.g., [12, 50]) or crawl only a limited selection of websites (e.g., [24]).

Finally, 22 papers collected JavaScript files to find malicious or vulnerable scripts. All of them will benefit from collecting more samples except for one. In Zeng et al. [83], the authors look for homepages that include a specific JavaScript library, and from those, they select only ten pages to conduct a user study. While navigating pages increases sample variety, that will not translate into a better dataset for user studies as participants are subject to fatigue, thus justifying a smaller sample set.

Crawling parameters — Of the 403 papers, 388 papers specify at least one experimental parameter, such as navigation depth, navigation limit, or page load limit. Conversely, 204 papers define at most two parameters. Finally, only 199 papers specify all the parameters. Still, the majority of the papers describing the crawling techniques specify a parameter.

Collected resources — One of the most collected types of resources is network messages and fields (229 papers) to extract domain names and IP addresses (e.g., [71]) or HTTP headers like cookies (e.g., [38, 55]). The second most common type of collected resource is information about the browser execution environment (148 papers), which is then used to study the relevance and prevalence of vulnerabilities or assess new defense mechanisms. For example, Lekies et al. [46] collected dynamic JavaScript to evaluate the cross-site scripting inclusion vulnerability risks. Another example is Soni et al. [70], where a crawler collected JavaScript from popular websites to evaluate their proposed JavaScript signing method. Finally, HTML code (139 papers) and screenshots (30 papers) are common too, especially in works analyzing malicious pages such as phishing (e.g., [33, 41]). The rest (19 papers) collect miscellaneous metrics, such as memory snapshots or power consumption (e.g., [47, 69]).

2.2 Crawling Algorithms

2.2.1 Analysis Methodology

Survey criteria — For the second part of our systematization, we focused on papers proposing new crawlers by adding to our previous seed of papers the ones published over the past three years in the top three software engineering conferences, i.e., IEEE/ACM ICSE, the ACM ESEC/FSE, and the IEEE/ACM ASE. We downloaded all titles, abstracts, and papers matching one of these keywords: crawling, testing, web automation, web application scanning, large-scale web measurements, in-the-wild and large-scale analyses. Then, we read and searched for the pseudo-algorithm, diagrams, techniques, tool names, URLs to the source code, or any other references detailing how the technique works. If the paper included the URL to the code, we downloaded it and manually reviewed it, searching for the module implementing the crawling logic.

We extended our survey to *referred papers, tools, and prototypes* listed in the evaluation and related work sections. For example, the evaluation section of `Black Widow` [25] contains back-to-back experiments with two web application scanners from prior academic works, i.e., Enemy of the State (`EotS`) [20] and `jÄk` [61], and four non-academic tools, i.e., `Arachni` [43], `Skipfish` [82], `GNU Wget` [57], `w3af` [65] and `ZAP` [76]. Whenever we identified a new paper, we reviewed it using the same criteria as before. Starting from 24 seed papers across security and software engineering venues, we identified additional 35 papers. However, the resulting 59 pa-

	Page similarity						Navigation strategy					
Name	*URL(s)*	*DOM tree*	*DOM str.*	*Screenshot*	*Events*	*HTTP*	*DFS*	*JAW*	*BFS*	*Random*	*RL-based*	*Code-cov.*
`Crawljax` [14, 51–53, 80]		×	×	×			×		×			
`Dagger` [77]		×	×									
`jÄk` [61]		×			×				×			
`WebExplor` [85]	×		×								×	
`Artemis` [7]												×
`AutoBlackTest` [49]		×									×	
`Black Widow` [25]	×									×		
`EotS` [20]		×					×					
`FeedEx` [27]		×										×
`JAW` [37]	×							×				
`KAFE` [13]		×							×			
`LigRE` [22, 23]		×					×					
`ProCrawl` [67]		×					×					
`SecuBat` [36]	×								×			
Fetterly et al. [28, 30]	×		×							×		
Lucca et al. [17, 18]		×	×									
Broder et al. [10, 30]			×									
Crescenzi et al. [15]		×							×			
Manku et al. [48]			×									
`Arachni` [43]	×	×				×			×			
`Skipfish` [82]			×			×				×		
`Wapiti` [74]	×					×			×			
`GNU Wget` [57]	×								×			
`w3af` [65]	×								×			
`ZAP` [76]	×					×			×			
Total	10	12	8	1	1	4	4	1	10	3	2	2

Table 4: Overview of the identified tools and algorithms.

pers included eight papers that were previously analyzed in Section 2.1 and another 24 papers that didn't include names and details of the algorithms deployed. As a result, we studied the remaining 27 papers.

Systematization criteria — We analyzed the tools and techniques and decomposed them into their building block algorithms. For that, we reviewed the pseudo-algorithms, text descriptions, and source code when available, looking for the algorithms used for page similarity and navigating web pages.

Collected sources — Our survey identified 27 papers from security and software engineering venues that contain sufficient details about the used algorithms. The analysis of these papers identified, in total, 25 tools and techniques.

2.2.2 Results

The results of our systematization are in Tables 4 and 11. Table 4 lists and maps the names of crawling techniques and the building block algorithms. Table 11 lists and briefly describes the individual building block algorithms.

Page similarity — Our survey identified 27 distinct page comparison techniques, which we organize in six groups.

Many tools, i.e., 14, use exactly one algorithm. Ten tools support multiple algorithms, two of which allow to select one at a time (i.e., `Crawljax` [51], and Lucca et al. [17]) whereas the others combine multiple algorithms in a single function. For example, `jÄk` [61] sums the metrics of the DOM trees and of the registered JavaScript event handlers. The remaining paper (i.e., `Artemis` [7]) does not specify the page comparison algorithm used when visiting websites.

When looking at the objects used for page similarity, prior works and tools covered multiple types and levels of granularity. For example, URL-based approaches range from comparing the full URL to comparing URL components (i.e., URL path, fragment, and query string). We can make a similar observation on the DOM-based method, where current approaches can use entire DOM trees, e.g., for the tree-edit distance (RTED algorithm [60]) or a node-by-node comparison. Alternatively, existing approaches determine similarity by looking at specific HTML tags such as input tags, button tags, HTML forms, hyperlinks, and text. The HTTP response messages' various fields have also been considered for comparison. Among these, we have the HTTP response code, the cookie headers, all HTTP headers, and the HTTP request method. Finally, prior work has proposed using algorithms from the computer vision domain to create image distance functions as a metric for page similarity.

Table 11 presents all the identified algorithms grouped by the object. We note that the DOM tree-based category is the most popular. However, there is a wide range of approaches. The most popular DOM tree-based approach is tree edit distance [60]. The most popular algorithm is the URL exact match algorithm, used by most of the surveyed papers, both by crawlers proposed by prior work (i.e., [25, 85]), tools (i.e., [43, 57, 65, 76]) and in large-scale, in-the-wild studies (i.e., [37]).

Navigation Strategies — For the navigation strategy, all tools except for four (`Dagger`, `Manku`, `Lucca`, and `Broder`) implement a navigation strategy which we can group in six families, i.e., BFS, DFS, depth-limited BFS (JAW), randomized navigation, reinforcement learning-based navigation, and code-coverage driven navigation. Of these, 20 tools implement one strategy only, and one tool (i.e., `Crawljax`) implements multiple ones, which can be selected with a configuration parameter.

The BFS and randomized algorithms are the two most used navigation strategies among all the surveyed papers and tools. Most randomized strategies are based on a random selection of links to visit, whereas the other two combine systematic exploration with randomized elements. The first is a variant of the BFS strategy with shuffled URLs within a node. The second one selects a random URL after path exhaustion.

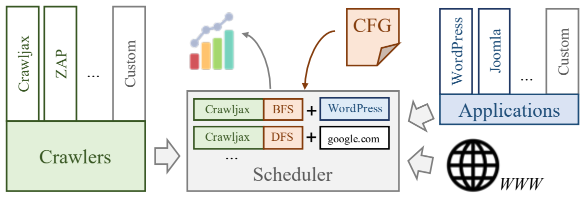


Figure 1: Architecture of `Arachnarium`.

3 `Arachnarium`: An Evaluation Framework

In this section, we present our experimental setup, implemented by our tool `Arachnarium`.

3.1 Architecture Overview

`Arachnarium` is a fully-extensible and scalable framework that enables the comparative evaluation of hundreds of parallel crawler configurations against real websites and web applications, at scale. Figure 1 shows an overview of the architecture. `Arachnarium` has four main components, i.e., the crawler module, the web application module (optional), the scheduler module, and the analysis module. The crawler and web applications modules rely on Docker [32], allowing us to scale experiments by running parallel experiments of the same web application and crawlers concurrently. The web application module can be disabled when testing live websites.

The primary input of `Arachnarium` is a configuration file specifying the tests to execute. An `Arachnarium` test defines the crawler, the web application to test (local or live), and the test parameters. The parameters can be command-line arguments for the tool or configuration options for the web application. The configuration file also specifies global experiment parameters, such as the time budget assigned for individual experiments or the number of concurrent executions. When running a test, the scheduler first deploys the web application container if needed, performs initial health checks to determine if the application is up and running (i.e., HTTP GET probes), and then executes the crawler. When the test reaches the maximum execution time, `Arachnarium` halts the test. The data generated by the crawlers (navigation maps and log files) and by the web application (coverage data and log files) remains in the storage for post-processing.

Users can extend `Arachnarium` with new crawlers or applications by providing a `docker-compose` file—nowadays a commonly available artifact, and enabling the provided code instrumentation module. This paper's version of `Arachnarium` fully supports PHP-based web applications and can instrument the PHP interpreter to measure code coverage through the XDebug interface [64]. Finally, the analysis module collects each test's activity log and returns statistics about crawler performance.

3.2 Testbeds

The evaluation in this paper covers two types of web applications: standalone web applications and live websites.

DS1: Standalone Web Applications — DS1 consists of web applications from prior works (i.e., [20, 25, 80]), which contain both modern applications (e.g., WordPress without plugins) and old ones but repeatedly-used in prior work (e.g., SCARF). We initialized these webapps with database records and configuration shared by the authors of [25] and [80] and deployed them locally. Each web application of DS1 is fully supported and integrated into `Arachnarium`.

DS2: Popular Websites — The second dataset is a list of popular websites. Reproducing results on live websites is generally challenging because of the transient nature of webpages' content and structure, which can change over time. In the recent work, Hantke et al. [29] showed that web archives could help improve reproducibility. However, they may not be an adequate choice for our evaluation. Web archives are created and maintained by crawling live websites; hence the used crawlers would bias our evaluation, and most importantly, our results would have been limited by the coverage of these crawlers. Accordingly, in this paper, we opt for live websites from *Chrome User Experience Report* (CrUX) [1], which is one of the most accurate lists reflecting sites popularities [66]. `Arachnarium` fully supports DS2.

3.3 Metrics

We intend to evaluate crawling algorithms' performance which can be helpful in common and practical deployment scenarios. We identify these scenarios and metrics from our systematization in Section 2, which are all implemented by `Arachnarium`.

M1: Code coverage — One of the application scenarios of crawlers is automated web testing, where crawlers are used in black-box application scanners to explore the attack surface to collect endpoints for the detection of vulnerabilities via ad-hoc tests. Crawlers also support other vulnerability detection techniques, such as server-side code analysis techniques [5], where a crawler provides execution traces to a concolic execution engine. In all these scenarios, crawlers are a means to discover new pages and reach deeper states, which ultimately can be measured via code coverage. `Arachnarium` obtains code coverage of a web application via the XDebug interface [64].

M2: JavaScript source coverage — Our survey showed that artifacts from the JavaScript engine, e.g., JavaScript code, are the second most collected data items. For example, Lekies et al. [46] collect dynamic JavaScript to evaluate the cross-site scripting inclusion vulnerability risks. Therefore, the second metric we consider in our experiments is the amount of unique JavaScript code retrieved by a crawler. In this paper,

Type	Description	Metric
DS1	WordPress v5.1.0 (CMS), OwnCloud v10.10.0 (Cloud storage), PrestaShop v1.7.5-1 (eCommerce), Joomla v3.9.6 (CMS), Drupal v8.6.15 (CMS), Vanilla v2.0.17.10 (Forum), phpBB v2.0.23 (Forum), SCARF v2007 (Conf mgmt), HotCRP v2.102 (Conf mgmt), WackoPicko v2021 (Benchmark for scanners), AddressBook v8.2.5 (Benchmark used by [80])	M1
DS2	CrUX, random 2000 websites from Top10K	M2-3

Table 5: Datasets for our experiments.

`Arachnarium` extracts all script tags from the visited pages and calculates the SHA-512 hashes on the inline and external JS code to determine uniqueness.

M3: Link coverage — The most common collected artifacts are network messages like HTTP requests the crawler generates when navigating a page, like clicking on links. Link coverage is also an established metric when benchmarking web application scanners in black-box settings (see, i.e., [20, 25, 61]) as the server-side component is unaccessible and thus unable to determine coverage via executed code. Accordingly, `Arachnarium` uses link coverage as the third metric by extracting all anchor tags from the visited pages and using exact string matching of the `href` attribute.

3.4 Evaluated Crawlers

Table 6 shows the list of techniques that we evaluated and are available in `Arachnarium`. We identified 39 candidate techniques, of which 35 are candidate building block algorithms. Section 3.4.1 presents the reimplementation rules that we followed when creating `Arachnarium`[3]. The remaining four techniques are tools implementing both page similarity and navigation strategy. These algorithms are interconnected from one to another, for which we could not decouple them tested as a whole. Section 3.4.2 presents these tools.

3.4.1 Candidate Building Block Algorithms

Our survey identified 35 basic algorithms that we considered for the integration into `Arachnarium` extensions of `Crawljax`. The integration was difficult because of incomplete information, missing code, or reference implementations, resulting in 23 implementations.

Code Available — If we were able to run the source code, then we integrated the code in `Arachnarium` for the evaluation. In two cases, i.e., `jÄk` and `EotS`, we were unable to run the code successfully. For `jÄk`, we could not resolve the no-longer supported version of the QT library. In this case, we used the paper and the source code as a reference implementation to reimplement the basic algorithms. For `EotS`, we obtained a Docker image from the authors; however, the code kept crashing. We tried to reimplement `EotS`; however, we could not find a way to isolate the algorithms without affecting the implementation.

[3]We reimplemented the algorithms of this section as extensions of `Crawljax`.

	Features		Code		
Techniques	*Page sim.*	*Nav.*	*Impl. exist*	*Ref. Impl.*	**Eval**
Building block algorithms					
URL Equality	×	-	×	×	×
RTED	×	-	×	×	×
SimHash	×	-	×	×	×
TLSH	×	-	×	×	×
Color histogram	×	-	×	×	×
Perceptual hash	×	-	×	×	×
Block-mean	×	-	×	×	×
PDiff	×	-	×	×	×
SSIM	×	-	×	×	×
SIFT	×	-	×	×	×
`jÄk`	×	-	×	×	×
`ProCrawl`	×	-	-	×	×
`FeedEx`	-	×	-	×	×
DFS	-	×	×	×	×
`JAW`	-	×	×	×	×
BFS	-	×	×	×	×
Rnd BFS	-	×	×	×	×
Rnd State	-	×	×	×	×
Variants					
URL Path Eq.	×	-	-	-	×
URL Path Eq. & QS	×	-	-	-	×
RTED Tr. 1	×	-	-	-	×
¬RTED Tr. 2	×	-	-	-	×
¬RTED Tr. 3	×	-	-	-	×
Tools					
`Arachni`	×	×	×	×	×
`ZAP`	×	×	×	×	×
`Wapiti`	×	×	×	×	×
`Skipfish`	×	×	×	×	×
Unable to reimplement					
Tree Equality	×	-	-	×	-
UI Controls	×	-	-	-	-
Root-Link Paths	×	-	-	-	-
Common Shingles	×	-	-	-	-
TAF	×	-	-	-	-
LevenSeq	×	-	-	-	-
`Dagger`	×	-	-	-	-
`LigRE`	×	-	-	-	-
`Fetterly`	×	-	-	-	-
`Artemis`	-	×	-	-	-
`WebExplor`	×	×	-	-	-
`EotS`	×	×	×	×	-

Table 6: Evaluated algorithms.

Code Not Available — When the source code was unavailable, we used the reference implementation to reimplement the crawlers. We fully reimplemented the algorithms in two cases, i.e., `ProCrawl` and `FeedEx`.

Missing Details — For all other cases, we did not have sufficient details to reimplement the algorithms. Appendix A.2 list the missing details. Also, we discarded the *Tree Equality* algorithm because it compares two DOM trees node by node and returns false as soon as one single node differs. We consider such an approach too brittle and did not consider it for the evaluation.

Proposed Variants — We also considered five techniques derived from popular and promising algorithms by recent works. We designed variants for URL equality, as the most used technique across all papers of our surveys, and RTED, the best performing in Yandrapally et al. [80].

Variants of URL Equality — URL Equality is one of the most popular page similarity techniques identified according to our survey. A drawback of this approach is its sensitivity to small changes, e.g., path or query string alterations result in a different page. That may not be true in practice, as different URL paths and parameters may not indicate a state change. Accordingly, we consider two variants. The first one, called **URL Path Equality**, is a string comparison between URLs considering only the domain and the path. The second one called **URL Path Equality & QS**, includes in the comparison the query string of the URL; however, it ignores the value of the query string parameters.

Variants of RTED — RTED calculates the tree edit distance between two trees, which prior work showed it outperforming other techniques [80]. RTED implementations currently run in polynomial time, which can be impractical in many scenarios. Accordingly, we consider three variants to simplify the complexity of comparing two trees. We observe that a DOM tree can include multiple identical subtrees (widgets) in lists, e.g., menu items and lists of products. Therefore, we could reduce the size of a DOM tree by collapsing lists of identical widgets. We call this variant **RTED with DOM Transformation #1**. We further observe that similar pages can share widgets, and we could use those widgets for the comparison. We say that two pages are similar if they share the same widgets. We call this variant ¬ **RTED with DOM Transformation #2**. Finally, we also relax the widget definition to any subtree on a page, even when the subtree appears only once. This way, a page can be seen as a set of widgets. We say that two pages are similar if both have the same set of widgets. We call this variant ¬ **RTED with DOM Transformation #3**.

3.4.2 Compound Tools

Finally, we integrated in `Arachnarium` four tools, i.e., `Arachni`, `Skipfish`, `Wapiti`, and `ZAP`. As these tools are used for security testing, they run tests while crawling, which are likely to exercise new branches or exceptions, increasing coverage. Accordingly, we turned off all vulnerability detection features. We could not evaluate `WebExplor` because we were not able to resolve the exact way the Gestal algorithm is used when processing a webpage. We asked the authors for the source code, but they never answered our emails. Finally, while we managed to run `EotS`, it was unstable with a high rate of crashes, making it particularly challenging to collect data reliably; thus, we discarded it.

4 Experiments and Results

In this section, we evaluate the crawling algorithms' performances and methodological choices when used to collect data. After presenting the design of our experiments (Section 4.1), we present our results. First, we determine, if any, the best-performing algorithms looking at total coverage at the end of each run and at the coverage speed (Section 4.2) . Then, we compare coverage results to determine how much surface a crawler can discover than others, using two baselines (Section 4.3). Finally, we assess the methodological decisions our survey identified and evaluate their impact (Section 4.4).

Reducing Experiment Complexity — `Arachnarium` implements 17 page similarity algorithms and six navigation strategies, which amounts to a total of 102 crawler configurations to test. As a preliminary step, we reduce the number of page similarity algorithms to test, discarding those that perform poorly. We run all the page similarity implementations using only BFS three times against DS1 (self-hosted web applications) instead of DS2 (live sites) to reduce traffic load on them. At the end of each run, we calculated the maximum across three runs and the total unique LoCs executed. We selected the top ten page similarity algorithms with the highest average weighted rank based on code coverage and used them for the main experiments (see below). The complete results of this experiment are in the data repository. The top ten algorithms selected for the rest of this paper are ¬RTED Tr. 2, TLSH, ¬RTED Tr. 3, SimHash, URL Path Eq. & QS, URL Path Eq., URL Eq., Phash, `ProCrawl`, and Block-mean.

4.1 Experiment Design

This section presents the design of experiments and the measurements of this paper.

4.1.1 Experiments

The first experiment (*Exp1*) measures code coverage in DS1. For that, we combine the top 10 page similarity algorithms from our preliminary experiment with all six navigation strategies. The second experiment (*Exp2*) measures the coverage of JavaScript source code and links against dataset DS2. Running all crawler configurations multiple times against the same sites can generate unwanted traffic. Accordingly, we sample increasingly large random samples from disjoint CrUX Top 10K buckets. More specifically, we randomly select 200 domains from the Top 1K, 800 from the Top 5000 (excluding the Top 1K), and 1K from the Top 10K (excluding the Top 5K), resulting in a dataset of 2K random domains. We visit these 2K domains using the same top 10 performing page similarity algorithms and top five navigation strategies of Exp1 by code coverage.

4.1.2 Coverage Data

In our experiments, we run a crawler configuration three times against the same target site, i.e., a self-hosted web application (DS1) or a live website (DS2). For each action the crawler performs, we collect the timestamped coverage data, i.e., executed unique lines of code or discovered unique script and link tags. Then, we sum the unique coverage data points per run and pick the greatest. Collecting coverage data from the live sites (Exp2) was not as reliable as from self-hosted web applications (Exp1) because of external factors such as connection timeouts or temporary site unavailability, resulting in sites with partial coverage data. Instead of scheduling more runs to compensate for the missing data points, and thus increasing website loads, we use the coverage data of the first successful run among the three attempts.

4.1.3 Experiment Measurements

Absolute coverage — We calculate the total coverage of a crawler on a given dataset as the sum of the coverage for each target site in the dataset. We also calculate the coverage of individual algorithms, i.e., page similarity or navigation, by averaging the total coverage data runs over the other algorithm, e.g., page similarity over navigation and vice versa.

Relative coverage — This definition of total coverage is *absolute* and cannot show whether two crawlers discovered the exact same states or different ones. For example, if two crawlers covered 100 unique lines of code each, the absolute coverage metric will rank them as equivalent. However, as each crawler implements a different algorithm, it can happen that while both total 100 lines of code, these lines may not be the same ones. Accordingly, we introduce the notion of *relative* total coverage to quantify the extent to which two crawlers covered the same and different areas of a target site. We define the relative coverage as the increment (or decrease) of the total coverage of a crawler over a *baseline*.

Baseline 1: Popularity — As we aim to assess methodologic decisions, e.g., status quo, we set as a first baseline the coverage for the most *popular* algorithms from our surveys. Our survey on crawling algorithms shows that DOM tree techniques are the most used similarity algorithm used by crawlers. However, the DOM tree is a family including several specific implementations, e.g., RTED or tree equality. None of these implementations is more popular than URL Eq. – the second most popular one. The survey of web measurement papers also reveals that URL Eq. is the most commonly used technique to deduplicate pages. Accordingly, we select URL Eq. for the baseline. The web measurement survey shows that randomized strategies are mostly used for the navigation strategy. However, many papers do not clarify the exact type of randomized algorithms,e.g., Rnd BFS and Rnd State. The crawler algorithms survey, however, shows that BFS is the most popular one. Accordingly, we select BFS for the baseline.

Baseline 2: Global coverage — Finally, we compare the coverage of each crawler against the union of the coverage of all crawlers except for the one evaluated. For the comparisons, we use the difference set for the increments and decreases and the intersection set for the known surface. Increment or decrease from this *global* baseline can precisely determine unique features of crawlers in discovering the target sites' surface that others cannot discover.

Hardware and software configuration — We run `Arachnarium` with a maximum of 12 parallel workers, a wait time after page reloads of 500 ms, and an interval between actions of 500 ms. We run our experiments on a GNU/Linux Debian 11 system running on two AMD EPYC 7h12 64-Core Processors with 2TB of RAM. We configured `Arachnarium` to run inside on RAMFS of 300GB to speed up disk I/O operations and avoid disk write bottlenecks.

4.2 Best-performing Algorithms

We now compare crawler configurations and algorithms using the total absolute coverage to determine the best-performing algorithms from two angles: the total coverage at the end of each run and the coverage growth over time.

4.2.1 Total Coverage

First, we determine the total coverage of individual configurations and individual algorithms at the end of each run.

Configurations — We first start with configurations, i.e., pairs of a navigation strategy and a page similarity. Table 9 shows an excerpt of the results with the top two deterministic and non-deterministic best-performing configurations in each metric. The complete table is in Table 12 (Appendix). When looking at the total coverage (code, link, and JS), the randomized algorithms, especially the random BFS, are consistently among the top best-performing configurations. For example, the randomized BFS covers ranks from five to eight of the best-performing configurations. It ranks first when collecting URLs and JavaScript code when using URL comparison, i.e., ignoring URL QS values. When switching the page similarity to ¬RTED Tr. 3, the randomized BFS ranks first also in code coverage. Among the deterministic navigation strategy, the best-performing configuration is BFS with ¬RTED Tr. 2, ranking only fifth. For link coverage, `JAW` is second when used with URL Path Eq. & QS. BFS with ¬RTED Tr. 3 is ranked first when collecting JavaScript code.

In general, the links and JavaScript metrics are positively correlated with LoCs, meaning that high coverage in the LoCs very likely results in a high coverage in the other two metrics, too. The pairwise Pearson coefficients and p-values of Table 12 (Appendix) are $0.5454/8.05E-03$ for LoC-Links and $0.6587/1.59E-65$ for LoC-JS.

Page Similarity	LoC Sum	R.	+/-%	Links Sum	R.	+/-%	JS Sum	R.	+/-%
Block-Mean	220.482	8	-3.1%	186.049	9	-37.6%	23.847	9	-25.2%
Phash	220.451	9	-3.2%	121.267	10	-59.4%	14.383	10	-54.9%
`ProCrawl`	214.134	10	-3.2%	189.492	8	-36.5%	24.262	8	-23.9%
SimHash	232.092	4	+2.0%	296.302	3	-0.7%	26.415	7	-17.2%
TLSH	240.015	3	+5.4%	265.314	7	-11.1%	31.351	5	-1.7%
URL Eq.(baseline)	227.634	6	-	298.345	2	-	31.902	4	-
URL Path Eq.	221.873	7	-2.5%	276.634	6	-7.3%	28.737	6	-9.9%
URL Path Eq. & QS	230.282	5	+1.2%	**338.922**	1	+13.6%	33.350	2	+4.5%
¬RTED Tr. 3	240.089	2	+5.5%	288.962	5	-3.1%	**34.524**	1	+8.2%
¬RTED Tr. 2	**242.509**	1	+6.5%	292.565	4	-1.9%	33.158	3	+3.9%

Table 7: Average sum coverage, ranking, and % increase over the baseline of all navigation strategies.

Navigation	LoC Sum	R.	+/-%	Links Sum	R.	+/-%	JS Sum	R.	+/-%
BFS (baseline)	**228.748**	3	-	**250.373**	2	-	27.080	4	-
DFS	226.236	5	-1.1%	248.389	4	-0.8%	**27.243**	2	+0.6%
`FeedEx`	214.781	6	-6.1%	-	-	-	-	-	-
`JAW`	227.983	4	-0.3%	248.247	5	-0.8%	27.139	3	+0.2%
Rnd BFS	**247.207**	1	+8.1%	**280.467**	1	+12.0%	**32.853**	1	+21.3%
Rnd State	228.781	2	0.0%	249.450	3	-0.4%	26.650	5	-1.6%

Table 8: Average coverage and ranking of all page similarity algorithms.

Configuration	LoC		Links		JS	
BFS + ¬RTED Tr. 2	**245.816**	5	298.696	14	31.594	13
BFS + ¬RTED Tr. 3	**244.946**	9	289.184	20	**33.960**	7
DFS + ¬RTED Tr. 2	241.565	16	291.458	17	**32.372**	10
`JAW` + URL Path Eq. & QS	226.454	30	**347.421**	2	31.297	15
DFS + URL Path Eq. & QS	229.749	27	**328.731**	4	30.732	19
Rnd BFS + ¬RTED Tr. 3	**259.600**	1	313.742	9	**41.691**	2
Rnd BFS + URL Path Eq. & QS	244.838	10	**369.810**	1	**43.319**	1
Rnd BFS + URL Eq.	252.530	3	**339.134**	3	39.655	3
Rnd BFS + ¬RTED Tr. 2	**259.028**	2	324.179	6	37.850	4

Table 9: Top two deterministic and non-deterministic best-performing configurations.

Individual Algorithms — When looking at the average coverage per algorithm, three page similarity algorithms emerge as the top performing ones: ¬RTED Tr. 2 for LoCs, URL Path Eq. & QS for links, and ¬RTED Tr. 3 for script tags, whereas the worst performing ones are `ProCrawl` for code coverage and Phash for both links and JS. Similarly, Rnd BFS is by far the best-performing algorithm in all metrics. Table 7 and Table 8 show the average code coverage for each page similarity and navigation technique, respectively.

Total Coverage Increase — We now look at the coverage increase over the most popular configuration, i.e., BFS and URL Eq., should one select a different navigation strategy or page similarity algorithm. The results are in Tables 7 and 8.

In general, five similarity algorithms obtain a better code coverage than the full URL equality, with a percentage increase ranging from a +1.2% for URL Path Eq. & QS to +6.5% for ¬RTED Tr. 2 (Table 7). When collecting links and JavaScript source code, fewer algorithms perform better, however, with a more evident increase in coverage. For example, switching to URL Path Eq. & QS can increase, on average, coverage by 13.6%. When collecting JavaScript code instead, three similarity algorithms perform better than URL Eq., from +3.9% for ¬RTED Tr. 2 to +8.2% for ¬RTED Tr. 3. No other page similarity algorithm helps improve LoCs coverage URL Eq. We note, however, that algorithms such as Block-Mean, Phash, and `ProCrawl` can instead degrade performances significantly, especially for links and JavaScript. For example, Phash can decrease link coverage by more than a half, e.g., almost -60%.

When looking at the navigation strategy results (Table 8), in all cases, switching to random BFS can increase coverage significantly, with a gain of +8.1%, +12%, and +21% for code, links and JavaScript coverage. Only `FeedEx` recorded a minimum low among the six strategies, with a -6.1% code coverage. Other strategies have an almost-negligible loss of performance, i.e., from about -1.6% to -0.3%.

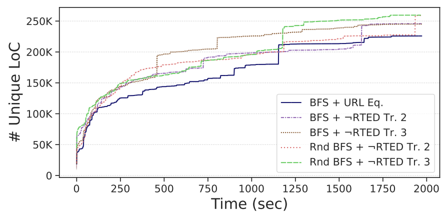
4.2.2 Coverage Growth over Time

Total coverage does not include the temporal dimension and cannot be used to examine the velocity a crawler can explore the surface of a site. In this section, we look at the timestamped coverage data and calculate the cumulative coverage over time. We focus only on the two deterministic and non-deterministic best-performing configurations in each metric using as a reference point the most common configuration from our survey, i.e., BFS with URL Eq. (Figure 2). All configurations tend to perform consistently better than BFS with URL Eq. over time. However, the differences are less appreciable with short crawls, and the breakpoints of these short crawls vary with the metrics. For example, when collecting links, the benefits of the selected configurations start being noticeable after 30 seconds. For JavaScript, the breakpoint is about 50 seconds, whereas for code coverage is about 100 seconds.

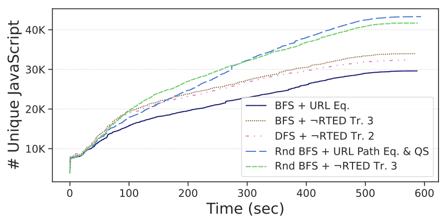
4.3 Coverage Analysis w/ Baselines

We now look at the coverage of each configuration relative to two baselines to precisely determine the fraction of the surface covered by crawlers over the most popular algorithms and against the global coverage.

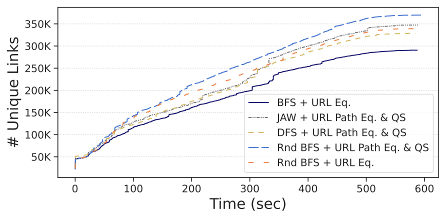
Increment over Popular Algorithms — In terms of unique lines of code, the surface discovered by both the baseline and the configurations ranges from 73.2% by random state with ¬RTED Tr. 2 to the almost identical overlap of 98.2% by `JAW` with URL Eq. When looking at the set differences, the randomized BFS with ¬RTED Tr. 3 almost includes the entire baseline with only 0.9% of missed LoCs and, at the same time, extends it with a considerably large new surface of about 25.4% of the total.



(a) LoC



(b) JavaScript



(c) Links

Figure 2: Cumulative coverage over time of the two deterministic and non-deterministic best-performing configurations in each metric with a baseline on BFS + URL Eq.

The relative coverage over links and JavaScript follows a different distribution, where the overlaps with the baseline lose relevance against the differences, i.e., missed and unique links/scripts. For example, at most, a fifth of the unique links and JavaScript code is shared with the baseline configuration, i.e., 8.7% of the baseline surface. The largest overlap is 21.4% by `JAW` with URL Eq. The coverage differences between configurations and the baseline are far more significant, covering from 76% up to 91% of the discovered links and JavaScript code. Such a stark difference is likely caused by the presence of dynamically generated links and JavaScript, which are generated and discovered in one run only, thus being missed by the others. We explore this aspect later.

Increment over Global Coverage — Similarly to the baseline coverage, the configurations' global coverage varies greatly across metrics. The best-performing configuration (random BFS with ¬RTED Tr. 3) covers 78.9% of the global coverage, whereas the worst-performing one (`FeedEx` with SimHash) reaches 54.9%. In comparison, the coverage is significantly lower when counting links and JavaScript. The best-performing configuration for links and JavaScript (random BFS with ¬RTED Tr. 2) is well below the worst LoC coverage, i.e., 19.4% and 23.8%.

4.3.1 Commonly Discovered Surface

To help interpret coverage results, we look at the coverage following a resource-centric approach. We count how many times a configuration run has hit a unique line of code, link, and script tag. Figure 3 shows the histogram of the number of hits in a log scale. Live websites contain several dynamically generated resources, including links and JS code. Dynamic content deflates total coverage as these links and JS scripts are unique to one crawling session. These resources are visible in Figure 3, where more than 1.9M links and 376K scripts were never visited by more than one crawler. Code coverage shows slightly different properties. On the one hand, we see also 196K unique lines of code being executed exactly once. These LoCs can be, for example, instructions regenerating cached data like UI templates. However, we can also see the dominant effect of boilerplate code, such as initialization procedures like loading configuration files and plugins, that is executed at each HTTP request. At $x = 60$, we see all configurations (ten page similarity algorithms × six navigation strategies) executed 181K unique lines of code.

	LoC		JavaScript		Links	
# crawls	Rnd	Det	Rnd	Det	Rnd	Det
1	100%	100%	100%	100%	100%	100%
2	+4.76%	+2.00%	+65.08%	+60.34%	+62.11%	+58.43%
3	+8.14%	+3.47%	+101.66%	+94.51%	+95.87%	+87.91%

Table 10: Average cumulative increment for the consecutive re-crawls for random and deterministic configurations.

4.4 Crawling Parameters

Navigating vs no navigating — 273 papers of our survey stop at the first page during a visit. Increasing the limit to two pages, the average coverage increases by +29.34% new lines of code, +101.96% new links, and +83.65% for LoCs, links, and JavaScript code, respectively. When pushing the limit to 10 pages, we observe a more significant increase of +100.91%, +253.95%, and +194.36%. Figure 4 shows the cumulative increment of coverage over the number of visited pages up to 50 pages.

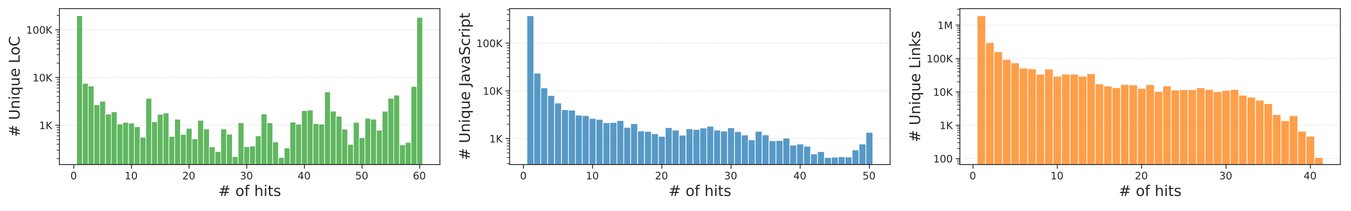
Single- vs re-crawling — Table 10 shows that additional re-crawls are likely to increase coverage across all metrics. Moreover, on average, randomized algorithms gained a more significant increase in comparison to deterministic ones.

5 Discussion

5.1 Lesson learned

5.1.1 Significant Challenges to Reimplementations

Our systematization in Section 2 showed a rather complex landscape in presenting the methodology, mostly characterized by scattered and incomplete descriptions, which ultimately affected our reimplementation.



(a) LoC (b) JavaScript (c) Links

Figure 3: Distribution of the number of times an element, e.g., unique line of code, link, and script, is hit by a configuration run.

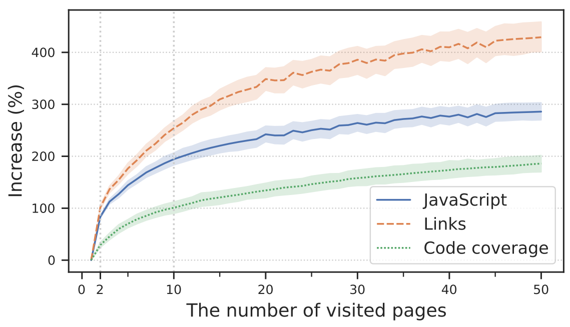


Figure 4: Cumulative increment percentage over the number of visited pages.

#1 - Insufficient descriptions — Insufficient descriptions occur in many papers and manifest themselves differently. The first one is not specifying all parameters, including the specific algorithms used in a study. In total, our review of large-scale studies in Section 2.1 shows that 38, 29, 34, 25, and 196 papers did not specify, respectively, the navigation strategy, page similarity, navigation depth, navigation limits, and page load waiting time (See Table 1). The second one is insufficient or missing details of crawling algorithms. For example, our systematization of existing algorithms in Section 2.2 could not determine the algorithms of 24 papers.

#2 - Scattered and heterogeneous descriptions — The second insight of our survey is that the text describing the crawler configuration or other parameters is scattered across the papers, making their localization hard and an error-prone task. Here, we welcome standardized, well-marked areas of a paper describing the methodology and algorithms in papers.

#3 - Lack of code and algorithmic details — We could not integrate ten of the analyzed algorithms into `Arachnarium` as we could not reimplement them. The main reason was the lack of sufficient details in the text and the source code's absence to attempt a new implementation. We strongly encourage our community to adopt even more stringent publication rules where source code and artifacts accompanying research become the norm, not the exception.

5.1.2 No Winner, Complex Landscape

Our results revealed a rather complex landscape, with randomized algorithms generally performing better.

#4 - Commonly used algorithms do not perform the best — URL Eq. is a commonly used page similarity algorithm that does not rank first in any metric. On average, switching to ¬RTED Tr. 2can improve coverage for LoCs (+6.5%), URL Path Eq. for links (+13.6%), and ¬RTED Tr. 3 for JavaScript (+8.2%). On the other hand, BFS is the commonly used navigation strategy, which also does not rank first on average. Switching to the randomized version, we observe an increase from +8.1% (LoCs) to +21.3% (JS).

#5 - No outperforming configuration — Our results show that no single configuration ranked first on both absolute and global coverage metrics. However, randomized BFS combined with many page similarity algorithms is consistently among the top-performing configurations.

#6 - Randomized navigation is a good choice also with limited resources — 36 large-scale web measurements used randomized visit strategy with strict navigation limits to collect resources from a landing page under limited time constraints, e.g., five links only. Our results indicate that such a data collection strategy allows us to balance coverage well with the given resources.

#7 - Algorithms not suited for the job — Our results also identified algorithms performing poorly across all metrics, especially for live crawls. Examples are Phash, `ProCrawl`, Block-Mean. Results also suggest that SimHash may be unfit when collecting JavaScript source code. Among the crawling strategy, `FeedEx`, which combines a coverage-driven metric, ranked last with a -13.11% from random BFS.

#8 - Crawling with metrics — We observed that crawler configurations perform differently across the metrics, and we recommend selecting the algorithm based on the targeted metrics.

#9 - Crawling with limited resources — Running crawlers takes, in general, time and computing resources. With a tight budget, e.g., crawling sessions below 30, 50, and 100 seconds, we notice that the algorithm choice does not significantly affect coverage. However, we note that coverage can dramatically improve by using a navigation technique with a depth of five or more.

#10 - No crawling — Most papers, i.e., 168, that do not navigate pages work on attack or defense techniques against website fingerprinting and online tracking and collect HTTP headers. For these papers, crawling could have provided more sample variety. However, crawling live websites can be expensive, and authors often need to balance the number of websites of their study and navigation depth to keep the data collection manageable. Most of these works also discuss such a decision and the impact on the results, scope, and conclusions. Here, our empirical measurement of coverage over the number of visited pages estimates the coverage loss for these methodologic choices.

5.1.3 Future Directions

Not having a silver-bullet solution indicates that new efforts in creating better crawling techniques are needed.

#11 - Alternatives to crawling are unexplored — Crawlers are not the only approach that could be used to explore websites. For example, over the past decades, many UI testing approaches have been proposed (e.g., [26, 54]) where they tend to reason on the UI structure and logic to explore deeper behaviors. Prior work has shown improvement over random test generation [26]; however, these techniques have not been used in web measurements yet, and their potential and impact are largely unexplored.

#12 - No better than random crawling — Perhaps, the most disappointing result is that randomized strategies performed, in average, better than all other techniques, including security testing tools. That suggests that, despite the efforts, visiting with random clicks performs better than current heuristics. Not being better than random algorithms calls for additional efforts to devise and build better crawling algorithms. At the same time, our results indicate that randomized algorithms should be included in all evaluations when proposing a new technique to precisely measure when a new idea achieves a significant and relevant result.

#13 - Security testing tools lag behind — None of the commercial-grade security testing tools are at par with the crawler configurations. The best-performing tool is `Arachni`, which ranked 38th in code coverage whereas (-24.5% from the first, i.e., random BFS with ¬RTED Tr. 3), by contrast, `ZAP` ranked last, i.e., 63rd (-33% from the first).

#14 - New web application surface as a shared contribution — We can also observe interesting properties of the current state-of-the-art of web crawling in global coverage. The total number of lines of code discovered at least by two configurations is 87% (non-unique LoC). Interestingly, for the remaining 13%, no two crawlers discovered it, suggesting that this fraction results from the union of disjoint, unique capabilities of each algorithm.

5.2 Ethical Considerations and Threat to Validity

Ethical Considerations — Our experiments intend to provide a better view of crawler behavior in real-world scenarios. However, automated interaction with public-facing websites necessitates careful risk evaluation. We anticipated three possible risks.

The first risk is the accidental collection of PIIs in webpages and URLs. Our crawlers visit webpages that can be reached by any human user when browsing the public, unauthenticated pages of a website with a browser. Our crawlers neither forge URL strings nor log in and stay on the website, thus minimizing the risk of collecting sensitive data. The risk of collecting PIIs is further reduced by the types of websites used in our study and by our sampling strategy. The top 10K CrUX tends to contain websites of global corporations, which are unlikely to store PIIs in webpages or URLs in public pages reachable from a homepage. In addition, we did not collect pages from all top 10K sites but from a random sample of 2K sites, further mitigating the risk of collecting PIIs accidentally.

Second, our experiments collected web pages via hundreds of crawler runs, which can flood websites with thousands of requests per second. To avoid hampering website availability, we implemented in `Arachnarium` the following constraints: (i) `Arachnarium` runs at most 12 concurrent crawlers, (ii) two crawlers cannot visit the same domain in parallel, (iii) each crawler runs in a single-thread and single-browser mode, executing actions sequentially, not concurrently, and (iv) each action is executed every 0.5s. ASN-based checks can further mitigate the risk of flooding smaller websites; however, small sites are unlikely to be in our dataset as the top 10K CrUX tends to contain globally available sites, using robust hosting, network, and CDN providers, e.g., top shared ASNs of our 2K dataset include Cloudflare, Amazon, and Akamai. As smaller websites are more likely to be present in low-ranked domains, researchers wanting to repeat our experiments on those sites should avoid running two crawlers against sites hosted by the same provider.

The third risk is collecting content website owners want to avoid being indexed and shared, e.g., the price of products, and they can specify their policy in `robots.txt`. However, our crawlers collect and extract URLs and script tags from the HTML code and do not index and share web content. Hence, when considering the interaction between harm and benefits of our work, we concluded that not honoring `robots.txt` was an acceptable trade-off in this specific instance, given the precautions we took.

Finally, the artifacts that we will share with the community will not contain the raw data collected (URLs or code, which are subject to removal from our systems) but redacted data, i.e., SHA256 hashes of URLs.

Threat to Validity — The DS1 dataset contains 11 web applications, three orders of magnitude smaller than the 2K websites in DS2. Confirming the full generalizability of our experiments requires expanding the DS1 dataset, which is hard to do at scale. We mitigated this external validity threat by selecting web applications that are relevant and real-size, e.g., WordPress, Joomla, and Drupal, and used in the literature [20, 25, 80]. Similarly, our evaluation used three coverage metrics, and future researchers can already use our results when planning their experiments. While we observed a strong correlation between LoC and link and JS coverage, the correlation may not hold for other metrics, and additional experiments with new metrics are needed. However, to minimize this risk, we designed Arachnarium to be extendable and configurable, providing a platform for researchers to select experimental parameters. Finally, Figure 3 shows a large fraction of JavaScript and links, each discovered by one crawler run. These links and JavaScript code are likely temporary, e.g., dynamically generated links, which can result in overestimating the covered and missed surface.

6 Related Works

Crawler evaluations — To the best of our knowledge, this is the first systematization of knowledge on crawlers and their impact on security measurements. The two closest prior works to ours are Yandrapally et al. [80] and Ahmad et al. [4]. Yandrapally et al. [80] evaluated different page similarity algorithms selected from different domains: information retrieval, web testing, and computer vision. Our work differs in two main aspects. First, we cover both page similarity algorithms and navigation strategies, exploring their combinations. Second, we consider algorithms that are used in practice. Ahmad et al. [4] evaluated a mix of crawlers, e.g., GNU Wget and ZAP, and web automation tools, e.g., Selenium and Puppeteer, focusing on aspects such as the ability to manipulate HTTP parameters or disabling cookies. As opposed to this work, our paper focuses on the algorithmic nature of crawlers, looking at their ability to find and identify pages, i.e., via navigation and page de-duplication, also exploring their combinations.

Prior works [20, 25, 61] evaluate the presented tool against the other tools, e.g., Black Widow is tested against Skipfish, w3af and ZAP, using different testbeds, parameters, and metrics. While these evaluations show us the performance of one tool against the other, results can hardly be transferred across papers. Our paper addresses this issue by proposing a comparative evaluation framework, Arachnarium, that can be used to define and execute reproducible experiments against web applications..

Reproducibility in web measurements — Reproducibility in web measurements is an open problem. For example, Jueckstock et al. [34] and Demir et al. [16] pointed out that reproducing prior measurements is hard in practice, whereas Hantke et al. [29] showed that web archives could be used to reproduce web security results. As opposed to these works, our paper does not reproduce prior works. Instead, it intends to evaluate the adequacy of the algorithms used to collect data in web measurements and provide insights to future researchers to help select the parameters of their studies.

Alternatives to crawling — Crawling is one of the many dynamic analysis techniques that could be used in web measurements, including both white-box and black-box approaches. Unfortunately, white-box approaches such as [5, 8, 9] require the server-side source code for the analysis, which is not available in practice. On the other hand, black-box approaches do not require the source code and, like crawlers, implement a variety of heuristics to explore and test the target web application. Among those, a line of work closely related to crawlers is UI testing, where many automated approaches have been proposed to test UIs for functional errors (e.g., [26, 54]. The goal of our study is to systematize and test techniques that are used in empirical studies. Our survey could not find the use of UI testing approaches; therefore, we did not consider them in our study.

7 Conclusion

This paper presents a systematization of knowledge about web crawling algorithms for web security measurements. After identifying and analyzing 403 papers from top venues in security, privacy, web, and measurements, and 27 papers from the top venues in software engineering, we created a taxonomy with 35 algorithms and mapped their use in empirical analyses. Then, we proposed and used Arachnarium, a framework for comparative analysis of crawlers, in a comprehensive empirical evaluation of proposed crawling techniques and parameters. From our systematization and evaluation, we distilled 14 insights, lessons learned, and recommendations for our community and future researchers.

Acknowledgments

This work received funding from the European Union's Horizon 2020 research and innovation programme under the TESTABLE project (grant agreement 101019206), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation, project number 452850842).

References

[1] Chrome User Experience Report. https://developer.chrome.com/docs/crux/.

[2] Puppeteer. https://github.com/puppeteer/puppeteer.

[3] Selenium Browser Automation. https://www.seleniumhq.org/.

[4] Syed Suleman Ahmad, Muhammad Daniyal Dar, Muhammad Fareed Zaffar, Narseo Vallina-Rodriguez, and Rishab Nithyanand. Apophanies or epiphanies? how crawlers impact our understanding of the web. In *Proceedings of The Web Conference 2020*, pages 271–280, 2020.

[5] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic

web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, August 2018. USENIX Association.

[6] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce M. Maggs. On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 680–695, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 571–580, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[9] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[10] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997. Papers from the Sixth International World Wide Web Conference.

[11] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: Fortifying web protocols via Browser-Side security monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, Baltimore, MD, August 2018. USENIX Association.

[12] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content security problems? evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1365–1375, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. Detecting and localizing keyboard accessibility failures in web applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 855–867, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 171–180, 2012.

[15] Valter Crescenzi, Paolo Merialdo, and Paolo Missier. Clustering web pages based on their structure. *Data Knowl. Eng.*, 54(3):279–299, sep 2005.

[16] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. Reproducibility and replicability of web measurement studies. In *Proceedings of the ACM Web Conference 2022*, pages 533–544, 2022.

[17] G.A. Di Lucca, M. Di Penta, and A.R. Fasolino. An approach to identify duplicated web pages. In *Proceedings 26th Annual International Computer Software and Applications*, pages 481–486, 2002.

[18] Giuseppe Antonio Di Lucca, Massimiliano Di Penta, Anna Rita Fasolino, and Pasquale Granato. Clone analysis in the web era: An approach to identify cloned web pages. In *Seventh Workshop on Empirical Studies of Software Maintenance*, volume 107, 2001.

[19] Vladan Djeric and Ashvin Goel. Securing Script-Based extensibility in web browsers. In *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.

[20] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A State-Aware Black-Box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, August 2012. USENIX Association.

[21] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1953–1970, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, page 37–48, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Fabien Duchène, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 252–261, 2013.

[24] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401, 2016.

[25] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1125–1142, 2021.

[26] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 82–93, 2016.

[27] Amin Milani Fard and Ali Mesbah. Feedback-directed exploration of web applications to derive test models. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 278–287, 2013.

[28] Dennis Fetterly, Mark Manasse, and Marc Najork. On the evolution of clusters of near-duplicate web pages. In *Proceedings of the First Conference on Latin American Web Congress*, LA-WEB '03, page 37, USA, 2003. IEEE Computer Society.

[29] Florian Hantke, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock. You call this archaeology? evaluating web archives for reproducible web security measurements. In *ACM CCS*, 2023.

[30] Monika Henzinger. Finding near-duplicate web pages: A large-scale evaluation of algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, page 284–291, New York, NY, USA, 2006. Association for Computing Machinery.

[31] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1701–1713, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Solomon Hykes. Docker. https://www.docker.com/.

[33] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 743–758. IEEE Computer Society, 2016.

[34] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. Towards realistic and reproducible web crawl measurements. In *Proceedings of the Web Conference 2021*, pages 80–91, 2021.

[35] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. Measuring the privacy vs. compatibility trade-off in preventing third-party stateful tracking. In *Proceedings of the ACM Web Conference 2022*, WWW '22, page 710–720, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: A web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, page 247–256, New York, NY, USA, 2006. Association for Computing Machinery.

[37] Soheil Khodayari and Giancarlo Pellegrino. JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2525–2542. USENIX Association, August 2021.

[38] Soheil Khodayari and Giancarlo Pellegrino. The state of the samesite: Studying the usage, effectiveness, and adequacy of samesite cookies. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1590–1607. IEEE, 2022.

[39] I Luk Kim, Weihang Wang, Yonghwi Kwon, Yunhui Zheng, Yousra Aafer, Weijie Meng, and Xiangyu Zhang. Adbudgetkiller: Online advertising budget draining attack. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 297–307, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[40] Taeri Kim, Noseong Park, Jiwon Hong, and Sang-Wook Kim. Phishing url detection: A network-based approach robust to evasion. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1769–1782, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Brian Kondracki, Babak Amin Azad, Oleksii Starov, and Nick Nikiforakis. Catching transparent phish: Analyzing and detecting mitm phishing toolkits. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 36–50, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1714–1730, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Tasos Laskos. Arachni. https://www.arachni-scanner.com/.

[44] Mathias Lecuyer, Riley Spahn, Yannis Spiliopolous, Augustin Chaintreau, Roxana Geambasu, and Daniel Hsu. Sunlight: Fine-grained targeting detection at scale with statistical confidence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 554–566, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1193–1204, New York, NY, USA, 2013. Association for Computing Machinery.

[46] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic JavaScript. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 723–735, Washington, D.C., August 2015. USENIX Association.

[47] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *PoPETs*, 2018(4):141–158, 2018.

[48] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 141–150, New York, NY, USA, 2007. Association for Computing Machinery.

[49] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[50] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. Uncovering http header inconsistencies and the impact on desktop/mobile websites. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 247–256, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[51] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134, 2008.

[52] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Trans. Web*, 6(1), mar 2012.

[53] Ali Mesbah, Arie van Deursen, and Danny Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, 2012.

[54] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78, 2014.

[55] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and confused: Web cache deception in the wild. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 665–682. USENIX Association, August 2020.

[56] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 736–747, New York, NY, USA, 2012. Association for Computing Machinery.

[57] Hrvoje Nikšić. GNU Wget. https://www.gnu.org/software/wget/.

[58] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer: Protecting users from stateful third-party web tracking with trackingfree browser. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[59] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 653–665, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[61] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, pages 295–316, Cham, 2015. Springer International Publishing.

[62] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[63] Ram Sundara Raman, Adrian Stoll, Jakub Dalek, Reethika Ramesh, Will Scott, and Roya Ensafi. Measuring the deployment of network censorship filters at global scale. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[64] Derick Rethans. Xdebug - Debugger and Profiler Tool for PHP. https://xdebug.org/.

[65] Andres Riancho. w3af - Open Source Web Application Security Scanner. http://w3af.org/.

[66] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. Toppling top lists: Evaluating the accuracy of popular website lists. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 374–387, New York, NY, USA, 2022. Association for Computing Machinery.

[67] Matthias Schur, Andreas Roth, and Andreas Zeller. Procrawl: Mining test models from multi-user web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 413–416, New York, NY, USA, 2014. Association for Computing Machinery.

[68] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. Leaky forms: A study of email and password exfiltration before form submission. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1813–1830, Boston, MA, August 2022. USENIX Association.

[69] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based Side-Channel defenses. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2863–2880. USENIX Association, August 2021.

[70] Pratik Soni, Enrico Budianto, and Prateek Saxena. The SICILIAN defense: Signature-based whitelisting of web javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1542–1557, New York, NY, USA, 2015. Association for Computing Machinery.

[71] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. Can i take your subdomain? exploring Same-Site attacks in the modern web. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2917–2934. USENIX Association, August 2021.

[72] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[73] Karthika Subramani, William Melicher, Oleksii Starov, Phani Vadrevu, and Roberto Perdisci. Phishinpatterns: Measuring elicited user interactions at scale on phishing websites. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 589–604, New York, NY, USA, 2022. Association for Computing Machinery.

[74] Nicolas Surribas. Wapiti. https://github.com/wapiti-scanner/wapiti.

[75] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The long "Taile" of typosquatting domain names. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 191–206, San Diego, CA, August 2014. USENIX Association.

[76] ZAP Dev Team. OWASP Zed Attack Proxy. https://www.zaproxy.org/.

[77] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and dagger: Dynamics of web search cloaking. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 477–490, New York, NY, USA, 2011. Association for Computing Machinery.

[78] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1376–1387, New York, NY, USA, 2016. Association for Computing Machinery.

[79] Tarun Kumar Yadav, Akshat Sinha, Devashish Gosain, Piyush Kumar Sharma, and Sambuddho Chakravarty. Where the light gets in: Analyzing web censorship mechanisms in india. In *Proceedings of the Internet Measurement Conference 2018*, IMC '18, page 252–264, New York, NY, USA, 2018. Association for Computing Machinery.

[80] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 186–197, New York, NY, USA, 2020. Association for Computing Machinery.

[81] Ronghai Yang, Xianbo Wang, Cheng Chi, Dawei Wang, Jiawei He, Siming Pang, and Wing Cheong Lau. Scalable detection of promotional website defacements in black hat SEO campaigns. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3703–3720. USENIX Association, August 2021.

[82] Michal Zalewski. Skipfish. https://code.google.com/archive/p/skipfish/.

[83] Eric Zeng, Rachel McAmis, Tadayoshi Kohno, and Franziska Roesner. What factors affect targeting and bids in online advertising? a field measurement study. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, page 210–229, New York, NY, USA, 2022. Association for Computing Machinery.

[84] Eric Zeng, Miranda Wei, Theo Gregersen, Tadayoshi Kohno, and Franziska Roesner. Polls, clickbait, and commemorative $2 bills: Problematic political advertising on news and media websites around the 2020 u.s. elections. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 507–525, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 423–435. IEEE Press, 2021.

A Appendix

A.1 Classification of the Identified Approaches

Table 11 shows a classification of the identified building block algorithms.

A.2 Excluded Algorithms

This section presents the reasons why we could not implement certain crawling algorithms.

`AutoBlackTest`, `KAFE`, `Dagger`, and `LigRE` do not specify which DOM elements are used or filtered to determine similarity. `WebExplor` and Fetterly et al [28] provided insufficient details on the algorithm application. Lucca et al. [17, 18] and Broder et al. [10] do not specify the threshold used in the similarity algorithms. `Artemis` does not provide enough parameters to reimplement the algorithm without the source code. We also could not modify our reference implementation to accomodate the state clustering approach used in Crescenzi et al. [15].

Name	Algorithm	Tools
Page Similarity: Page URL		
URL Equality	True if the URL strings are the same	Black Widow, JAW, SecuBat, GNU Wget, w3af
Page Similarity: DOM Tree		
Tree Equality	True if the two trees are identical	KAFE
RTED	True if $\texttt{RTED}(t_i,t_j) > c$ for a threshold c, where RTED calculates the minimum of node edit operations that transform one tree into the other one	Crawljax, FeedEx
UI Controls	True if the ratio of common UI controls (e.g., input tags) is greater than a threshold c	AutoBlackTest
Root-Link Paths	True if the ration of common root-to-link paths is greater than a threshold c	Crescenzi
Page Similarity: HTML Code		
SimHash	True if the Hamming distance of two 64-bit fingerprint digests is greater than a threshold c	Crawljax, Manku
TLSH	True if the distance of two locality-sensitive hash digests is greater than a threshold c	Crawljax
Common Shingles	True if the fraction of common shingles is greater than a threshold c	Broder
TAF	True if $\texttt{TAF}(t_i,t_j) > c$, where TAF is the difference of the tag and attribute frequency function of two trees	Lucca
LevenSeq	True if $\texttt{LevenSeq}(s_i,s_j) > c$, where LevenSeq is the Levenstein distance between the sequences of the tags and attributes	Lucca
Page Similarity: Screenshots		
Color histogram	True if χ^2 distance between two color histograms is greater than a threshold c	Crawljax
Perceptual hash	True if Hamming distance of two 128-bit hash digests is greater than a threshold c	Crawljax
Block-mean	True if Hamming distance of two 256-bit hash digests is greater than a threshold c	Crawljax
PDiff	True if the number of common pixels is greater than a threshold c	Crawljax
SSIM	True if the structural distortion value is greater than a threshold c	Crawljax
SIFT	True if the common SIFT key-points are greater than a threshold c	Crawljax
Page Similarity: Combined Algorithms		
jÄk	True if the mean value of the fractions of common forms, hyperlinks, and event handlers is greater than a threshold c	jÄk
ProCrawl	True if buttons, text, and links are the same	ProCrawl
WebExplor	True if page URL strings are the same and the Gestalt Pattern Matching of the HTML codes is greater than a threshold	WebExplor
Dagger	Iterative use of different approaches, respectively, the fraction of common shingles (sequence of hashes of a page), the fraction of different tags, and the fraction of different tags per DOM level	Dagger
LigRE	True if the form prefix trees are the same and the common rooted depth link prefix trees are greater than a threshold c	LigRE
EotS	True if the rooted link prefix trees are the same (precondition)	EotS
Fetterly	True if the URL Rabin fingerprints match and SimHash is greater than a threshold c	Fetterly
Arachni	True if the HTML code and the cookie sets are the same	Arachni
ZAP	True if the HTML code, the HTTP headers, and the request methods are the same	ZAP
Wapiti	True if URL strings (without query string values) and the HTTP methods are the same	Wapiti
Skipfish	True if the word length distributions and the HTTP response codes are the same	Skipfish
Navigation Strategies		
DFS	The Depth-First Search traverses a website by navigating to the most recently discovered page after completing the actions on the current one. The actions are executed in the order of encounter	EotS, Crawljax, ProCrawl, LigRE
JAW	The Iterative Deepening Depth-First Search inspired technique traverses a website iteratively using a BFS approach with a depth limit. The actions are executed in the order of encounter	JAW
BFS	The Breath-First Search traverses a website by navigating to the most early discovered page after completing the action on the current one. The actions are executed in the order of encounter	Crawljax, jÄk, Cookie Hunter, Cached+Confused, KAFE, SecuBat, Crescenzi, Arachni, Wapiti, GNU Wget, w3af, ZAP
Rnd BFS	The Breath-First Search with random action selection traverses a website using the regular BFS for state selection and follows it by random action selection at the selected state	Crawljax
Rnd State	The Random State traverses a website by randomly selecting a state and executing actions on that state in the order of encounter	Black Widow
RL-based	The Reinforcement Learning-based approach learns a policy with an appropriate definition of reward and state	WebExplor, AutoBlackTest
FeedEx	FeedEx calculates a score for each visited page: a combination of code coverage impact, path diversity, and DOM diversity. When selecting the next page, it chooses the page with the highest score	FeedEx
Artemis	Artemis fires sequences of events on the web page. The algorithm prioritizes sequences with low branch coverage.	Artemis

Table 11: Classification and presentation of the identified building block algorithms.

		LoCs (M1)							JavaScript (M2)							Links (M3)						
				Avg. app coverage			Baseline				Avg. app coverage			Baseline				Avg. app coverage			Baseline	
Navig.	Page sim.	Sum unique	Rank	Total	Unique	Rank	Unique	Missed	Sum unique	Rank	Total	Unique	Rank	Unique	Missed	Sum unique	Rank	Total	Unique	Rank	Unique	Missed
BFS	Block-mean	218,136	44	66.26%	0.14%	49	9.56%	5.45%	24,391	38	21.42%	0.85%	33	35.15%	46.59%	191,027	40	18.37%	0.43%	34	30.22%	54.13%
BFS	¬RTED Tr. 3	244,946	9	75.89%	0.22%	10	20.59%	1.12%	33,960	7	22.88%	0.99%	23	45.01%	36.94%	289,184	20	18.73%	0.44%	32	40.68%	40.96%
BFS	Phash	217,551	45	66.52%	0.14%	49	10.32%	6.03%	14,529	47	17.84%	0.27%	47	22.45%	61.95%	125,247	46	12.92%	0.21%	48	20.52%	65.74%
BFS	ProCrawl	210,782	48	65.98%	0.14%	43	10.64%	6.50%	23,725	43	21.80%	0.79%	36	33.73%	46.91%	196,392	38	18.48%	0.43%	35	31.04%	53.39%
BFS	SimHash	232,798	24	58.53%	0.16%	26	7.28%	12.97%	25,167	36	20.52%	0.66%	44	36.67%	46.18%	270,506	30	15.74%	0.35%	43	39.38%	43.57%
BFS	TLSH	244,326	11	72.16%	0.17%	18	16.35%	1.86%	30,363	23	20.98%	1.10%	11	43.30%	41.87%	258,037	33	15.54%	0.42%	36	36.80%	43.88%
BFS	URL Eq.	225,926	32	63.73%	0.14%	37	-	-	29,614	24	21.49%	1.02%	18	-	-	290,564	18	16.33%	0.78%	11	-	-
BFS	URL Path Eq.	219,009	39	60.03%	0.14%	45	1.21%	7.92%	26,520	30	20.91%	0.91%	27	36.42%	43.06%	263,163	32	15.16%	0.58%	19	36.82%	42.78%
BFS	URL Path Eq. & QS	228,191	28	64.69%	0.22%	12	5.69%	3.70%	30,939	18	21.56%	1.12%	9	42.32%	39.74%	320,915	7	16.43%	0.77%	12	43.62%	37.74%
BFS	¬RTED Tr. 2	245,816	5	69.92%	0.16%	23	18.14%	7.32%	31,594	13	22.34%	0.90%	28	42.45%	38.60%	298,696	14	19.24%	0.56%	23	41.44%	39.80%
DFS	Block-mean	212,105	47	64.44%	0.14%	49	8.96%	8.71%	23,605	44	20.89%	0.75%	37	33.43%	46.94%	176,729	43	18.01%	0.34%	44	28.07%	56.25%
DFS	¬RTED Tr. 3	238,313	20	74.22%	0.16%	21	18.25%	1.24%	31,698	12	22.64%	0.95%	26	43.06%	39.05%	274,046	26	18.77%	0.52%	28	38.90%	42.38%
DFS	Phash	218,439	43	70.28%	0.14%	49	15.86%	5.17%	14,131	49	17.71%	0.27%	48	21.56%	62.57%	124,388	47	12.48%	0.32%	46	20.92%	66.15%
DFS	ProCrawl	208,310	49	64.58%	0.14%	44	9.40%	7.41%	24,167	39	21.61%	0.72%	39	34.24%	46.34%	191,407	39	18.14%	0.52%	29	30.51%	54.23%
DFS	SimHash	224,339	34	55.28%	0.15%	31	3.21%	15.38%	25,970	32	20.17%	0.63%	45	37.25%	44.97%	301,721	13	16.56%	0.40%	40	40.99%	38.72%
DFS	TLSH	243,842	13	76.81%	0.22%	11	23.36%	1.78%	30,416	22	20.55%	1.04%	14	43.38%	41.85%	227,784	35	15.11%	0.36%	42	37.23%	50.79%
DFS	URL Eq.	226,495	29	64.63%	0.14%	34	3.90%	2.47%	31,163	17	21.93%	1.11%	10	42.79%	39.79%	277,129	23	16.17%	0.56%	22	39.18%	41.99%
DFS	URL Path Eq.	219,206	38	59.83%	0.15%	29	1.20%	7.94%	28,175	29	21.09%	0.97%	25	39.35%	42.29%	290,496	19	16.21%	0.66%	16	40.57%	40.58%
DFS	URL Path Eq. & QS	229,749	27	64.67%	0.19%	15	5.66%	3.44%	30,732	19	21.34%	0.99%	24	42.04%	39.86%	328,731	4	17.17%	0.67%	15	44.89%	37.65%
DFS	¬RTED Tr. 2	241,565	16	69.20%	0.51%	5	17.37%	8.20%	32,372	10	22.42%	0.84%	34	43.20%	37.91%	291,458	17	18.50%	0.53%	26	40.33%	40.15%
JAW	Block-mean	220,228	36	66.88%	0.14%	35	9.45%	4.19%	22,394	45	20.89%	0.70%	42	32.64%	49.06%	172,053	44	17.14%	0.33%	45	26.76%	56.63%
JAW	¬RTED Tr. 3	238,705	18	72.41%	0.16%	24	15.07%	1.38%	32,031	11	22.85%	0.99%	22	42.93%	38.27%	274,533	25	19.17%	0.57%	21	38.93%	42.30%
JAW	Phash	219,787	37	67.49%	0.14%	42	11.05%	5.09%	14,071	50	17.95%	0.27%	49	21.08%	62.50%	115,089	50	11.35%	0.17%	49	18.27%	67.63%
JAW	ProCrawl	214,338	46	65.61%	0.14%	36	9.38%	5.89%	23,754	42	21.66%	0.71%	40	34.04%	47.09%	177,791	42	17.31%	0.52%	30	28.66%	56.35%
JAW	SimHash	232,356	26	58.42%	0.15%	30	7.10%	12.96%	26,355	31	20.46%	0.69%	43	37.79%	44.64%	288,914	21	16.10%	0.43%	33	41.01%	41.34%
JAW	TLSH	242,988	15	71.78%	0.17%	17	15.59%	2.14%	31,166	16	20.63%	1.14%	8	43.70%	40.75%	264,489	31	15.27%	0.29%	47	38.87%	44.35%
JAW	URL Eq.	224,854	33	63.41%	0.14%	40	0.61%	1.20%	30,549	20	22.06%	1.05%	12	40.93%	39.06%	297,981	15	16.56%	0.79%	10	40.06%	38.53%
JAW	URL Path Eq.	218,667	41	59.97%	0.14%	45	1.15%	7.93%	28,358	27	20.99%	0.87%	30	38.37%	40.98%	270,850	29	15.00%	0.64%	17	37.56%	41.79%
JAW	URL Path Eq. & QS	226,454	30	63.77%	0.14%	45	4.22%	3.70%	31,297	15	21.06%	1.01%	20	42.09%	38.80%	347,421	2	16.48%	0.58%	18	45.04%	34.28%
JAW	¬RTED Tr. 2	241,448	17	69.32%	0.15%	28	17.94%	7.99%	31,413	14	22.22%	0.87%	31	42.54%	39.04%	273,345	28	18.74%	0.41%	38	39.37%	42.96%
Rnd BFS	Block-mean	245,700	6	75.81%	0.59%	3	21.42%	2.17%	24,868	37	22.27%	1.01%	21	36.27%	46.48%	204,953	37	17.44%	1.07%	8	33.16%	52.86%
Rnd BFS	¬RTED Tr. 3	259,600	1	78.86%	0.39%	7	25.40%	0.85%	41,691	2	23.69%	2.13%	4	52.54%	33.18%	313,742	9	18.98%	2.08%	6	45.99%	41.69%
Rnd BFS	Phash	237,993	21	69.86%	0.38%	8	16.39%	6.60%	14,993	46	18.16%	0.41%	46	23.27%	61.16%	121,661	48	12.56%	0.40%	39	20.93%	66.89%
Rnd BFS	ProCrawl	245,255	8	74.64%	0.21%	14	19.07%	1.69%	25,590	34	22.07%	1.02%	17	36.76%	45.36%	209,870	36	18.55%	0.95%	9	33.07%	51.66%
Rnd BFS	SimHash	245,471	7	62.51%	0.86%	2	11.47%	10.59%	29,296	25	21.47%	1.63%	7	42.40%	43.02%	316,574	8	16.97%	2.14%	5	47.94%	43.29%
Rnd BFS	TLSH	246,426	4	74.83%	0.46%	6	20.46%	2.16%	36,602	5	22.07%	2.07%	5	49.86%	38.02%	302,670	11	16.71%	2.03%	7	46.36%	44.12%
Rnd BFS	URL Eq.	252,530	3	76.53%	0.55%	4	22.09%	1.40%	39,655	3	23.25%	2.65%	1	51.46%	35.00%	339,134	3	18.12%	3.14%	1	48.43%	39.81%
Rnd BFS	URL Path Eq.	235,231	23	67.80%	0.25%	9	10.99%	4.16%	34,665	6	22.13%	2.35%	3	47.21%	38.21%	302,073	12	16.48%	2.34%	3	44.63%	42.44%
Rnd BFS	URL Path Eq. & QS	244,838	10	70.18%	0.21%	13	14.00%	2.74%	43,319	1	22.98%	2.56%	2	54.49%	33.43%	369,810	1	17.62%	2.80%	2	51.57%	38.36%
Rnd BFS	¬RTED Tr. 2	259,028	2	76.46%	1.31%	1	23.66%	1.93%	37,850	4	23.82%	1.93%	6	49.28%	35.17%	324,179	6	19.38%	2.19%	4	47.10%	40.98%
Rnd State	Block-mean	218,518	42	68.20%	0.14%	48	13.44%	6.13%	23,977	41	21.76%	0.74%	38	34.11%	46.65%	185,483	41	17.37%	0.53%	27	29.39%	54.93%
Rnd State	¬RTED Tr. 3	238,446	19	75.92%	0.17%	20	21.38%	1.65%	33,240	8	22.50%	1.03%	15	43.67%	36.78%	293,303	16	19.13%	0.56%	24	40.82%	40.26%
Rnd State	Phash	218,722	40	69.71%	0.14%	49	15.34%	5.62%	14,189	48	17.51%	0.26%	50	21.29%	62.29%	119,948	49	11.99%	0.13%	50	20.47%	67.17%
Rnd State	ProCrawl	206,816	50	63.92%	0.14%	39	8.68%	7.77%	24,075	40	21.06%	0.85%	32	34.42%	46.68%	171,998	45	17.44%	0.42%	37	27.99%	57.37%
Rnd State	SimHash	235,734	22	59.32%	0.16%	22	8.98%	12.95%	25,285	35	20.59%	0.70%	41	36.20%	45.52%	303,797	10	15.40%	0.57%	20	42.11%	39.48%
Rnd State	TLSH	243,257	14	76.56%	0.17%	19	22.78%	1.69%	28,209	28	20.74%	1.04%	13	40.53%	43.35%	273,591	27	14.29%	0.40%	41	41.22%	44.65%
Rnd State	URL Eq.	226,266	31	64.42%	0.14%	33	4.20%	3.22%	28,529	26	20.92%	1.02%	16	39.83%	42.03%	286,916	22	16.18%	0.72%	13	40.20%	40.95%
Rnd State	URL Path Eq.	223,208	35	60.30%	0.16%	25	1.72%	7.62%	25,969	33	20.39%	0.88%	29	35.70%	43.62%	256,588	34	15.42%	0.47%	31	36.79%	44.18%
Rnd State	URL Path Eq. & QS	232,676	25	66.03%	0.15%	27	7.11%	2.52%	30,462	21	21.28%	1.02%	19	41.39%	39.71%	327,732	5	17.12%	0.71%	14	45.48%	38.51%
Rnd State	¬RTED Tr. 2	244,168	12	70.37%	0.17%	16	19.16%	7.62%	32,561	9	21.79%	0.83%	35	43.22%	37.57%	275,148	24	16.77%	0.54%	25	39.46%	42.67%
FeedEx	Block-mean	208,206	59	62.07%	0.13%	57	6.03%	9.26%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	¬RTED Tr. 3	220,522	38	64.80%	0.14%	49	6.87%	6.23%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	Phash	210,216	55	62.82%	0.13%	57	6.01%	8.16%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	ProCrawl	199,303	63	59.81%	0.14%	49	6.29%	12.55%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	SimHash	221,853	37	54.98%	0.14%	40	2.90%	15.78%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	TLSH	219,249	42	63.51%	0.15%	32	6.71%	6.81%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	URL Eq.	209,735	56	56.70%	0.09%	60	1.46%	14.47%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	URL Path Eq.	215,914	51	59.26%	0.13%	57	0.68%	8.81%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	URL Path Eq. & QS	219,785	41	61.03%	0.14%	38	0.90%	5.75%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
FeedEx	¬RTED Tr. 2	223,029	36	60.68%	0.14%	56	8.26%	13.67%	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Tool	Arachni	208,753	57	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Tool	Skipfish	202,231	62	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Tool	Wapiti	206,587	61	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Tool	Zap	193,977	64	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 12: Coverage results of all configurations.