Maelstrom Crawler: A feedback-driven web vulnerability scanner

Improving web vulnerability scanning with dynamic strategies

Master's thesis in Computer Science and Engineering

Oscar Morisbak Olsson
Roj Mert Tekin

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Master's Thesis 2026

Maelstrom Crawler: A feedback-driven web vulnerability scanner

Improving web vulnerability scanning with dynamic strategies

Oscar Morisbak Olsson
Roj Mert Tekin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Maelstrom Crawler: A feedback-driven web vulnerability scanner
Improving web vulnerability scanning with dynamic strategies
Oscar Morisbak Olsson, Roj Mert Tekin

Supervisors:
Eric Olsson Department of Computer Science and Engineering

Examiner:
Ahmed Ali-Eldin Hassan Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026 Maelstrom Crawler: A feedback-driven web vulnerability scanner
Improving web vulnerability scanning with dynamic strategies

Oscar Morisbak Olsson
Roj Mert Tekin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Web application vulnerability scanners rely on crawlers to explore applications and discover potential attack surfaces. The effectiveness of these scanners is therefore heavily influenced by the crawler's ability to navigate diverse applications. Most existing crawlers use a fixed navigation strategy, despite different strategies often being more effective in different contexts. This thesis investigates whether a crawler that dynamically switches between navigation strategies based on feedback it receives at runtime can improve web application vulnerability scanning. To address this problem, we design and implement the Maelstrom Crawler, a feedback driven extension of the open-source Black Widow vulnerability scanner. Maelstrom dynamically alternates between randomised BFS and randomised DFS exploration using feedback channels that monitor the progress during execution and suggest when to switch strategy. An initial set of nine candidates for feedback channels was analysed using principal component analysis and an ablation study, resulting in five selected channels: growth, duplicate candidates, URL diversity, error rate, and novel code. These channels together guide the strategy-switching decisions through a majority voting based mechanism. To prevent excessive switching and thrashing, the design includes comparison and baseline windows together with a cooldown period. The final design was evaluated on three open-source applications: OsCommerce, WordPress, and Kanboard. Results show that Maelstrom Crawler achieved an average code coverage improvement of 9.63% compared to Black Widow, 91.68% compared to OWASP ZAP, and 70.74% compared to EvoCrawl. Internal experiments further demonstrated that feedback-driven strategy switching outperformed purely random switching. However, the improved code coverage did not consistently translate into increased vulnerability discovery, highlighting a gap between exploration effectiveness and vulnerability detection. The results indicate that feedback-driven strategy adaptation can improve crawler exploration efficiency and code coverage in web applications. Furthermore, they suggest that dynamically combining multiple navigation strategies is a promising direction for future web vulnerability scanners.

Acknowledgements

We would like to thank our supervisor Eric Olsson, for all his feedback, guidance, and abundant assistance throughout this thesis. We would also like to thank our examiner Ahmed Ali-Eldin Hassan for his insights.

Oscar Morisbak Olsson & Roj Mert Tekin, Gothenburg, 2026-06-22

Contents

List of Figures

List of Tables

1

Introduction

In 1993, the World Wide Web was first made public, and has since become an increasingly integral part of our lives. As new web applications are introduced and modified year after year, an unfortunate by-product is new vulnerabilities in the web, an issue exacerbated by the rise of AI-generated code [1].

A particularly dangerous yet commonly introduced vulnerability class is injection-based issues, ranking at 5th place in OWASP's top 10 list of web vulnerabilities [2]. Injection attacks such as Cross-Site Scripting (XSS) can cause enormous damage [3] by giving malicious actors a way to execute arbitrary code on remote machines. These vulnerabilities can be used to escalate privileges and access accounts or other sensitive data on the target device, such as session cookies.

While web application developers want to remove vulnerabilities in their code, finding them is a challenging task. Web vulnerability scanners are one tool that developers can use to discover vulnerabilities in their own code. These programs crawl a running web application by traversing it automatically and testing for vulnerabilities within it. Vulnerability scanners have not only proven effective for vulnerability discovery [4], but can also be preferred due to their ease of use. A core component of scanners is crawling, as their ability to discover vulnerabilities depends on the vulnerable code first being discovered by the crawler. This thesis focuses on improving this crawling component in an XSS vulnerability scanner.

1.1 Crawling in vulnerability scanners

Web crawlers are software that automatically navigate through a web application, starting with a list of seed pages (URLs) as starting nodes. A crawler then iteratively navigates through these nodes, gaining more nodes to traverse. The crawler's strategy decides how it will navigate these nodes. The crawling process attempts to maximise its coverage of unique application states [5] found in these nodes. A security scanner extends this notion of a crawler by also scanning each node for vulnerabilities. In practice, a vulnerability scanning web crawler tests for security flaws such as XSS vulnerabilities by injecting payloads in input fields, and then monitoring the application to discover which payloads reveal vulnerabilities. Therefore, as these crawlers'[1] coverage of application states increases, including vulnerable inputs and outputs, more vulnerabilities can be discovered.

Despite the effectiveness of crawlers, they typically only implement a single strategy [5]. A fixed strategy can fail to match the diversity of web applications, implementations, and behaviours. Code coverage [5–7] and vulnerabilities [7] discovered can vary greatly across crawlers on the same web application. For instance, YuraScanner and SpiderSapien find different vulnerabilities within the unique code they separately discovered in the same application [7]. This indicates that the results of scanners with different scanning strategies, especially navigation strategies, can often be complementary within larger applications. However, the process of combining results can often be time-consuming, as combining results across different crawlers and different stateful runs often requires demanding manual work.

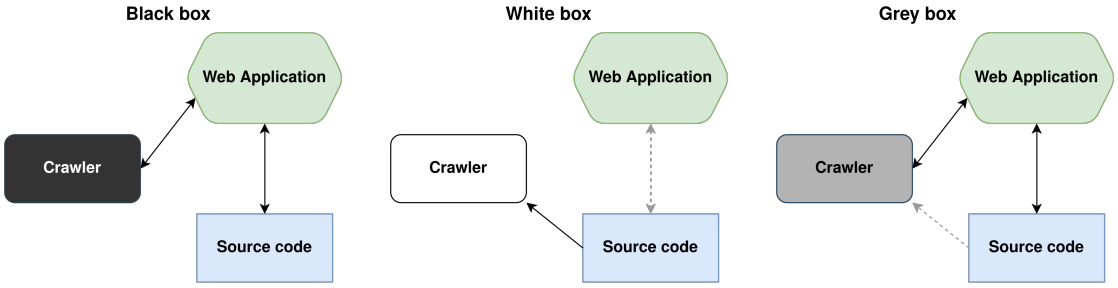
1.2 Black-, white-, and grey-box scanners

Scanners can be distinguished by their access to web application internals into black-, grey- or white-box [4]. This access is illustrated in Figure 1. Black-box vulnerability scanners only use client-side data, such as URLs and resulting pages, to dynamically analyse a running application and therefore require guidelines or heuristics to crawl. On the other hand, white-box scanners have access to the application's source code. White-box scanners typically discover vulnerabilities with static analysis of source code, though they can also integrate dynamic analysis to reduce their false positives. Finally, grey-box scanners have access to some internal signals from the web application, but not the source code, such as unique lines of code executed [4].

1.2.1 Black-box crawling relies on heuristics

Black-box crawlers operate with incomplete information. They therefore infer their navigation decisions from the client-side interface and network responses. This limited feedback makes it hard for black-box crawlers to correctly prioritise and decide on what action to take and which are more valuable to explore. Moreover,

[1] In this thesis, we use the term crawler interchangeably with "*vulnerability scanning web crawler*".



(a) Black-box scanner **(b)** White-box scanner **(c)** Grey-box scanner

Figure 1: Visualisation of the web application components accessed by black-, white- and grey-box scanners.

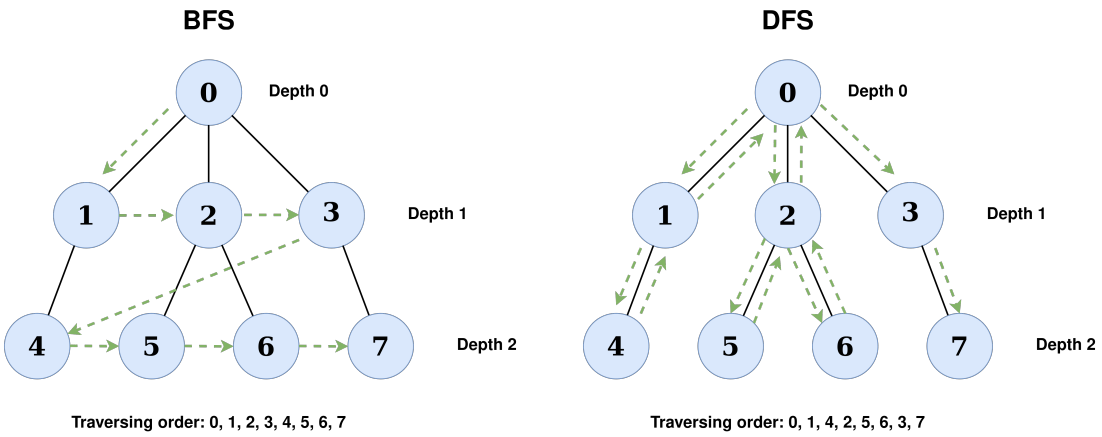
an exhaustive exploration is not only impractical but also unrealistic for non-trivial web applications because their state space could be unbounded. A common example of this is a calendar interface where the user can click "next month" or "next year" indefinitely, effectively generating an infinite sequence of distinct states.

Web applications often have many potential actions on any given page, especially now that most also include dynamic client-side elements. The statefulness of web applications makes the order of actions matter, causing the number of possible application states to explode. Therefore, black-box crawlers rely on heuristics to guide their navigation strategy.

One common heuristic is modelling the navigation strategy as a graph search. Breadth First Search (BFS) is often used as a navigation strategy, which explores all available actions on the current page before their neighbours, as seen in Figure 2a. Another common heuristic is Depth First Search (DFS), which instead follows one branch of actions deeply in the web application, as seen in Figure 2b [8]. These heuristics act as practical shortcuts that enable the crawler to increase its coverage and discover more vulnerabilities.

1.2.2 One heuristic is not optimal

No single heuristic can consistently perform well across all web applications due to their sheer diversity. While a heuristic that prioritises clickable objects may work well for most applications, it could also incorrectly prioritise a logout action, destroying the scanning run [6]. Certain heuristics can better suit specific applications or parts of one. However, across applications, no single heuristic may be optimal, as seen by the results of Stafeev et al. [5]. A crawler that dynamically incorporates runtime information to select a heuristic or tune its parameters throughout the crawling process has the potential to improve crawling performance.



(a) BFS search **(b)** DFS search

Figure 2: An illustration of how, in a BFS search, each known node in the current depth will be visited before proceeding to nodes in the next depth. Meanwhile, in DFS search, it will proceed to deeper nodes until it cannot find a deeper unvisited node before backtracking.

Examples of such parameters include:

- Maximum depth limits and thresholds.
- Loop-control parameters, including state similarity thresholds and repetitive actions caps.
- Budget parameters to limit the maximum number of actions per state.

1.3 Approach

To address the prior limitation of crawlers employing a single, static navigation strategy, We design a method that can instead switch between randomised BFS and DFS strategies at runtime, by using signals from the application as feedback channels.

First, we must define and select the appropriate runtime signals that are available and can be used as feedback channels to the crawler from the application. We initially define nine feedback channels, which can be useful for measuring how well a crawler is performing on an application at any given time, such as increasing server-side code coverage. We evaluate all of these feedback channels on runtime data to ensure they measure different things and are not correlated, while also measuring how much each channel contributes to performance. We then ensure that the feedback channels do not overlap by a signal analysis with PCA, and remove correlated channels, reducing the number of feedback channels to five. An ablation study finally verifies that these feedback channels are also gathering useful information for the crawler and contribute to the overall coverage achieved.

Now that we have a good selection of feedback channels, we implement a switching mechanism that compares a comparison-window of recent measurements to a baseline-window of historic measurements, while incorporating a cooldown value to prevent thrashing. Appropriate values for these parameters are chosen based on empirical analysis of sample crawler runs. A set of signal analyses determines the correct sizes for the *comparison-* and *baseline-windows*. We also perform a parameter sweep to determine an appropriate cooldown time. Finally, we implement this method in our own Maelstrom Crawler. Maelstrom Crawler is an extension of the prior open-source Black Widow crawler.

1.4 Evaluation

We evaluate the crawling performance of Maelstrom on three open-source web applications, compared to three other web application vulnerability scanners. We achieve 9.63% more coverage compared to Black Widow, 91.68% compared to ZAP and 70.74% compared to EvoCrawl. We also find a total of 6 XSS vulnerabilities, finding the same number of vulnerabilities as EvoCrawl and Black Widow in both WordPress and OsCommerce. However, Black Widow is the only crawler that manages to find vulnerabilities within Kanboard, where it finds 2. The performance increase in coverage is likely due to Maelstrom Crawler's ability to reduce redundant actions by swapping strategies and navigating out of areas within the web applications where it is stuck and does not achieve any progress, something that happens regularly to all the other tested crawlers.

1.5 Our contributions

Our contributions in this thesis are:

- We design a system where compatible strategies can be swapped at runtime without losing previously stored navigation information.
- We design and implement feedback channels in a modular fashion.
- We evaluate a selection of feedback channels on their similarity and performance improvement.
- We implement a complete system that combines the best feedback channels and multiple strategies.
- We evaluate the performance gain from purely switching navigation strategies to a comparative feedback-driven switching.

2

Background

2.1 Background on heuristic strategies

The heuristic we will focus on in this thesis is the crawling strategy, used by the crawler to decide which parts of a web application to explore. Crawlers traditionally rely on simple heuristics like **BFS** or **DFS**, often combined with randomisation. Some crawlers may also vary their strategy by starting with a BFS overview and then switching to a more directed search [5]. This can generally fit the structure of many applications, which begin with a landing page before branching into deeper, more specific functionality.

However, fixed heuristics have their limitations. Some applications can reveal more functionality only after specific interactions or state transitions, which means that some key areas may remain hidden during an initial search. Examples of this include multi-step forms or shopping flows. A single heuristic or sequence of heuristics cannot reliably cover all parts of an application.

Ultimately, a crawler that dynamically switches its strategy based on context has the potential to achieve greater coverage and reveal more vulnerabilities.

2.1.1 Web applications and XSS

Web applications typically consist of a browser-based frontend and a backend server connected to persistent storage, for example, a database. Users can then interact with pages by navigating, clicking elements, and filling and submitting forms. These actions will often trigger HTTP requests to the backend. The backend then processes the input, reads and writes data, and returns responses that the browser renders.

Since many features in web applications depend on data supplied by users, these programs can have a large and complex attack surface.

XSS is a vulnerability where attacker-controlled input is inserted into a page in a way that causes the browser to execute unintended JavaScript. This can happen when input from forms, URLs or stored database fields is rendered or reflected without the correct output encoding. XSS can often be categorised as either reflected, with an immediate response, stored, which is persisted and later served, or DOM-based. Reflected XSS often happens when attacker-controlled input is included in an HTTP response, for example, as a part of an error message or search result. The essence is that the payload is not stored server-side but rather is "reflected" back immediately. Stored XSS occurs when malicious input is persisted by the application, usually in a database, but it could also be in logs, comments or profiles that later render to other users. Since the payload is served from the application, stored XSS can often be more impactful. DOM-based XSS is primarily triggered on the client side when JavaScript is running in the browser and reads data from a controllable source and then writes it to a dangerous sink. If exploitable, XSS can give an attacker options to enable session theft, account takeover or data exfiltration.

Even though XSS is a well known vulnerability, consistently preventing it can be very difficult in practice because it depends on where and how data is used through different elements such as HTML, attributes, JavaScript, and URLs. Since many XSS-relevant flows are state-dependent, a vulnerable path may only be reachable after specific sequences of actions, making it very hard to detect. These issues can make static analysis less pragmatic, and we therefore require an alternative way to detect XSS vulnerabilities. Because of their state-dependent nature, detecting all XSS dynamically would require the ability to navigate and try payloads in each possible area, but this process would be prohibitively time-consuming. Instead, an automated scanner that attempts multiple payloads is commonly used, and as the code itself is hard to statically analyse, these scanners use a black-box approach where the system explores the application like a user and observes runtime behaviour.

2.2 Black-box crawling prior work

The web has evolved throughout the years, which in turn led to the necessary improvement of crawlers. Various works have focused on navigation strategies, payload generation, or form validation. For instance, since Web 2.0, AJAX (Asynchronous JavaScript and XML) has been used to dynamically update webpages without changing the URL. This required a new approach to identify dynamically and index these different states and DOM structures, something that Crawljax managed to accomplish [9]. The discovered dynamic pages, combined with the previous server-side URLs, nowadays fall under the umbrella term *state*. Adding awareness of this to a crawler is an aspect *Enemy of the State* [10] improved upon. They took a black box approach to identify application state, through inspecting similar web pages and the

deviations in responses, which proved itself effective on web applications with many different possible states [10].

The jÄk paper addressed a key limitation of traditional black-box scanners at the time in how they miss large portions of JavaScript-heavy applications [11]. The problem with previous scanners was how they extracted URLs by parsing HTML and applying regexes. The jÄk paper instead proposed crawling driven by dynamic analysis of client-side JavaScript. jÄk runs the application's JavaScript and then hooks key browser APIs to see what the app does at runtime, mainly which UI events are registered and which network requests the code tries to call. Empirically, jÄk was evaluated against several existing scanners on 13 web applications, and was shown to explore an 86% larger surface than other competing approaches. This method demonstrated how dynamic and event-aware crawling can translate into improved testing reach for web applications.

The Black Ostrich crawler improved form validation by overcoming prior problems with input validation on modern web forms. Instead of relying on brute force or fixed input dictionaries, Black Ostrich extracts validation constraints from regular expressions and uses an SMT-based string solver. This solver synthesises inputs to satisfy constraints, which will allow the crawler to progress past states that otherwise were blocked [12]. The key contribution from Black Ostrich is a validation-aware crawling platform built by integrating the Black Widow crawler with the Ostrich string solver. Furthermore, new automata-based techniques enable sound and complete solving of ECMAScript regular expressions, which a typical SMT string solver does not support natively. With these improvements, Black Ostrich showed a 25% improvement in form validation compared to the union result of the other tested scanners.

Improving navigation and state validation has shown minor improvements to code coverage in the work by Cazzaro et al [13], using Reinforcement Learning (RL) methods. However, the same paper showed that combining efforts from different crawling strategies showed better improvements. By using a mix of BFS, DFS, and randomised navigation strategies, an improvement upwards of 20% was observed. But arguably more impressive was the efficiency of this approach, as the ceiling of code coverage on certain applications was reached faster.

3

Challenge

Black-box crawling still faces significant hurdles to becoming effective across web applications. Not having access to the backend makes it challenging to identify the application state, which is critical for knowing when to re-navigate already explored areas of the crawled application. Actions taken by the crawler often change how many parts of the application behave and potentially expose new areas with possible vulnerabilities. For instance, one could create a project on Kanboard, which adds actions to a previously navigated URL, meaning the same URL is effectively a new edge that should be explored once more. In Figure 1, we can see that creating a project populates the project list with an item, which brings with it new actions, all within the same URL.

This creates a mismatch between the crawler's understanding of the web application's state and the actual state in the web application: the crawler does not know when it should re-navigate to these URLs, as they now contain new actions.

But an even more prevalent issue are repeated actions, a natural consequence of not being able to distinguish server-side code from client-side interfaces. The same piece of code can be utilised in multiple areas; for instance, the same input sanitisation code could be used in both the name of a project and within its identifier field. An example of this code reuse is in the project creation workflow of the aforementioned Kanboard application.

Again, this mismatch between the crawler's understanding of application state and reality leads to the crawler repeatedly testing the same piece of code, as it sees these multiple URLs as different states. This wastes the limited crawling time,

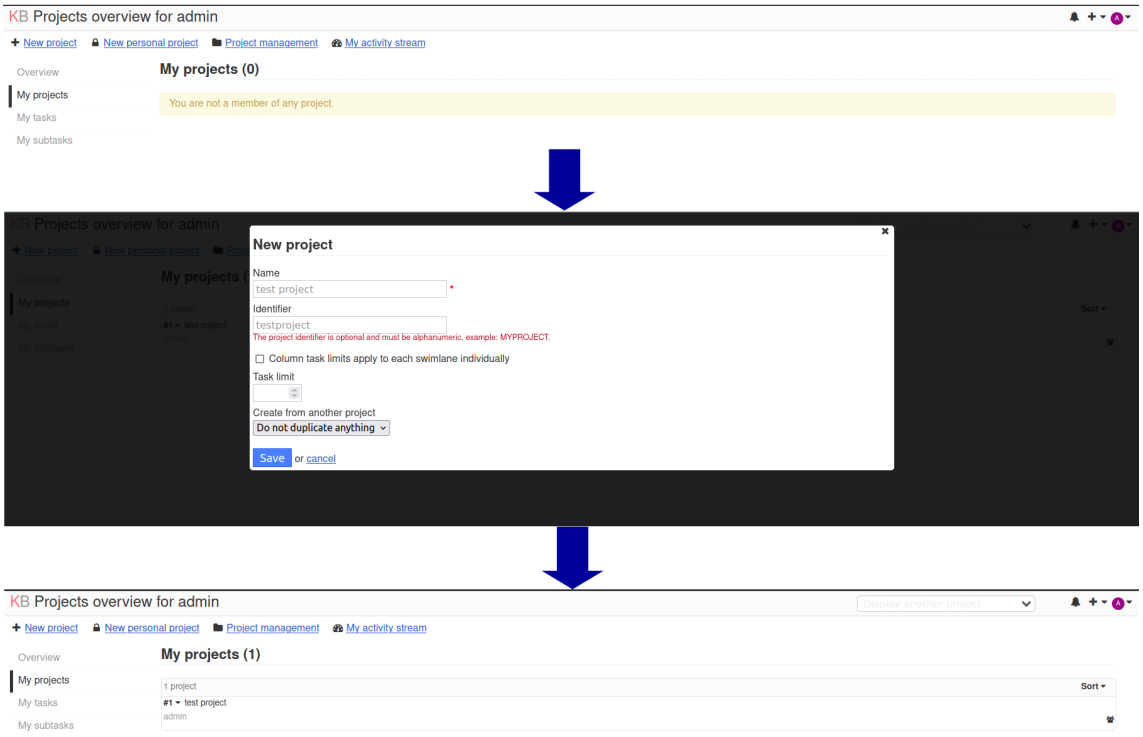


Figure 1: Example of when creating a project in Kanboard, a page with the same URL now has new available actions.

instead of analysing different vulnerable code and deriving payloads that expose these vulnerabilities. To remedy this, black-box crawlers use different heuristics to guide the navigation process, but they still struggle with incomplete coverage, getting stuck in areas, and overall failing to navigate into novel parts of the application and thereby adequately explore the application's vulnerabilities [14].

3.1 Navigation

Navigation strategies have consistently been one of the largest hurdles for security scanners; many security tools even entirely forgo navigation and only scan the current page [5]. It has been observed that, despite best efforts, a simple pseudo-randomised navigation strategy generally performs best [5]. However, strategies based on this heuristic are still imperfect, and often fail in multiple ways, be it doing repeated actions, or not discovering the far corners of an application where vulnerabilities can be exceptionally hard to find.

To remedy this, a crawler can perform minute changes to its crawling strategy during runtime, guided by differences in executed code or by pattern recognition [15, 16]. But even these dynamic strategies tend to ignore many responses to their actions and keep the same overall navigation strategy throughout a crawling run. A crawler that changes its navigation strategy throughout execution time has the potential

to combine the advantages of different navigation strategies to achieve higher code coverage, and thereby discover more vulnerabilities.

First, the scanner must be made aware at runtime of when to switch strategies. This can be achieved by integrating feedback channels such as novel executed code, following the assumption that novel code reveals that we are either in a different application state or in a hard-to-reach corner. In either case, the scanner must remain within the given space to ensure proper scanning, and using a strategy that stays within this area of the application to do a thorough scan is ideal.

3.1.1 Prior work pitfalls

Prior work in the area has not thoroughly evaluated the possible performance gains from switching navigation strategies at runtime. This limits both their coverage and efficiency. For instance, while random BFS has been proven very successful [5], the unfortunate side effect of this is that crawlers often exclusively rely on this for their navigation, and do not try to improve upon it. These works typically disregard navigation and focus on another aspect of the problem, such as payload generation. However, the core problem of navigation is still unsolved, and these crawlers are missing critical points within applications to test their other improvements.

There are a small number of prior works that show a positive correlation between combining different strategies, such as BFS and DFS, to improve crawler performance [13]. However, they do not include random DFS and confound their results by only testing different reinforcement learning methods internally, instead of comparing to other published crawling methods to evaluate the performance gain of switching strategies in a crawler.

4

Methods

We design the *Maelstrom* crawler to incorporate a context-aware switching mechanism, based on runtime feedback from the web application available to the crawler. Our proposed crawling method switches between its strategies based on a majority-vote mechanism comparing a current window to a historical window of the selected feedback channels. We propose a set of feedback channel candidates and remove redundant ones through Principal Component Analysis (PCA), and use a set of ablation studies to measure how much each channel contributes to performance. We describe how we select the parameters for the crawler, including the window sizes and cooldown, based on a set of signal analyses. Finally, we implement Maelstrom Crawler as an extension of the prior open-source Black Widow crawler.

4.1 Design

The Maelstrom Crawler aims to adapt and extend the existing *Black Widow* crawler [6] by integrating a dynamic context-aware navigation mechanism. Currently, Black Widow has a very streamlined navigation strategy. It starts with an initial "early-gets" strategy, where it gathers multiple "seed" nodes that, after the hardcoded 100 iterations, switches to a Randomised BFS strategy. Thereafter, it continues indefinitely, never switching its navigation strategy again, as shown in Figure 1.

The extended Maelstrom Crawler will start with the same "early-gets" strategy but instead uses a dynamic method to pick between a BFS or DFS strategy afterwards. The internal feedback gathered during runtime, as outlined later in Section 4.3.1, will provide context to the crawler of the web application. With this gathered information, these feedback channels are used to instruct the crawler to switch navigation strategy when appropriate, by continually updating the information from

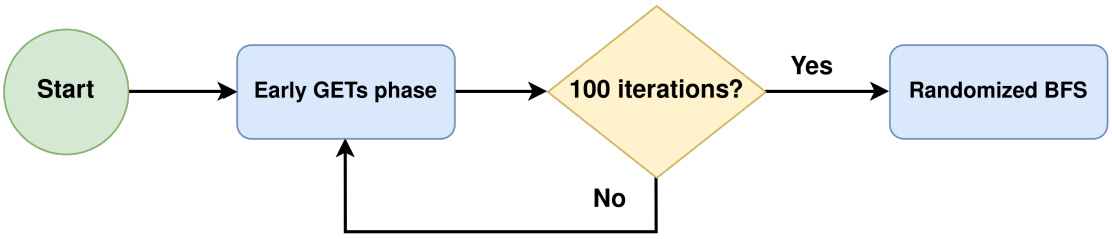


Figure 1: Block scheme illustrating Black Widow's navigation strategy.

the feedback channels to ensure relevant feedback. The switching of navigation strategy will transfer any relevant information, such as previous observations of feedback channels and the navigation graph, between them, as shown in Figure 2. The BFS and DFS strategies prioritise different actions, with BFS tending to switch to unrelated states of the application, while DFS tends to explore states that descend from the current one. Including both strategies allows us to match the strategies to their specialised purposes.

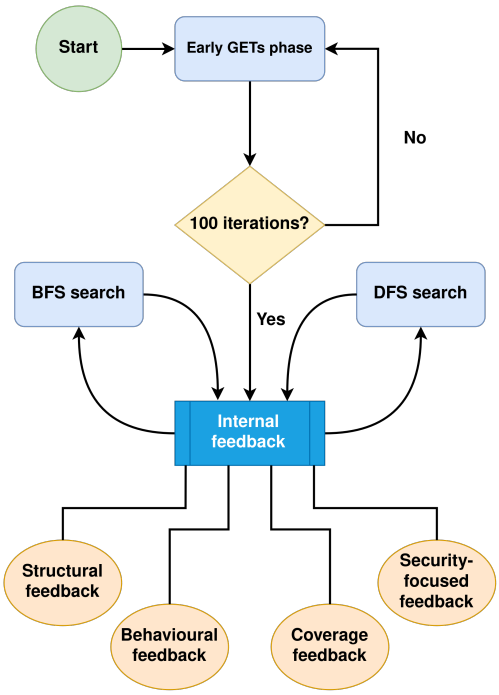


Figure 2: Illustration of how our extended scanner compiles multiple feedback channels before proceeding with a navigation strategy, continuously looping back to integrate new feedback throughout runtime.

4.2 Switching

The core of the Maelstrom Crawler is its ability to adapt to an application's structure by switching traversal behaviours in real-time. To facilitate this, the base BFS and DFS strategies and their algorithms were modified into interchangeable state-managed components. Rather than running strategies to completion, each strategy

at different times operates as the sole "active driver", which allows the controller to toggle the active strategy without losing the global state of the graph or queues. The switching mechanism is governed by a feedback loop, as illustrated in the decision graph seen in Figure 3.

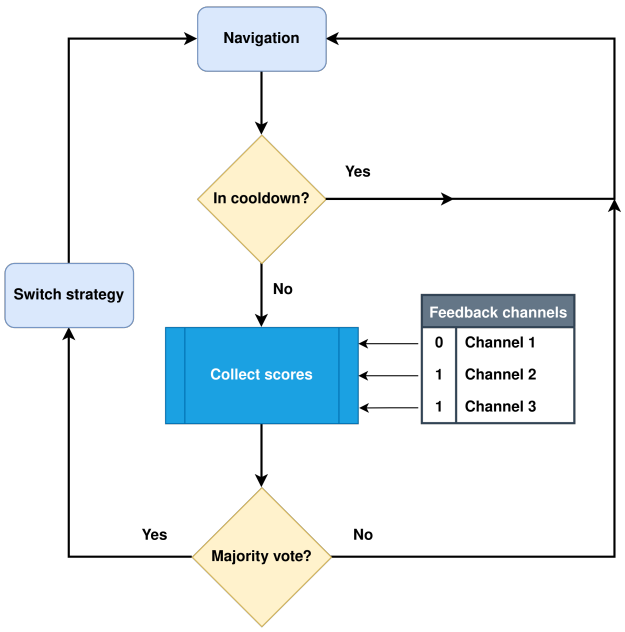


Figure 3: Overview of the crawlers switching mechanism, showing the decision flow from cooldown check into collecting scores and then into the majority vote check.

4.2.1 Majority-vote switching logic

The Maelstrom Crawler uses a switching controller that aggregates suggestions from each feedback channel. Based on the results from the PCA and ablation studies, we use 5 feedback channels, see Section 4.3.2 and Section 4.3.3. Each channel evaluates if the crawler is improving on its metric or not by comparing itself to a baseline-window, which is derived by averaging feedback values from an adequately large amount of recently observed feedback values from the previously derived feedback channel. If the average of the last few iterations (excluding the baseline-window) is worse than the value from the baseline-window, it suggests a switch by setting the suggestion value to 1, otherwise 0. We call these 5 latest iterations the comparison-window. Rather than calculating individual "scores" for BFS and DFS in order to decide which one is the most suitable, it instead only evaluates the performance of the active strategy. The controller does not care which strategy is the active one; it only cares about whether the current one performs poorly and then switches accordingly.

A majority-vote threshold governs the decision to switch. If N is the number of active feedback channels, then a strategy switch will only occur if at least half of those N channels vote for it. This threshold ensures that the crawler does not switch too early due to a single noisy metric. By requiring a majority of channels to signal

for a switch, the crawler ensures that a switch to a different strategy could result in potentially more productive exploration.

4.2.2 Anti-thrashing mechanisms

"Thrashing" is a potentially debilitating problem for adaptive strategies. Thrashing occurs when the system rapidly switches between strategies because of noise in short-term measurements, causing the system to not make proper progress. In order to prevent this, switching is gated by a few mechanisms to make sure it doesn't happen too often. These mechanisms consist of:

- **Early GETs phase:** the crawler will start in a BFS early GETs phase for the initial 100 steps, which is very similar to how Black Widow starts. This period will allow the crawler to establish baseline averages for what values could be considered normal for the feedback channels on this specific application. It also allows the crawler to discover an initial set of reachable pages and states.
- **Cooldown:** after a switch, the controller will enforce a minimum number of steps before another switch can occur. This is a simple yet effective way to limit how often a switch can occur. It is important to choose a cooldown value that is not too restrictive yet not too loose.
- **Comparison-window:** to determine the scores for the previously mentioned majority-vote switching scheme, it checks the average signal value of the feedback channels over a few iterations instead of singular ones. This avoids scenarios when outlier values from signal(s) can cause an unnecessary switch.

These mechanisms help avoid unnecessary switching and ensure that switching reflects more consistent evidence.

4.3 Feedback channels

One of the central design decisions for the crawler is how to switch between strategies; to do so, we require feedback channels which incorporate dynamic information. This is achieved by identifying signals that can be used as feedback for our decision algorithm. Since web applications may differ widely in their behaviour, size, and structure, we want to combine several channels that aren't overly sensitive to these variations, and ensure the switching process is beneficial. We identified several channels that observe different elements of the traversed web application. Additionally, a web application can differ vastly even within itself, as one part may behave very differently from another, requiring us to discard previously observed results from our feedback channels. We achieve this by recording the 5 most recent observations that we call the comparison-window and comparing them to a baseline-window, a record of the 25 prior iterations. A channel adds a score if the 5 most recent observations are relatively larger than the baseline-window. These window sizes are chosen based on the analysis

whose methodology is described in Section 4.4.1 and evaluated in Section 5.1.1.

4.3.1 Channel details

- **Growth** measures how quickly the interaction graph expands during a crawl step. An edge is considered a newly discovered actionable item, such as a new URL request, a form submission, or an iframe transition. For each visited state or page, the crawler extracts candidates and attempts to add them to the graph and then counts how many of them became **new unique edges**.

  Let E_t^{new} denote the number of extracted candidates during step t that become previously unseen edges in the graph, and let V_t denote the number of visited states in that step. Since one crawl step corresponds to one visited state in this implementation, typically $V_t = 1$. The growth signal is defined as:

  $$\text{Growth}_t = \frac{E_t^{new}}{V_t}$$

  A high growth value indicates that the current area of the application still exposes many unexplored transitions, which suggests that further exploration nearby may still be productive.

- **Duplicate Candidates** measures the wasted extraction effort based on how many extracted candidates were already known relative to the total extracted candidates.

  Let C_t be the total number of extracted candidates in step t, and let D_t be the number of candidates that do not result in a new edge, for example because they were already known. The duplicate rate is defined as:

  $$\text{Dup}_t = \frac{D_t}{C_t}$$

  A high duplicate rate suggests that the crawler is repeatedly encountering the same navigation structure or equivalent actions, which can indicate diminishing returns in the current region.

- **URL diversity** is an estimate of how structurally diverse the discovered URLs are, rather than variations of the same pattern. Instead of comparing concrete URLs directly, the crawler maps each URL to a simplified template by normalising variable path segments and query values.

  Let U_t be the number of unique concrete URLs extracted in step t, and let T_t be the number of unique URL templates among them. The diversity signal is

defined as:

$$\mathrm{Div}_t = \frac{T_t}{\max(U_t, 1)}$$

A high value means that the crawler is seeing many structurally different endpoint patterns, whereas a low value means that it is mainly observing many instances of the same route shape.

- **DOM change rate** measures how often interactions cause meaningful client-side changes in the page structure. When the crawler encounters an interactive edge, such as events, forms, UI forms and iframes, it records a lightweight DOM fingerprint before and after this interaction. If they differ, it counts as a DOM change.

  Let M_t denote the number of interactions in step t that lead to a changed DOM fingerprint, and again, V_t denotes the number of visited states in that step. The DOM change rate is defined as

  $$\mathrm{DOM}_t = \frac{M_t}{V_t}.$$

  A high DOM change rate suggests that deeper client-side interactions are exposing new interface states or behaviours, which typically indicates that there are more things to explore.

- **Error rate** is a signal capturing how often the crawler triggers behaviour that could be considered problematic. It aggregates browser console errors, HTTP errors from performance logs and detected error traces in HTML. It works by counting errors over a step or window and then normalising it by the number of visited states.

  Let J_t be the number of browser console errors, H_t the number of HTTP responses with status code at least 400, and R_t the number of detected error-trace signatures in the returned HTML during step t, V_t denotes the number of visited states in that step. The error signal is defined as:

  $$\mathrm{Err}_t = \frac{J_t + H_t + R_t}{V_t}$$

  A high error rate could indicate that the current strategy is pushing the crawler into failing states with broken flows, invalid form submissions, or dead endpoints. This could suggest switching away from the current strategy as the current state is unstable, and instead achieve a more stable section to potentially explore more entry points.

- **Latency** is a measure of the time cost per step and is aggregated inside a window and then divided by the visited states in that window. When latency is high, deep interaction strategies can become expensive because each additional step in a sequence costs time. This could bias towards BFS because it tends to make cheaper progress by sampling more entry points rather than committing to long interaction sequences.

- **Novel endpoints** measures how often the crawler discovers new server-side request targets during exploration by tracking response URLs. Otherwise similar to URL diversity, it collects response URLs (instead of request URLs) and normalises them into templates. It counts how many templates are new compared to what has been seen before. High scores could indicate that the current strategy is still giving good results because it shows that exploration still reaches new resources.

- **URL similarity** measures how many times a particular URL has appeared. To not make this channel overly rigid, we use SimHashing (similarity hashing), where normalised URLs are hashed into a smaller domain to determine URL similarity [17]. High URL similarity suggests the crawler is in an already explored application space and encourages switching the current strategy.

- **Novel Code** is a channel that measures backend code coverage, meaning how often the crawler discovers new code during exploration. It analyses Xdebug log files [18] and compares them to what has been seen before. High novel code favours staying in the current strategy because it would indicate that we are still finding unexplored code.

These metrics are chosen because, except for novel code, they can all be observed in a black-box environment, are relatively cheap to compute, and have intuitive explanations why they can correlate with exploration progress. Novel code is also included, even though it may not be available for all web applications, because it is the clearest indicator of a new behaviour and therefore a useful channel.

4.3.2 Principal Component Analysis on feedback channels

Because the feedback channels previously mentioned in Section 4.3.1 are derived by manual analysis of observable behaviour within web applications and not through a formal process, they may carry overlapping or redundant information. To identify and eliminate these potentially redundant channels, we conduct 3 runs on **OsCommerce** [19], each spanning 8 hours. Each run has with a different navigation configuration: once with only DFS, once with only BFS, and once with the regular switching version. We then do a Principal Component Analysis (PCA) on the full

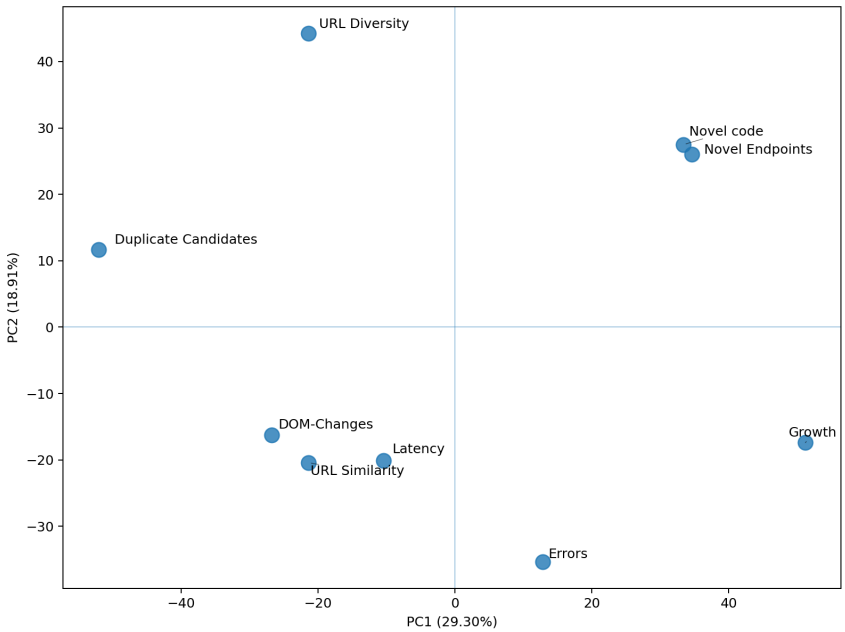


Figure 4: A 2-dimensional PCA plot with combined results between both BFS and DFS exclusive runs.

set of feedback channel data from these runs. PCA is a method that translates a dataset into a new coordinate system whose axes, called **Principal Components** (PCs), are ordered by how much variance each one explains [20]. Points that cluster together in the new space indicate a similar underlying signal and can therefore be treated as interchangeable.

Different channels operate on very different scales, for instance `latency` is a noisy continuous measurement, whereas `error rate` is a nearly discrete channel. To account for this variance and ensure PCA captures the correlation rather than the raw magnitude, each channel is standardised across all collected samples. We get a resulting score matrix, which can be plotted on a graph to identify similar signals.

The resulting PCA graph is based on the two principal components that account for the most variance, which are 29.30% and 18.91% each. Together, this accounts for a total of 48.21% of the total variance. This immediately reveals two clusters that need to be addressed.

Our channel selection proceeds in two steps:

1. **Remove either `novel endpoints` or `novel code`.**

   These two points occupy nearly identical positions in the plot, indicating

strongly correlated behaviour across all samples. Therefore, retaining both channels provides no additional information. Given that *novel code* is intuitively reliable across applications, as it directly measures coverage, we prioritise keeping it. But instead of completely removing *novel endpoints*, we use it as a fallback option when *novel code* cannot be tracked in a fully black-box setting.

2. **Remove `DOM-change`, `URL-similarity` and `latency`.**

   These three channels with *errors* form a tight cluster with each other in at least one of the plots (see Section A.1), indicating that they capture the same underlying signal, at the very least when in particular strategies. We therefore remove 3 of them to achieve proper separation in all graphs.

The remaining five channels, `growth`, `duplicate candidates`, `URL diversity`, `errors`, and `novel code`/`novel endpoints`, are the final feedback channels we used in the Maelstrom Crawler.

4.3.3 Ablation study

To ensure each remaining channel gives useful feedback, we do a series of ablation studies. As previously mentioned, when `novel code` cannot be tracked, the crawler has a fallback option where it exchanges the channel with `novel endpoints`. Therefore, we did a study to see if this is a necessity or not. Thereafter, we conduct a separate study for each of the channels, including either exclusively `novel code` or `novel endpoints`, where they are individually tested to see if they contribute to increased code coverage.

4.3.3.1 Novel- code and endpoints

We let our crawler run for 8 hours with our implementation tracking `novel code` during runtime (see Section 4.3.1), and then ran 8 hours with `novel endpoint` channel instead. We repeated this process 3 times for each tested web application.

Table 4.1: Lines of unique code executed during each run on OsCommerce.

OsCommerce	with Novel Code	with Novel Endpoints
Run 1	84 790	88 297
Run 2	89 603	88 934
Run 3	82 446	83 537
Average per run	**85 613**	**86 917**

Table 4.2: Lines of unique code executed during each run on WordPress.

WordPress	with Novel Code	with Novel Endpoints
Run 1	82 356	69 747
Run 2	85 045	81 332
Run 3	83 184	70 133
Average per run	**83 528**	**73 737**

`Novel code` either gives an improvement over `novel endpoints` in certain applications, such as WordPress (see Table 4.2) or has almost no significant performance difference, such as in Table A.1. We therefore apply the `novel endpoints` variant as a fallback option.

4.3.3.2 Remaining feedback channels

The remaining channels were excluded sequentially and then compared with a baseline in which all channels were included. `Novel code` and `novel endpoints` are excluded simultaneously but only one is indicated, see Table 4.3.

Table 4.3: Percentage change in average lines of total unique code executed relative to baseline for each system and ablation. Positive values indicate an increase over baseline; negative values indicate a decrease.

Web application	**¬Novel code**	**¬Growth**	**¬Dup-Cand**	**¬URL diversity**	**¬Error**
OsCommerce	+3.76%	+0.72%	+2.70%	+3.09%	+3.60%
WordPress	+0.60%	-4.30%	-14.10%	-2.22%	-6.83%
Kanboard	-15.27%	-0.58%	-15.30%	-14.62%	-0.29%

Table 4.4: Per crawler iteration performance, presenting how much novel code was discovered on average when each channel was excluded and a comparative baseline value, only after the shared Early-GETs stage was complete.

Web application	**Baseline**	**¬Novel code**	**¬Growth**	**¬Dup-Cand**	**¬URL diversity**	**¬Error**
OsCommerce	22.52	28.69	23.66	28.46	27.38	28.82
WordPress	42.42	32.49	35.96	28.67	39.11	46.79
Kanboard	1.51	0.86	1.15	1.02	0.50	1.69

For WordPress specifically, the deviations from the average of a particular subset of runs could be vast, limiting the unique lines of code to around 60000 (see Table A.2). This occurred in rare instances where the crawler identified and registered many edges within the CSS colour configuration settings that relied on the same code. These runs were discarded as statistical anomalies if a rerun didn't run into the

same problem; while this could occur on any of the runs, it only happened twice when run without the `Duplicate Candidates` feedback channel. Another type of anomaly happened when running without the `Error rate` feedback channel, where its Early-GETs section performed significantly worse, resulting in per-step unique code discovered when excluding the Early-GETs to increase when including this anomaly run, even within WordPress, see Table 4.4. When this run is excluded, the per crawler step unique code drops from 46.79 to 32.24 and then follows the same pattern as Table 4.3.

If we disregard OsCommerce, then removing any feedback channel shows a near-neutral impact on performance or a considerably negative one, with the smallest non-neutral impact being 4.30% negative impact. This gives a great incentive to retain all feedback channels. However, this pattern is reversed for OsCommerce, where all but `Growth` give a positive impact on performance when removed. This could indicate that OsCommerce incentivises more actions over effective actions. To investigate this possibility, we test a completely randomised switching strategy and present its results in Section 5.6.

However, we retain all feedback channels as the discovered negative impact on performance when tested on WordPress and Kanboard is larger than the relatively minor positive impact perceived on OsCommerce.

4.4 Parameter selection

The Maelstrom Crawler has key parameters that apply to cooldown, comparison-window and baseline-window. For these values to be universally applicable across different web applications, we conduct a window-size analysis to decide the window sizes and a parameter sweep on cooldown across different applications.

4.4.1 Window-size analysis

To determine the correct sizes of the baseline-window and comparison-window, we did a *window-size analysis*. We accomplish this by running our crawler in their single navigation strategy modes, where they are either in their DFS or BFS mode (see Section 4.5.1). Each run was conducted on OsCommerce and spanned 8 hours.

The comparison of the comparison-window of length C steps and baseline-window of length B steps is a central component for the strategy controller. The quality and reliability of the switching decisions are fundamentally dependent on the chosen values of B and C. If C is too small, the current window may contain insufficient observations to distinguish genuine change from regular per-step noise. If C instead is too large, the average value might be too diluted as the span is much larger than a typical stable interval. The value of B faces similar challenges, where it must be

long enough to represent a coherent history of values yet short enough that it does not dilute the history.

Neither the value of B nor C can be set reliably by intuition alone, because they depend on the properties of the feedback channels and their noise level. Therefore, we apply a principled and data-driven methodology to estimate these parameters directly from observed crawl data.

While the following sections will present the theoretical basis for each method used, all analyses were implemented and run with a Python script. This script accepts raw signal value data in a CSV format and then implements all three methods sequentially. By combining these methods, we aim to identify well-suited values for B and C by identifying statistically different sections using:

- *Sliding-window KS divergence*, which identifies the natural timescale at which the signal distribution changes.
- *PELT change-point detection*, which locates structural breaks in the signal and then yields estimates of stable segment lengths.
- *Maximum Mean Discrepancy (MMD)*, which provides a cross-validation of the results from the first two methods.

Together, these three methods bracket the values of B and C from both sides, which creates a principled and data-driven recommendation for their values. For each method, the script generates both numerical summaries and visual plots, which have been used to decide the final window parameters.

Method 1

A natural measure of how quickly the signal distribution changes is the statistical divergence between two adjacent, non-overlapping windows of equal width W. For each candidate window size W and each step $t \in [2W, N)$, the two adjacent windows are defined as:

$$A(t, W) = \{X_{t-2W}, ..., X_{t-W-1}\}$$

$$B(t, W) = \{X_{t-w}, ..., X_{t-1}\}$$

The distributional distance between A and B at step t is quantified using the two-sample Kolmogorov–Smirnov (KS) statistic [21–23], defined for two samples P and Q as:

$$\mathrm{KS}(P, Q) = \sup_x |F_P(x) - F_Q(x)|$$

where F_P and F_Q are the empirical cumulative distribution functions of P and Q respectively, and $\sup_x$ denotes the supremum over all values of x. Since the signal

consists of K channels, the statistic is applied independently to each channel and the results are then averaged to produce a single divergence measure:

$$D(t, W) = \frac{1}{K} \sum_{k=1}^{K} \mathrm{KS}\Big(A^{(k)}(t, W),\ B^{(k)}(t, W)\Big)$$

where K is the total number of active channels and $A^{(k)}$, $B^{(k)}$ denote the observations of channel k within windows A and B respectively.

The *divergence growth curve* is then defined as:

$$\bar{D}(W) = median_t D(t, W)$$

which is evaluated over the candidate set:

$$W \in \{3, 5, 7, 10, 15, 20, 30, 40, 50, 75, 100\}$$

As W increases from small values, $\bar{D}(W)$ initially grows. Once W gets larger and exceeds what is the typical *regime* length, $\bar{D}(W)$ will consequently plateau or decrease. The inflexion point of this curve, noted as W^*, corresponds to the *natural regime timescale*, meaning the window width at which adjacent windows most reliably detect a genuine change in signals. This inflexion point is identified computationally using the *Kneedle algorithm* [24].

Method 2

The second method used is PELT change-point detection. The idea is that this method identifies specific locations of signal change and estimates stable segment lengths.

Structural breaks in the multivariate channel signal are identified using the Pruned Exact Linear Time (PELT) algorithm [25]. Let $\{\tau_1, ..., \tau_m\}$ denote the detected change-points, and let $\ell_i = \tau_{i+1} - \tau_i$ denote the length of the i-th stable segment. Let $Q_p(\{\ell_i\})$ denote the p-th quantile of the empirical distribution of segment lengths. Then the following window selection is made:

- $\boldsymbol{Q_{0.25}(\{\ell_i\})}$ **:** Setting $C \leq Q_{0.25}(\{\ell_i\})$ will ensure that the comparison-window is narrower than at least 75% of all observed stable segments.

- $\boldsymbol{Q_{0.50}(\{\ell_i\})(Median)}$ **:** Setting $B \geq Q_{0.50}(\{\ell_i\})$ will ensure that the baseline-window is at least as wide as the typical (median) stable segment.

Method 3

The third method used to determine values for baseline-window and comparison-window is a **Maximum Mean Discrepancy for Joint Distributional Analysis**. **Maximum Mean Discrepancy (MMD)** provides complementary joint measures that can capture inter-channel dependencies. The squared MMD between two distributions P and Q in a *reproducing kernel Hilbert space* $\mathcal{H}$ with kernel $k(.,.)$ is defined as [26, 27]:

$$MMD^2(P,Q) = \mathbb{E}_{x,x\prime\sim P}[k(x,x\prime)] - 2\mathbb{E}_{x\sim P,y\sim Q}[k(x,y)] + \mathbb{E}_{y,y\prime\sim Q}[k(y,y\prime)]$$

With given finite samples as $X = \{x_1, ..., x_m\}$ from P and $Y = \{y_1, ..., y_n\}$ from Q. The empirical estimator is then as follows:

$$\widehat{MMD}^2(X,Y) = \frac{1}{m(m-1)}\sum_{i\neq j} k(x_i,x_j) - \frac{2}{mn}\sum_{i,j} k(x_i,y_j) + \frac{1}{n(n-1)}\sum_{i\neq j} k(y_i,y_j)$$

The kernel used in the process is the **RBF** (Gaussian) kernel $k(x,y) = exp(-||x-y||^2/2\sigma^2)$, with the bandwith σ set by the median heuristic: $\sigma = median\{||x_i - x_j|| : i \neq j\}$. This choice of a data-adaptive bandwidth helps the kernel appropriately scale to the observed inter-point distances.

MMD primarily serves as a cross-validation of the other methods and increases confidence in the other recommendations.

4.4.2 Parameter sweep on cooldown

To determine the optimal size of **cooldown**, we did a simple parameter sweep, testing in 10-step intervals between 0 and 100. As we are investigating the possibility of thrashing, which is immediately shown, we conducted shorter tests, limiting them to 4 hours on WordPress, an adequately large application.

Table 4.5: Cooldown values and associated switch-related information, such as mean time between switches, or full crawler steps between switches.

Cooldown value	Average time (s)	Median time (s)	Average steps	Median steps	Total switches
0	92.6	19.0	2.2	1	152
10	471.3	382.0	14.1	11	30
20	1021.9	828.0	25.4	24	14
30	1141.4	865.0	30.8	30	13
40	1120.8	804.5	40.6	40	13
50	1374.4	1361.0	55.8	52	10
60	1683.9	1207.5	62.2	60	9
70	1558.3	1565.0	73.7	73	10
80	2145.3	2007.0	87.0	84	7
90	2237.5	2197.5	93.2	92	7
100	1967.5	2076.5	101.8	100	7

4.5 Implementation

Implementing our method first required we implement the base DFS and BFS strategies. After ensuring a basic extension of Black Widow for both BFS and DFS strategies, we implemented a method for the strategies to communicate and switch between each other, all while retaining context in an effective way. After achieving a process for switching and communicating between strategies, we implemented the various feedback channels. Finally, we incorporated these feedback channels into automatically switching strategies and tested their performance against previous iterations of the scanner. This was to ensure the combination of strategies would achieve better performance than running each strategy in isolation before our evaluation.

4.5.1 Randomised DFS and BFS

Deciding on these base strategies was a time-consuming process as many variations of BFS and DFS were considered, but eventually we settled on having both base navigation strategies utilise randomisation. Each strategy has a different probability distribution over which action type to select next, for example:

- **GET:** a normal navigation request to a discovered URL
- **Fresh GET:** a GET request whose target URL has not been visited before.
- **Form:** a submission of an HTML form with extracted input fields and parameters.
- **UI form:** a form-like interaction built from interface elements that do not appear as a standard HTML form.
- **Event:** a client-side JavaScript interaction.
- **Iframe:** a transition into content embedded inside an iframe.

This allows the crawler to preserve the overall character of the strategy while still introducing enough variation to avoid repetitive traversal patterns.

A further refinement in the implementation is the notion of a fresh GET. Rather than treating all GET edges as equally valuable, the crawler explicitly distinguishes GET requests whose target URL has not yet been successfully visited from those that have been visited. This is implemented through dedicated queue-pop operations for BFS and DFS, which check the target URL of a candidate GET edge against a set of previously visited URLs. If the URL has already been visited successfully, that GET edge is skipped and marked as visited instead of being explored again.

4.5.2 Feedback Channels

The most time-consuming process was implementing the feedback channels, even though some were trivial. Some channels, such as *novel code*, required more work to ensure their functionality without also being prohibitively slow.

- **Growth** is implemented by recording how many previously unseen candidates from the current crawl step are inserted into the backlog. After the crawler loads a state, it extracts navigation and interaction candidates such as URLs, forms, events, iframes, and UI forms. For each candidate, the crawler attempts to construct and connect a corresponding edge in the graph. If this succeeds and the edge did not already exist, the per-step counter for `new_unique_edges` is increased.

- **Duplicate Candidates** is implemented by extracting candidates and comparing them to previously recorded ones. During each crawl step, every discovered candidate is first registered as an item, which the crawler attempts to add to the graph. If the corresponding edge already exists, the per-step duplicate counter is increased. This applies to **URLs**, **forms**, **events**, **iframes**, and **UI forms**.

- **URL diversity** is implemented by maintaining two per-step sets, one containing all unique concrete URLs encountered during the step, and one containing their corresponding normalised URL templates. Whenever a candidate with a URL is extracted, the crawler inserts the concrete URL into the first set and computes a simplified template for insertion into the second set. The template generation preserves scheme and host, while replacing variable-looking path components such as numeric identifiers, UUIDs, and long hexadecimal strings with placeholders, retaining only query keys rather than query values. The sizes of these two sets are then used to characterise how many structurally different URL patterns were seen relative to the number of distinct concrete URLs.

- **DOM change rate** is implemented by computing a lightweight fingerprint of the page DOM immediately before and immediately after an interaction-oriented crawl action. The fingerprint is constructed from the current page source after whitespace normalisation and truncation, and then hashed to obtain a compact representation of the DOM state. If the fingerprint after the action differed from the fingerprint before the action, the interaction was counted as having produced a DOM change.

- **Error rate** is implemented as an aggregate of several error-related signals col-

lected after each crawl step, recorded through built-in functions of the Google Chrome development drivers [28]. First, browser-side JavaScript errors are obtained from Selenium [29] through `driver.get_log("browser")`, where log entries marked as `SEVERE` or `ERROR` are counted. Second, HTTP-level failures are extracted from Chrome `performance` logs by parsing `Network.responseReceived` messages and counting responses whose status code is at least 400. Third, the crawler scans the returned HTML for error-trace signatures using a collection of regular expressions matching common stack traces, framework debug pages, and database error messages. The counts from these three sources are then accumulated into the per-step error counters and combined into a single error signal.

- **Latency** channel is implemented by measuring the clock time spent executing a single crawl step, such as a **GET** action.

- **Novel Endpoints** tracking is implemented by observing the browser's network activity via Selenium's Chrome `performance` logs. After the crawler follows an edge and a page finishes loading, the crawler calls `driver.get_log("performance")` and parses the emitted Chrome DevTools Protocol messages, filtering specifically for `Network.responseReceived` events. Each such event contains the URL of a resource whose HTTP response was actually received by the browser. For every observed **response URL**, we compute an *endpoint template* using a lightweight normalisation function: we preserve scheme and host, replace dynamic-looking path segments (UUIDs, long hex strings, and numeric IDs) with placeholders, and discard query *values* while retaining only the set of query *keys*. These templates are inserted into a global set of previously seen endpoint templates. The per-step novelty signal (`new_endpoints`, used by the `endp` channel) is then defined as the number of templates observed in the current step that were not already present in this global set.

- **URL similarity** is tracked by first normalising each URL by lowercasing the scheme and host, removing default ports, cleaning up slashes, sorting query parameters, and dropping volatile parameters such as session or token values. It then tokenises the normalised URL on non-alphanumeric characters and computes a 64-bit SimHash. Each token t gets weight $w(t) = \max(1, |t|)$, and its MD5 hash contributes to each bit position:

$$v_i \mathrel{+}= \begin{cases} +w(t) & \text{if } \mathrm{bit}_i(\mathrm{MD5}(t)) = 1, \\ -w(t) & \text{if } \mathrm{bit}_i(\mathrm{MD5}(t)) = 0. \end{cases}$$

  The final fingerprint is $f_i = \mathbf{1}[v_i > 0]$. This produces similar fingerprints for similar URLs (locality-sensitive hashing). The similarity between two URLs is then measured as the Hamming distance between their fingerprints.

- **Novel Code** tracking is implemented through enabling Xdebug, a tool that, when connected to PHP applications, makes it possible to get code coverage [18]. We first ensure that each web application does this by outputting these logs into small text files that we can track during run-time. We take each log file, read each line of code executed and place them in a set. As a set only records unique objects, we simply track its size, resulting in a live view of novel lines of code executed.

4.6 Evaluation

To evaluate the Maelstrom Crawler, we measure its performance on **OsCommerce** [19], **Kanboard** [30] and **WordPress** [31], 3 open-source web applications commonly used in prior research. We then compare our crawler's performance against **Black Widow**, as our crawler is built upon its source code, as well as **EvoCrawl** [16] and **ZAP** [32]. Performance is measured with two key metrics: server-side code coverage and the number of vulnerabilities detected.

We also test the individual strategies used within the Maelstrom Crawler against its final switching strategy, following the same metrics: server-side code coverage and the number of vulnerabilities detected.

Tests using a fully randomised switching mechanism for the Maelstrom Crawler are also be conducted and compared to runs using the regular Maelstrom Crawler. These tests intend to see whether a feedback-based switching mechanism is actually useful or if a fully randomised switching mechanism could perform just as well or even better.

5

Results

This chapter first presents the results of our tests to discover the optimal comparison-window, baseline-window and cooldown values. We then compare the performance of Maelstrom Crawler with Black Widow [6], ZAP [32] and EvoCrawl [16] on 3 open-source web applications, presenting the difference in code coverage and vulnerabilities found. Lastly, we present our internal tests comparing the base DFS and BFS navigation strategies of Maelstrom Crawler, with the switching version, as well as a strategy that randomly switches between the base strategies. Our results show that the Maelstrom Crawler outperforms the other crawlers on average. It gives around a 9.63% average improvement in code coverage compared to Black Widow and shows benefits from switching between navigation strategies throughout runtime. However, the Maelstrom Crawler gets outperformed by Black Widow in the number of vulnerabilities found, where Black Widow finds 2 vulnerabilities within Kanboard, while the Maelstrom Crawler finds none. It should be noted that these vulnerabilities were also found by the Maelstrom Crawler during our ablation studies. EvoCrawl finds the same number of vulnerabilities on OsCommerce and WordPress, while ZAP finds no vulnerabilities.

5.1 Anti-thrashing analysis

This section presents the results of the methods we used to derive values for the previously mentioned anti-thrashing mechanisms (see Section 4.2.2). This includes the final values for cooldown, comparison-window and baseline-window used throughout the rest of the evaluation.

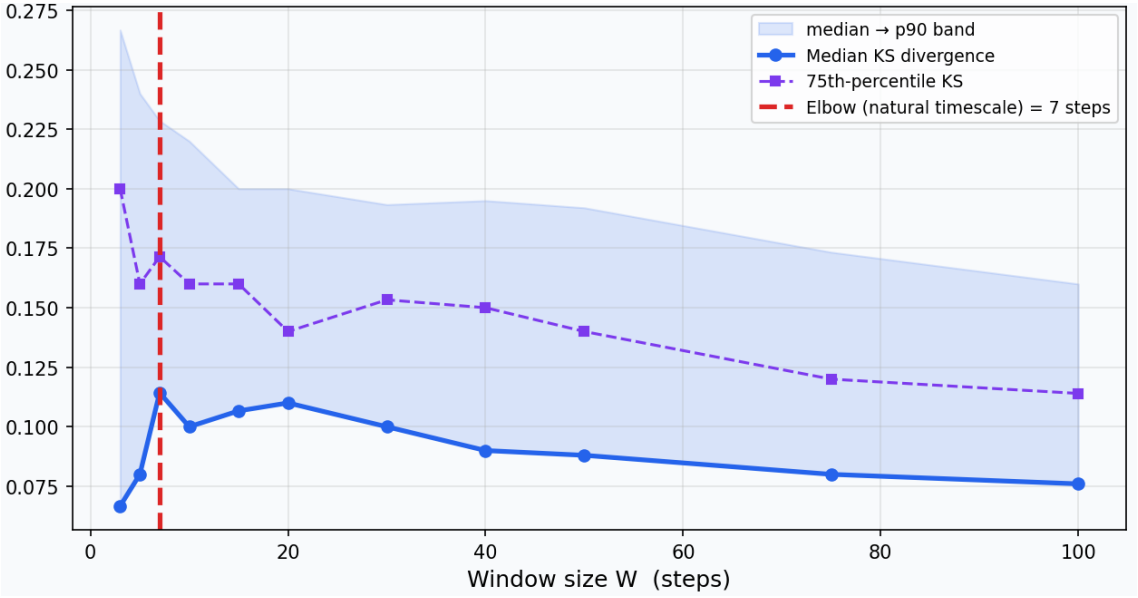


Figure 1: The red dashed line shows the elbow for the natural timescale based on the Divergence Growth Curve for the DFS run.

5.1.1 Window-size analysis

The analysis of window-sizes was conducted on the OsCommerce application with two independent runs using the Maelstrom Crawler. One run only used BFS traversal, while the other run used DFS traversal exclusively, which yields two different recommendations for both the baseline-window B and the comparison-window C. The results of both runs are presented below and are followed by the final parameter selection.

DFS run

The DFS crawl ran for 2491 crawler-loop steps across five active channels. The divergence growth curve for the DFS run is shown in Figure 1. The median KS divergence rises from 0.067 at $W = 3$ to a local peak of 0.114 at $W = 7$, and it then declines monotonically toward 0.076 at $W = 100$. The Kneedle algorithm then identifies the elbow point at $W^* = 7$ steps, which indicates that 7 steps is the natural regime timescale for the DFS traversal. This is further supported by the KS divergence timeseries. In Figure 2, when $W = 7$, the figure shows clearly structured variation with identifiable peaks.

The change-point detection identified 69 structural breaks across the signals in the DFS strategy; for further detail, see Section A.1.1. The resulting stable segment length distribution is shown in Table 5.1. It shows a 25th percentile of $Q_{0.25} = 11$ steps, a median of $\bar{\ell} = 27$ steps, and a maximum of 146 steps. This further confirms that the recommended comparison-window $C = 7$ lies below $Q_{0.25} = 11$. The recommended baseline-window $B = 30$ also just slightly exceeds the median segment

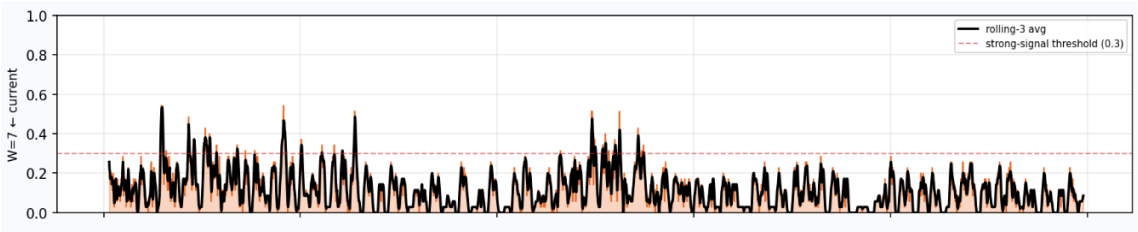


Figure 2: KS divergence timeseries for the DFS run at $W = 7$. Peaks above the strong-signal threshold (dashed red, $D = 0.3$) indicates genuine regime transitions rather than noise.

length of 27 steps, which ensures that it covers at least one full typical stable period.

Table 5.1: Stable segment length distribution for the DFS run, derived from the 69 PELT-detected change-points. $Q_{0.25} = 11$ steps defines the upper bound on C, and the median of 27 steps defines the lower bound on B.

Statistic	**Value (steps)**
Minimum segment length	1
$Q_{0.25}$ (upper bound on C)	11
Median $\tilde{\ell}$ (lower bound on B)	27
$Q_{0.75}$	55
Maximum segment length	146
Recommended C	7
Recommended B	30

The MMD^2 divergence surface for the DFS run in Figure 3 shows a pattern consistent with the KS analysis. High-divergence events in the red columns are localised and vertically coherent structures across multiple window-sizes. The signal is predominantly low-divergence (green) during the middle portion of the run, which is consistent with the longer stable segments identified by PELT in those regions. This further confirms that the KS-based elbow at $W = 7$ corresponds to genuine joint distributional change.

Applying the window recommendation procedure described in Section 4.4.1 yields the summarised values seen in Table 5.2.

BFS run

The BFS crawl ran for 1588 crawler-loop steps across the same five active channels. The divergence growth curve for this run is shown in Figure 4. Unlike the DFS run, the curve here decreases monotonically from $\bar{D}(3) = 0.200$ without an upward-rising phase, reaching a minimum near $W = 30$ before flattening. The Kneedle algorithm places the elbow at $W^* = 30$ steps, reflecting that the BFS strategy might change

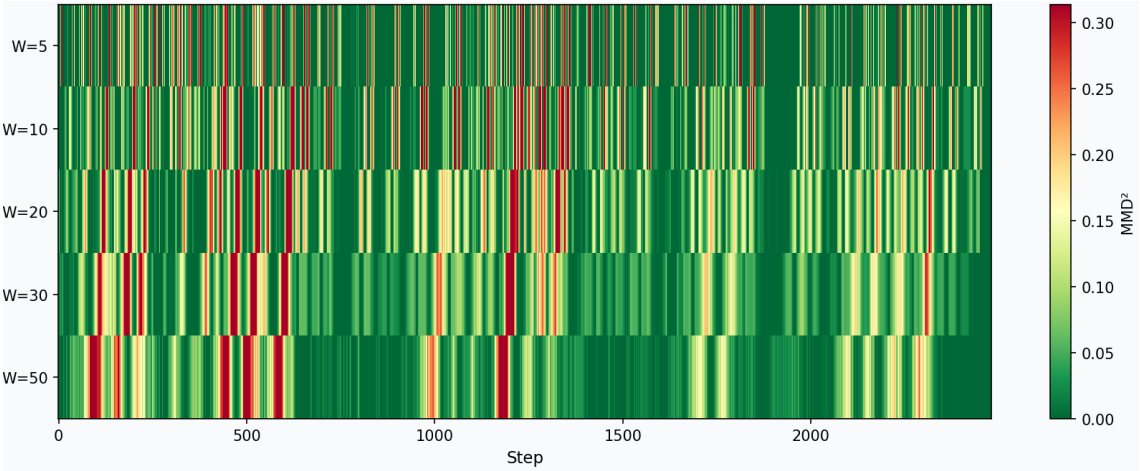


Figure 3: MMD^2 divergence surface for the DFS run. Each row corresponds to a window size W, and each column to a crawl step. Red indicates high divergence (regime change) while green indicates stability.

Table 5.2: Summarising the values from the DFS run.

Parameter	Value
Current window C	7 steps
Baseline window B	30 steps
Ratio B/C	4.3×
Divergence elbow	7 steps
$Q_{0.25}$ segment length	11 steps
Median segment length	27 steps

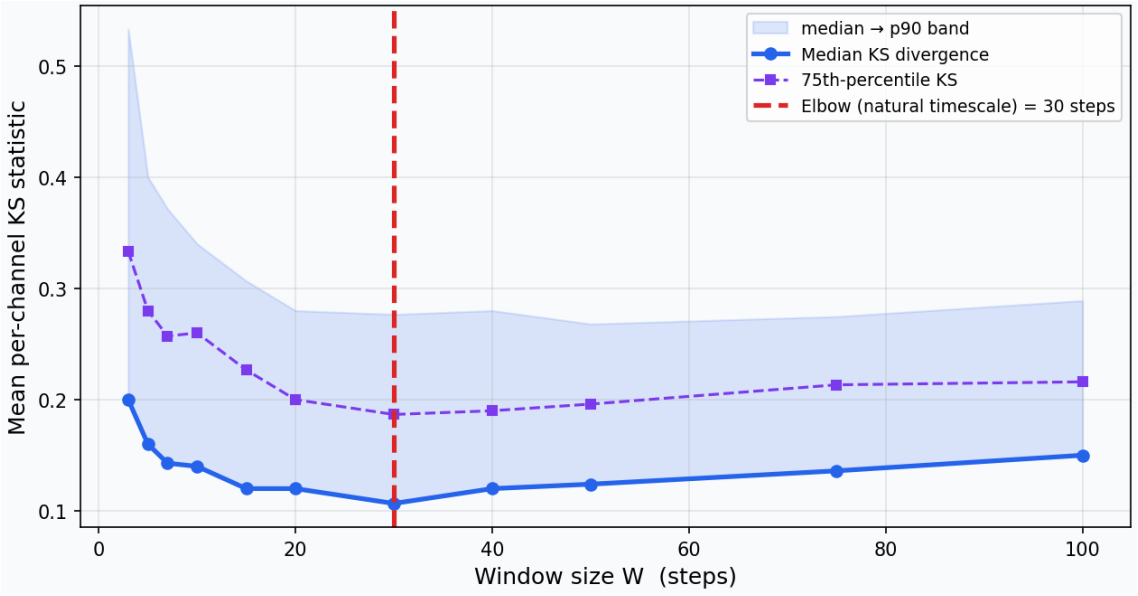


Figure 4: Red dashed line shows the elbow for the natural timescale based on the Divergence Growth Curve.

more slowly or gradually than the DFS strategy at short timescales.

For the DFS run, 52 structural breaks were detected; for further details see Section A.1.1. An inspection of the change-point overlay reveals that the majority of these breaks are concentrated within the first 400 steps of the run, and seem to stabilise after this. This structure could produce a skewed segment length distribution seen in Table 5.3. The majority of segments are very short with a median $\bar{\ell} = 5$ steps, and 25th percentile $Q_{0.25} = 3$ steps, and are driven by the dense breakpoints in the early phase. As a consequence, the segment-length statistics are dominated by the early-phase behaviour, which could limit their direct interpretability for window sizing.

Table 5.3: Stable segment length distribution for the BFS run, derived from the 52 PELT-detected change-points. The distribution is heavily right-skewed, driven by dense change-points in the early exploration phase.

Statistic	**Value (steps)**
Minimum segment length	1
$Q_{0.25}$ (upper bound on C)	3
Median $\tilde{\ell}$ (lower bound on B)	5
$Q_{0.75}$	8
Maximum segment length	880
Recommended C	3
Recommended B	20

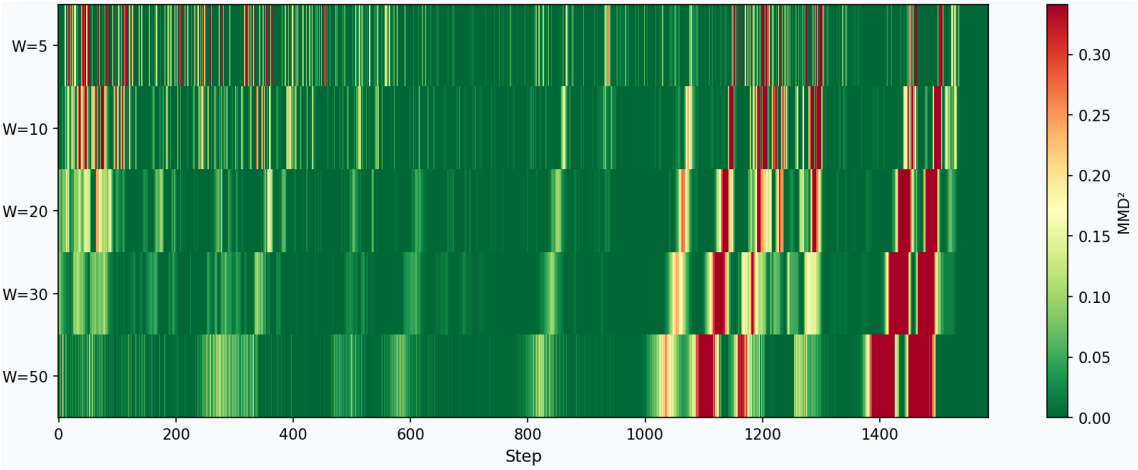


Figure 5: MMD^2 divergence surface for the BFS run. Each row corresponds to a window size W, and each column to a crawl step. Red indicates high divergence (regime change) and green indicates stability.

Table 5.4: Summarising the values from the BFS run.

Parameter	Value
Current window C	3 steps
Baseline window B	20 steps
Ratio B/C	6.7×
Divergence elbow	30 steps
$Q_{0.25}$ segment length	3 steps
Median segment length	5 steps

The BFS MMD^2 surface in Figure 5 reflects the structure of the BFS run clearly. The first 400 steps show broad and high-divergence activity across all window sizes. From step 400 onward, the surface seems to be mostly stable with a few exceptions at the end. In short, we get the recommendations from the BFS run seen in Table 5.4.

Comparison and final selection

The two runs yield divergent recommendations, summarised in Table 5.5. The disagreement between the two reflects actual differences in the strategy dynamics. The DFS strategy shows clear short-timescale regime transitions with a median stable segment of 27 steps, motivating a larger comparison-window. The BFS signal is dominated by a noisy early exploration phase that seems to produce very short segments and pulls the recommendation downward.

Since the adaptive Maelstrom Crawler operates in both BFS and DFS modes, the window parameters must perform reasonably under both traversal strategies. Selecting the DFS recommendations ($B = 30, C = 7$) would risk that over-smoothing occurs during BFS phases. At the same time, selecting the BFS recommendation ($B = 20, C = 3$) could result in the comparison-window being too narrow during the

Table 5.5: Window parameter recommendations from the DFS and BFS regime analyses.

	DFS run	BFS run
Divergence elbow W^*	7	30
Median segment length $\bar{\ell}$	27	5
$Q_{0.25}$ segment length	11	3
Recommended C	7	3
Recommended B	30	20
Grid search CV at recommended pair	0.134	0.053

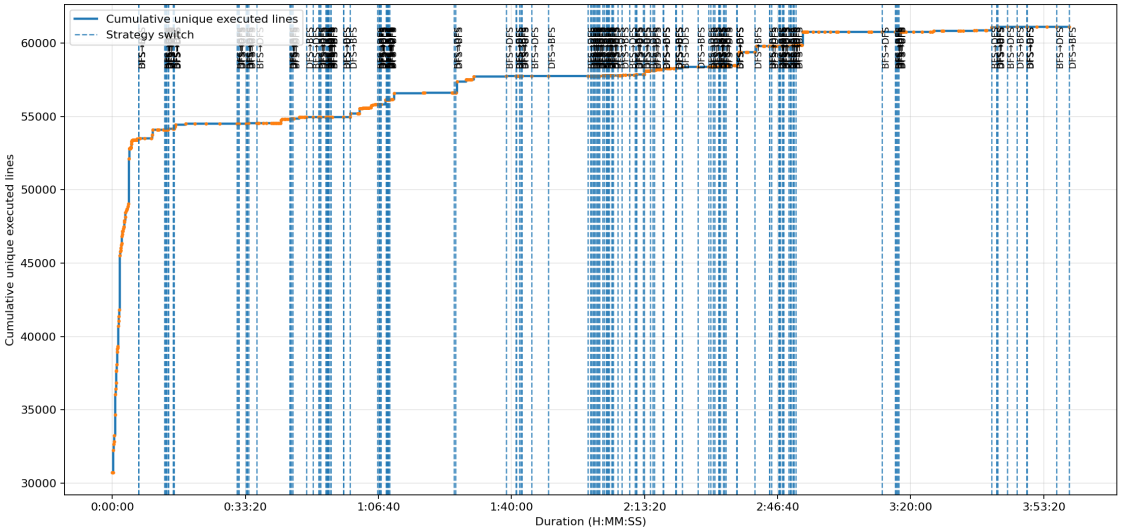


Figure 6: Code coverage and strategy switching between BFS and DFS visualised throughout runtime with cooldown set to 0 on WordPress.

DFS phase, making it more sensitive to per-step noise. A midpoint compromise of $C = 5$ and $B = 25$ is therefore adopted for the Maelstrom Crawler.

5.1.2 Cooldown

The results, shown in Table 4.5, reveal four distinct behavioural regions as the cooldown parameter increases. The first region consists of cooldown values 0 and 10. In this range, the crawler exhibits clear thrashing behaviour, rapidly switching targets while still discovering new code paths. This behaviour is especially pronounced for cooldown 0, resulting in 152 total switches and a median step value of 1 crawler steps between switches, see Figure 6. This behaviour quickly subsides, starting at the cooldown value of 20, see Figure 7. Beyond this initial unstable region, the remaining cooldown values naturally group into three clusters, see Table 5.6.

A clear inverse relationship can therefore be observed between cooldown length and switch frequency. Increasing the cooldown consistently reduces the number of switches performed during execution. However, the results also show that the crawler

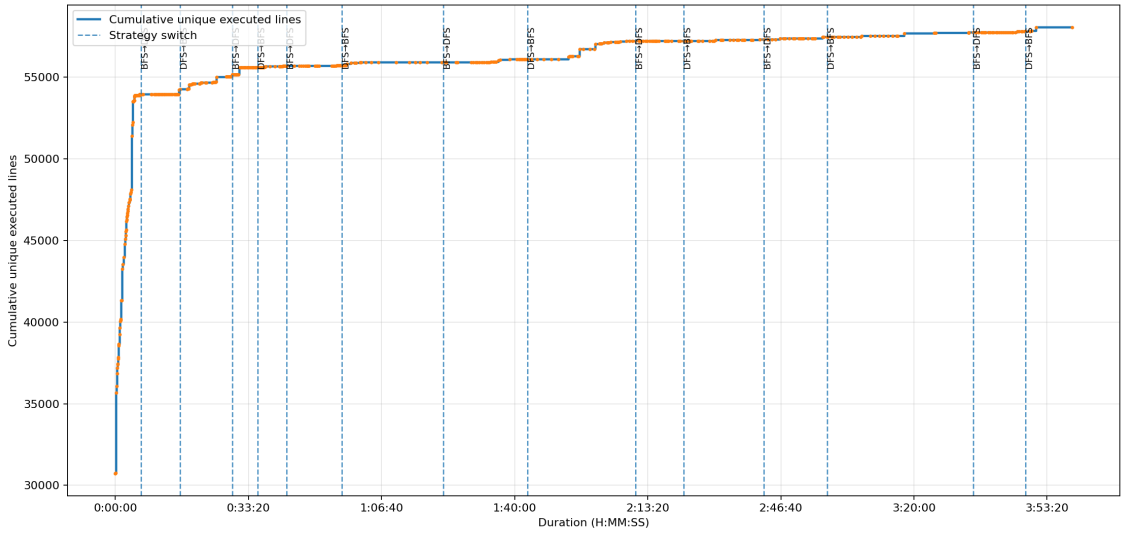


Figure 7: Code coverage and strategy switching visualised throughout runtime, with cooldown set to 20 on WordPress.

Table 5.6: Observed behavioural regions for different cooldown intervals.

Cooldown range	Switch range
20–40	13–14
50–70	9–10
80–100	7

adapts strongly to the configured cooldown value itself.

For nearly all tested cooldowns, both the average and median number of crawler steps between switches remain very close to the configured cooldown parameter. This indicates that the scheduler often delays switching until the cooldown requirement is fully satisfied, effectively turning the cooldown into a dominant control mechanism rather than a soft limitation.

To better evaluate this effect, Table 5.7 compares the deviation between the cooldown value and the observed average and median step counts. The low deviation percentages for most cooldown values demonstrate that switching behaviour becomes increasingly deterministic as the cooldown increases. In particular, cooldowns of 30 and above exhibit median deviations close to zero, meaning that switches on average occur immediately after the cooldown threshold has been reached.

This suggests that while high cooldown values reduce unnecessary switching effectively, they may also over-constrain exploration by preventing opportunistic target changes before the cooldown expires.

Cooldown	Average steps	Median steps	Average deviation (%)	Median deviation (%)
10	14.1	11	41.0	10.0
20	25.4	24	27.0	20.0
30	30.8	30	2.7	0.0
40	40.6	40	1.5	0.0
50	55.8	52	11.6	4.0
60	62.2	60	3.7	0.0
70	73.7	73	5.3	4.3
80	87.0	84	8.8	5.0
90	93.2	92	3.6	2.2
100	101.8	100	1.8	0.0

Table 5.7: The deviation between configured cooldown values and observed switch intervals.

The chosen metric for determining the cooldown value is chosen to be median deviation, as we want the cooldown to have as little impact as possible on the switching mechanism and primarily prevent thrashing. Here, the cooldown value of 20 is a clear winner with a deviation of 20% and is therefore chosen as the cooldown value in the Maelstrom Crawler. Coincidentally, its average step value is closest to the chosen comparison-window in Section 5.1.1 despite the tests being conducted on different web applications.

5.2 Coverage

When tested on all three web applications, Maelstrom Crawler yields 9.63% improvement on average, compared to the Black Widow crawler, and improves by 91.68% over ZAP and by 70.74% over EvoCrawl. However, the Maelstrom Crawler does not show an improvement over Black Widow and Evocrawl on OsCommerce. This is the only web application for which our crawler performs worse when compared to the other tested vulnerability crawlers. On OsCommerce, EvoCrawl performed 15.4% and Black Widow 4.58% better when compared with the Maelstrom Crawler. This gap is reduced when we compare the union of discovered code across three runs; here EvoCrawl shows a reduced 11.77% improvement, while Black Widow shows a 4.1% improvement (see Section A.1.3).

On the other two web applications, Maelstrom Crawler shows more promising results. On WordPress, Maelstrom Crawler interacts with 2.51% more unique lines of code than Black Widow, 19.92% compared to ZAP and 21.80% compared to EvoCrawl. On the Kanboard application, Maelstrom Crawler achieves 23.49% more unique lines of code, when compared against Black Widow, a 50.12% improvement compared to ZAP and a 66.41% improvement compared to EvoCrawl. An interesting observerion on Kanboard is that all three final runs from EvoCrawl had exactly 3 192 unique lines of code executed. This could indicate that EvoCrawl consistently converged to

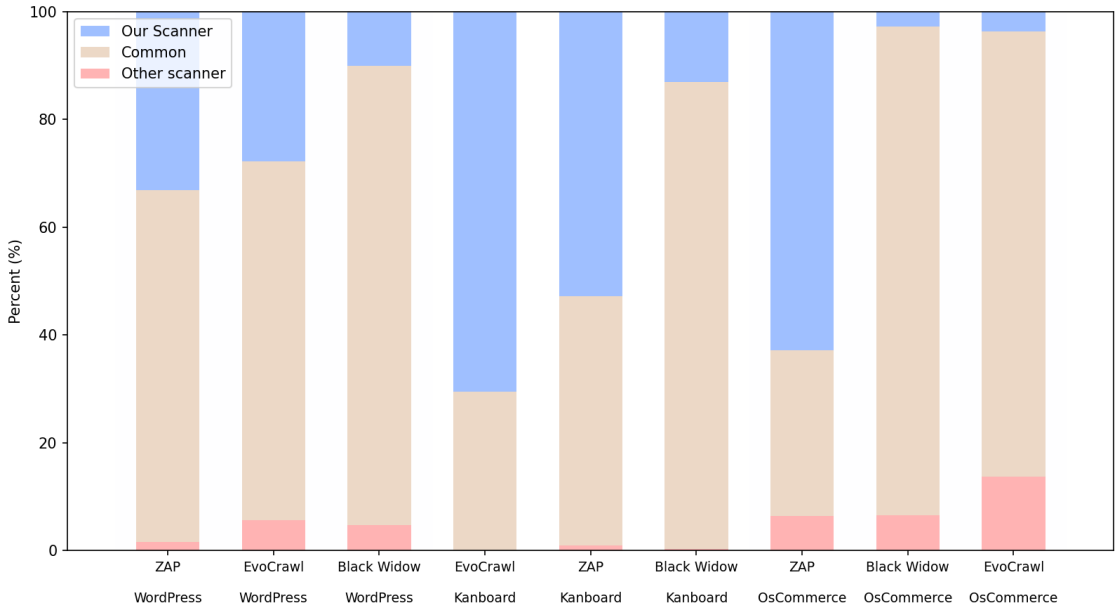


Figure 8: A visual representation of how much coverage each crawler is finding when compared to ours in WordPress, Kanboard, and OsCommerce

the same confined portion of the application and never managed to break out. While the underlying cause was not investigated further, this behaviour could have been a contributing factor to EvoCrawl's low coverage compared to the other crawlers on Kanboard specifically. These averages, however, do not always tell the whole story. When considering the union of code reached across three repeated runs for each crawler, Maelstrom remains the strongest performer on WordPress and Kanboard in terms of coverage (see Section A.1.3).

At the same time, if we inspect Figure 8, a visual representation can be seen of what code each crawler found compared to the Maelstrom Crawler on each application. This highlights the fact that there is code which the other crawlers reach that Maelstrom Crawler does not reach, and vice versa. Although this additional potential coverage is negligible on Kanboard, it remains visible on both WordPress and OsCommerce.

5.3 Vulnerabilities

The total number of verified XSS vulnerabilities that were discovered by each crawler across the three final runs is presented in Table 5.8. Overall, Black Widow found the highest number of verified vulnerabilities, with a total of 8, while Maelstrom Crawler and EvoCrawl each found 6. ZAP did not find any vulnerabilities in any of its three final runs.

On OsCommerce, Maelstrom Crawler, Black Widow and EvoCrawl all performed

equally by each finding a total of 5 unique and verified vulnerabilities. A similar pattern can be observed on WordPress, where Maelstrom, Black Widow and EvoCrawl each found 1 verified vulnerability. The main difference arises on Kanboard, where only Black Widow managed to find any vulnerabilities during its three final runs, by finding 2 unique verified vulnerabilities.

This means that, when considering the final three runs, Maelstrom Crawler and EvoCrawl match each other in total vulnerabilities found, but remain below Black Widow due to the results from Kanboard. At the same time, it is worth noting that the two Kanboard vulnerabilities found by Black Widow were also observed in earlier experimental runs and during the ablation study with Maelstrom Crawler. However, since these vulnerabilities were not rediscovered in any of Maelstrom Crawler's three final evaluation runs, they are not included in the results in Table 5.8.

Table 5.8: Total amount of verified XSS vulnerabilities each crawler found on each application.

	XSS vulnerabilities found			
Application	Maelstrom Crawler	Black Widow	ZAP	EvoCrawl
OsCommerce	5	5	0	5
WordPress	1	1	0	1
Kanboard	0	2	0	0
Total	**6**	**8**	**0**	**6**

5.4 Coverage per request

A request is identified by any HTML request that interacts with the PHP code, resulting in Xdebug getting invoked. This metric shows how efficient the actions of the crawlers are. When presenting the performance based on the number of requests, the best run from each crawler in terms of coverage is selected. It should be noted that the recorded requests are specifically the server-side requests, which are not equal to client-side requests, as a single client-side request could invoke multiple server-side actions and thereby requests.

Despite the fact that Maelstrom Crawler does not achieve the highest final coverage on OsCommerce, the request-based analysis shows that it is the most efficient crawler of the tested ones in terms of coverage gained per request. As seen in Figure 9, Black Widow and EvoCrawl eventually overtake Maelstrom Crawler in total unique executed lines. However, this additional coverage is obtained only after a substantial number of additional requests. Maelstrom reaches its final coverage after 3 657 requests for the 8-hour-long run, meanwhile Black Widow and EvoCrawl have 9 363 and 24 509 requests respectively after the same duration. ZAP is considerably

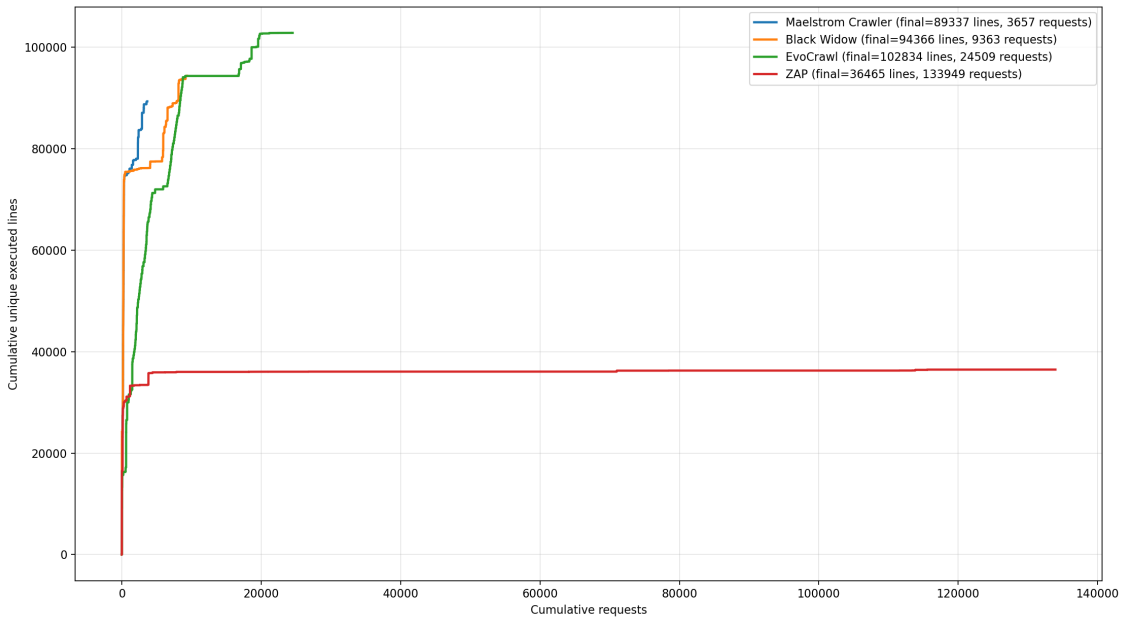


Figure 9: Each crawler's coverage per request from their best performing run on OsCommerce.

less efficient, with by far the worst performance in coverage despite having many more requests. This indicates that, while Maelstrom Crawler has slightly less final coverage on OsCommerce, it gets more new code for each request it makes compared to the other crawlers.

This pattern is more evident in Figure 10, where the data is truncated at the number of requests used by Maelstrom Crawler. Under this constraint, Maelstrom Crawler gets the highest coverage. These results show that Maelstrom's primary strength on OsCommerce is with the quality of its interactions rather than the volume of them. The same figure also clearly shows how Maelstrom Crawler and Black Widow share a similar development in terms of coverage in the first third part of the run, and then deviate from each other. This shows how Maelstrom Crawler is based on the Black Widow Crawler, especially for its early navigation.

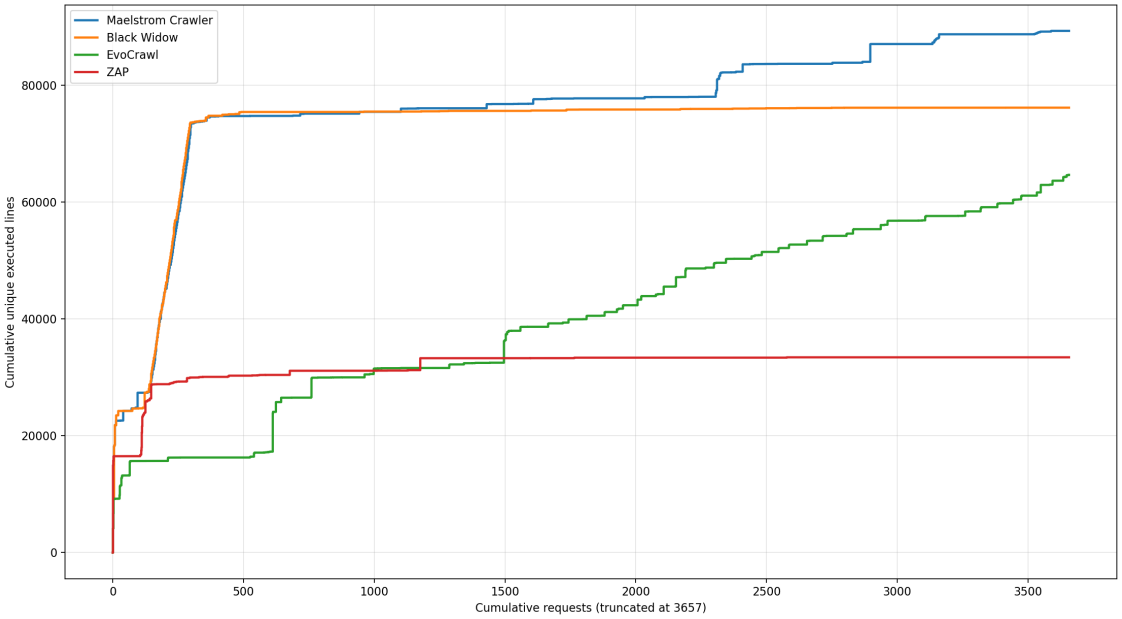


Figure 10: Each crawler's coverage per request from their best performing run on OsCommerce, but truncated at the number of requests from Maelstrom Crawler.

5.5 Comparing single strategies and switching

When we compare DFS and BFS exclusive runs to the Maelstrom Crawler's switching strategy, we see clear improvements from switching. Despite only performing three 8-hour runs on WordPress, there's a code coverage improvement of 12.0% for BFS exclusive and 15.4% for DFS exclusive (see Table 5.9 for details).

Table 5.9: Detailed code coverage results of 3 WordPress runs, comparing BFS exclusive and DFS exclusive runs, to the standard Maelstrom Crawler switching strategy.

WordPress

Run	Switching	BFS exclusive	DFS exclusive
1	82 157	66 213	68 717
2	84 125	77 119	74 622
3	70 610	68 204	61 884
Average	**78 964**	**70 512**	**68 401**
ΔBaseline	0%	-10.7%	-13.4%
∩ code	63 382	61 808	60 744
∪ code	93 641	81 107	77 756

The base DFS and BFS strategies, when compared to one another, find plenty of code not discovered by the other; BFS finds 12 750, and DFS finds 4 682 lines of code, respectively, which the other does not. The union of code between all runs

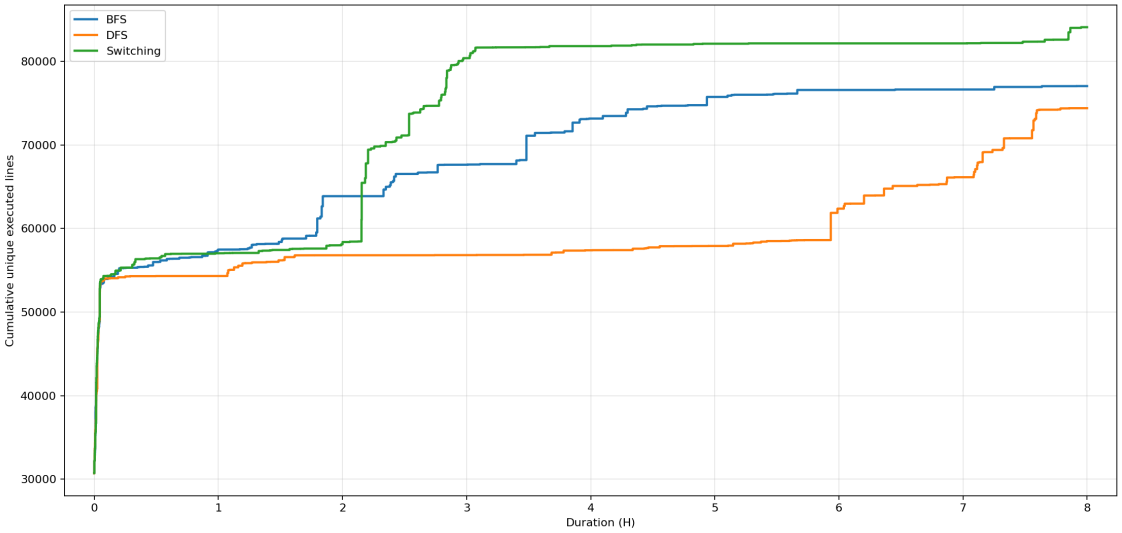


Figure 11: Timeline graph of novel code executed in WordPress for BFS, DFS, and switching strategies.

for BFS and DFS is 82 841, more than 10 000 less than the union of the switching strategy. This indicates that combining the base DFS and BFS strategies into the singular switching strategy makes them better than the sum of their parts, helping remedy their individual weaknesses. If we inspect Figure 11, where we visualise the best singular run for each strategy, we see how DFS often has hours of plateaus but also has strong spikes of code, almost catching up with BFS in the last 2 hours of the crawl. The Maelstrom Crawler's default switching strategy already surpasses both base strategies within 3 hours. It should also be noted that the selected BFS and DFS runs find at least 6 000 more lines of code than they do on average, meaning the average run performs significantly worse than visualised.

5.6 Randomised switching

Tests using the Maelstrom Crawler with a fully randomised switching mechanism were conducted and compared to the regular Maelstrom runs. We ran three 8-hour runs on Kanboard and WordPress. The randomised switching mechanism was achieved by having a 4% chance of switching strategy on each new iteration step; 4% was chosen as it would achieve a switch on average every 25 steps, which we found optimal during our parameter sweep (see Section 5.1.2).

On Kanboard, the randomised switching version produced 7 113 unique lines of code on average across three runs. This is 25.14% worse in terms of coverage than the regular Maelstrom Crawler, which produced 9 502 unique lines of code across three runs. When conducted on WordPress, a comparatively large application, the randomised switching achieves an average of 71 081 unique lines of code, a 9.98% decrease in performance from the 78 964 unique lines of code from the regular Maelstrom Crawler. On OsCommerce, the randomised switching version reached, on

average, 86 100 unique lines of code across three runs, which is 3.74% lower than the 89 101 achieved by the Maelstrom crawler.

6

Discussion

There are many aspects of this thesis that could be taken into consideration in this section, given the scope of our method and the results of our evaluation. Firstly, we did not explore any non-XSS classes of vulnerabilities or other modules in the scanner, such as payload generation. Varying these other modules could improve or worsen the resulting code coverage. This, accompanied by small sample sizes, shows a need for further testing on the proposed feedback channels and their effects. Despite these shortcomings, a pattern supporting the combining of navigation strategies through a method of guided switching appears, but how exactly this should be done, which feedback channels are optimal, and if they should be weighted more granularly can be further explored.

6.1 Limitations

Given the time frame, we prioritised testing on applications in varying sizes while attempting to keep the total number of them small; despite this, we only had adequate time to do 3 runs for each permutation. This necessitates further research, not only on the same applications but also on entirely different ones, to adequately assess whether switching strategies improve code coverage across applications because of the injected randomness, or if smart switching is a method to combine synergistic crawler strategies for the observed improvements. A limited number of web applications and runs is a compounding issue throughout the evaluation; to remedy this, we performed other methods such as window-size analysis (see Section 4.4.1) or extracted more context clues such as the union of discovered code (see Section A.1.3) when possible.

While we attempted to include applications with a diverse set of known vulnerabilities, structures and sizes, the results may not generalise to all web applications as only 3

were tested. The reproducibility of our evaluation is also affected by several different non-deterministic elements that exist in web applications, including dynamic content rendering, asynchronous events, and timing-related events, not to mention the inherent randomness in the crawler itself. Repeated tests during our evaluations helped mitigate this, but some variability in the results is still prevalent. As we were limited in time, we were forced to do a few thorough tests on a only these web applications, which would reduce variability but limit the breadth of our results. Alternatively, we could test many web applications with a shorter testing window, giving us a high variety of web applications, and possibly showcasing how it performs on different types of applications, such as forums, file storage, and more. However, we chose longer tests on a select few applications, WordPress, OsCommerce, and Kanboard, believing it better explores the potential benefits of switching rather than the overall effectiveness of the Maelstrom Crawler.

6.2 Feedback channels

The method we used to derive our candidate feedback channels does not consider all possible feedback channels. As they were mostly derived from intuition and easily accessible logs from Selenium, there could be other promising candidates we have not evaluated. If we had found the optimal combination of all available feedback channels, then the Maelstrom Crawler could not only be more effective per crawler iteration but could also benefit from excluding certain feedback channels that take more computational power, such as `Novel code`. This could explain the inconsistency within our ablation study, where OsCommerce does not improve with any feedback channel. Alternatively, OsCommerce could be such a large application that the benefits from the feedback-driven switching have not appeared yet, meaning we might have had an optimal configuration (or at the very least a beneficial one), but 8 hours wasn't enough to adequately assess if this is true. But it is probable that a beneficial configuration of feedback channels exists, such that we outperform Black Widow even within the 8-hour mark. This configuration could have appeared if we had conducted our PCA and window-size analysis on other web applications or if identified more feedback channels to increase our options.

6.2.1 Error channel implementation

We chose to implement an error-rate channel for switching feedback to avoid navigational instability from error-prone regions. Intuitively, when a high concentration of browser errors and failing HTTP responses occurs, it indicates that the crawler is repeatedly entering states that are broken, dead or for other reasons hard to progress from. From a navigational perspective, if the current strategy repeatedly leads to failing states that potentially prohibit increased coverage, then switching away from that strategy to help the crawler reach more productive regions seems like a good thing. On the other hand, this signal might not be unambiguously positive from a security-testing perspective. Error-prone areas could seem like a good place to expose vulnerabilities. Consequently, treating a high error rate as a reason to move away from a region should result in a bias toward stable and well-behaved parts of

the application, while reducing time spent in areas that could be interesting from a security perspective.

In other words, using the error-rate channel might improve navigational efficiency while at the same time potentially reducing the crawler's focus on areas that might be promising from an attacker's perspective. This creates a tension between two objectives, maximising exploration and, at the same time, finding as many unique vulnerabilities as possible. A high error rate region does not necessarily mean that it is an unproductive region from a coverage perspective, but rather implies the region may be unstable. Overall, the error-rate channel can be seen as a trade-off that helps the crawler avoid becoming trapped in failing flows and also helps improve overall coverage efficiency.

6.2.2 Novel code

There are two main things to consider connected to the novel code feedback channel, firstly we implemented a runtime tracking by running a process in parallel, which would be a non-issue if we assume there would exist a lightweight way to implement it in the future. However, the current implementation would not scale as well as implicitly assumed, as it currently occupies a processor in parallel. Secondly, applications implemented across different programming languages would require different methods to track executed code in a lightweight manner

6.3 Anti-thrashing

The limited timeline made it necessary to use fixed values for our window sizes and cooldown. The current values inherently bias performance towards the application they were determined on, as their values are based on gathered data from DFS and BFS exclusive runs on a particular application. Therefore, a better solution should include a method that either determines these values throughout the Early-GETs phase and possibly updates them throughout runtime, or alternatively, one could rerun the same analysis done as presented in Section 4.4.1 and Section 4.4.2 for each application, and then add more specific hard-coded values. While both methods are viable, we recommend exploring a dynamic method (see Section 6.8).

6.3.1 Window sizes

The selection of window sizes for the Maelstrom Crawler revealed minor conflicts between the two navigation strategies. The DFS-only and BFS-only analyses yielded slightly different results; the DFS run showed a preference for slightly longer intervals than the BFS run. This also indicates that these parameter values are not fixed even within a single application, but also vary between different navigation strategies. The final choice of $C = 5$ and $B = 25$ should therefore be seen more as a compromise than the perfect choice for both navigation strategies. At the same time, this result could also suggest that these window sizes are application-dependent as well. Values

that are calibrated for OsCommerce may not transfer optimally to WordPress or Kanboard. A promising direction for future work could be to make the window sizes adaptive, either by conditioning them on the current strategy or by recalibrating them during the crawling progress to fit the current application.

6.3.2 Cooldown

A notable result from Table 4.5 is that the chosen value of 20 has the average steps value of 25.4, or approximately 25, which is the same value as the baseline-window. These two values were derived from different applications, which could indicate two primary possibilities: applications around the same size have a preference for similar values for window-sizes and cooldown values and therefore want cooldown values slightly smaller than baseline-window. Alternatively, there's an optimal value for cooldown and thereby window sizes that generalise across combinations of feedback channels.

6.4 Coverage comparison

The comparison of the coverage shows several interesting characteristics of both the Maelstrom Crawler and the evaluated applications. One observation is that the crawler seems to perform differently depending on the size and complexity of the target application. On Kanboard, Maelstrom Crawler consistently outperforms the other tested Crawlers, and the code that is uniquely discovered by the others is negligible. This could suggest that Kanboard may be approaching a completed crawl and that the 8-hour long tests conducted in the evaluation are enough for Maelstrom Crawler to benefit from its dynamic strategy.

The situation is different for WordPress and especially on OsCommerce. These are substantially larger and more complex applications that leave possibilities for more interaction paths. The fact that Maelstrom Crawler continues to discover code that the others do not reach and vice versa suggests that none of the tested crawlers fully explores the application within the 8-hour time limitation. One possible explanation is that the benefits of Maelstrom's feedback-driven switching mechanism become more apparent as exploration progresses. During the early stages of a crawl, particularly on large applications, it is intuitive to assume that many of the interactions will yield previously unseen code regardless of which exploration strategy is active. In such situations, the advantage of dynamically selecting between strategies may be limited, and perhaps have the inverse effect, because any reasonable action will continue to discover new behaviour. As the crawl progresses and the application becomes increasingly explored, the remaining undiscovered functionality becomes harder to reach. It is in these later stages that the feedback-driven approach could potentially provide greater benefits by helping the crawler identify promising and unexplored regions.

6.4.1 Fully randomised switching

This interpretation is further supported by the randomised switching experiments. The performance difference between Maelstrom Crawler and the randomised variant decreases as the application size increases, with the same time constraints. On Kanboard, randomised switching performed 25.14% worse than the regular Maelstrom Crawler, which indicates that feedback-driven switching provides benefit in that scenario. On WordPress, the gap decreases to 9.98%, while on OsCommerce it decreases even further to only 3.74%. One explanation is that the larger applications contain such a large amount of unexplored functionality that even a random strategy-switching mechanism is able to continue discovering new code throughout most of the experiment.

Consequently, the advantage of a feedback-driven switching mechanism may not have enough time to fully give benefits within the 8-hour time constraint. Given longer runtimes, it is possible that the difference between the regular Maelstrom Crawler and the randomised variant would continue to grow as the amount of easily reachable code decreases and the importance of navigation decisions increases, further strengthening the central argument within Section 6.4. Investigating this potential impact on performance would require longer experiments and is left for future work.

6.4.2 Vulnerabilities

The results from the discovered vulnerabilities present a different picture than the coverage results. While Maelstrom Crawler generally achieves higher coverage than the competing approaches, it is hard to see any improvement in discovered vulnerabilities. This highlights an important difference between coverage and vulnerability discovery. Reaching more code is valuable because it increases the opportunity to also discover more vulnerabilities, but all code may not be equally likely to contain vulnerabilities. This indicates that more coverage does not necessarily translate to proportional improvements in finding vulnerabilities. An important consideration is that the vulnerability-finding capabilities are heavily dependent on the attacks that the crawler performs during the exploration. Since Maelstrom Crawler builds upon the Black Widow code, both crawlers share the same underlying attack mechanisms and vulnerability checks. Therefore, the primary contribution of Maelstrom lies in improving navigation rather than improving the attack techniques.

The limited number of vulnerabilities discovered across the evaluated applications may also indicate limitations in the attack mechanisms themselves. Black Widow's vulnerability-detection techniques were developed several years ago, which means it is highly likely that there are parts of it that are outdated. Additionally, it is not unlikely that some of the applications may have patched those kinds of vulnerabilities. This also means that there may exist vulnerabilities within the applications that remain undiscovered simply because the implemented attack functionalities are not capable of triggering them. Furthermore, some vulnerabilities may only be reachable

through very specific sequences of actions, which makes them difficult to trigger even if the affected code is reached. The Kanboard results show another important aspect of evaluating vulnerability discovery. Despite that, Black Widow found two vulnerabilities that were not found by Maelstrom Crawler during the final evaluation runs; both vulnerabilities had been identified by Maelstrom Crawler during earlier tests and experiments. This suggests that the difference is likely because of the inherent non-determinism of web crawling, rather than a systematic inability of Maelstrom Crawler to detect those vulnerabilities. This further highlights that vulnerability discovery often can depend on a specific combination of navigation choices and actions in sequence, meaning that individual runs do not always produce identical findings.

6.5 Per-request results

The per-request coverage results across the tested crawlers highlight that total coverage and exploration efficiency are two very different things. On OsCommerce, the Maelstrom Crawler does not obtain the highest final coverage, since both Black Widow and EvoCrawl eventually reach more unique executed lines on this specific application. When coverage is instead evaluated relative to the number of requests required to obtain it, a different picture emerges. In short, Maelstrom Crawler reaches a large fraction of the reachable code base with fewer requests than the competing crawlers. This suggests that the interaction choices that Maelstrom Crawler makes are, on average, more productive because each request made by Maelstrom Crawler is more likely to drive execution into unexplored code compared to the other tested crawlers. Meaning that Maelstrom Crawler sacrifices some long-run exhaustiveness in exchange for a more selective exploration.

An interesting implication of this result is that request efficiency should not automatically be interpreted as time efficiency. Since Maelstrom Crawler appears to spend longer between requests, it is fair to assume that its advantage comes from additional processing and careful selection before each action. This raises an interesting question of how much hardware performance influences the comparison. If the main bottleneck lies within the local computation being performed by the crawler, then stronger hardware (or longer crawling runs) would benefit Maelstrom Crawler more than other crawlers. If, however, the biggest bottleneck lies in browser execution, rendering or network latency, then better hardware might not give any notable effect on the relative performance compared to other crawlers.

6.6 BFS and DFS exclusive

When it comes to combining strategies, a relatively unexplored area, it is difficult to find appropriate strategies to combine. With the belief that strategies have to either be substantially different and/or be synergistic, we choose to develop our crawler around BFS and DFS exclusively. These strategies are also more more likely to

exhibit BFS and DFS qualities, for instance the Maelstrom Crawler's BFS strategy has a stronger tendency to navigate away from its current area of the web application through its increased likelihood of actions that either retain or decrease its "depth", especially when compared to a more traditional implementation like Black Widow. Vice versa, the DFS has a stronger likelihood of navigating further within the same area of the target application. This adaptation to the randomised BFS strategy can explain why our randomised BFS strategy performs worse than Black Widow but better when combined with randomised DFS. Essentially, we can rely on each strategy to be more specialised because the other will compensate where the other lacks. But while we present evidence in Sections 5.5 and 5.6 that combining these strategies shows a performance gain, in a similar vein as Cazzaro et al [13], adapting different strategies or parameters within them is still an unexplored area and would require further experimentation.

6.7 Ethical considerations

The project involves discovering security vulnerabilities, which naturally raises important ethical concerns. We will manage our research and will disclose our findings responsibly. We will only describe vulnerabilities that don't endanger active users, and these will instead be reported privately, and details about them will be kept anonymous.

All our tests were conducted on applications we had permission to scan. They were run on local machines to ensure that no unwanted interactions with external systems or users occurred.

6.8 Further research

While the Maelstrom Crawler demonstrates that using feedback-driven strategy switching can improve exploration efficiency and coverage, there is still room for future research.

One natural extension would be to investigate the potential use of additional exploration strategies and switching mechanisms. The current implementation only switches between BFS and DFS, but other complementary strategies could potentially provide further improvements. Furthermore, the concept of feedback-driven adaptation could also be extended beyond just within navigation. A similar mechanism could be used to dynamically select which attack payloads or vulnerability-testing techniques to prioritise during a crawling run.

Another area for future investigation involves the relationship between hardware performance and crawler effectiveness. The idea is that different crawlers may benefit

differently from increased computational resources. For example, crawlers that spend more time analysing collected information or making navigation decisions may scale differently than crawlers that prioritise issuing requests as quickly as possible. Evaluating crawler performances across different hardware configurations could help provide a better understanding of how exploration efficiency and coverage are affected.

The results also indicate that runtime duration may influence the performance of the crawlers. Several observations suggest that none of the tested crawlers fully explored the larger applications within the 8-hour evaluation time. Future work could therefore involve performing substantially longer experiments to help determine when individual crawlers begin to plateau. It could also be a way to see if the relative performance differences observed in the results persist over longer time periods.

The design of the feedback channels themselves presents another promising research direction. While the selected channels were motivated by observable crawler behaviour and mostly produced positive results, it remains unclear whether they represent the most informative signals for guiding exploration. Alternative metrics could be investigated, and it is possible that other signals would provide stronger guidance for exploration or be more suitable for specific types of applications.

In particular, the error-rate channel could be interesting to investigate further. Despite the fact that errors can indicate that a crawler is entering unstable or unproductive regions of an application, they may also be associated with behaviour that exposes vulnerabilities. By discouraging exploration in error-prone areas, the crawler could potentially improve coverage efficiency while simultaneously reducing opportunities to trigger security flaws. Future studies could therefore investigate whether certain feedback channels introduce trade-offs between exploration effectiveness and vulnerability discovery.

The current implementation also relies on globally selected parameters for the baseline-window, comparison-window and cooldown. While these values performed well overall, different applications may have different exploration characteristics. Future work could investigate application-specific parameter tuning, where the values for these parameters are selected based on observed behaviour instead. An even more ambitious extension would be to make these parameters adaptive rather than static. Instead of relying on fixed values, the crawler could dynamically adjust them during execution based on observations. Adaptive mechanisms could allow the crawler to respond more effectively to changing application characteristics and also further improve the performance.

7

Conclusion

This thesis set out to explore whether a crawler that dynamically switches between navigation strategies with the help of feedback channels can outperform a crawler with only one navigation strategy in terms of code coverage and found (XSS) vulnerabilities.

To answer this we extended the open-source web vulnerability crawler, Black Widow and created our own Maelstrom Crawler. The Maelstrom Crawler dynamically switches between randomised BFS and DFS navigation strategies by getting feedback through 5 channels: growth, duplicate candidates, URL diversity, error rate, and novel code. These feedback channels each cast a vote to suggest switching if their associated signal does not identify progress throughout the crawling process, and if a majority of channels agree that switching of strategies occurs. These channels were narrowed down from an original 9 candidates to the finalised 5 through first conducting a principal component analysis to determine similar underlying signals, with a subsequent ablation study to determine their benefit.

To eliminate thrashing within the crawler we implemented a baseline-window and comparison-window which store recent signal values. The average of these values is compared in each feedback channel to determine a vote; the window values were derived through window-size analysis. A cooldown value also forces the crawler to stay for a fixed number of steps within the recently switched to strategy, eliminates the issue of thrashing. The cooldown value was determined through a parameter sweep.

When evaluated across OsCommerce, WordPress, and Kanboard, Maelstrom Crawler

achieved a 9.63% average code coverage improvement over Black Widow, 91.68% over ZAP, and 70.74% over EvoCrawl. This was not uniform across applications; on OsCommerce, Maelstrom Crawler underperformed Black Widow by 4.58% and EvoCrawl by 15.40% in average code coverage. Maelstrom Crawler during its allocated 8 hours reached 3 657 requests compared to 9 363 for Black Widow and 24 509 for EvoCrawl, this pattern of fewer requests repeats for every tested application. This indicates higher coverage per request and potential for better comparative performance with better hardware.

The improvement in code coverage did not translate into better vulnerability discovery. Across the three tested applications, no additional vulnerabilities were found by the Maelstrom Crawler, even after accounting for our additional experimental runs we only discover the exact same vulnerabilities as Black Widow (see Section 5.3). In the final evaluation runs, Maelstrom Crawler found 2 fewer in total than Black Widow, although it is believed that a more thorough evaluation stage would also discover these.

Internal crawler testing, where we exclusively ran the separate navigation strategies, showed at least a 11.99% improvement on code coverage once combined. We also compared our switching mechanism with a fully randomised one, which showed in the worst case a 3.74% decrease in performance compared to the feedback driven switching.

Taken together, these results suggest that a feedback driven switching mechanism matters most once a crawler has exhausted the coverage that any reasonable strategy would reach quickly, and instead accounts for its context to find harder to reach areas within targeted applications. However, further research is necessary not only to further identify the intricacies of combining strategies to improve code coverage, but also to improve the vulnerability finding process.

Bibliography

[1] Veracode, *AI-Generated Code Poses Major Security Risks in Nearly Half of All Development Tasks, Veracode Research Reveals*, https://www.businesswire.com/news/home/20250730694951/en/AI-Generated-Code-Poses-Major-Security-Risks-in-Nearly-Half-of-All-Development-Tasks-Veracode-Research-Reveals, Business Wire press release, Accessed: 2026-02-02, Jul. 2025.

[2] OWASP. "A05:2025 Injection (OWASP Top 10:2025)." Accessed: 2026-02-02, Open Web Application Security Project. [Online]. Available: https://owasp.org/Top10/2025/A05_2025-Injection/.

[3] T. Zhang, H. Huang, Y. Lu, K. Zhu, and J. Zhao, "State-sensitive black-box web application scanning for cross-site scripting vulnerability detection," *Applied Sciences*, vol. 13, no. 16, p. 9212, 2023. DOI: 10.3390/app13169212. [Online]. Available: https://www.mdpi.com/2076-3417/13/16/9212.

[4] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "Webfuzz: Grey-box fuzzing for web applications," in *Computer Security – ESORICS 2021*, E. Bertino, H. Shulman, and M. Waidner, Eds., Cham: Springer International Publishing, 2021, pp. 152–172, ISBN: 978-3-030-88418-5.

[5] A. Stafeev and G. Pellegrino, "Sok: State of the krawlers – evaluating the effectiveness of crawling algorithms for web security measurements," in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*, USENIX Association, Aug. 2024, pp. 719–734, ISBN: 978-1-939133-44-1. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity24-stafeev.pdf.

[6] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1125–1142. DOI: 10.1109/SP40001.2021.00022.

[7] E. Olsson, "Spidering the modern web: Securing the next generation of web sites and browser extensions," Available at https://research.chalmers.se/en/publication/546193, Licentiate, Chalmers University of Technology, Gothenburg, Sweden, May 2025.

[8] S. H. Kim, "Understanding dfs vs bfs in web crawling: A practical perspective," *Medium*, Sep. 2024, Accessed: 2025-12-09. [Online]. Available: https://medium.

com/@seilylook95/understanding-dfs-vs-bfs-in-web-crawling-a-practical-perspective-8129c93bfb02.

[9] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling ajax by inferring user interface state changes," in *2008 Eighth International Conference on Web Engineering*, 2008, pp. 122–134. DOI: 10.1109/ICWE.2008.24.

[10] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed., USENIX Association, 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[11] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "Jäk: Using dynamic analysis to crawl and test modern web applications," in *Research in Attacks, Intrusions, and Defenses*, H. Bos, F. Monrose, and G. Blanc, Eds., Cham: Springer International Publishing, 2015, pp. 295–316, ISBN: 978-3-319-26362-5.

[12] B. Eriksson, A. Stjerna, R. D. Masellis, P. Rümmer, and A. Sabelfeld, "Black ostrich: Web application scanning with string solvers," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds., ACM, 2023, pp. 549–563. DOI: 10.1145/3576915.3616582. [Online]. Available: https://doi.org/10.1145/3576915.3616582.

[13] L. Cazzaro, S. Calzavara, M. Kovalkov, A. Stafeev, and G. Pellegrino, "Less is more: Boosting coverage of web crawling through adversarial multi-armed bandit," in *55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2025, Naples, Italy, June 23-26, 2025*, IEEE, 2025, pp. 183–192. DOI: 10.1109/DSN64029.2025.00030. [Online]. Available: https://doi.org/10.1109/DSN64029.2025.00030.

[14] B. Xiang, Y. Zhang, F. Liu, H. Huang, Z. Lin, and M. Yang, "Pig in a poke: Automatically detecting and exploiting link following dictionaries vulnerabilities in windows file operations," in *USENIX Security Symposium*, USENIX Association, 2025, pp. 7019–7038.

[15] C.-H. Liu, S. You, and Y.-C. Chiu, "A reinforcement learning approach to guide web crawler to explore web applications for improving code coverage," *Electronics*, vol. 13, p. 427, Jan. 2024. DOI: 10.3390/electronics13020427.

[16] L. Demetrio et al., "Evocrawl: Exploring web application code and state using evolutionary search," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2025-366-paper.pdf.

[17] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proceedings of the 16th international conference on World Wide Web*, ser. WWW'07, ACM, May 2007, pp. 141–150. DOI: 10.1145/1242572.1242592. [Online]. Available: http://dx.doi.org/10.1145/1242572.1242592.

[18] Xdebug Project, *Xdebug: A Debugger and Profiler Tool for PHP*, https://xdebug.org, Accessed: 2026-04-23, 2026.

[19] osCommerce, *Oscommerce v4*, https://github.com/osCommerce/osCommerce-V4, Accessed: 2026-05-26, 2026.

[20] F. L. Gewers, G. R. Ferreira, H. F. D. Arruda, F. N. Silva, C. H. Comin, D. R. Amancio, and L. D. F. Costa, "Principal component analysis: A natural approach to data exploration," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021, ISSN: 0360-0300. DOI: 10.1145/3447755. [Online]. Available: https://doi.org/10.1145/3447755.

[21] A. N. Kolmogorov, "On the empirical determination of a distribution function," in *Breakthroughs in Statistics: Methodology and Distribution*, ser. Springer Series in Statistics, S. Kotz and N. L. Johnson, Eds., Authorised English translation of: Kolmogorov, A.N. (1933). Sulla determinazione empirica di una legge di distribuzione. *Giornale dell'Istituto Italiano degli Attuari*, 4, 83–91., New York, NY: Springer, 1992, pp. 106–113. DOI: 10.1007/978-1-4612-4380-9_10.

[22] N. V. Smirnov, "Table for estimating the goodness of fit of empirical distributions," *The Annals of Mathematical Statistics*, vol. 19, no. 2, pp. 279–281, 1948. DOI: 10.1214/aoms/1177730256. [Online]. Available: https://projecteuclid.org/euclid.aoms/1177730256.

[23] W. J. Conover, *Practical Nonparametric Statistics*, 3rd. New York, NY: Wiley, 1999, The two-sample KS test is in Section 6.3, pp. 456–466., ISBN: 978-0-471-16068-7.

[24] V. Satopää, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "Kneedle" in a haystack: Detecting knee points in system behavior," in *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops (ICDCSW 2011)*, IEEE, 2011, pp. 166–171. DOI: 10.1109/ICDCSW.2011.20. [Online]. Available: https://ieeexplore.ieee.org/document/5961514.

[25] R. Killick, P. Fearnhead, and I. A. Eckley, "Optimal detection of changepoints with a linear computational cost," *Journal of the American Statistical Association*, vol. 107, no. 500, pp. 1590–1598, 2012. DOI: 10.1080/01621459.2012.737745. [Online]. Available: https://arxiv.org/abs/1101.1438.

[26] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. Smola, "A kernel two-sample test," *Journal of Machine Learning Research*, vol. 13, no. 25, pp. 723–773, 2012. [Online]. Available: https://jmlr.org/papers/v13/gretton12a.html.

[27] D. Garreau, W. Jitkrittum, and M. Kanagawa, *Large sample analysis of the median heuristic*, 2017. arXiv: 1707.07269 [math.ST]. [Online]. Available: https://arxiv.org/abs/1707.07269.

[28] Google Chrome Labs, *Chrome for Testing*, https://googlechromelabs.github.io/chrome-for-testing/, Accessed: 2026-04-23, 2026.

[29] SeleniumHQ, *Selenium*, https://www.selenium.dev/, Accessed: 2026-04-23, 2026.

[30] Kanboard, *Kanboard*, https://github.com/kanboard/kanboard, Accessed: 2026-05-27, 2026.

[31] WordPress Contributors, *Wordpress develop*, https://github.com/WordPress/wordpress-develop, GitHub repository, accessed 2026-05-27, 2026.

[32] The ZAP Team, *Zed attack proxy (zap)*, Version 2.17.0, accessed 2026-05-13, ZAP, 2026. [Online]. Available: https://www.zaproxy.org/.

A

Appendix

A.1 Principal Component Analysis (PCA)

Here we present extra plots retrieved during our principal component analysis.

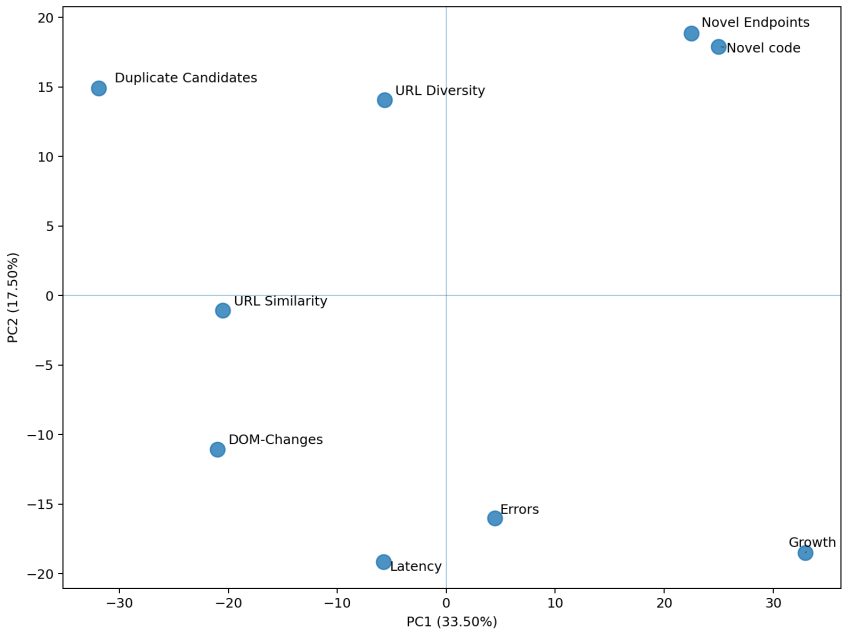


Figure 1: A 2-dimensional PCA plot from the BFS exclusive run.

Figure 2: A 2-dimensional PCA plot from the DFS exclusive run.

A.1.1 Window size plots

Extra plots visualising the feedback channel change-points for both the DFS and BFS run are presented here.

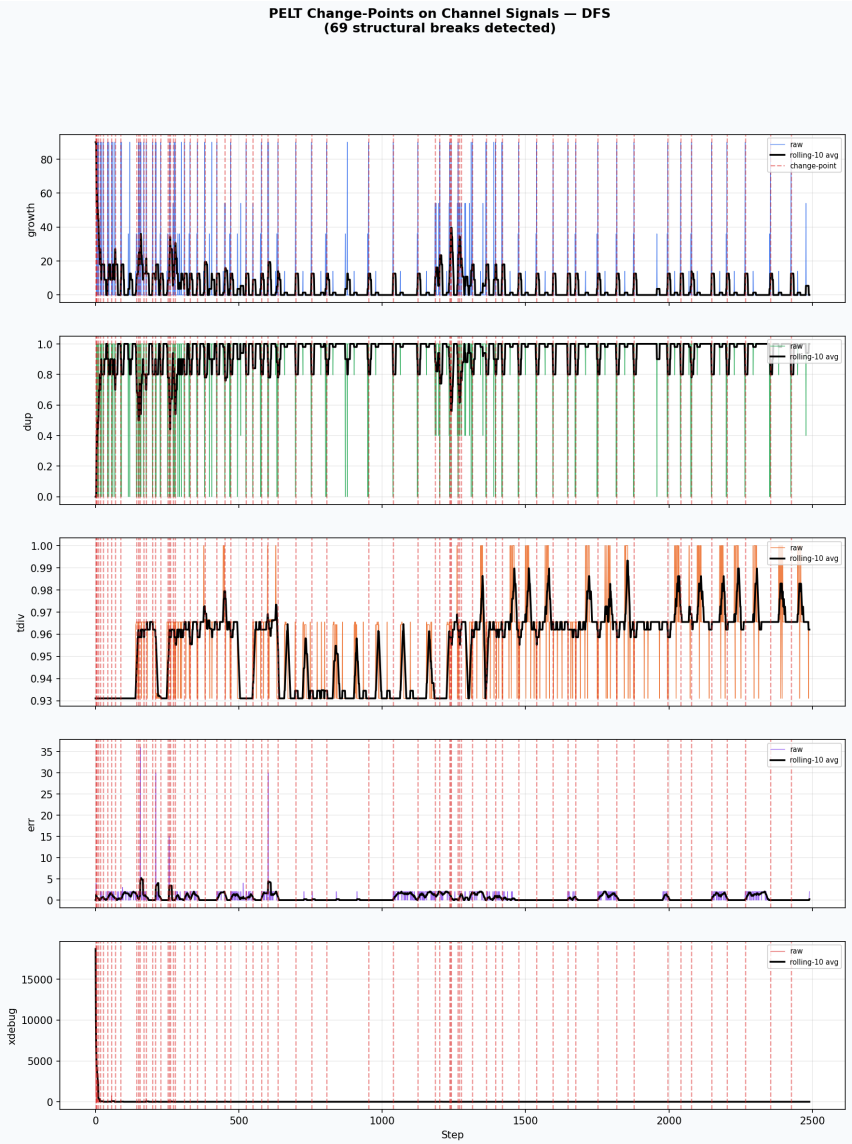


Figure 3: Feedback channel change-points from the DFS exclusive run.

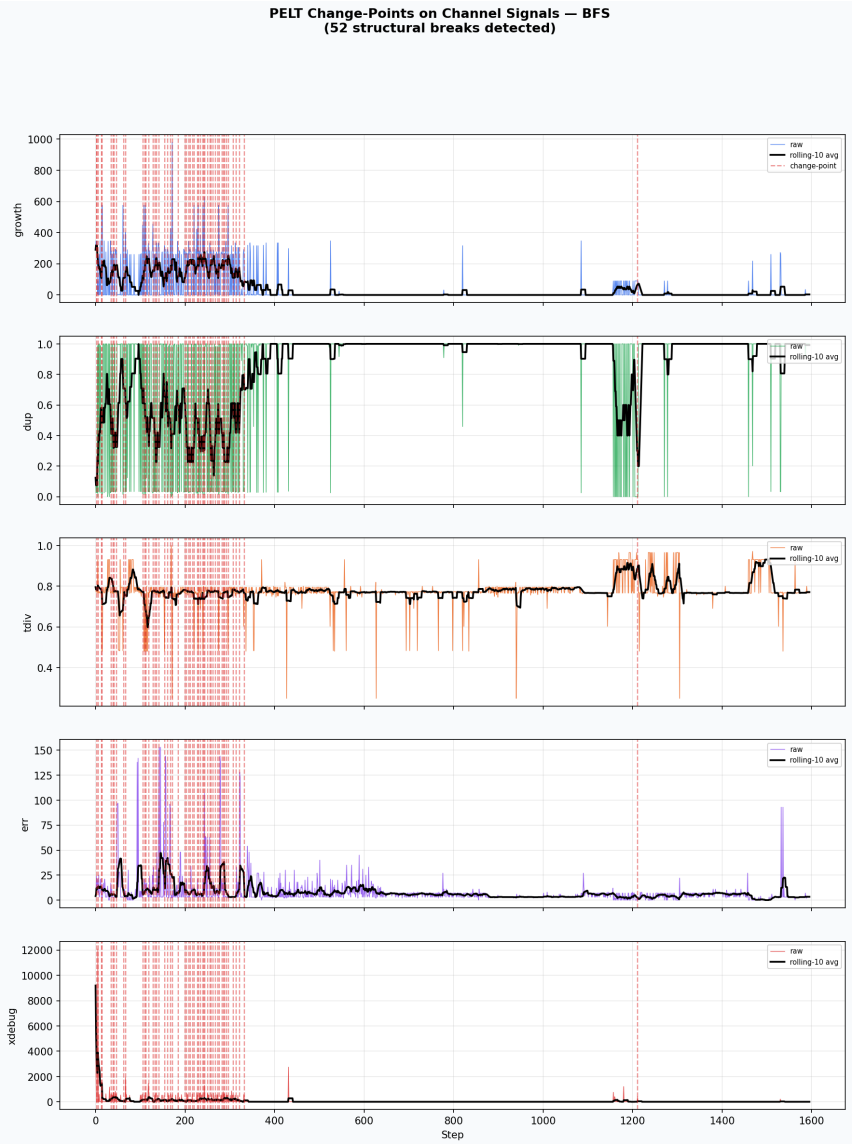


Figure 4: Feedback channel change-points from the BFS exclusive run.

A.1.2 Ablation study results on feedback channels

Here we present more in-depth results from the ablation study on feedback channel. Each run spans 8 hours.

Table A.1: Lines of unique code executed during each run on OsCommerce, baseline includes all feedback channels; the remainder indicate excluded channels.

OsCommerce

Run	Baseline	¬Novel code	¬Growth	¬Dup-URLs	¬URL diversity	¬Error
1	84 790	87 681	84 130	88 386	89 685	89 683
2	89 603	89 108	85 926	88 945	87 302	87 947
3	82 446	89 706	88 627	86 449	87 796	88 463
Average	**85 613**	**88 832**	**86 228**	**87 927**	**88 261**	**88 698**
ΔBaseline	0%	3.76%	0.72%	2.70%	3.09%	3.60%
∩ code	77 705	85 161	82 928	83 606	85 209	85 652
∪ code	91 994	92 635	90 019	92 516	92 166	91 742

Table A.2: Lines of unique code executed during each run on WordPress, baseline includes all feedback channels; the remainder indicate excluded channels.

WordPress

Run	Baseline	¬Novel code	¬Growth	¬Dup-URLs	¬URL diversity	¬Error
1	82 356	83 092	82 873	61 243	87 372	81 687
2	85 045	87 105	77 502	77 262	73 945	76 744
3	83 184	81 885	79 430	76 729	83 692	75 042
Average	**83 528**	**84 027**	**79 935**	**71 745**	**81 670**	**77 824**
ΔBaseline	0%	0.60%	-4.30%	-14.1%	-2.22%	-6.83%
∩ code	74 434	78 458	69 514	58 092	69 366	71 208
∪ code	91 498	90 639	91 313	86 009	92 788	86 421

Table A.3: Lines of unique code executed during each run on Kanboard, baseline includes all feedback channels; the remainder indicate excluded channels.

Kanboard

Run	Baseline	¬Novel code	¬Growth	¬Dup-URLs	¬URL diversity	¬Error
1	10 766	7 135	7 111	7 111	7 118	10 702
2	7 136	6 949	7 111	7 124	7 110	7 136
3	7 113	7 111	10 649	6 952	7 128	7 105
Average	**8 338**	**7 065**	**8 290**	**7 062**	**7 119**	**8 314**
ΔBaseline	0%	-15.27%	-0.58 %	-15.30%	-14.62%	-0.29%
∩ code	7 108	6 924	7 106	6 914	7 110	7 075
∪ code	10 794	7 137	10 654	7 138	7 133	10 742

A.1.3 Coverage and vulnerabilities results

Here we present detailed results from the evaluation of Maelstrom Crawler and compare them to similar tests conducted on Black Widow, ZAP and EvoCrawl. Each run spans 8 hours.

Table A.4: Lines of unique code executed during each run on OsCommerce, baseline includes all feedback channels; the remainder indicate excluded channels.

OsCommerce

Run	Maelstrom Crawler	Black Widow	ZAP	EvoCrawl
1	88 854	92 575	35 252	102 820
2	89 112	94 366	35 339	102 802
3	89 337	92 597	36 465	102 834
Average	**89 101**	**93 179**	**35 685**	**102 819**
ΔMaelstrom Crawler	0%	4.58%	-59.95%	15.40%
∩ code	86 656	89 843	35 096	102 797
∪ code	92 026	95 796	36 520	102 854
Average number of request	4 360	8 599	116 368	24 689
Novel code per request	20.44	10.84	0.31	4.16
∪ Vulnerabilities	5	5	0	5

Table A.5: Lines of unique code executed during each run on WordPress, baseline includes all feedback channels; the remainder indicate excluded channels.

Wordpress

Run	Maelstrom Crawler	Black Widow	ZAP	EvoCrawl
1	82 157	80 290	63 530	60 854
2	84 125	80 346	63 350	70 392
3	70 610	70 310	62 819	54 002
Average	**78 964**	**76 982**	**63 233**	**61 749**
ΔMaelstrom Crawler	0%	-2.51%	-19.92%	-21.80%
∩ code	63 382	62 002	62 624	52 845
∪ code	93 641	88 424	63 588	71 656
Average number of request	12 863	17 975	62 693	28 792
Novel code per request	6.14	4.28	1.01	2.14
∪ Vulnerabilities	1	1	0	1

Table A.6: Lines of unique code executed during each run on Kanboard, baseline includes all feedback channels; the remainder indicate excluded channels.

Kanboard

Run	Maelstrom Crawler	Black Widow	ZAP	EvoCrawl
1	10 618	6 110	4 420	3 192
2	7 102	6 343	4 736	3 192
3	10 787	9 358	5 064	3 192
Average	**9 502**	**7 270**	**4 740**	**3 192**
ΔMaelstrom Crawler	0%	-23.49%	-50.12%	-66.41%
∩ code	7 043	6 025	4 279	3 192
∪ code	10 831	9 443	5 166	3 192
Average number of request	5 482	4 466	263 345	42 335
Novel code per request	1.73	1.63	0.02	0.08
∪ Vulnerabilities	0	2	0	0