# Scrooge: Detection of Changes in Web Applications to Enhance Security Testing

Fabio Büsser[1], Jan Kressebuch[1], Martín Ochoa[1], Valentin Zahnd[2] and Ariane Trammell[1]
[1]*Institute of Computer Science, Zurich University of Applied Sciences, Winterthur, Switzerland*
[2]*Secuteer GmbH, Zurich, Switzerland*

Keywords: Security Testing, Black-Box Testing, Software Evolution.

Abstract: Due to the complexity of modern web applications, security testing is a time-consuming process that heavily relies on manual interaction with various analysis tools. This process often needs to be repeated for newer versions of previously tested applications, as new functionalities frequently introduce security vulnerabilities. This paper introduces *scrooge*, a tool that automates change detection in web application functionality to enhance the efficiency and focus of the security testing process. We evaluate *scrooge* on various platforms, demonstrating its ability to reliably detect a range of changes. *Scrooge* successfully identifies different types of changes, showcasing its applicability across diverse scenarios with high accuracy.

## 1 INTRODUCTION

The ever-evolving landscape of web applications presents a continuous challenge for security testing. Among various testing strategies, black-box testing is commonly performed, where the source code of a web application is unknown, simulating external attackers (Bau et al., 2010). In black-box testing, a combination of automatic and manual tasks are often employed to achieve high coverage of the application under test, as automatic crawling has well-known limitations (Doupé et al., 2010). Additionally, a combination of manual and automatic testing is typically needed to test for both well-known vulnerabilities (such as various injection types) and logical vulnerabilities that are inherent to the application and are more difficult to automate (Pellegrino and Balzarotti, 2014).

Testing web applications is time-consuming and resource-intensive. Furthermore, when an application evolves due to new functionality, the security testing process needs to be repeated. It has been observed that changes to software tend to introduce new vulnerabilities to existing versions (Mitropoulos et al., 2012; Williams et al., 2018). Therefore, it is advantageous for security testers to quickly identify which parts of a web application have changed since the last security testing, to prioritize their efforts accordingly.

This paper introduces *scrooge*, a tool designed to automatically identify changes between two versions of the same web application in a black-box fashion. By automating change detection, scrooge has the potential to significantly improve the efficiency and effectiveness of security testing.

We propose a data-gathering architecture aimed at abstracting key interfaces and the state of a web application. The resulting graph data structure allows us to perform various change analyses, from parameter changes to changes in HTML or JSON responses, to changes in how a given page is visually rendered. We evaluate our approach on three web applications (two popular e-shop applications and one application used in security training and teaching) and measure its accuracy with respect to changes, either artificially introduced or resulting from enabling new functional modules. Our evaluation shows that scrooge can precisely detect changes without false positives. Reaching all changes may depend on the selection of the crawling strategy, which we also evaluate using two popular crawlers and manual crawling. Our tool is open source (Kressebuch and Büsser, 2024).

The rest of this paper is structured as follows: We introduce our approach in Section 2. We then discuss our implementation in Section 3. Section 4 contains our evaluation. We discuss limitations and future work in Section 5. Related work is summarized in Section 6, and we conclude in Section 7.

## 2 APPROACH

The overall integration of scrooge in the penetration testing tool flow is shown in Fig. 1. A penetration



Büsser, F., Kressebuch, J., Ochoa, M., Zahnd, V. and Trammell, A.
Scrooge: Detection of Changes in Web Applications to Enhance Security Testing.
DOI: 10.5220/0013139600003899

In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 2*, pages 48-59
ISBN: 978-989-758-735-1; ISSN: 2184-4356

tester performs a test of a web application in a given version $x$. If a new version $x+1$ is available, she uses scrooge to determine the changes between the version she tested last and the new version. Scrooge analyses the versions in question and determines the changes which can be represented as an annotated graph or a text report. The penetration tester reviews the change analysis and plans the test of the new version to analyse the detected changes.

While this approach has the potential to increase efficiency in the security testing, it is crucial that changes are detected reliably. If a change would not be detected the penetration tester might ignore relevant new functionality and therefore a vulnerability might remain undetected. Also, this approach must be as precise as possible, because false positives would need to be manually inspected and discarded, which would require additional work.

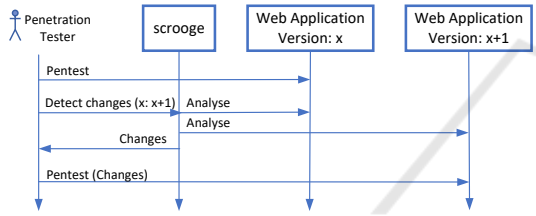



Figure 1: Integration of scrooge in Penetration Testing Flow.

In the following we describe the architecture of scrooge and how it was designed to accurately detect various families of changes in a black-box fashion.

## 2.1 Architecture

The overall architecture of scrooge is shown in Fig. 2.

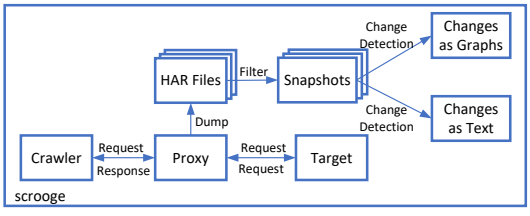



Figure 2: Design Overview of the Overall System.

Functionality in modern web applications is complex, given that typically one part of the logic is defined in the front-end (via JavaScript) and another part on the back-end or external services (via synchronous or asynchronous requests). In order to identify changes between two versions of a web application, ideally we should be able to exercise as much of the applications' logic as possible. Since a manual navigation by a human would be time consuming and possibly incomplete, it is natural to involve an automatic crawler in this process. This crawler navigates the website independently and tries to reach as many endpoints as possible. Note however that our architecture is independent from the type of crawler used and is compatible with manual crawling as we will see in Section 4.

The crawler accesses the target web application with a proxy in order to dump all requests and received answers. This data is stored in so called HAR-Files (HTTP Archive format), which is a JSON formatted archive file that stores all data relevant for the interaction of a web browser with a web site. This data is then analysed and filtered to only keep relevant information in a data structure suitable for the analysis of different versions of the same web application.

This data structure is a graph where nodes are aggregated URLs and edges represent links or asynchronous calls to other URLs. Moreover for each node we store sent parameters (if any) and responses. Those graphs are referred to as *snapshots*. Two snapshots can be compared by using various metrics to detect changes. The detected changes are stored as text or as graphical files which can be used by a penetration tester to identify the changed areas of a web application. The comparison algorithms to detect changes are described in the next section.

## 2.2 Detection of Changes Between Snapshots

In order to detect different types of changes in a web application, several detection mechanisms need to be combined. Scrooge combines approaches from the following three different areas:

- **Changes in Parameters and Requests.** It is analysed whether the crawler generates different requests to the web application indicating that the scope of the web application changed.
- **Changes in the Graphical Representation.** Changes are detected by comparing screenshots of supposedly similar web pages.
- **Changes in the HTML Structure.** The structure of supposedly similar HTML pages are compared to detect differences.

The comparison algorithms used are described in Section 3. At the basis of the comparison algorithms we use well known visual, textual and tree comparison algorithms that we briefly recap here. Note that these algorithms will give a similarity score and

are not binary (equal or different). This is a common practice when assessing the similarity of web sites (Mallawaarachchi et al., 2020) given that a strict comparison (i.e. using hashes of HTML) would potentially yield false positives in common situation such as dynamic content (banners, state of database). Non-binary scores allow users to configure thresholds and adjust accuracy to various scenarios. We nevertheless assume that in a penetration testing scenario, the application under tests in different versions has more or less a stable or minimalist database content (for instance an e-shop has more or less the same inventory in the two versions), as we will discuss in the evaluation in Section 4.

*DHash.* The dHash algorithm (difference hash) (Krawetz, 2013) is a hashing function for recognizing image similarities. It converts an image into a hash value that represents the visual appearance of the image. This hash value can be used to quickly compare images to determine if there is a similarity.

*Jaccard Similarity.* The Jaccard similarity (Jadeja, 2022) describes the similarity of two sets based on the intersection divided by the union of the elements of both sets. A high overlap between the intersection and the union results in a high match, while a low overlap results in a low match.

*Tree Edit Distance.* The tree edit distance describes the similarity of two data structures arranged as a tree. Starting from the root node, the tree to be compared is searched for differences in the underlying nodes and the steps required to reach the target state are calculated.

# 3 IMPLEMENTATION

We implemented scrooge as a prototype in python which is available as an open source project (Kressebuch and Büsser, 2024). The following sections describe the most important implementation aspects.

## 3.1 Crawler and Proxy

As discussed previously, in order to detect as many changes as possible, it is important to fully navigate the web application under test. Therefore, in our work we implemented scrooge with the following three web crawler approaches. In order to prepare for our evaluation, we have chosen state of the art crawlers, both following a deterministic and a randomized strategy, and manual crawling. Note that scrooge can be integrated with any other crawler.

- **CrawlJax** (Mesbah et al., 2012) is a deterministic crawler that automatically analyzes user interface state changes in the web browser. The crawler scans the DOM tree of the application and identifies elements that can potentially change the state of the application. These elements are then automatically activated by triggering events such as clicks. This gradually creates a state machine that models the various navigation paths and states within the web application.
- **Black Widow** (Eriksson et al., 2021) is a randomized crawler based on a data-driven approach to crawling and scanning web applications. The tool addresses three central pillars for in-depth crawling and scanning: Navigational Modeling, Traversal, and State Dependency Tracking. The effectiveness of Black Widow is illustrated by significant improvements in code coverage compared to other crawlers, with coverage between 63% and 280% higher depending on the application (Eriksson et al., 2021).
- **Manual Crawling** was used as a baseline to evaluate the performance of the two automated crawlers.

As a proxy server, we use mitmproxy (@maximilianhils et al., 2013) that saves all observed connections as a HAR file.

## 3.2 Detection of Changes

In the snapshot data structure, nodes are aggregated URLs. We have chosen to aggregate URLs with the same identifier in order to abstract the state of the application. We are aware that this abstraction may have an impact in terms of precision depending on how the application under tests is coded, but it has advantages in terms of efficiency. In the following, an example of the generation of an identifier for a GET method is shown.

```
GET-Request:
https://example.domain/product?cart=2&quantity=5

⇓

GET/product?cart&quantity
```

The detection of changes between snapshots is performed by a set of comparison algorithms. As introduced in 2.2, the different approaches can be divided into metrics that are based on changes in the parameters and requests, changes that are based on the graphical representation and changes that are based on the structure of the HTML file. In the following paragraphs we describe the chosen metrics more precisely.

#### 3.2.1 Detection of Changes in Parameters and Requests

The following metrics were used to analyze changes in the parameters or in the requests themselves.

- **ParameterChange** is a method for detecting changes by comparing the parameters sent in different requests. It works by creating an alphabetically sorted list of parameters for each snapshot and then comparing the lists. If the lists differ, it is considered a change. This method is useful because changes in the parameters often indicate changes in the way the application is working.
- **MissingRequestChange** is a method for detecting changes by identifying requests that are present in one snapshot but not another. It works by iterating over all requests in Snapshot 1 and checking if each request is also present in Snapshot 2. If a request is not found in Snapshot 2, it is considered a change. This method is useful for identifying changes in the flow of a web application, as the removal of a request can indicate a change in the way the application is structured.
- **NewRequestChange** is a method for detecting changes by identifying requests that are present in one snapshot but not another. It works by iterating over all requests in Snapshot 2 and checking if each request is also present in Snapshot 1. If a request is not found in Snapshot 1, it is considered a change. This method is useful for identifying changes in the flow of a web application, as the addition of a request can indicate a change in the way the application is structured.
- **AsyncRequestsChange** is a method for detecting changes by identifying differences in the execution of asynchronous requests associated with a static page. It does not consider the parameters or responses of asynchronous requests but instead focuses on whether new requests are being executed or previous requests are no longer being executed. This can indicate that new features have been added to the corresponding static page.
- **AsyncRequestParamChange** is a method for detecting changes by identifying differences in the JSON schema of asynchronous request parameters. It works by generating a JSON schema from the request body (Content-Type JSON) for each snapshot and then comparing the schemas using a standard library (xlwings, 2024). If differences are found, it is considered a change. Only asynchronous requests with the Content-Type `application/json` are supported. This method is useful for identifying changes in the data that is being passed between the web application and the client, as changes in the request parameters can indicate changes in the way the application is interacting with the user.
- **AsyncResponseChange** is a method for detecting changes by identifying differences in the JSON schema of asynchronous response bodies. It works by generating a JSON schema from the response body (Content-Type JSON) for each snapshot and then comparing the schemas (again using (xlwings, 2024)). If differences are found, it is considered a change. Only asynchronous responses with the Content-Type `application/json` are supported. This method is useful for identifying changes in the data that is being returned from the web application to the client, as changes in the response body can indicate changes in the way the application is providing information to the user.

#### 3.2.2 Detection of Changes in the Graphical Representation

The following metric was used to detect changes in the graphical representation of a webpage.

- The **DHashStructureChange** detects changes by comparing the visual structure of a static request using a difference hashing algorithm. It works by loading the HTML structure for a request with an identical identifier from both snapshots into a web browser and taking a screenshot of the window. The resulting hash from the images is then compared, and the change rate is determined based on the difference in the hash. The value $N = 8$ was chosen for calculating the DHash.

#### 3.2.3 Detection of Changes in the HTML Structure

The following two metrics were used to detect changes in the structure of the HTML code.

- **JaccardStructureChange** is a method for detecting changes in web application functionality by comparing the HTML structure using the Jaccard Similarity algorithm. It works by creating a set for each HTML document from Snapshot 1 and Snapshot 2, and then comparing the sets using the Jaccard Similarity algorithm to determine a change rate.
- **TreeDifferenceStructureChange** is a method for detecting changes in web application functionality by comparing the HTML structure using a tree edit distance algorithm. It works by converting the

HTML structure into a node tree for each snapshot and then traversing the trees synchronously. An edit distance is counted throughout the traversal. For each node, it is checked whether there is a corresponding node in the opposite snapshot. If the corresponding node is missing, the edit distance is increased by one. Similarly, the number of child tags is checked, and if there is a difference, the edit distance is also increased by this difference. This method is useful for identifying changes in the hierarchical structure of a web page, as changes in the HTML structure can indicate changes in the organization of the content on the page.

# 4 EVALUATION

In the previous sections, we introduced the architecture and overall approach of scrooge. Given the complexity of modern web applications, accurately estimating the performance of our methodology is challenging. Our change detection features could potentially produce false positives (being too sensitive to apparent changes) or false negatives (failing to detect certain types of functional changes). Therefore, in this section, we apply our methodology to two versions of three different popular web applications, which offer sufficient complexity to preliminarily evaluate our approach in terms of accuracy.

Note that following choices have been made for the three applications: the state of the application database has been preserved between the two versions of an application as much as possible (modulo necessary changes for the new functionality). We have decided not to set a predetermined configuration on the thresholds of the non-binary similarity metrics, but to manually inspect any detected difference by scrooge and assess whether it was the result of an intended change in the new version or a false positive.

## 4.1 Evaluation on WordPress & WooCommerce

WooCommerce (WooCommerce, 2024) is a widely-used open-source e-commerce plugin for WordPress. It allows users to create and manage online stores within their WordPress sites, offering features such as product management, order processing, payment gateways, and shipping options. WordPress (version 6.4.3) with WooCommerce (version 8.7.0) as the installed e-commerce plugin was used as the test application. For the evaluation, the installation was cloned and targeted functional changes were made to the cloned version. Both instances have the same version numbers and the same data status. The unchanged version is hereinafter referred to as *Original*, while the modified version is referred to as *Modified*. The following is a list of introduced changes in order to evaluate the effectiveness of scrooge at detecting them.

*C1 - Additional Parameter in POST Request "Add to Cart"* The form used to add a product to the cart was supplemented with an additional parameter.
*Goal:* Detect parameter changes in static requests (POST)

*C2 - Additional Parameter in AJAX Request "Add to Cart"* The asynchronous addition to the cart was supplemented with an additional parameter.
*Goal:* Detect parameter changes in asynchronous requests

*C3 - Additional Field in AJAX Response "Add to Cart"* The response for the asynchronous "Add to Cart" was supplemented with a new field.
*Goal:* Detect parameter changes in asynchronous requests

*C4 - Additional Step in Checkout Process* The normal checkout process proceeds as follows:
Cart $\longrightarrow$ Checkout $\longrightarrow$ Order Confirmation
In Modified, an additional step was added.
Cart $\longrightarrow$ Cross-Selling $\longrightarrow$ Checkout $\longrightarrow$ Order Confirmation
*Goal:* Detect a flow change in a process

*C5 - New AJAX Request on Product Page* Product pages now send an additional asynchronous request.
*Goal:* Detect new AJAX requests

*C6 - Template Adjustment on Product Page* The HTML structure on the product page was adjusted. New HTML elements were added to the "Add to Cart" form.
*Goal:* Detect a structural change

*C7 - Sample Page "Sample Page" Deleted* The content page "Sample Page" was deleted in Modified.
*Goal:* Detect that content/functions are missing

*Snapshot Comparison and Interpretation* Figure 3 illustrates the output of our tool when comparing the two versions of the web application. Green nodes represent new functionality and red nodes removed functionality. Table 1 summarizes the introduced changes vs. the change detection algorithms that were able to detect them. All changes C1-C7 were detected by one or more change detection algorithms. For instance, C6 "Template Adjustment on Product Page" triggered a `JaccardStructureChange`, a `TreeDifferenceStructureChange` and a visual `DHashStructureChange` on values above the pre-

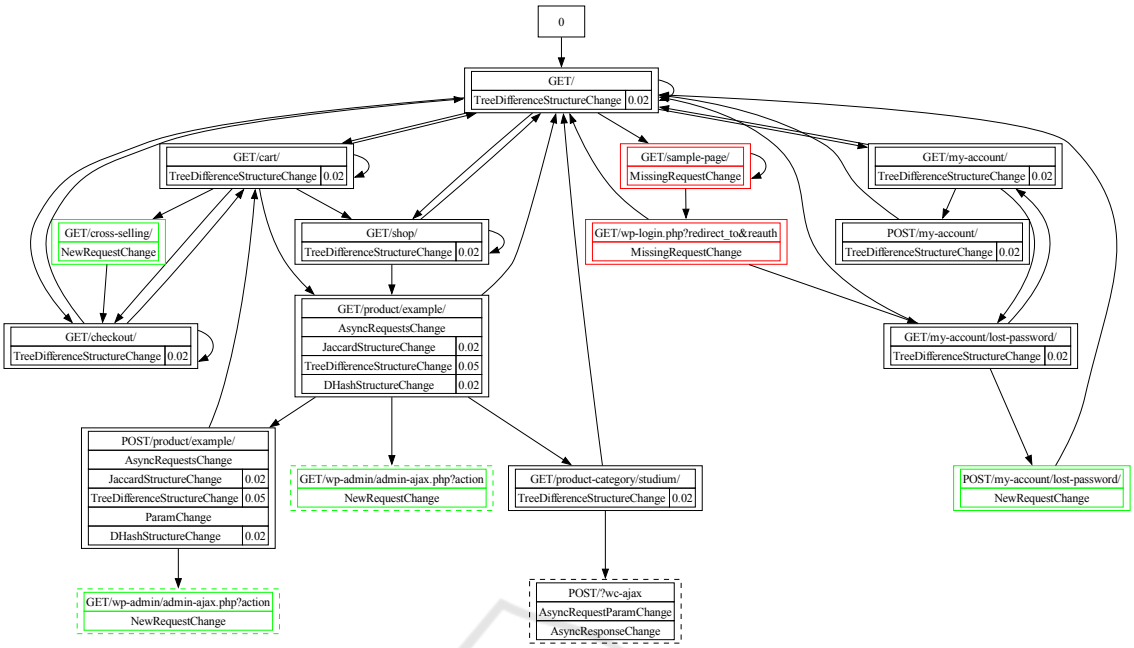



Figure 3: Evaluation Graph WordPress Original - Modified (CrawlJax). Green nodes represent new functionality and red nodes removed functionality.

configured thresholds.

In sum, all changes were detected and no false positives where found, that is, all indicators of change could be associated to one of the changes C1-C7. From a crawling perspective, for the WooCommerce webshop, both CrawlJax and Black Widow worked well. Some fine-tuning was required to ensure the complete run of the crawlers. For example, CrawlJax gets stuck on the WordPress search form. If this is removed, CrawlJax runs completely through all pages.

## 4.2 Evaluation on PrestaShop

PrestaShop (version 8.1.5) is a popular e-commerce platform (PrestaShop, 2024). In order to evaluate our approach, a copy of a webshop was created, with some new modules activated in the cloned version. The unchanged version is hereinafter referred to as Original, while the modified version is referred to as Modified. The goal is now to check if the newly active modules can be detected using the comparison algorithms.

*Activated Modules in the modified version* The following modules were activated in the modified version in order to test the change detection capabilities of our approach. Note that the changes introduced are therefore not under our control, which should represent a more realistic situation compared to the previously described evaluation on Wordpress.

*M1 Wishlist* Allows customers to create wishlists to save their favorite products for later

*M2 Customer "Sign in" link* Allows customers to easily register for the webshop

*M3 Contact form* Adds a contact form to the contact page

*M4 Newsletter subscription* Allows customers to sign up for the newsletter in the footer of the webshop

*Snapshot Comparison and Interpretation*The result of running scrooge on both version is depicted in Fig. 4. For instance, the modification M3 (Contact form) was detected at GET/contact-us using JaccardStructureChange, TreeDifferenceStructureChange, and DHashStructureChange. The new request POST/contact-us also reveals the new form on the contact page. Figure 5, illustrates the before and after rendering of this website, which explains why DHashStructureChange has detected this change.

*Insights PrestaShop* The newly introduced modules in the modified version were identified accurately. Since these are entirely new modules and no existing functions were modified, the NewRequestChange and AsyncRequestsChange functions are particularly useful in this case.

As with the evaluation with WooCommerce in the previous subsection, automatic testing using crawlers proved challenging. The constant crashes or hang-ups

Table 1: Evaluation of detected changes.

| Evaluation of introduced changes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Change | C1 | C2 | C3 | C4 | C5 | C6 | C7 | False-Positives |
| MissingRequestChange | | | | | | | X | 0 |
| NewRequestChange | | | | X | | | | 0 |
| AsyncRequestsChange | | | | | X | | | 0 |
| JaccardStructureChange | X | | | | | X | | 0 |
| DHashStructureChange | | | | | | X | | 0 |
| TreeDifferenceStructureChange | | | X | | | X | X | 0 |
| ParamChange | X | | | | | | | 0 |
| AsyncRequestParamChange | | X | | | | | | 0 |
| AsyncResponseChange | | | X | | | | | 0 |

Table 2: Evaluation of found changes depending on crawler used.

| Evaluation Crawler | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Change | C1 | C2 | C3 | C4 | C5 | C6 | C7 | False Positives |
| CrawlJax | X | X | X | X | X | X | X | 0 |
| BlackWidow | X | X | X | X | X | X | X | 0 |
| Manual Crawling | X | X | X | X | X | X | X | 0 |

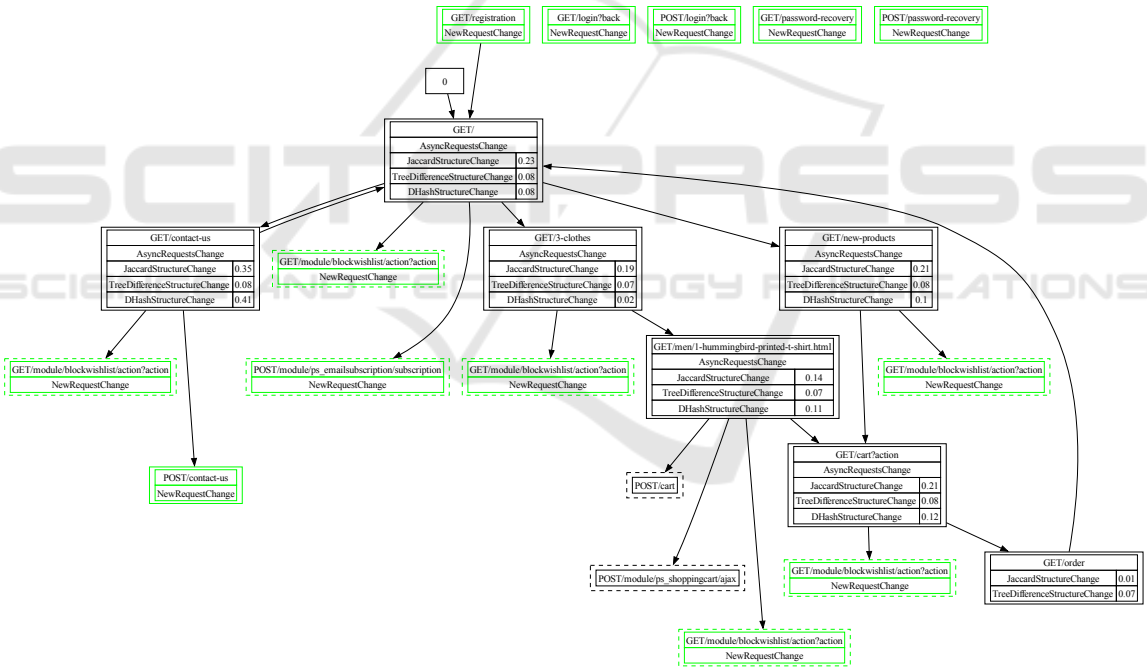



Figure 4: Evaluation Graph PrestaShop Original - Modified (Manual Crawling).

of the crawlers require numerous adjustments to the crawler as well as the test application. No clear pattern was identified as to where the crawlers fail. Ultimately, once those issues were fixed however, both crawlers were able to run through the entire shop and detect all changes without false positives.

## 4.3 Evaluation on DVWA

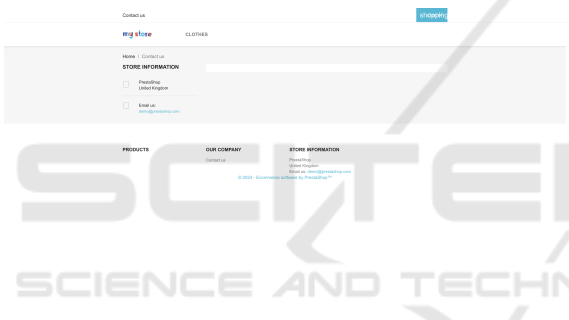
To test the functionality of change detection on general websites, a local version of the Damn Vulnerable Web Application (DVWA) (dvw, 2010) was installed. This application illustrates various vulnerabilities through customizable difficulty levels. Each difficulty level has a slightly different implementation

Table 3: Evaluation of Changes per Module.

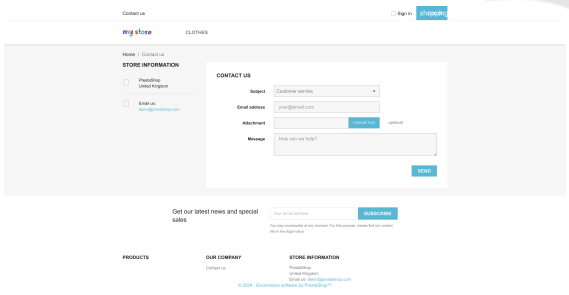
| Evaluation of Types of Changes | | | | | |
|---|---|---|---|---|---|
| Type of Change | M1 | M2 | M3 | M4 | False-Positives |
| MissingRequestChange | | | | | 0 |
| NewRequestChange | X | X | X | X | 0 |
| AsyncRequestsChange | X | | | X | 0 |
| JaccardStructureChange | | X | X | X | 0 |
| DHashStructureChange | | X | X | X | 0 |
| TreeDifferenceStructureChange | | X | X | X | 0 |
| ParamChange | | | | | 0 |
| AsyncRequestParamChange | | | | | 0 |
| AsyncResponseChange | | | | | 0 |

Table 4: Evaluation of Detected Modules by Crawler.

| Evaluation of Crawlers | | | | | |
|---|---|---|---|---|---|
| Type of Change | M1 | M2 | M3 | M4 | False-Positives |
| CrawlJax | X | X | X | X | 0 |
| BlackWidow | X | X | X | X | 0 |
| Manual Crawling | X | X | X | X | 0 |



(a)



(b)

Figure 5: Comparison “Contact us” (a) **M3** deactivated (b) **M3** activated.

which makes it suitable for evaluating our approach. Thus, for testing purposes, the modified functionality was defined by adjusting the difficulty level.

*Insights from DVWA Crawler Comparison* The DVWA application was crawled three times. Initially, manual interaction was used with the application, and the resulting requests were collected via the proxy. In the second step, Black Widow was used as the crawler, followed by CrawlJax in the third step.

Using CrawlJax as the crawler for DVWA did not yield the desired results in the tests. CrawlJax consistently terminated without accessing any subpages. In tests using Black Widow, the crawler navigated autonomously through the menus. The website navigation with Black Widow visually confirmed accessing all subpages accessible in the website’s navigation menu. However, visible forms were only partially filled out and submitted, thus not all functionalities of the website were tested with Black Widow.

A baseline was formed from the union of all detected request identifiers in Snapshots Snapshot 1, Snapshot 2, and Snapshot 3. The baseline was derived from manual interaction, crawling with Black Widow, and crawling with CrawlJax. Table 5 (in Appendix) displays the existing URLs in the snapshots. The resulting baseline includes **35 URLs**, with **34 URLs** reached through manual interaction. Crawling with Black Widow as the crawler reached **23 URLs**. It became apparent that Black Widow reached all subpages, but did not test all forms, thus missing requests for form submissions in Snapshot 2. The request `GET/instructions.php?doc` was reached in the crawl with Black Widow, but not in manual interaction. Crawling with CrawlJax as the crawler resulted in coverage of only three URLs, including the login page `GET/login.php`, the landing page `GET/index.php`, and the home page `GET/`.

Due to Black Widow’s inability to correctly fill

out all forms, it was decided to manually navigate DVWA for evaluating differences between security levels Low and High. It is expected that precisely the changes in behavior during form submission will provide insights into the changes in the application's implementation. Therefore, as many existing forms as possible must be filled out for evaluation purposes.

In a direct comparison of security levels Low and High, several differences in the website's behavior were identified as depicted in Fig. 6.

*Insights DVWA.* The automatically identified differences by scrooge between the security levels Low and High were manually analyzed in order to determine their accuracy. All differences could be explained by functionality changes between these two levels. For instance the page GET/vulnerabilities/sqli/ was implemented fundamentally differently in both security levels. While an input field was used directly for input at security level Low, at level High a link points to a pop-up window that contains the input mask for the ID. This leads to structural changes in the direct comparison and to new requests that open and send the pop-up window. From a manual inspection of the inspected functionality, it all seems scrooge was able to detected most changes between the two security level. However we did not have a ground truth definition of all changes implemented, so there could be potentially false negatives in the automatic analysis.

# 5 DISCUSSION

As we have seen in the previous section, scrooge exhibits promising capabilities for detecting changes in the functional scope of web applications. However, several limitations hinder its broader applicability and require further investigation.

**Abstraction of Functionality and Application's State.** By construction, our graph data structure aggregates URLs with the same identifier but different parameter values. Depending on the application's logic this abstraction may affect the precision of our analysis. Similarly, the state of persistent storage may affect accuracy of a comparison (i.e. an empty e-shop vs. a shop with several items in inventory). We believe however that our preliminary evaluation is promising in the sense that for the evaluated applications the achieved accuracy of the comparison was high. A more thorough evaluation on other applications and architectures constitutes interesting future work.

**Crawler Challenges:** The current reliance on external crawlers introduces a significant limitation. These crawlers exhibit inconsistencies in performance across different web applications. Some function flawlessly, while others struggle entirely. To address this, future work should explore several avenues. Combining multiple crawlers can leverage their strengths and mitigate weaknesses by creating snapshots from various tools, ensuring comprehensive analysis. Developing a dedicated crawler optimized for scrooge, particularly for detecting specific change types and handling Single Page Applications (SPAs), would enhance precision and effectiveness. Additionally, utilizing parallel crawling techniques can significantly reduce analysis time, making scrooge more efficient for large-scale applications by running multiple crawlers simultaneously for extensive coverage in a shorter period.

**SPA Integration Hurdles:** Single-page applications (SPAs) pose a unique challenge due to their dynamic nature. Scrooge currently struggles to effectively capture changes within SPAs. To improve this, identifying page changes within a Single Page Application (SPA) is crucial. This requires developing logic in the crawling process to detect when a new page loads, which can be achieved by continuously monitoring the current URI and its history to recognize adjustments triggered by AJAX requests. Additionally, exploring alternative storage solutions beyond HAR files for capturing the dynamic state of SPAs is necessary. Evaluating the feasibility and effectiveness of these alternative storage methods will be essential for improving the analysis process.

Addressing these limitations constitutes interesting future work, as well as a more comprehensive evaluation of the tool's precision.

# 6 RELATED WORK

There are various research topics closely related to our work such as change detection, software evolution, as web crawling, and automated black-box testing. In the following we give an overview of the works most related to ours and how we compare against them.

**Change Detection in Web Pages.** Closest to our work are studies and implementations of change detection in web pages (see for instance (Mallawaarachchi et al., 2020) for a survey in this domain). This line of work has been inspired by the practical need to track changes in web sites to get notifications related to important updates (i.e. news, government announces etc.) or potential attackes (i.e. defacements). Today there exist several closed-source change detection services, such as Google Alerts (Google, 2024) and some open source

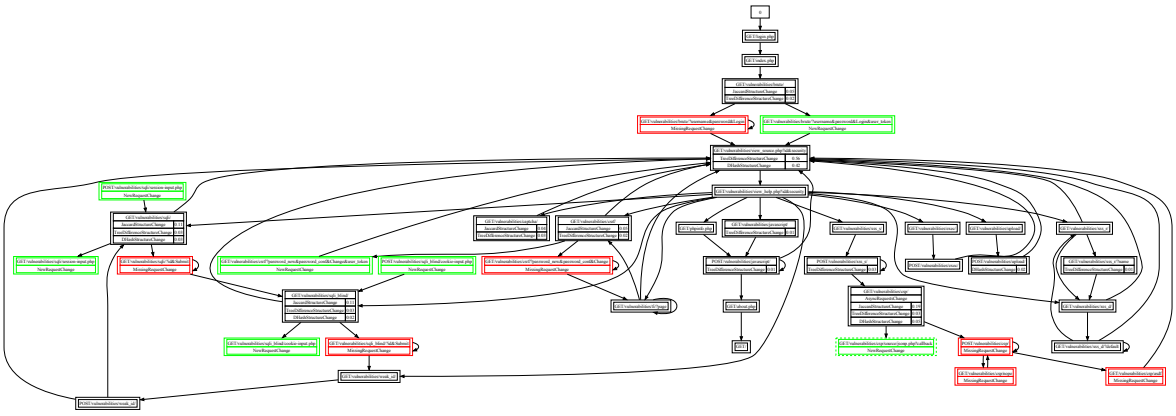


Figure 6: Evaluation Diagram DVWA Crawling with Low and High Level.

ones such as changedetection.io (dgtlmoon, 2024). However note that different from that line of work, we are concerned about changes in an application as a whole, and not only on individual websites. The graph data structure defined in our work is more general and allows one to reason for instance on changes in navigation paths, and thus on more abstract application control flows. Last, our implementation can run locally, which is important in order to guarantee the privacy of the system under test.

**Software Evolution.** Software evolution research provides valuable insights into understanding software changes and predicting future development. D'Ambros et al.'s (D'Ambros et al., 2008) work demonstrates a detailed approach for analyzing software repositories to gain insights into software evolution. Mitropoulos et al.'s (Mitropoulos et al., 2012) work highlights the importance of tracking software evolution to identify security-related bugs. However most works in this domain are white-box approaches that assume the source code is known, whereas our work treats the application under test as a black-box.

**Web Crawling.** As illustrated in our work, effectiveness of web crawling is crucial for identifying changes in web applications. Stafeev and Pellegrino's (Stafeev and Pellegrino, 2024) work provides a comprehensive survey of web crawling algorithms and their effectiveness for web security measurements. In our work, we do not claim a contribution in the crawling domain, since the proposed approach in this paper is designed to be independent of the specific crawler used.

**Automated Black-Box Testing.** Automated black-box testing approaches, such as EvoMaster (Arcuri, 2021) and RestTest-Gen (Corradini et al., 2022), provide techniques for testing RESTful APIs. However, these methods typically require API documentation. This work addresses this limitation by generating API documentation for the identified endpoints, enabling more comprehensive black-box testing.

Overall, the proposed approach extends existing research by combining web crawling, change detection and software evolution concepts, and black-box testing techniques to effectively detect changes in web application functionality, particularly in the context of black-box testing scenarios.

# 7 CONCLUSION

In this work, we present scrooge, a prototype tool designed to identify changes in web application functionality. Scrooge aims to improve security testing efficiency by detecting differences between web page versions. We evaluated scrooge on e-commerce platforms and a security application, demonstrating its ability to reliably detect changes. The effectiveness however, relied on the data generation method. Manual interaction provided the most consistent results, highlighting the need for crawler optimization. Furthermore, scrooge successfully identified various change types, showcasing its applicability across diverse scenarios. Our findings suggest that combining crawling methods could improve coverage, and future work should focus on crawler optimization and single-page application compatibility. Additionally, implementing more differentiation algorithms could increase the number of detectable webpage features. Overall, scrooge demonstrates the potential of automatic change detection for improved security testing efficiency, laying a foundation for further research and development in this domain.

## ACKNOWLEDGEMENTS

DeepL and ChatGPT were used to translate some sections of this work. Gemini was used to shorten the text.

## REFERENCES

(2010). Damn vulnerable web application (DVWA). http://www.dvwa.co.uk/. Accessed: 2024-07-01.

Arcuri, A. (2021). Automated black- and white-box testing of restful apis with evomaster. *IEEE Software*, 38.

Bau, J., Bursztein, E., Gupta, D., and Mitchell, J. (2010). State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE symposium on security and privacy*, pages 332–345. IEEE.

Corradini, D., Zampieri, A., Pasqua, M., and Ceccato, M. (2022). Resttestgen: An extensible framework for automated black-box testing of restful apis.

D'Ambros, M., Gall, H., Lanza, M., and Pinzger, M. (2008). *Analysing software repositories to understand software evolution*.

dgtlmoon (2024). changedetection.io. https://github.com/dgtlmoon/changedetection.io. Accessed: 2024-07-01.

Doupé, A., Cova, M., and Vigna, G. (2010). Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer.

Eriksson, B., Pellegrino, G., and Sabelfeld, A. (2021). Black widow: Blackbox data-driven web scanning. volume 2021-May.

Google (2024). Google alerts. https://www.google.com/alerts. Accessed: 2024-07-01.

Jadeja, M. (2022). Jaccard similarity made simple: A beginner's guide to data comparison. Accessed: 2024-05-26.

Krawetz, N. (2013). Kind of like that. *The Hacker Factor Blog*.

Kressebuch, J. and Büsser, F. (2024). Scrooge source code. https://github.com/secuteer/scrooge.

Mallawaarachchi, V., Meegahapola, L., Madhushanka, R., Heshan, E., Meedeniya, D., and Jayarathna, S. (2020). Change detection and notification of web pages: A survey. *ACM Computing Surveys (CSUR)*, 53(1):1–35.

@maximilianhils, @raumfresser, and @cortesi (2013). mitmproxy/mitmproxy.

Mesbah, A., Deursen, A. V., and Lenselink, S. (2012). Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6.

Mitropoulos, D., Gousios, G., and Spinellis, D. (2012). Measuring the occurrence of security-related bugs through software evolution. In *2012 16th Panhellenic Conference on Informatics*, pages 117–122.

Pellegrino, G. and Balzarotti, D. (2014). Toward black-box detection of logic flaws in web applications. In *NDSS*, volume 14, pages 23–26.

PrestaShop (2024). Prestashop. https://prestashop.com/. Accessed: 2024-07-01.

Stafeev, A. and Pellegrino, G. (2024). Sok: State of the krawlers-evaluating the effectiveness of crawling algorithms for web security measurements.

Williams, M. A., Dey, S., Barranco, R. C., Naim, S. M., Hossain, M. S., and Akbar, M. (2018). Analyzing evolving trends of vulnerabilities in national vulnerability database. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3011–3020. IEEE.

WooCommerce (2024). Woocommerce. https://woocommerce.com/. Accessed: 2024-07-01.

xlwings (2024). jsondiff. https://github.com/xlwings/jsondiff. Accessed: 2024-07-03.

# APPENDIX

Table 5: Found Request Identifiers per Snapshot in DVWA.

| URLs | Manual | BlackWidow | CrawlJax |
|---|---|---|---|
| GET/ | X | X | X |
| GET/about.php | X | X | |
| GET/index.php | X | X | X |
| GET/instructions.php?doc | | X | |
| GET/login.php | X | X | X |
| GET/phpinfo.php | X | X | |
| GET/vulnerabilities/brute/ | X | X | |
| GET/vulnerabilities/brute/?username&password&Login | X | X | |
| GET/vulnerabilities/captcha/ | X | X | |
| GET/vulnerabilities/csp/ | X | X | |
| POST/vulnerabilities/csp/ | X | | |
| GET/vulnerabilities/csp/nope | X | | |
| GET/vulnerabilities/csrf/ | X | X | |
| GET/vulnerabilities/csrf/?password_new&password_conf&Change | X | | |
| GET/vulnerabilities/exec/ | X | X | |
| POST/vulnerabilities/exec/ | X | | |
| GET/vulnerabilities/fi/?page | X | X | |
| GET/vulnerabilities/javascript/ | X | X | |
| POST/vulnerabilities/javascript/ | X | | |
| GET/vulnerabilities/sqli/ | X | X | |
| GET/vulnerabilities/sqli/?id&Submit | X | | |
| GET/vulnerabilities/sqli_blind/ | X | X | |
| GET/vulnerabilities/sqli_blind/?id&Submit | X | | |
| GET/vulnerabilities/upload/ | X | X | |
| POST/vulnerabilities/upload/ | X | X | |
| GET/vulnerabilities/view_help.php?id&security | X | | |
| GET/vulnerabilities/view_source.php?id&security | X | | |
| GET/vulnerabilities/weak_id/ | X | X | |
| POST/vulnerabilities/weak_id/ | X | | |
| GET/vulnerabilities/xss_d/ | X | X | |
| GET/vulnerabilities/xss_d/?default | X | | |
| GET/vulnerabilities/xss_r/ | X | X | |
| GET/vulnerabilities/xss_r/?name | X | | |
| GET/vulnerabilities/xss_s/ | X | X | |
| POST/vulnerabilities/xss_s/ | X | X | |
| 35 URLs | 34 URLs<br>97.1% | 23 URLs<br>65.7% | 3 URLs<br>8.6% |