

# SWaTEval: An Evaluation Framework for Stateful Web Application Testing

Anne Borchering<sup>1,3</sup> <sup>a</sup>, Nikolay Penkov<sup>1</sup> <sup>b</sup>, Mark Giraud<sup>1</sup> <sup>c</sup> and Jürgen Beyerer<sup>1,2,3</sup>

<sup>1</sup>Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB, Karlsruhe, Germany

<sup>2</sup>Vision and Fusion Laboratory (IES), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

<sup>3</sup>KASTEL Security Research Labs, Karlsruhe, Germany

**Keywords:** Web Application Testing, Stateful Testing, Blackbox Testing, State Machine Inference, Fuzzing, Clustering.

**Abstract:** Web applications are an easily accessible and valuable target for attackers. Therefore, web applications need to be examined for vulnerabilities. Modern web applications usually behave in a stateful manner and hence have an underlying state machine that determines their behavior based on the current state. To thoroughly test a web application, it is necessary to consider all aspects of a web application, including its internal states. In a blackbox setting, which we presuppose for this work, however, the internal state machine must be inferred before it can be used for testing. For state machine inference it is necessary to choose a similarity measure for web pages. Some approaches for automated blackbox stateful testing for web applications have already been proposed. It is, however, unclear how these approaches perform in comparison. We therefore present our evaluation framework for stateful web application testing, SWaTEval. In our evaluation, we show that SWaTEval is able to reproduce evaluation results from literature, demonstrating that SWaTEval is suitable for conducting meaningful evaluations. Further, we use SWaTEval to evaluate various approaches to similarity measures for web pages, including a new method based on the euclidean distance that we propose in this paper. These similarity measures are an important part of the automated state machine inference necessary for stateful blackbox testing. We show that the choice of similarity measure has an impact on the performance of the state machine inference regarding the number of correctly identified states, and that our newly proposed similarity measure leads to the highest number of correctly identified states.

## 1 INTRODUCTION

With the widespread use of Web Applications (WAs), it is necessary to expose and eliminate as many vulnerabilities in WAs as possible to ensure a minimal attack surface. This applies to WAs such as web shops as well as WAs provided by industrial devices for monitoring and configuration purposes. Especially for these industrial settings, a thorough test for vulnerabilities is necessary, since a successful attack on the WA can lead to a production stop or even physical harm (Pfrang et al., 2019). Stateful blackbox WA testing is crucial to a thorough test. In *blackbox* tests, information regarding the internals of the tested WA is unavailable. *Stateful* refers to tests that take the internal state of the WA into consideration. This provides

the possibility to conduct tests in an efficient and purposeful way (Doupé et al., 2012). As previously mentioned, however, a blackbox setting does not provide explicit information about the internal state machine of the WA. As a consequence, identifying the underlying state machine of the WA by different means is inevitable in a blackbox setting.

Various approaches to automatic state machine inference exist in the literature. For example, Raffelt et al. (Raffelt et al., 2005) suggest a general automata learning approach, while Doupé et al. (Doupé et al., 2012) propose a method for a WA-specific state inference. In general, the state machine of a WA is learned by looking for changes in the behavior of the WA when it is provided with different inputs. To identify changes in the behavior of the WA, we measure the similarity between the web pages presented by a WA with the help of a similarity measure. With this, we can determine whether the internal state of the WA has changed and what caused the state transition.

<sup>a</sup>  <https://orcid.org/0000-0002-8144-2382>

<sup>b</sup>  <https://orcid.org/0000-0002-5421-4253>

<sup>c</sup>  <https://orcid.org/0000-0002-2972-2758>

The aforementioned challenges can be solved with a set of different approaches. To evaluate the approaches in an individual or a combined manner, a modular framework is necessary. Moreover, an easily understandable Evaluation Target is needed to determine the performance and impact of the approaches.

The contribution of our work to the domain of Stateful Web Application Testing (SWAT) can be summarized as follows:

- We design and implement a modular framework (SWaTEval) to analyze different approaches for the multitude of challenges of stateful WA testing
- We design and implement a manageable and stateful Evaluation Target
- We implement various detection approaches with the help of various similarity measures and evaluate them using SWaTEval

Our evaluation is twofold. On the one hand, we perform a qualitative and quantitative evaluation of SWaTEval itself. This includes reproducing of results presented by a large study in literature (Yandrapally et al., 2020). We show that the results produced with SWaTEval are coherent with the results from this study, suggesting that SWaTEval produces meaningful results. On the other hand, we use SWaTEval to perform a new evaluation regarding the choice of similarity measure for stateful blackbox WA testing. We show that the choice of similarity measure has an impact on the state machine inference and that the similarity measure based on the euclidean distance leads to the best results.

## 2 BACKGROUND

State-awareness is generally needed to maximize test coverage of a WA (Doupé et al., 2012). Behind the requirement to perform state-aware testing lies the following intuition: If a testing tool is unaware of the existing states, it might skip some of them while testing and fail to reveal vulnerabilities in the missed states. This is true for different meanings of the word *state* in literature. Similar to Doupé et al., we consider the WA state as the underlying internal state. For example, the WA's internal state for an authenticated user usually differs from that of a non-authenticated user. In contrast, the general case of the WA showing different pages to the user does not correspond to different internal states. In the following, we will always refer to the internal state of the WA when using *state*.

Additionally, we use the following terminology. The target that should be tested and is used for our evaluation is a WA in its full form and functionality.

We refer to it as *Evaluation Target* or simply *target*. A WA consists of various *web pages* like a login page, or a register page. *Endpoints* expose functionality and allow communication with a WA. Generally, they point to web pages of a WA.

Since a blackbox WA does not reveal information about its internals directly, we need to infer the underlying state machine by communicating with the WA. There are various approaches to learn a state machine in a blackbox setting (e.g. (Raffelt et al., 2005; Doupé et al., 2012)). Most of them are based on the assumption that a state transition is present if a request on an Endpoint returns a response that is different from previously observed responses to the same request on the same Endpoint. We call a request-response pair an *Interaction* with the WA.

In general, it is necessary to decide which Endpoints of the WA should be tested. Similar Endpoints have to be clustered properly and omitted. For this, a method for clustering of similar Endpoints is necessary. Lastly, the inferred state machine has to be pruned by detecting similar states and merging them.

Summarized, the process of automatically learning the state machine of a WA in a blackbox setting consists of the following challenges: (I) cluster similar Endpoints, (II) detect a state change, and (III) cluster similar states (Doupé et al., 2012). Various approaches can be used for each of these challenges, which we discuss in more detail in Section 5.2. Especially, different approaches can be taken to define the distance between two Endpoints or states (or their similarity). To understand the impact of these choices, an evaluation framework for SWAT is necessary. This framework needs to focus on the state detection capabilities and provide an Evaluation Target with manageable complexity. In this work, we present SWaTEval, which aims to fulfill the need for a general evaluation framework for SWAT.

## 3 RELATED WORK

Our work is located in the domain of SWAT while also focussing on similarity measures for web pages. The following paragraphs give an overview of related work in these two areas.

**Stateful Testing of Web Applications.** SWAT is a topic that has been around for some time and is, nevertheless, still relevant. Doupé et al. proposed a first approach to include the underlying state of the tested WA into testing and fuzzing (Doupé et al., 2012). However, the authors chose a non-modular approach for their implementation, and their approach has not

yet been compared to other approaches that include the underlying state. Since then, various approaches for stateful WA testing and crawling have been proposed, for which Hassanshahi et al. provide a recent overview (Hassanshahi et al., 2022). Moreover, a recent work proposes a framework for SWAT (Drakonakis et al., 2020), but it only focuses on a single part of SWAT, namely authentication and authorization. To our knowledge, there is no work besides ours that aims to provide a modular framework to evaluate different aspects of the SWAT process.

Several intentionally vulnerable WAs and benchmarks exist that can be used as evaluation targets for the evaluation of WA testing. Some of the most prominent ones are WackoPicko (Doupé et al., 2010), JuiceShop<sup>1</sup>, XVWA<sup>2</sup>, DVWA<sup>3</sup>, and OWASP Benchmark<sup>4</sup>. These evaluation targets mainly focus on the vulnerabilities that a test might find. Although the discovery rate of these vulnerabilities can be used as an overall performance metric, a more specific evaluation target is necessary to understand the impact of stateful testing methods on the test performance. For this, it is particularly important that the state machine of the target is known and manageable in size.

There are several works focusing on different aspects of WA testing that are orthogonal to our approach. Borcharding et al. present an approach to improve the performance of WA fuzzers transparently by injecting additional information to the fuzzer’s traffic using a proxy (Borcharding et al., 2020). We use a similar approach to provide an interface for existing fuzzers in SWaTEval. Li et al. present an approach to test stateful web services based on a state machine. They derive the state machine from a specification of the web service provided in the Web Services Description Language (Li et al., 2018). In contrast, we build a state machine of the underlying WA based on the blackbox HTTP communication with the WA itself. Other approaches focus on stateful testing of the network protocol implementation itself (Aichernig et al., 2021; Pferscher and Aichernig, 2022). In contrast, our work focuses on testing the interface with which a user interacts directly. In contrast to the blackbox testing used in this work, another approach to dynamic testing is to conduct gray-box testing, which requires additional information about the WA under test (Gauthier et al., 2021).

**Similarity of Web Pages.** As mentioned in Section 1, it is essential to calculate the distance (or

<sup>1</sup><https://github.com/juice-shop/juice-shop>

<sup>2</sup><https://github.com/s4n7h0/xvwa>

<sup>3</sup><https://github.com/digininja/DVWA>

<sup>4</sup><https://owasp.org/www-project-benchmark/>

similarity) between two web pages when performing SWAT. Since we focus our evaluation on similarity measures, we give a short overview of related work in this paragraph. Lin et al. propose a similarity measure based on the similarity of links on the web pages (Lin et al., 2006). Other works apply the Levenshtein distance to calculate the distance between two web pages (Popescu and Nicolae, 2014; Mesbah et al., 2008). In contrast, Doupé et al. use a distance metric based on a prefix tree (Doupé et al., 2012). More recent works propose to measure the similarity between web pages by using their content, such as input fields (Lin et al., 2017), or buttons, anchors, and images (Alidoosti et al., 2019). Yandrapally et al. present an extensive study on the performance of different similarity measures based on various real world WAs and nine open source WAs (Yandrapally et al., 2020). Note that the understanding of *state* used by Yandrapally et al. is different compared to ours. We consider the internal state of the WA, similar to Doupé et al. (see Section 2). In contrast, Yandrapally et al. consider the dynamic webpage of the WA as the state. Nevertheless, the concepts of web page similarity described by Yandrapally et al. can be directly transferred to our work, and we use the results of their extensive work for comparison purposes in Section 6.2.

Oliver et al. present a method for generation of a locality-sensitive hash called Trend Micro Locality Sensitive Hash (TLSH) (Oliver et al., 2013). When calculating the TLSH of an input, small changes in the input lead to small changes in the output hash. This property is beneficial since it provides a method to represent data in a compressed way. Moreover, it allows the calculation of meaningful distances between data which is represented in this fashion. Oliver et al. define a distance measure on TLSH which approximates the hamming distance. We choose TLSH and the corresponding distance measure (TLSH Score) as one of the possibilities to represent web pages and calculate similarities (see section 5.2).

## 4 FUNDAMENTALS

As a basis for the following sections, we present our research goals, our methodology, and the requirements we used for our approach in the following sections.

### 4.1 Research Statements

With our research, we aim to (I) design and implement an evaluation framework for SWAT and (II) conduct an evaluation and analysis of the different simi-

larity measures and thus also verify the usefulness of SWaTEval regarding its evaluation capabilities. We formulate the following two research questions.

- RQ1:* Can we reproduce the results of Yandrapally et al. by using SWaTEval?
- RQ2:* How do the different approaches for page similarity measurement influence the number of correctly detected states?
- RQ3:* Which combination of similarity measures performs better in terms of the number of correctly detected states?

## 4.2 Methodology

To achieve our previously stated research goals, we first infer requirements from literature and formulate our requirements for a framework for the evaluation of SWAT. Afterwards, we design and implement a framework that meets these requirements. Finally, we evaluate our resulting framework qualitatively by using two different approaches. On the one hand, we evaluate SWaTEval by using it to reproduce the results of Yandrapally et al. (Yandrapally et al., 2020). We thus show that SWaTEval is suited to produce meaningful results. On the other hand, we use SWaTEval to evaluate different web page similarity measures. We show thus that SWaTEval can successfully be used to conduct evaluations in the domain of SWAT and that the similarity measure based on the Euclidean distance performs best.

## 4.3 Requirements

As a basis for our design and implementation, we gathered requirements for an evaluation framework by investigating existing work. We used insights from Doupé et al. (Doupé et al., 2012) and Hassanshahi et al. (Hassanshahi et al., 2022) as well as our own analysis of existing tools and frameworks. In the following, we present the resulting requirements for the framework (*FR*) and the Evaluation Target (*TR*).

- FR1 Modularity:* The state machine inference and testing approaches are modular and can easily be modified or replaced.
- FR2 Interaction:* The crawling and fuzzing of the framework can work in an interlaced fashion.
- FR3 Traceability:* All generated data can be traced back and reproduced easily.
- TR1 Similar Pages:* The Evaluation Target has similar (but not equivalent) web pages that should be treated as the same web page.

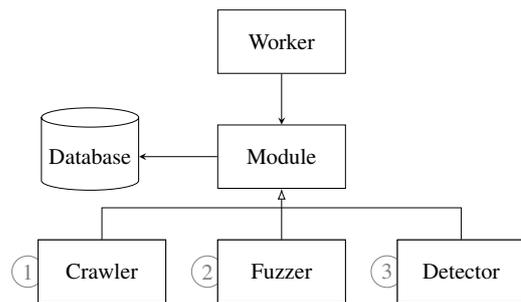


Figure 1: Overview of SWaTEval showing a modular approach regarding the Modules and the execution logic. The Modules are run in the order given by the numbers.

*TR2 Different Pages:* The Evaluation Target has web pages that are significantly different and should be treated as different web pages.

*TR3 Statefulness:* The Evaluation Target provides the possibility to change its underlying state.

*TR4 Complexity:* The state machine is understandable and manageable by humans.

## 5 APPROACH

In this section, we present the approach and the technical details for SWaTEval. First, we give an overview of the framework and the used Modules. We then present the various Modules of SWaTEval in further detail. This includes a description of the new similarity measure we present in this work. At last, we describe the features and details of the Evaluation Target.

### 5.1 SWaTEval

We applied the four principles of object-oriented programming: encapsulation, abstraction, inheritance, and polymorphism. The result is a compact framework that avoids tight coupling, but allows compound configurations with interchangeable Modules.

#### 5.1.1 Overview

As shown in Figure 1, the Modules of SWaTEval are divided into three categories: (I) Crawlers, which are used for state selection and state traversal, (II) Fuzzers, which are used for payload generation and attack evaluation, and (III) Detectors, which are used for pattern detection and information inference. All Modules have access to a centralized database where they can read and write data. This facilitates the interaction of the Modules and the sharing of information between them. Especially, this allows the Crawlers

Table 1: Implemented Modules of the different categories.

| Name of Module      | Source                     |
|---------------------|----------------------------|
| Basic Crawler       | (Doupé et al., 2012)       |
| Dummy Fuzzer        | New work                   |
| External Fuzzers    | (Borcherding et al., 2020) |
| Clustering Detector | New work                   |

and Fuzzers to interact with one another as suggested by literature (Hassanshahi et al., 2022; Doupé et al., 2012). Further details on each module are available in Sections 5.1.2 to 5.1.4.

Further, we encapsulate the execution logic of the Modules in Workers. This creates the advantage that the Modules can be dynamically spawned and run in a distributed way, allowing SWaTEval to scale to for bigger workloads.

In the following, we present the Modules we designed and implemented. This includes approaches from literature as well as our own approaches. Table 1 shows an overview of the Modules as well as their source.

### 5.1.2 Crawlers

The Crawlers' goal is to walk through the WA and find new Endpoints. The information found by the Crawlers is saved in the database and is later used by the Detectors which generate the preliminary state machine. This state machine is then used by the Crawler again to decide which states to explore next.

**Basic Crawler.** This Crawler implements a simple approach for crawling. It navigates the WA to the first available state that is not marked as *explored*, and visits all available Endpoints in this state. To ensure that the WA is always in the selected state, we reset the WA and navigate to the selected state before each crawl iteration. While crawling, the Crawler saves all generated data in the database. In the case of the BasicCrawler, we assume that the WA can be crawled thoroughly and assume convergence when all detected states and Endpoints are visited, and no further states and Endpoints are detected.

### 5.1.3 Fuzzers

The Fuzzers' goal is to generate payloads to test the WA and to analyze whether a vulnerability has been found. In our setting, they are also used to trigger new states of the WA by trying out different inputs. Similar to the Crawlers, the data generated by the Fuzzers is also saved in the database. To differentiate between

the data generated by the Fuzzers and the Crawlers, we flag the data in the database accordingly.

**Dummy Fuzzers.** To validate our concept, we implemented a dummy Fuzzers for the Evaluation Target we used in our evaluation. They mock the functionality of a real Fuzzer by detecting the current state of the WA and executing requests that are known to cause state transitions. In addition, they also send non-fuzzy requests that will not trigger a state transition, mimicking the behavior of a real Fuzzer.

**External Fuzzers.** Since SWaTEval has the goal to integrate different approaches for SWAT, we also provide an interface for external Fuzzers such as Wapiti<sup>5</sup> or Nikto<sup>6</sup>. This configuration is based on the work by Borcherding et al. (Borcherding et al., 2020). To interact with the external Fuzzer, we implement a Fuzzer-specific strategy that starts the external Fuzzer and intercepts its traffic using the MITM proxy<sup>7</sup>. This allows us to inject state data in the requests of the external fuzzer such as cookies and headers, and simultaneously retrieve the responses of the WA.

### 5.1.4 Detectors

The Detectors compare the output of similar requests before and after fuzzing to infer information on the state machine. The intuition behind this decision is the following. If a response to a request differs from the responses that we received to the same request earlier, a state change has likely happened between those two requests. The detection approach presented below is based on this assumption and uses it to infer the state machine.

**Clustering Detector.** Our newly proposed approach is based on clustering. The clustering algorithm uses Endpoints, Interactions, and States as input, which we will denote as *content* in the following. For this Detector type, the content is represented by a locality sensitive hash. To measure the similarity of two content entries, a distance score between their respective hashes is calculated. The clustering algorithm first calculates this distance, and then clusters the web pages accordingly. The detection of a new cluster is interpreted as a relevant change in the content, which represents an anomaly. This anomaly can indicate a state transition, a state collapsing, or the existence of an unseen Endpoint. The whole method

<sup>5</sup><https://wapiti-scanner.github.io/>

<sup>6</sup><https://github.com/sullo/nikto>

<sup>7</sup><https://mitmproxy.org/>

is described in further detail in the next section (Section 5.2).

## 5.2 Clustering Based Detection

This section provides deeper insight into the inner workings of our clustering based Detectors. Our approach aims to achieve the same goals as the approach presented by Doupé et al.. Doupé et al. represent a web page as a *page link vectors* which incorporate information on the DOM path, anchors, forms, parameters, and values (Doupé et al., 2012). In contrast, we create a latent representation of the data and apply direct clustering methods on it. With these clustering based methods, we aim to achieve a more adaptive analysis that adjusts the clustering results to the newly available data. We represent *content* generated in our framework with the help of the locality sensitive hash called TLSH (see Section 3). To be more specific, we apply this hash to different parts of the available content as shown in the following.

### 5.2.1 Data Types

To represent the *content* of the WA and make use of it during SWAT, we define three data types: *Endpoint*, *Interaction*, and *State*. Each of these data types has a hash attribute that contains the compact TLSH representation of the contained data. Furthermore, we define for our framework the data types Request and Response to represent HTTP Requests and HTTP Responses, respectively.

**Endpoints** contain information about web addresses found while scanning a WA. They include an address location, the parameters found in the URL and the body, the HTTP method, the DOM location, the corresponding state, and the source. Here, the source is defined as the Interaction that created a Response which contained the corresponding Endpoint. To calculate the hash of an Endpoint, we concatenate its URL, parameters, state, and DOM location in a string and apply TLSH to it.

**Interactions** represent the communication with the WA and are put together from an Endpoint, and the corresponding Request, Response, and State. To calculate the hash of an Interaction, we first parse the body of its Response and extract links provided as anchors and forms. Then, we concatenate them with the received response code, URL, and HTTP method, and use the final string to generate a TLSH.

**States** contain information about the Interaction that caused the transition, and paths from other States that lead to the corresponding State. The state hash is calculated by concatenating the related Interaction hashes in a String and using it to generate a TLSH.

### 5.2.2 Clustering Based Detectors

We orient the SWAT challenges, that the Detectors of SWaTEval should solve, on the challenges identified by Doupé et al. (Doupé et al., 2012). The Endpoint-Detector has the goal to decide which Endpoints are to be classified as identical in order to discard duplicated Endpoints and such reduce the time needed for a test. The StateChangeDetector aims to identify state transitions in order to build the state machine. Similar to the EndpointDetector, the StateDetector helps reducing the testing time by deciding which states of the identified state machine are considered equivalent and can be merged. As has been mentioned, the Workers execute the Modules in a sequential order. After a Crawling or a Fuzzing step, the Detectors are run in the following order: (I) EndpointDetector, (II) StateChangeDetector, (III) StateDetector. Each of the Detectors makes use of our clustering approach which we will explain in more detail in Section 5.2.3.

**EndpointDetector.** The EndpointDetector decides which Endpoints should be visited in the upcoming crawling iterations, since not every newly found Endpoint should be visited by the Crawler. Some Endpoints lead to Interactions that contain similar information even if the Endpoint addresses differ. Since duplicate information does not contribute to the state machine inference, we want to avoid interacting with duplicate Endpoints. For example, WAs sometimes provide dynamically generated links on webpages, which point to the same URL with slightly different parameters. Such is the case in WackoPicko, an intentionally vulnerable WA (Doupé et al., 2010). This WA contains a calendar page that always generates a link pointing to the same calendar page, but with the next date selected. With this, it essentially creates a never ending chain of Endpoints which will get a crawler stuck in the corresponding state.

We solve this issue by calculating clusters of Endpoint hashes. We consider Endpoints in the same cluster to be the same. By labelling the Endpoints accordingly, the EndpointDetector helps the Crawler to filter out identical Endpoints when selecting an Endpoint for the next Interaction. Thereby, we ensure that the subsequently visited Endpoints are considerably different from any previously visited Endpoint and the crawling procedure will eventually converge.

**StateChangeDetector.** The purpose of the StateChangeDetector is to detect state transitions. To achieve this, it clusters all Interaction hashes generated when visiting an Endpoint and looks for outliers.

As stated earlier, we assume that a state change has occurred when the same Request results in a different Response than in previous Interactions. To decide whether a Response is different from the ones seen before, we apply our clustering approach again. We calculate clusters using data from all Interactions that have been carried out on the same Endpoint. If the Interactions on the same Endpoint have more than one cluster, we assume that a state change has happened. Based on this observation, we infer the state machine as proposed by Doupé et al. (Doupé et al., 2012), and a new state is created. As soon as the Fuzzer creates a new Interaction, the StateChange-Detector compares this Interaction with Interactions having the same Endpoint and being created by the Crawler. This ensures that the StateChangeDetector can detect a state transition on time. As a result, it ensures that future Interactions will be mapped to the correct state.

**StateDetector.** An additional important step in the state machine inference is the merging of similar states (Doupé et al., 2012). This pruning operation is crucial, since duplicate States will slow down the scanning process without adding new information.

The StateDetector runs after the StateChange-Detector and identifies states that appear equivalent. The StateDetector allows only the earliest state to persist and labels the identical ones as duplicates. This ensures that the duplicate states are filtered out in the following crawling steps.

Again, we use the clustering approach to solve this challenge and calculate the clusters of states. If multiple States are considered to be in the same cluster, we assume that these states are duplicates. By labeling them as duplicates, we allow only the initial distinct state to persist. Additionally, we ensure that no previous state graph paths are broken by appending the paths of the duplicate states to the persisted state. We also transfer the Interactions and Endpoints of the duplicate states accordingly.

### 5.2.3 Clustering

For the Detectors, we apply clustering on TLSH hashes to find patterns in the content. In the following, we describe an issue coming from the requirements of TLSH as well as our solution of augmenting the input data with padding. Afterwards, we describe how we preprocess the hashes for our clustering algorithms.

**Padding.** By design, the TLSH algorithm requires a minimum input of 50 bytes to generate a hash. To fulfil this requirement, we use a random but fixed

padding for the inputs to the TLSH algorithm. However, augmenting the TLSH input data with random data might dilute the information of the actual data and make the clustering result less reliable. Also, the concrete choice of the padding may influence the clustering results. We acknowledge the impact of our padding choice during our evaluation (Section 6).

**Preprocessing.** We propose the following two ways to preprocess the TLSH hashes in order to use them as an input for the clustering algorithms:

**Integer Representation.** Since the TLSH algorithm always results in a hash with 72 characters, we can interpret the hash as a feature vector in  $\mathbb{R}^{72}$ . Each entry of the vector is converted from a character to an Integer by using its ASCII representation. With this, we receive a numerical representation of the TLSH hash which can be used for clustering.

**Text-based Distance.** The second approach is to directly apply a text distance on the hashes and create a distance matrix, which then can be used as an input for the clustering algorithm. In this case, we use the TLSH Score proposed by the authors of TLSH, and apply it pairwise on the content hashes to create the distance matrix. The TLSH Score approximates the Hamming distance and is calculated as the distance of every two bits. Additionally, the score punishes large differences with higher distance values (Oliver et al., 2013).

**Clustering Algorithm.** We choose DBSCAN as the algorithm for the clustering since it fits the properties of our clustering task (Ester et al., 1996). It is not dependent on the cluster shape and count, it is capable of detecting densely positioned clusters, and robust to noise and outliers. DBSCAN has two input parameters that need to be chosen.  $\epsilon$  defines the maximal distance between two samples in a neighbourhood, and  $minPts$  determines the minimal number of neighbours for a core point sample. We set the  $minPts$  parameter to 1, since each Endpoint, Interaction, or state can be a cluster on its own. This is in line with Schubert et al., who state that  $minPts$  should be chosen based on domain knowledge (Schubert et al., 2017).  $\epsilon$ , however, is optimized with the help of the *Silhouette Score* (Shahapure and Nicholas, 2020) every time we run the clustering procedure.

### 5.2.4 Evaluation Target

Complementary to SWaTEval, we created an Evaluation Target, which is used to evaluate the performance

of the Modules. Similar to Doupé et al. we implemented a server-side rendered WA, meaning that the DOM elements for each request are generated in the back end and sent in the response each time (Doupé et al., 2012). Based on the requirements defined in Section 4.3, we integrated the following use cases in the Evaluation Target.

**User Login.** The Evaluation Target supports login functionality including a normal user and an admin.

**Chained Links.** The Evaluation Target includes an Endpoint that takes a number as input parameter and contains a link that points to the same page with an incremented number. This mimics the previously mentioned case of a Crawler getting stuck in a feedback loop of links (see Section 5.2). This challenges the EndpointDetector, which should detect the Endpoints generated after the first visit as duplicates.

**Pages with Links and Content.** The Evaluation Target includes pages that can have links or content which can be dynamically generated or constant. All four possible combinations exist and can be used to evaluate the StateChangeDetector and the EndpointDetector.

**State Machine.** Furthermore, the Evaluation Target includes additional states which are accessible only when a proper keyword is given (see Figure 2). The state machine can be used to test the features of a StateChangeDetector or a Fuzzer. The state machine also includes a state transition that resets the state machine to the initial state. This state transition creates a challenge for the StateChangeDetector, as the StateChangeDetector has to merge the new state and the initial one. Failing to do so will replicate the state machine indefinitely.

## 6 EVALUATION

Our goal in evaluating SWaTEval is to verify the functionality of the implemented Modules and their interoperability. To achieve this, we aim to ensure that the requirements defined in Section 4.3 are fulfilled. Moreover, we evaluate different similarity measures and examine the performance of multiple framework configurations. We conduct our experiments on a machine with 128GB of RAM and the following CPU: Intel(R) Xeon(R) E5-1650 v3 @ 3.50GHz. Note that the used hardware exceeds the needs of SWaTEval.

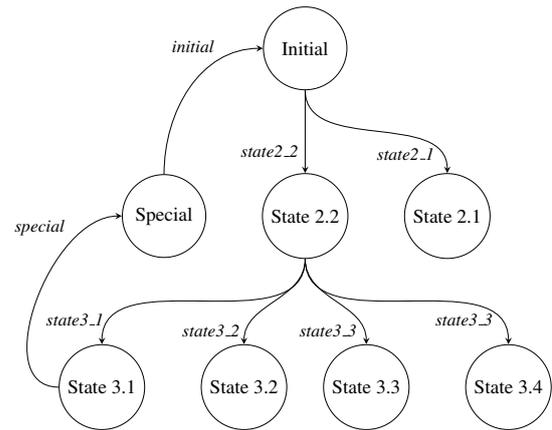


Figure 2: State machine of the Evaluation Target. Keywords needed to access a new state are presented in italic.

### 6.1 Qualitative Evaluation

As the first step of our evaluation, we analyze how SWaTEval fulfills the requirements for a framework enabling SWAT evaluations as stated in Section 4.3.

Dividing the framework into Modules with the same base functionality and encapsulating their individual policy allows us to hide complexity and create abstract workflows. By doing so, we ensure that SWaTEval covers *FR1 Modularity*.

Interaction between the Modules of SWaTEval (*FR2 Interaction*) is achieved by establishing the database as a core point of information exchange. The Modules are run in a sequential order, and they execute their logic only if the conditions specified in their policy match the current state of the framework. For example, a Fuzzer will only generate a fuzzing request if the Crawler and the Detectors have marked the current state as fully explored. In order to satisfy *FR1 Modularity*, we implement the policy of each Module separately, which allows us to avoid tight coupling and still have interoperating Modules.

*FR3 Traceability* aims to make the influence and behavior of the different Modules and their interlaced functionality visible. By separating the generated data from the Modules, we allow for a centralized analysis of the current state of SWaTEval. In addition, this allows for manual data editing during runtime, enabling experiments with edge cases and a deeper analysis of the Modules' behavior. To trace the various instances of states, Endpoints, and Interactions, each instance is identified by its hash. The database contains all relevant information on the behavior of the Modules and as such the content of the database can be stored and used for documentation or analysis purposes. With this, SWaTEval fulfills *FR3 Traceability*.

The Evaluation Target incorporates different web

pages (see Section 5.2.4 for details). First, it includes web pages with similar content which should be classified as the same web page (*TR1 Similar pages*). This holds, for example, for `/views/const-content/const-links/random-page`. Besides, our Evaluation Target includes pages with different content, such as `views/dynamic-content/const-links/random-page` (*TR2 Different pages*). Moreover, it implements the possibility to change the underlying state of the WA by (I) a user login, and (II) the artificial state machine controlled by different keywords, as shown in Figure 2 (*TR3 Statefulness*). The state machine has eight states and the transitions of the states can be retraced easily. With this, our Evaluation Target fulfills the requirements *TR1 - TR4*.

### 6.2 Quantitative Evaluation

The goal of our quantitative evaluation is to analyze whether SWaTEval can be used to reproduce results from literature. We compare the results regarding similarity measures based on our Evaluation Target with results from Yandrapally et al. (Yandrapally et al., 2020). Note that Yandrapally et al. use a different terminology for the state of a WA, as each individual web page is considered a state. This definition maps to our understanding of an Endpoint. As a result, the evaluation of similarity measures for *states* by Yandrapally et al. can be compared to our evaluation of similarity measures for *Endpoints*.

We run various experiments for our comparison of the similarity measures. First, we obtain the baseline data by applying the similarity measures provided by us (Section 5.2.3) and Yandrapally et al. to the targets given by Yandrapally et al.. Second, we apply the same similarity measures to our Evaluation Target. As a result, we obtain insights on the performance of different similarity measures for the large study of Yandrapally et al. as well as on our Evaluation Target.

Figure 3 shows the comparison of the accuracy of similarity measures calculated on the targets used by Yandrapally et al. and our Evaluation Target. The x-axis presents the similarity measures and the data representation they are based on. The y-axis displays the achieved accuracy values of the different similarity measures (see Section 5.2.3). Since the data by Yandrapally et al. consists of more diverse WAs, it is expected that the similarity measures show overall less accuracy. Our results support this expectation.

Except for the differences in absolute values, our results show the same trends and relative results for the similarity measures. The TLSH Score results in the lowest accuracy, while Levenshtein distance on DOM data scores second best. The highest accu-

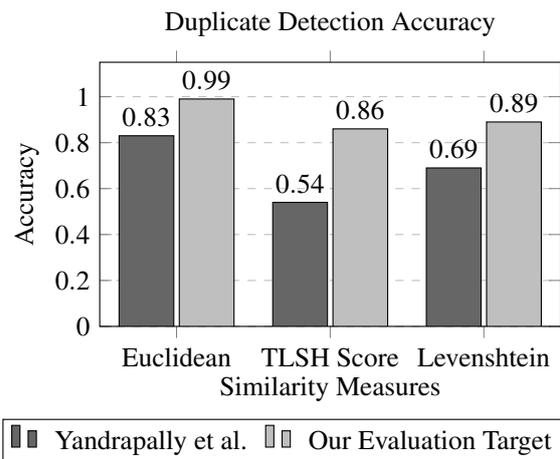


Figure 3: Duplicate detection accuracy of various similarity measures based on data by Yandrapally et al. and our Evaluation Target. The relative performance of the similarity measures is the same.

racy is achieved by our proposed method using Euclidean distance and the integer vector representation of TLSH. The results show that the performance is maximized both on the data from Yandrapally et al. and from our Evaluation Target.

Our evaluation shows that our artificial Evaluation Target leads to similar results as the study conducted by Yandrapally et al.. Therefore, the conclusion can be drawn that our Evaluation Target successfully incorporates the relevant features of real-world WAs. With respect to *RQ1*, our results suggest that our Evaluation Target achieves similar results to those presented in literature.

### 6.3 Similarity Measures

In contrast to the evaluations presented in the previous subsections, which aim to evaluate SWaTEval itself, this subsection presents an evaluation of various similarity measures by using SWaTEval. Nevertheless, this evaluation also shows that SWaTEval can successfully be used for evaluations of different approaches for SWAT. The goal of this evaluation is to analyze the influence of the choice of similarity measure on the results of the state machine inference.

Table 2 shows the similarity measures we considered for the Detectors in our Evaluation. For each of the Detectors, we choose either Euclidean or Levenshtein as similarity measure (see Section 5.2.3), and use them to calculate the distance between content entries represented as TLSH. For the StateChange-Detector, we additionally evaluate calculating the Levenshtein distance of the body of the Responses.

Thus, we end up with  $2 \cdot 2 \cdot 3 = 12$  configurations

Table 2: Similarity measures used for the evaluation.

| ID          | Representation   | Sim-Measure |
|-------------|------------------|-------------|
| TLSH Score  | TLSH as String   | TLSH Score  |
| Euclidean   | TLSH as Integers | Euclidean   |
| Levenshtein | Response Body    | Levenshtein |

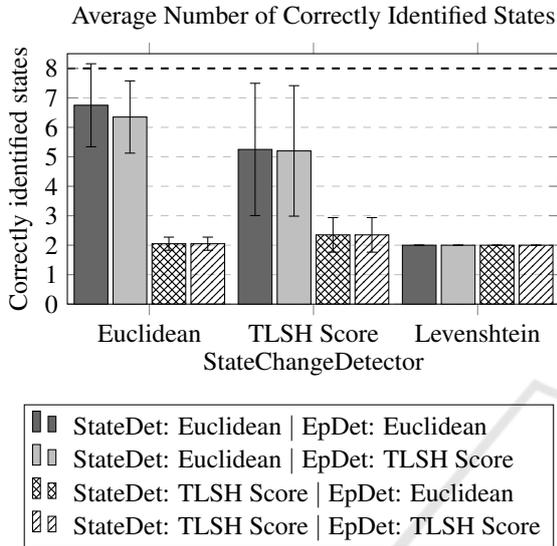


Figure 4: Means average number of correctly identified states for the different Detector configurations for the StateChangeDetector, the StateDetector (StateDet), and the EndpointDetector (EpDet). The euclidean similarity measurement leads to higher numbers.

which we run 20 times each. This repeated execution accounts for one additional parameter which is the padding for the TLSH which might affect the performance of our Modules (see Section 5.2.3).

For each of the configuration combinations, we analyzed the count of correctly identified states, the count of Endpoints, and the count of Interactions. Figure 4 shows the count of correctly identified states, which is the most important metric to measure the performance of the configurations. The configurations for the StateChangeDetector are shown on the x-axis. The bars correspond to the different combinations for StateDetector and EndpointDetector, and the error bars show the standard deviation calculated on the data from the 20 runs of each configuration. The correct number of states in the state machine is marked with the dashed black horizontal line.

The graph in Figure 4 shows three main findings: (I) The euclidean similarity measure leads to the highest number of correctly identified states, (II) the usage of TLSH increases the variance of the results, and (III) the choice of the similarity measure for the EndpointDetector is the choice which has the lowest impact on

the number of correctly identified states.

The first finding can especially be seen for the Euclidean StateChangeDetector (bars on the left), and for the Euclidean StateDetector (plain bars). Overall best results are achieved by the configuration where all three Detectors use the Euclidean similarity measure. Especially, the Euclidean distance is better suited as a similarity measure than the TLSH Score developed specifically for TLSH. This finding is interesting since literature shows that distance measures based on the Manhattan distance are better suited for data with higher dimensions than the Euclidean distance (Aggarwal et al., 2001). In our case of 72 dimensions, one would assume that the TLSH Score, which approximates the Hamming distance, would behave similar to the Manhattan distance and such perform better than the Euclidean approach. A possible explanation might be the different nature of the data. Aggarwal et al. use synthetic data drawn from a normal distribution as well as data taken from the UCI machine learning repository. In contrast, we use TLSH as input for the distance measure and the clustering. We believe that an analysis of the classification performance on various types of data would be an interesting starting point for future work, but we consider it to be out of scope for this paper.

The second finding is that the use of TLSH increases the variance of the results. This can be seen especially in comparison to the Levenshtein StateChangeDetector (bars on the right) which is not using TLSH. The observation that the usage of TLSH leads to a higher variance is expected since it is affected by the padding mentioned before. Note that the Euclidean distance measure is influenced by the padding as well, since it also operates on the TLSH. This shows that the choice of padding indeed has an influence on the identified states.

The third finding concerns the impact of the choice of similarity measure. As a measurement for the impact, we choose to analyze the difference between the minimum and maximum values of correctly identified states from the different configurations. This impact can already be seen in Figure 4. We show these values in Table 3 in a more compact form. For the StateDetector, for example, this table shows the maximum difference of the mean numbers of correctly identified states for all configurations for which the StateDetector is using Levenshtein and Euclidean, respectively. Based on this measurement, the choice of the distance measure for the StateChangeDetector and the StateDetector is more important than the one for the EndpointDetector. This insight is also reflected by the data depicted in Figure 4, where a change in the EndpointDetector still leads to simi-

Table 3: Impact of the choice regarding similarity measures for the different Detectors. The choice for the Endpoint-Detector has the lowest impact.

| Detector            | Detection impact |
|---------------------|------------------|
| StateChangeDetector | 2.30             |
| StateDetector       | 2.46             |
| EndpointDetector    | 0.08             |

lar results, when the plain bars, and respectively the patterned bars, are compared. For a deeper analysis, we also evaluated the number of detected Endpoints and the performed Interactions with the WA. These numbers highly correlate to the number of correctly detected states. In regard to *RQ3*, we can state that applying different distance measurements in the presented approaches do influence the performance of the StateChangeDetector and StateDetector, but only have a small impact on the EndpointDetector.

#### 6.4 Summary

Our evaluation had two objectives. On the one hand, we showed that SWaTEval fulfills the requirements for a SWAT framework (see Section 4.3). In addition, we showed that SWaTEval is able to reproduce results by a study from literature by reproducing the results by Yandrapally et al.. On the other hand, we used SWaTEval to evaluate various similarity measures used for SWAT. We showed that the choice of similarity measures for StateChangeDetector and StateDetector indeed has an impact on the state machine inference and that, for our data, the euclidean similarity measure lead to the best results.

## 7 FUTURE WORK

We created SWaTEval with the goal to create a foundation and initial building block for future SWAT research. On the one hand, SWaTEval is suited to evaluate the impact of various choices for the state detection. First, one could evaluate the impact of the chosen state detection algorithm. E.g., Vaandrager et al. proposed a new approach to automated automata learning which could be evaluated regarding its suitability for WA automata learning by using SWaTEval (Vaandrager et al., 2022). Second, one could approach the question of how much the quality of the inferred state machine influences the performance of the fuzzer. Third, the feature selection for the representation of Endpoints, Interactions, and States could be evaluated and tuned. On the other hand, additional approaches for Detectors could

be implemented and analyzed. A possible approach would be to base new detectors on the anomaly detection capabilities of autoencoders such as presented by Mirsky et al. (Mirsky et al., 2018). In addition, other challenges of stateful fuzzing can be evaluated using SWaTEval, such as the challenge of deciding which state to focus on during the next fuzzing phase (Liu et al., 2022).

The current scope of SWaTEval are server-side rendered WAs. Even though this is still the standard in industrial contexts, many of the modern WAs apply a client-based approach. In the future, SWaTEval could be extended with Detectors and an appropriate Evaluation Target for client-side WAs.

## 8 CONCLUSIONS

In this work, we present SWaTEval, a novel evaluation framework for Stateful Web Application Testing. It consists of modular implementations for Detectors, Crawlers, and Fuzzers, and an Evaluation Target. In our evaluation, we show that (I) the Evaluation Target is a suitable substitute for an evaluation of various real-world WAs, and that (II) SWaTEval is suitable to conduct evaluations. By further using SWaTEval for an evaluation of similarity measures, we show that the choice of similarity measures has an impact on the state machine inference, and that, for our data, our euclidean similarity measure leads to the highest number of correctly identified states. This insight is invaluable for the design and implementation of future SWAT tools. Regarding the limitations of SWaTEval, we would like to highlight two points. First, our Evaluation Target currently only includes HTML web pages and as such does not cover JavaScript based content. Second, SWaTEval currently focuses on server-side rendered WAs and does not support, for example, Progressive Web Applications. Both points represent possible enhancements of SWaTEval that should be addressed in future work.

One of our main goals for SWaTEval is that it will be useful for future evaluations in the domain of Stateful Web Application Testing. Hence, we published the source code of SWaTEval<sup>8</sup>. SWaTEval thus provides a solid basis for future enhancements, implementations, and evaluations.

<sup>8</sup><https://github.com/SWaTEval>

## ACKNOWLEDGEMENTS

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF).

## REFERENCES

- Aggarwal, C. C., Hinneburg, A., and Keim, D. A. (2001). On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer.
- Aichernig, B. K., Muškardin, E., and Pferscher, A. (2021). Learning-based fuzzing of iot message brokers. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 47–58. IEEE.
- Alidoosti, M., Nowroozi, A., and Nickabadi, A. (2019). Blprom: A black-box approach for detecting business-layer processes in the web applications. *Journal of Computing and Security*, 6(2):65–80.
- Borcherding, A., Pfrang, S., Haas, C., Weiche, A., and Beyerer, J. (2020). Helper-in-the-middle: Supporting web application scanners targeting industrial control systems. In *17th International Joint Conference on e-Business and Telecommunications*, pages 27–38.
- Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538.
- 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.
- Draonakis, K., Ioannidis, S., and Polakis, J. (2020). The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1953–1970.
- Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *The Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231.
- Gauthier, F., Hassanshahi, B., Selwyn-Smith, B., Mai, T. N., Schlüter, M., and Williams, M. (2021). Backrest: A model-based feedback-driven greybox fuzzer for web applications. *arXiv preprint arXiv:2108.08455*.
- Hassanshahi, B., Lee, H., and Krishnan, P. (2022). Gelato: Feedback-driven and guided security analysis of client-side web applications. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 618–629. IEEE.
- Li, Y., Sun, Z.-G., and Jiang, T.-T. (2018). An automated test suite generating approach for stateful web services. In *International Conference on Software Analysis, Testing, and Evolution*, pages 185–201. Springer.
- Lin, J.-W., Wang, F., and Chu, P. (2017). Using semantic similarity in crawling-based web application testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 138–148. IEEE.
- Lin, Z., King, I., and Lyu, M. R. (2006). Pagesim: A novel link-based similarity measure for the world wide web. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06)*, pages 687–693. IEEE.
- Liu, D., Pham, V.-T., Ernst, G., Murray, T., and Rubinstein, B. I. (2022). State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 720–730. IEEE.
- Mesbah, A., Bozdog, E., and Van Deursen, A. (2008). Crawling ajax by inferring user interface state changes. In *2008 Eighth International Conference on Web Engineering*, pages 122–134. IEEE.
- Mirsky, Y., Doitshman, T., Elovici, Y., and Shabtai, A. (2018). Kitsune: an ensemble of autoencoders for online network intrusion detection. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.
- Oliver, J., Cheng, C., and Chen, Y. (2013). Tlsh—a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE.
- Pferscher, A. and Aichernig, B. K. (2022). Stateful black-box fuzzing of bluetooth devices using automata learning. In *NASA Formal Methods Symposium*, pages 373–392. Springer.
- Pfrang, S., Borcherding, A., Meier, D., and Beyerer, J. (2019). Automated security testing for web applications on industrial automation and control systems. *at-Automatisierungstechnik*, 67(5):383–401.
- Popescu, D. A. and Nicolae, D. (2014). Determining the similarity of two web applications using the edit distance. In *International workshop soft computing applications*, pages 681–690. Springer.
- Raffelt, H., Steffen, B., and Berg, T. (2005). Learnlib: A library for automata learning and experimentation. In *10th international workshop on Formal methods for industrial critical systems*, pages 62–71.
- Schubert, E., Sander, J., Ester, M., Kriegel, H. P., and Xu, X. (2017). Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21.
- Shahapure, K. R. and Nicholas, C. (2020). Cluster quality analysis using silhouette score. In *2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA)*, pages 747–748. IEEE.
- Vaandrager, F., Garhewal, B., Rot, J., and Wißmann, T. (2022). A new approach for active automata learning based on apartness. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 223–243. Springer.
- Yandrapally, R., Stocco, A., and Mesbah, A. (2020). Near-duplicate detection in web app model inference. In *ACM/IEEE 42nd international conference on software engineering*, pages 186–197.