ReScan: A Middleware Framework for Realistic and Robust Black-box Web Application Scanning

Kostas Drakonakis
FORTH
kostasdrk@ics.forth.gr

Sotiris Ioannidis
Technical University of Crete
sotiris@ece.tuc.gr

Jason Polakis
University of Illinois Chicago
polakis@uic.edu

***Abstract*—Black-box web vulnerability scanners are invaluable for security researchers and practitioners. Despite recent approaches tackling *some* of the inherent limitations of scanners, many have not sufficiently evolved alongside web browsers and applications, and often lack the capabilities for handling the inherent challenges of navigating and interacting with modern web applications. Instead of building an alternative scanner that could naturally only incorporate a limited set of the wide range of vulnerability-finding capabilities offered by the multitude of existing scanners, in this paper we propose an entirely different strategy. We present ReScan, a *scanner-agnostic* middleware framework that *transparently* enhances scanners' capabilities by mediating their interaction with web applications in a realistic and robust manner, using an orchestrated, fully-fledged modern browser. In essence, our framework can be used in conjunction with *any* vulnerability scanner, thus allowing users to benefit from the capabilities of existing and future scanners. Our extensible and modular framework includes a collection of enhancement techniques that address limitations and obstacles commonly faced by state-of-the-art scanners. Our experimental evaluation demonstrates that despite the considerable (and expected) overhead introduced by a fully-fledged browser, our framework significantly improves the code coverage achieved by popular scanners (168% on average), resulting in a 66% and 161% increase in the number of reflected and stored XSS vulnerabilities detected, respectively.**

I. Introduction

Web application scanners play a crucial role for security engineers and developers for uncovering vulnerabilities in applications and patching them in a timely manner. Black-box scanners can be extremely useful since they do not require any *a priori* knowledge of the target application. However, as the web ecosystem continues to evolve at a breakneck pace, modern applications incorporate more complex functionalities, features [32], APIs [42], and client-side code, and therefore need a fully-fledged, modern browser environment for their functionality to be fully exercised and the applications to be accurately tested.

State-of-the-art academic [19], [49], [43] and community-developed scanners [51], [57], [5], [56], which have seen wide recognition, suffer from core limitations that hinder their effectiveness and lead to incomplete scanning, lower coverage and missed vulnerabilities. First, many existing scanners use raw HTTP requests to interact with the application instead of a real browser, thus missing out on dynamically generated DOM content (e.g., new URLs or forms) and asynchronous requests. Second, many scanners are limited to a specific method of navigating the application (e.g., extracting static links and HTML forms) or rely solely on client-side code and events [49]. Applications, however, often make use of both. Moreover, existing tools typically simply replay requests when crawling or fuzzing the application, and do not adhere to the intended and correct execution of steps for moving the application from one state to another. Another major limitation is that while scanners can be configured to log into the application, they typically assume that the authenticated session remains intact for the duration of the scan, which quite often is not the case. In addition, scanners can be prone to false positives and negatives for certain types of vulnerabilities (e.g., XSS) due to their naive approach to verifying successful injections. Finally, since black-box scanners are not context-aware of applications' content and functionalities, they can spend a significant amount of time redundantly testing similar pages.

Recently, Eriksson et al. [25] highlighted some of these problems and the importance of taking them all into account when implementing a web application scanner. These limitations are further evidenced by the fact that certain scanners attempt to tackle *some* of them by offering the ability to be used as a proxy [9], [1], [3] between a user's browser and the application, so as to collect useful information (e.g., event originating requests). However, this is not a robust or effective strategy, as it requires significant manual effort and therefore does not scale, and is inherently unable to address all the limitations. Overall, while *certain* scanners attempt to address these limitations, they either only *partially* address them or only tackle a *subset* of the limitations.

Nonetheless, despite their limitations, the aforementioned tools offer a plethora of different scanning techniques and capabilities which are undoubtedly of great value. Ideally, overcoming these limitations would require redesigning these tools or collecting their individual techniques and re-implementing them from scratch. Unfortunately, this is an unlikely and impractical scenario, as it would require an exorbitant amount of time and engineering effort. Instead, we propose an alternative strategy for leveraging the capabilities of existing (and future) scanners while addressing their limitations.

Specifically, we design and implement ReScan, a scanner-agnostic black-box middleware framework that *transparently* enhances web application scanners and addresses the aforementioned limitations. In more detail, our framework intercepts scanner requests and provides a realistic state-of-the-art orchestrated browser environment with a rich set of additional capabilities (e.g., event triggering, HTTP request tampering). Our system detects new endpoints that reveal further endpoints or trigger asynchronous requests, to construct a navigation model of the target web application, and mirrors the scanner's requests through the browser and the model.

Additional enhancement modules operate concurrently to verify the validity of the authenticated session and re-authenticate if needed, detect *inter-state dependencies* (i.e., submitted values that

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24169
www.ndss-symposium.org

TABLE I: Scanners' features and capabilities.

Feature / System	w3af	wapiti	Enemy of the State	ZAP
Browser support	○	○	○	●
Navigation model	○	○	●	○
Inter-state dependencies	○	○	○	○
Client-side events	○	○	○	●
Authentication	◑	◑	●	●
FP / FN elimination	○	○	○	○
URL clustering	○	○	◑	○

●: feature supported, ◑: partially supported, ○: not supported.

appear on and affect other URLs) and cluster similar pages that would be redundant to audit. In general, ReScan does not require *any* information about the scanner's or app's internals and does not make any assumptions; it receives HTTP requests and attempts to accurately mirror them based on the learned model so as to respect the navigation workflow. This is done inside the browser, to ensure realistic interaction and response rendering.

Our extensive evaluation with state-of-the-art scanners shows that ReScan effectively facilitates the detection of more vulnerabilities, both for benchmark and modern applications, while offering a code coverage improvement between 3% and 935% (168% on average). Moreover, we outline several prominent vulnerability examples that demonstrate the practicality of our different enhancement techniques and also show that ReScan can handle more than a single class of vulnerabilities. While our system induces a considerable performance overhead, due to the numerous techniques it employs and the unavoidable cost of leveraging a fully-fledged modern browser, we show that our URL clustering algorithm can dramatically reduce the total scan time for a representative modern application, resulting in a 6.7x speedup.

In summary, our research contributions are the following:

- We propose ReScan, a novel black-box middleware framework that enhances existing vulnerability scanners by transparently addressing their core limitations. Our system has been open-sourced [12].
- We design a novel URL clustering algorithm that prevents scanners from spending valuable time and resources on testing redundant application endpoints.
- We extensively evaluate our system using popular and state-of-the-art scanners on a rich set of web applications. We have released our applications' Docker images to ensure reproducibility and facilitate further research [11].

II. Challenges and Design Requirements

Implementing a scanner-agnostic middleware framework requires solving numerous technical challenges and overcoming the aforementioned limitations in a way that is *transparent* to the scanner, while also providing functionality enhancements without understanding or tampering with scanners' internals. Essentially, the black-box interaction should be *bidirectional*; the scanner knows nothing about ReScan and vice-versa. Here we outline some of the main challenges that our system tackles.

Inter-state dependencies. Certain types of vulnerabilities, such as stored XSS, are not necessarily triggered directly on the landing page after the payload is delivered. On the contrary, successful exploitation (and detection) might require the scanner to visit a different URL in the application. For instance, consider editing a vulnerable field on a user's account page, which is then triggered when visiting that user's profile page. ReScan needs to uncover and keep track of these *inter-state dependencies* so as to enable detection of said vulnerabilities, and also account for the *order* in which a scanner fuzzes potential injection points and visits URLs they might affect.

Authentication. Modern web apps typically include functionality and resources that are only available post authentication, and also include account and session management features that terminate active sessions. When performing an authenticated scan, at some point the session might break, e.g., due to following the logout URL or sending a malformed request that the web application cannot handle and all session info is invalidated. This directly affects the coverage and vulnerability detection scanners can achieve, as they might not be able to detect such state changes on time, or even at all, and proceed to perform an incomplete scan. ReScan needs to account for this major limitation as well, by ensuring the session remains valid for the entire duration of the scan and for all tested functionalities.

False positives & negatives. Scanners typically assert successful exploitation of certain vulnerabilities by checking whether the injected payload appears as-is in the application's response. This is not foolproof and is prone to false positives, as the mere existence of the payload inside the DOM does not necessarily imply successful exploitation. Additionally, a scanner might successfully exploit a vulnerability but not be able to detect it as the payload's structure might have been slightly altered or even completely stripped *after* being executed, leading to false negatives. Since we cannot tamper with each scanner's internal detection mechanisms, we need to devise a mechanism so as to eliminate both false positives and false negatives, or provide additional information to the user about such possible cases to facilitate *triage*.

URL clustering. Web app scanners can typically spend a significant amount of time testing redundant application endpoints, i.e., pages that are conceptually similar and offer the exact same functionality. It is apparent that such behavior directly affects their performance and overall scanning times. Thus, ideally, we need to prevent scanners from ever learning the existence of such redundant endpoints, by efficiently and effectively comparing and clustering them under a single, representative URL.

Response enhancement. Even if we successfully overcome the core scanner limitations, we still need to communicate ReScan's findings back to the scanner, such as request-triggering or DOM-changing client-side events. Due to our black-box approach, ReScan cannot directly interact with the scanner and is restricted to the existing communication channel (i.e., the HTTP connection) for transmitting artifacts.

Limitations of prior work. In Table I we outline the capabilities of the different scanners that we evaluate our system on (§ *IV*), so as to paint a clear picture of how each one can benefit from ReScan's enhancement techniques. As can be observed, only a single scanner leverages a modern, full-fledged browser environment; the rest are oblivious to dynamic content, functionalities and client-side events. Similarly, *only one* scanner leverages a navigation model to properly traverse the application, and *none* of them consider dependencies between different endpoints of the app. All of them offer some mechanism to handle authentication, however, some do so partially, i.e., assume that the authenticated session remains valid throughout the scan. When it comes to false positives and negatives, none of them take steps to eliminate them. Similarly,

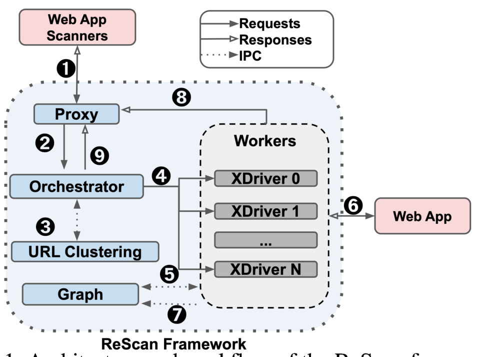

Fig. 1: Architecture and workflow of the ReScan framework.

for all scanners we find that they spend significant time testing redundant URLs or can incorrectly exclude pages from scanning due to deeming them to be similar to other tested pages. Overall, we find that all scanners do not take into account *at least four* aspects that can directly affect their vulnerability detection efficacy and achieved code coverage. This motivates our novel design approach of offering a framework that operates as a middleware component for enhancing the capabilities of any vulnerability scanner.

III. Design and Implementation

Figure 1 presents an overview of ReScan's architecture. Our system consists of a series of modules that operate concurrently and communicate with each other. The entry point to the system is an *intercepting proxy* which captures scanner requests and feeds them into the system. Next, the intercepted requests are loaded by the *orchestrator* and passed on to the *browser workers*, each of which has its own browser instance and attempts to accurately mirror the request in the application. They are also responsible for detecting new endpoints, e.g., links, forms and event originating requests, which are all wrapped in a single, enhanced HTTP response and sent back to the scanner. All discovered endpoints are stored in the application's navigation model by the *graph worker*, which is leveraged by the browser workers to properly retrace and execute all necessary steps when executing a request. Each worker utilizes the *authentication helper* module to ensure the session is valid. Meanwhile, the *background worker* is responsible for inspecting all data submitted to the application and uncovering *inter-state dependencies* (i.e., if a value submitted in a page appears on another one) which is necessary for detecting certain vulnerabilities (e.g., stored XSS). Finally, the *URL clustering* module employs a novel algorithm to detect similar pages, and notifies the orchestrator to filter out such requests and prevent the scanner from spending valuable time on redundantly testing them. Before describing each component in detail, we first briefly describe our navigation model (which is inspired by the approach of Eriksson et al. [25]); additional details can be found in Appendix B. It is important to stress that while *some* of our techniques are inspired by prior work, incorporating them into our middleware-style architecture is a demanding process that requires a methodical design strategy and addressing numerous implementation challenges, as we outline in the following sections.

Navigation model. Our model is realized as a directed graph, where each node represents the state of the application in terms of unique URLs, and edges describe the transitions between states – the specific actions required to move from one state to another, like client-side events or forms. Specifically, the edges are one of five types: `GET`, `FORM`, `EVENT`, `IFRAME` or `REDIRECT`. Thus, when visiting a page we collect all such edges and add them to the model. Each edge is assigned a unique ID, which consists of the edge type, the destination URL, and the normalized HTTP payload (i.e., the set of parameter names present in the payload). Finally, each edge is also connected to its parent edge, so we can construct workflows. Incoming HTTP requests from the scanner are mapped to individual edges, depending on their HTTP method and whether they carry a payload (e.g., POST data). Subsequently, to properly execute the request, we recursively construct its *workflow* by following the parent edges until we find a *safe* `GET` edge, which based on the HTTP specification [27] is not considered state-changing and can be safely executed as a starting point. For requests that cannot be mapped to any edge, i.e., *arbitrary* requests, we either directly execute them if they are a GET, or generate and submit an equivalent HTML form so as to get a response through the browser for payload-bearing requests.

A. System Components

Intercepting proxy. This component accepts incoming requests from a scanner, and presents *one of the two* requirements for our system's operation: the scanner *must* be configured to send its requests through this proxy. The other is setting the scanner's timeout per HTTP request to a large value, so ReScan has enough time to employ its numerous enhancement techniques. Both capabilities are supported by all scanners we have encountered so far. Our component is built as an add-on script on top of *mitmproxy* [17]. When a request is intercepted a new thread is spawned to handle it and our custom add-on code is executed ❶. Initially, the request is appended to all intercepted requests, along with all other relevant information (HTTP headers, method, payload and URL). The request thread then waits to receive its response from the system ❽, which will be sent back to the scanner. The only exception are requests towards static resources that do not have state-changing effects and are directly proxied to the target application. We achieve this by filtering requests based on known file extensions, which we list in Appendix A.

Orchestrator. This component periodically loads the intercepted requests populated by the proxy ❷, and enqueues any newly appended requests in a FIFO queue to ensure requests are served by the browser workers ❹ in the order they appear. It is also responsible for initializing and configuring all other components of the system.

Browser workers. The browser workers' goal is to accurately mirror each request through a fully-fledged, automated browser by executing the necessary workflow from the navigation model. We build the workers by leveraging XDriver [21], a robust Selenium-based browser automation tool with a rich set of security-oriented features pertinent to our goal (e.g., extracting HTTP redirection flows, spoofing request headers). The number of workers is a configurable parameter.

Serving requests. Initially, each worker reserves a request, constructs the corresponding edge ID based on the request information, queries the *graph worker* (described later on) so as to fetch the edge's workflow ❺ and proceeds to execute it ❻. Before executing the workflow the worker sets its cookie jar according to the request's *Cookie* header, so as to acquire the necessary state for authentication. Executing individual edges is rather straightforward; the worker simply fetches the page, fills and submits a form, and switches the browser's focus inside an iframe or triggers an event. We have also extended XDriver so as to capture events' asynchronous requests or redirects with the browser's internal proxy and make all necessary modifications before sending it to the application. However, in certain cases, some edges might not be required or cannot even be replayed. For instance, an intermediary edge corresponding to a login form that we previously used to log into the application will most likely *not* be present when executing the workflow of an admin dashboard's functionality. In such cases, we simply ignore non-existing, intermediary edges and proceed to the rest of the workflow. While at this point the scanner's request has been served ❻, we still need to enrich the response before sending it back to the scanner by leveraging any client-side events and the browser's JavaScript execution engine for triggering the events and recording any meaningful changes to the DOM or any asynchronous requests or redirects.

Event discovery. In order to trigger client-side events, we first need to identify which HTML elements have registered event listeners by hooking into them; we achieve this by using jÄk's JS library [49]. We then iterate over all captured events and attempt to trigger them; we expanded XDriver to accurately trigger each event. To detect DOM changes we utilize the MutationObserver API [6] by registering an observer on each document, which returns all changes caused by the last fired event. This is much more efficient compared to prior approaches of constantly scanning the DOM for relevant changes [25]. The DOM changes we consider are new links, forms and iframes, since they can reveal new endpoints of the application to the scanner, and at the same time constitute potential injection points. Similarly, for asynchronous requests, we modified jÄk's library to capture all the information our system needs. We also note that right before triggering each event, we block all asynchronous requests; after capturing the request information we do not send the actual request to the application so as to to avoid possible side-effects due to changing the application's state (e.g., logging out or deleting a user). Users can disable our event discovery module with a simple configuration option, when testing web applications that do not make heavy use of JavaScript.

Another aspect we have to take into account during this phase is that elements with events can produce *additional* elements, which can also produce others, make DOM changes and so on. To capture such *nested* events we follow a BFS approach and start by triggering all displayed events when loading the page (i.e., *level zero* events). After triggering each event we inspect whether any new events appeared and store this dependency link between them, thus constructing *event dependency chains*. In addition, if an event hides any level zero events (e.g., due to opening an overlapping menu), we immediately trigger it once again in an effort to make the base event reappear. When we are finished with all of a level's events, we proceed to the next level and repeat the process. It is important to note that for nested events we first check if they actually exist and are displayed in the page in order to trigger them immediately. This is needed since a previous event might have made permanent changes on the page (e.g., a sidebar with further events that remains open throughout the interaction with the page). If, however, the nested event cannot be triggered immediately,we proceed to recursively execute its dependency chain. The recursion is necessary since an intermediary event of the dependency chain might also not exist, thus its own chain should be executed in order to arrive to the final event. All dependency chains are also depicted in the navigation model as individual edges.

Finally, an exhaustive approach would require performing the event discovery for each unique edge we execute. In practice, however, we empirically observed that many similar edges (i.e., same base URL but different query) land on the same or similar pages and include the same or many common events; as such, triggering all

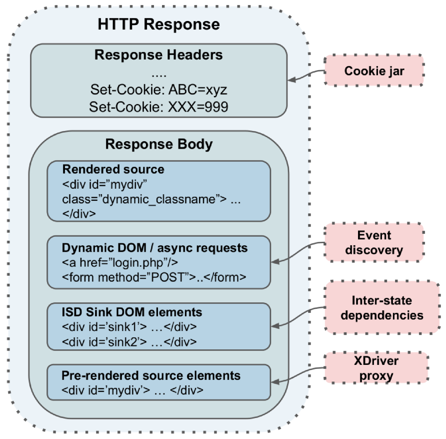
HTTP Response
Response Headers
....
Set-Cookie: ABC=xyz
Set-Cookie: XXX=999
Cookie jar
Response Body
Rendered source
<div id="mydiv" class="dynamic_classname"> ... </div>
Dynamic DOM / async requests
<a href="login.php"/>
<form method="POST">..</form>
Event discovery
ISD Sink DOM elements
<div id='sink1'> ...</div>
<div id='sink2'> ...</div>
Inter-state dependencies
Pre-rendered source elements
<div id='mydiv'> ... </div>
XDriver proxy

Fig. 2: Crafted HTTP response.

events again would be redundant. To address this, for each base URL we store each event along with its unique CSS selector. For ensuing requests to the same base URL with differing parameters we skip any previously encountered events and trigger only the new ones that may exist, thus significantly reducing the event discovery time.

Inter-state dependencies. It is common for certain parts of a web application to interact with and affect other parts. For instance, data submitted in a registration form can be reflected in other pages (e.g., the username being shown in the user's profile). The extraction of such *inter-state dependency* links (ISD) is crucial for discovering certain types of vulnerabilities that are not directly visible on the landing page after the payload injection but appear on other URLs, such as stored XSS. The core idea is to identify if a parameter value for a given POST request (the *ISD source*) appears on another page (the *ISD sink*). Thus, whenever a browser worker executes a POST request, mapped to either a `FORM` or `EVENT` edge, it will feed the request's edge ID, as well as all parameters and their values to the *Background Worker*, which is responsible for detecting such dependencies.

Assuming we have detected such ISD links, when executing a `POST` request the browser worker collects parameters whose values' entropy exceeds a threshold to capture the actual scanner payload being tested, which typically includes several different characters and has a greater length. Then, for every parameter it internally fetches ISD sinks associated with it and inspects whether the value truly appears in their source. If so, the value and its encapsulating HTML elements are stored, to be used in the final HTTP response and "carry" the necessary context for the scanner to properly evaluate its injection. We do not store the entire sink's DOM, as it can significantly increase the response size and corresponding processing time, especially given that a single POST request might be associated with more than one sinks.

Response enhancement. At this point, the system needs a way to inform the scanner of newly discovered endpoints or inter-state dependencies. Since our approach is fully scanner-agnostic, we treat each scanner as a black-box that produces HTTP requests and consumes responses. Thus, we need to transcribe each of these newly discovered artifacts into a final, *static* HTTP response, in a way that they are "detected" and leveraged by scanners. To that end, we initially append the ISD sinks' relevant HTML elements and the DOM changes "as is" to the response (e.g., a dynamically generated form is appended to the document's body). In contrast, simple asynchronous `GET` requests and redirects are transcribed as new links, while more composite requests are converted into an equivalent HTML form, with input fields matching those of the HTTP payload, as well as the HTTP method of the asynchronous request. During our analysis, we also observed that modern browsers may alter the structure of the original page source in regards to syntax semantics. For instance, if the browser encounters *stray* element attributes, i.e., only an attribute name without a value (which can be part of a scanner's payload), it will modify their structure, e.g., by appending the `=` sign and quotes. For example, consider the following scenario, where a form field vulnerable to XSS originally has the form `<input name="username" value="user">`. The scanner payload, trying to escape the value attribute, might look like `abc("'xyz`, which will leave the `'xyz` portion as a new, stray attribute inside the element and will be converted by the browser to `'xyz=""` with a leading space, thus preventing the scanner from detecting its otherwise successful injection. In addition, we use BeautifulSoup [7] with the `html5lib` standards-compliant parser [4], to parse and modify the final HTTP response (e.g., to append sink DOMs for ISD), which might further alter the payload structure by rearranging the order of such stray attributes. To overcome this challenge, much like the ISD approach, when submitting values, by appending the *pre-rendered* source's HTML elements that include the value in the final HTTP response, we allow the scanner to detect and evaluate its injections that may have been altered due to this behavior.

Moreover, the application might have set or unset cookies during the execution of the workflow either with *Set-Cookie* response headers or via JavaScript. Since we capture the response headers for the last executed edge (i.e., the initial scanner request), and several scanners do not include a JavaScript execution environment, we also need to pass this information to the scanner as well. To that end, we iterate over the browser's cookie jar, compose equivalent Set-Cookie headers, and append them in the response so as to inform the scanner of the current state. The worker then proceeds to craft the final HTTP response including all relevant information, i.e., the enriched response body, status code and response headers and sends it back to the proxy's request thread waiting for it ❽. A detailed depiction of how the response is crafted can be seen in Figure 2. Subsequently, the worker submits all discovered events, along with all links, forms, iframes and redirects of the page to the *graph worker* so they can be included as new edges in the navigation model ❼. We provide more technical details on browser workers and the challenges we had to solve in Appendix C.

Graph worker. This component is responsible for interacting with the navigation model, and consists of two separate threads. The first one operates in a read-only manner on the model: it awaits for browser workers' requests for specific edges, constructs and returns the necessary workflow ❺. The second thread, operating in a write-only fashion, receives newly submitted edges by the workers and adds them to the model ❼.

Background worker. Detecting inter-state dependencies is crucial for detecting certain types of vulnerabilities. The background worker's (BG worker) goal is to detect such dependency links in a timely manner. During its lifetime, the BG worker constantly observes the `POST` requests executed by the browser workers and identifies ISD sources that might appear in other parts of the application. While in practice other HTTP verbs might be used for state-changing requests as well (e.g. PUT), similarly to prior work [25] we only consider POST as others are not intended to be state-changing at all (e.g., GET [27]). Specifically, it initially keeps track of all submitted values that have an entropy higher that a given threshold. This is a necessary performance/detection trade-off, so as to eliminate commonly seen values (e.g., "1", "true") which would lead to a prohibitive number of candidate ISD sinks being fetched by the browser workers; we empirically found that a threshold of 1.4 effectively eliminates such values. It then queries the graph worker and collects *all* the `GET`, `IFRAME` and `REDIRECT` edges found so far, as they constitute potential ISD sinks. It then proceeds to iteratively fetch each of these edges and inspect whether any of the submitted, higher-entropy values appear in its source. If so, it has detected an ISD sink and immediately notifies the browser workers of the uncovered dependency, so they can fetch it whenever submitting the ISD source.

Before moving on to the next edge, the BG worker submits *all* edges it observes to the graph worker so they can be inspected as well. This is required as certain sinks might appear only *after* submitting the corresponding ISD source (e.g., the URL for editing a comment appears after posting the comment). The BG worker will repeatedly visit all potential sinks as long as there are POST requests executed by the browser workers that have not been fully inspected (indicating that not all potential sinks have been explored). While this approach involves a non-negligible number of additional requests it does not affect the performance, as the BG worker operates concurrently with the browser workers.

We note that a more straightforward approach would be to detect ISD links by leveraging the underlying scanner's requests. For instance, whenever we execute a workflow, we could inspect the landing page for previously submitted values. This, however, creates a strong dependency between our system's effectiveness and the underlying scanner; if the scanner decides to fuzz a form and does not later visit the corresponding ISD sink, either due to not having discovered it or because it had crawled it earlier, then we would have no chance of detecting the ISD link on time and the underlying vulnerability would be missed. With our approach we can effectively tackle such cases and detect inter-state dependencies on time.

Input pre-filling. Another technical challenge is scanners' default behavior when submitting a form. In more detail, when scanners start fuzzing a form, they will typically use any values already filled out by the application itself and will submit their own default values for empty fields. For instance, wapiti sends "default", while ZAP uses "ZAP" as their default values. This behavior, however, can lead to the detection of numerous *candidate* sinks during our ISD detection and can significantly affect its performance, as such values may appear in multiple areas of the application and could stem from multiple ISD sources. To tackle this issue, right before sending the final HTTP response back to the scanner, we iterate over all *empty* input fields in the page and assign a unique value we generate on the fly, with an entropy high enough so it can be detected in actual ISD sinks. This way, scanners will use our unique tokens instead of their defaults and we can precisely map ISD sources to sinks.

Authentication helper. A common problem with scanners that execute authenticated scans is that they will submit the valid, user-provided credentials once but will forget them in subsequent authentication requests and instead submit their default credentials. This can occur if the scanner decides to fuzz the login or an account settings form (i.e., change username or password); this is problematic since these default credentials do not correspond to a valid user, thus permanently losing the authentication state for the remainder of the scan. Similarly, scanners do not infer when they are logged out (e.g., due to a malicious HTTP request that the application cannot handle) and continue their (incomplete) scanning irregardless of the authentication state. To overcome these problems, we develop an authentication helper module, leveraged by the browser workers.

Credential detection. Initially, the module captures the *first* authentication request (i.e., includes a password in its `POST` data) and detects the valid user credentials based on common parameters' naming conventions. This is based on the assumption that the first authentication-like request will include the user-defined, valid credentials, which holds true for all scanners we evaluated. ReScan will then detect these fields in subsequent requests and will overwrite them with the valid credentials, thus ensuring the integrity of the authenticated session.

Oracle. Moreover, again on the first authentication request, the module will perform a series of steps to dynamically infer an authentication oracle, conceptually similar to the one proposed in [21]. In more detail, the worker that performed the login will send another request to the landing page without containing *any* cookies, to obtain an unauthenticated response. The module will then check if the detected username (or email) or any logout-related string appears in the authenticated response but not in the unauthenticated one. If so, it has deduced a robust authentication oracle. If they appear in both responses, the worker will then fetch the page containing the login form and check whether the form appears only in the unauthenticated response. Similar to the first step, if this yields a positive result we have again found the oracle. For each subsequent request, right after the execution of its workflow, we deploy the oracle in a new browser tab so as to check the validity of the established session. The new tab is required so as to maintain the main request's state (e.g., landing page) and execute the remaining components. During our experimental evaluation we observed that the overhead induced by executing the oracle in *every* request is negligible yet significantly increases robustness.

Relogin. If after the execution of a request the oracle detects that we have been logged out of the application, the module must attempt to transparently re-establish an authenticated session. This can happen if another concurrent worker triggered a logout and invalidated the session for everyone, or the session broke due to this specific request. However, at this point we cannot be certain which is the cause. In any case, the module will revisit the login URL and try to login with the valid credentials and consult the oracle to verify that the relogin succeeded. If it succeeds, the worker will retry the request at hand to ensure that its workflow was executed properly. If the session breaks again, indicating that this is indeed the faulty request, we relogin and skip the remaining components for this request; otherwise we proceed as normal. Finally, if the relogin fails, we inform all workers to shutdown since we cannot continue the authenticated scan, indicating that either the credentials are no longer valid (e.g., the user was deleted or blocked), or that the application is no longer responsive (e.g., it returns an error message). However, this behavior can be disabled with a simple configuration option, so as to allow the scan to continue regardless of a successful relogin. It is important to note that while the oracle is conceptually inspired by [21], the entirety of the authentication helper module and its capabilities is a new contribution, rooted in the challenges presented by the novel middleware architecture of ReScan.

False positive & negative elimination. Another common scanner limitation is their susceptibility to false positives (FPs) and negatives (FNs), especially when testing for XSS vulnerabilities. This is due to the fact that scanners often verify the success of their injections based solely on the payload's existence in the HTTP response, without verifying if the payload was actually executed. Therefore, it is clear that a payload that appears "as is" but was never executed (e.g., due to it being part of an input element's value attribute) will lead to a FP. Similarly, even if the payload is executed, but its structure is altered or completely removed by client-side code, the scanner will not be able to detect the successful injection, leading to a FN. Due to our black-box approach, we cannot tamper with each scanners' internals and tackle this; we can, however, eliminate or reduce such cases by providing the user with additional results as a separate report, which can be intersected with the scanner's reported vulnerabilities. To achieve this, we employ the following approach.

First off, for each incoming request, we try to identify if it is an injection attempt by collecting all query parameters for `GET` requests and payload parameters for `POST` requests, and check their values against common keywords used among scanners (e.g., `alert`, `prompt`, `javascript:`). The intuition is that regardless of the payload structure, most scanners will attempt to trigger an alert box with their injected code. Then, whenever an alert box opens throughout the rest of the scan, we extract its text and try to match it against all detected injections in the previous step. If the alert text does appear inside any of the injection values, we can be certain that the detected XSS is a true positive, since it leads to code execution. If it does not match any of the injection attempts, it is discarded, as it is a legitimate alert by the application. It is important to note that the effectiveness of this technique relies on the underlying vulnerability scanner.

For scanners that do *not* reuse payloads, i.e., do not alert the same value for different injection points (e.g., Wapiti), then all FPs and FNs can be eliminated as each alert box can be exactly mapped to one injected value. If the scanner does reuse payloads (e.g., ZAP always tries to execute `alert(1)`) then any alert box that occurs will be mapped to all attempted injections so far. Thus, we know there is *at least* one XSS vulnerability up to that point, but we cannot be certain which parameter is vulnerable, nor if there are more than one. For such cases, we simply inform the user that the confidence level for the legitimacy of the reported vulnerabilities is lower and they should account for possible FPs reported by the scanner. Moreover, if there are no alerts throughout the scan, indicating that no XSS vulnerabilities were triggered, then any reported vulnerabilities by the scanner can be safely considered as FPs.

B. *URL Clustering*

All the enhancement techniques we have devised so far aim to improve scanners' code coverage and vulnerability detection capabilities. However, during our empirical analysis we identified an orthogonal limitation which affects scanners' performance. We observed that scanners are not able to identify *similar URLs*, which may include the same or related content but essentially offer the exact same functionality. For instance, if a web application includes a series of product URLs of the form `/products.php?pid=X`, the scanner will crawl *and* audit each URL separately, which can eventually lead to the detection of other similar URLs, specific to each product (e.g., `/products.php?pid=X&action=edit`). Clearly this behavior impacts scanners' performance, as they spend resources and time repeatedly testing the same functionality.

Additionally, while some scanners offer some sort of control mechanism to limit this behavior, they are coarse-grained approaches that require careful configuration by the end user and can also wrongfully skip URLs. For instance, Wapiti offers an option to limit the crawl's depth, which can result in skipping important URLs that appear in deeper levels of the application while similar URLs that are at the same depth will be redundantly crawled. w3af provides an option to limit the number of URLs with the same path and set of query parameters that will be considered during the scan. While this can decrease the number of discovered similar URLs, it requires careful examination and proper configuration. More importantly, some URLs with the same query parameters might need a different threshold than others; for example setting the threshold to 1 would work for `/products.php?pid=X` URLs, as more than a single product page would be redundant, but for URLs of the form `/products.php?pid=X&action=Y`, where `action` can be one of `edit`, `update`, `delete`, a value of 1 would cause the scanner to consider only one of these functionalities, thus directly affecting its coverage and, potentially, the detected vulnerabilities.

It is worth mentioning that prior work on web application scanning has proposed page clustering algorithms [19], [49]; however, their approach relied solely on a page's link structure or did not consider URLs' query parameter values. Regarding the link structure, while such a strategy may have been adequate at the time, modern web applications utilize new methods for navigation and different functionalities (e.g., stray input or button elements that do not belong to a form) which are driven by client-side events. Ignoring these can lead to the incorrect clustering of different pages. On the other hand, ignoring URL parameter values can lead to subtle but important inconsistencies. For example, consider two product pages, with `pid=1` and `pid=2` and their corresponding sub-URLs for editing each one, `/products.php?pid=X&action=edit`. When the scanner encounters the product URLs, it deems them to be similar and clusters them under `pid=1`. When it visits the editing pages, if it does not take into account the `pid` value it used previously, it can end up clustering them under `/products.php?pid=2&action=edit`. Thus, if a vulnerability stems from editing a product's field and is reflected in that product's page, the scanner will miss it as it will edit the second product but inspect the page of the first one.

To overcome these limitations, we design an advanced URL clustering algorithm that aims to cluster similar URLs in real time when requested by the scanner and prevent it from ever learning the existence of redundant URLs, *without* requiring any specific configuration by the user.

Page similarity. The first requirement for our algorithm is to be able to *accurately* and *efficiently* identify whether two URLs and their respective DOMs are in fact similar and should be clustered together. We consider two URLs to be similar if they share the same path, regardless of the URL query parameters. If they are not similar they are not clustered together; if they are, we still have to investigate whether their DOMs are similar to decide if they should be clustered or not. To achieve this, we build upon the *normalized DOM-edit distance* metric (NDD), proposed by Vissers et al. [62], which essentially takes as input two DOM trees and computes the number of edit operations required to move from one tree to the other. Initially, we observed that the tree edit distance algorithm they used at the time (ZSS [63]) was rather slow even for relatively simple DOMs. Since we need to compute DOM similarity immediately when a page is requested by the scanner, we cannot afford such a performance penalty. Thus, we decided to replace the tree edit distance algorithm with current state-of-the-art, namely *APTED*, proposed by Pawlik et al. [47], [48], which offers a significant speed up. However, in certain cases of more complex DOMs, even this algorithm had a prohibitive processing time for our use case.

Since designing a more efficient tree edit distance algorithm is out of scope of this work, the only way to reduce the processing time is to reduce the size of the algorithm's input, i.e., the DOMs' sizes. While [62] adopted a horizontal pruning approach, discarding all tree nodes below a level of five, such an approach would not be suitable in our case. This could lead to subtle but important differences that indicate different functionality between the two pages (e.g., a form residing in deeper levels of the DOM) not being considered, leading to pages being incorrectly clustered together.

Thus, we devise our own fine-grained pruning methodology. When constructing the corresponding trees from the pages' DOMs we recursively discard leaf nodes that do not offer any sort of functionality (e.g., line breaks, paragraph, `span`, `div` or font formatting tags), while maintaining nodes that denote functionality (e.g., scripts, forms, iframes, buttons and inputs). If a node has all of its children removed, becomes a leaf node and is a non-functional tag, it is discarded as well. With this approach, we significantly reduce the tree size and thus the processing time required for the NDD computation, while maintaining the important parts of the tree structure that we should consider when deciding whether two pages should be clustered.

Moreover, this approach also helps us avoid not clustering similar pages together due to insignificant differences. For instance, consider two article pages that should be clustered together as they offer the exact same functionality and one of them includes a list of comments, while the other does not have any yet; this would lead to *not* clustering these pages together due to these (irrelevant) nodes. It is important to stress at this point, that two pages that do not include any functionality denoting nodes (e.g., two static pages) would be oversimplified and incorrectly clustered together; however, we employ our NDD variant *only* for pages that also share a similar URL, thus avoiding clustering together irrelevant pages. We refer to our NDD variant as *mNDD*. If the mNDD between two DOMs is lower than a predefined threshold, which we identified experimentally, they are considered similar.

Algorithm. We use an example application that includes a set of similar and different pages under the same base URL to illustrate our URL clustering algorithm. Assume an application that lists two different products pages, with URLs of the form `/products.php?pid=X` and `pid` denoting each product's unique ID (either `1` or `2`). For the remainder of this section, all example query strings will refer to the `products.php` page, which we omit for brevity. These pages include links to product-specific pages (`?pid=X&action=Y`) that offer different functionalities. The `action` parameter can be `edit`, `review` or `add`, and leads to a page that has a form for editing, reviewing, or adding the product to the cart, respectively.

At a high level, the core idea of the URL clustering algorithm is to prevent scanners from learning the existence of redundant URLs that point to similar pages and offer the same functionality. In this example, the scanner should either learn (i.e., receive a regular response) for `pid=1` or `pid=2` but not both. Moreover, the algorithm must ensure that the scanner will learn all other functionalities and sub-URLs for *the same product ID only*. For instance, if the scanner initially learns `?pid=1`, then it should also learn all actions for that product ID only. This is required since executing such a functionality will most likely have an effect only on that product's page. Thus, the algorithm needs to keep track of the different URL parameters and their values for which the scanner got a response. Finally, it is crucial that the algorithm also accounts for the arbitrary order the scanner might request such URLs; e.g., it is not restrained to requesting the `?pid=X` URLs first and then moving on to their specific sub-URLs. Having defined the algorithm's goals and an example scenario, we next detail the specifics of our URL clustering algorithm.

Firstly, the algorithm considers only `GET` requests that include query parameters. We do not include other types of requests (e.g., `POST`) as they do not carry information that is useful for the clustering process and would only lead to longer running times. The first time the scanner requests one of these URLs it will get a regular response and we store all parameters and their values that it learned. For subsequent requests towards the same base URL with differing query parameters (either with a different set of parameters or different values or both) the algorithm will perform the following steps:

1) Collect all parameters that we have seen before but have a *different* value than the one(s) the scanner has learned. We represent these parameters with set $\{X\}$, and another empty set with $\{X'\}$.
2) If there are any new parameters that the scanner has not requested before, leave them "as is" and store their values.
3) Swap the unknown values for the $\{X \setminus X'\}$ parameters (the difference of the two sets) with those previously learned by the scanner (initially all values are unknown).
4) Internally fetch the swapped and original URLs (if not already fetched) and compute the mNDD between them.
5) For similar pages generate a clustering rule (described below) and stop.
6) If the pages are *not* similar: (i) reset $\{X'\}$, (ii) pick a parameter to preserve its original value, (iii) add it to $\{X'\}$, and (iv) go to (3). If no more single parameters remain to preserve their value, iterate through all combinations of two parameters, then combinations of three, and so on.
7) If there are no more parameters to swap, stop, as this is indeed a new page and should be served normally. Store the original URL's parameter values for subsequent comparison and clustering decisions.

Assuming the scanner initially requested and learned `?pid=1`, we next provide a series of cases based on our example scenario to demonstrate our algorithm's correctness and to also describe the generated clustering rules and how they are applied. If `?pid=2` is requested next, since the `pid` parameter has already been seen before but with a different value (step 1), it will be swapped with the known value `1`. Then both URLs will be fetched internally, their mNDD score will be computed and they will be found similar, and the clustering rule shown in Listing 1 will be generated (steps 1-5).

```
{    "pid" : "*",
     "redirect" : "?pid=1"}
```

Listing 1: Example of a simple clustering rule.

Essentially, this rule indicates that any incoming request that only has the `pid` parameter, regardless of its value, should be redirected to `?pid=1`, thus effectively preventing the scanner from ever getting a response for the other product. This is achieved by sending a crafted HTTP response back to the scanner, with a 301 status code and the `Location` header set to the appropriate URL. The only exception is when the scanner requests `?pid=1`, which will be served as normal so as to obtain a fresh response for that product's page.

If the scanner requests `?pid=2&action=edit` next, we have a newly seen parameter (i.e., `action`) which is not tampered with throughout the process (step 2). The known `pid` parameter will be swapped and processed like before and the scanner will internally fetch the original URL and also `?pid=`**`1`**`&action=edit`, compare them using mNDD and infer that they are again similar, setting the rule shown in Listing 2.

```
{    "pid" : "*",
     "action" : "edit",
     "redirect" : "?pid=1&action=edit" }
```

Listing 2: Example of a subsequent clustering rule.

It is important to note, that the scanner might not have requested or even seen the swapped URL before; this, however, does not affect the algorithm's operation, as it can *proactively* infer its existence by considering the previously seen parameters and their values. It is also necessary, in order to ensure that the scanner will learn the editing functionality for the same product it learned before.

If the scanner then requests `?pid=1&action=review`, the `pid` parameter already includes a known value and is therefore left as-is. The `action` parameter, while known, has a different unknown value compared to before. Thus, it will be swapped with the value `edit`, the two URLs will be compared and the algorithm will infer that they are in fact different pages, since their mNDD score is higher than the threshold. This leads to not setting any new rules and the originally requested URL should be served normally (step 7) while storing the newly seen value for the `action` parameter.

Finally, in a more complex case, if `?pid=2&action=add` is requested, both parameters are known but yet contain values that the scanner has not learned. As a result, all parameters will be initially swapped and the URL `?pid=`**`1`**`&action=`**`edit`** will be fetched and compared. Since it leads to a truly different page, no rules will be generated and one of the parameters will be picked randomly (e.g., `pid`) to preserve its original value (step 6). The swapped URL that occurs is `?pid=`**`2`**`&action=`**`edit`**, which also leads to a different page. Next, the last remaining parameter will maintain its value while swapping the other one (`?pid=`**`1`**`&action=`**`add`**). This indeed lands on a similar page as the original URL and a rule similar to Listing 2 will be generated, for the `add` functionality. Even if these requests occurred in a completely different order, since scanners can prioritize them differently (e.g., based on when the URLs were discovered in the application during the crawl), the algorithm would end up inferring the exact same clustering rules, with a slightly different order and steps, depending on the known parameters and their values at the time of processing.

At this point, it is worth noting that a large number of URL parameters could *potentially* lead to a state explosion or a prohibitive processing time, due to the numerous parameter combinations that would have to be tested. In practice, however, this is a highly unlikely scenario due to several reasons. First, since the algorithm initially ignores newly seen parameters and parameters with known values, it would not have to test and compare *all* combinations. In addition, for that to happen all combinations would need to lead to different pages, as the algorithm stops if it finds a pair of similar pages. Finally, we have not come across any such case during our experimental evaluation. In §IV-B we experimentally measure the performance gain our algorithm offers, and also evaluate its correctness in terms of achieved code coverage and discovered vulnerabilities.

Implementation details. Our URL clustering functionality can be enabled with a simple command line parameter; no further configuration is needed. Initially, the Orchestrator inspects the scanner requests and locates each `GET` that includes a query. It then checks if any of the generated rules applies to the URL; if so,

Scanner	**w3af**		**wapiti**		**Enemy**		**ZAP**	
Vulnerability	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS	R-XSS	S-XSS
SCARF (2007)	-/-	4/**8**	-/-	3/**7**	-/-	**-/4**	-/-	3/**6**
WackoPicko (-)	1/**2**	**-/1**	2/**3**	1/1	2/2	1/1	2/2	1/1
Wordpress (5.1)	-/-	**-/1**	**-/1**	**-/1***	-/-	-/-	**-/1**	**-/1***
osCommerce (2.3.4.1)	**-/2**	**-/2**	3/3	**5/16**	-/-	-/-	-/-	2/2
Vanilla (2.0.17)	-/-	**-/1**	-/-	**-/1**	-/-	-/-	-/-	**-/1**
PhpBB (2.0.23)	-/-	-/-	-/-	**-/2**†	-/-	-/-	-/-	**-/4**†
Prestashop (1.7.5.1)	**-/1***	-/-	**-/1***	-/-	-/-	-/-	**-/1***	-/-
Joomla (3.9.6)	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
Drupal (8.6.15)	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
HotCRP (2.102)	**-/1**	-/-	**-/1**	-/-	-/-	-/-	-/-	-/-
Total	1/**6**	4/**13**	5/**9**	9/**28**	2/2	1/**5**	2/**4**	6/**15**

* The scanner was able to identify the vulnerability *only* with ReScan, but not during the maximum scan time.

† One of the vulnerabilities was found in a URL that broke the app and was eventually excluded.

TABLE II: Number and type of unique vulnerabilities discovered by each scanner without (left) and with ReScan (right) for each app.

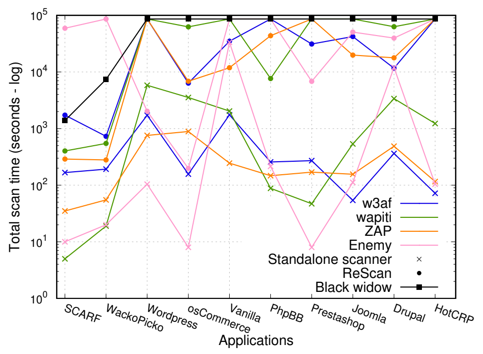

Fig. 3: Total scan time in seconds for each app/ scanner pair with and without ReScan.

it immediately crafts an HTTP redirect (❾ in Figure 1) and sends it to the request's proxy thread, which eventually sends it back to the scanner. If no rules apply to the URL (initially because none have been generated) it marks the request as pending and passes it to the clustering module.It then moves on to subsequent requests that need to be served, so as not to remain idle while pending requests are processed, and periodically checks if a new rule has been generated and applies to the pending URL or whether the request should be passed on to the browser workers.

The clustering module is comprised of a configurable number of threads that handle pending requests concurrently. When a request arrives, a thread reserves it and runs the algorithm to decide whether to serve it normally or infer clustering rules. These threads do not maintain their own browser instances, as it would be too expensive, but need a way to fetch the original and swapped URLs that occur during their operation. To that end, we leverage the BG Worker, which apart from ISD-detection also serves these requests by the mNDD threads. It also caches the responses in case the same URL is requested later on. The BG worker constantly checks requests, as they are issued from the scanner and need to be served as soon as possible.

C. API Abstraction for Future Scanners

As aforementioned, while ReScan can be transparently leveraged by any scanner as a black-box middleware, *future* scanners could greatly benefit from the ability to access ReScan's internal knowledge and alter its behavior based on their runtime needs. As such, in order to unleash ReScan's full potential and enable such a symbiotic interaction, we design and implement an abstraction layer in the form of an API. A scanner opting to use the API can request access to ReScan's 'internal data, such as the entire app navigation model, detected ISDs, discovered XSS and more. Moreover, it can alter ReScan's behavior, by enabling or disabling any module at runtime, e.g., disabling ISD detection and sink collection when testing for vulnerabilities that do not have ISD effects. We detail the various API endpoints in Appendix E.

IV. Experimental Evaluation

Experimental setup. For our system's evaluation, we use state-of-the-art vulnerability scanners that have seen wide adoption, allowing us to perform a direct comparison. Specifically, we evaluate ReScan on w3af [51], wapiti [57], Enemy of the State [19] and ZAP [5], which have been extensively evaluated by prior work [20], [19], [49], [24], [25]. We refrain from evaluating simple crawlers, e.g., wget [44], CRAWLJAX [43], even though they could benefit from ReScan, since our main goal is to enhance vulnerability scanners and measure both their coverage *and* detection capabilities. w3af and wapiti serve as a benchmark for more traditional vulnerability scanners as they mainly use raw HTTP requests and feature from minimal to no-Javascript execution engines, while ZAP has more advanced capabilities that are better suited for modern scanning requirements. Enemy of the state is a popular state-of-the-art academic scanner, which also introduced the concept of modelling application states. To be able to evaluate Enemy, we had to make some slight modifications so as to proxy all of its traffic through ReScan; its core functionality was left as-is. Finally, it is worth mentioning that we also attempted to setup jÄk [49] for our evaluation, but were unable to do so, due to the use of certain outdated packages that prevented it from executing properly. We contacted the authors to aid us in the setup process, but did not receive a response.

To obtain better code and functionality coverage in the tested applications, we run authenticated scans by configuring each scanner to log into them. For ReScan's configuration, we enabled four headless Chrome browser workers and all enhancement techniques described in §III (i.e., ISD detection, event triggering, as well as the authentication helper and URL clustering modules). For the event discovery process we consider the events by [25], (i.e., *(on)input*, *onchange* and *compositionstart*) and also extend them to include another set of prominent events that can trigger requests and cause DOM changes, namely *(on)click* and *(on)submit*.

When running the standalone scanners without ReScan, we enabled the audit plugins both for reflected and stored XSS. However, when evaluating ReScan we disabled the stored XSS plugins, as we rely on our ISD module as a substitute. Similarly, we enabled the AJAX spider plugin for ZAP when running without our system, but disabled it with ReScan, due to our event discovery module. These configuration changes however, do not work in favor of ReScan; on the contrary, by disabling modules we can only limit our coverage and detected vulnerabilities. Despite that risk, and in favor of reducing redundant operations as well as proving our approaches' practicality, we opt to rely on our own techniques. We also set the HTTP timeout for all scanners to 999 seconds. Finally, to avoid long lasting scans that may occur for more complex applications and complete our evaluation in a reasonable timeframe, we set each scanner's maximum scan time to one day. It is important to stress that apart from the aforementioned configuration options, all scanners and applications were configured in the *exact same way* when running with and without ReScan. We provide more details on the specific configuration options in Appendix F.

In Table II we list the web applications and the specific versions we used during our evaluation. We opted to use the same set of applications as [25], since it includes both legacy and intentionally vulnerable apps as well as modern, widely used applications. In addition, using the same applications allows for direct comparison. We created an individual Docker container for each application, allowing us to reset it back to a clean state after each scan; all containers have been released to facilitate further research in the field [11]. We also enabled XDebug [50] in each application for capturing precise coverage information in terms of unique lines of code (LoC) executed during each scan. Finally, we note that for each application we excluded any URLs that might affect the application's correct deployment (e.g., user deletion functionalities, version upgrading), as done in prior work as well (e.g., [22]) In more detail, we initially identified common URLs manually. Then, during our test runs our authentication helper module allowed us to identify more URLs (i.e., when being logged out and not able to re-login). Inspecting the traces showed that some functionality broke the app or the user was disabled/blocked. While such endpoints might also suffer from vulnerabilities, they pose a risk to correctly auditing other (and usually more) functionalities. In practice a separate scan should be performed for those endpoints. This is inherent to black-box scanning and is not a limitation of ReScan; it is rather a matter of correct scanner configuration.

Experiments were done on a commodity desktop with an 8-core Intel Core i7-4790 CPU 3.60GHz and 12 GB of RAM.

Discovered vulnerabilities. Table II details the results of our evaluation; we manually verified every discovered vulnerability and report on the true positives. To deduplicate scanners' results and provide a fair comparison, we cluster vulnerabilities following the same approach as prior work [25]. Regarding false positives, we found that wapiti reports only one, while ZAP is the scanner most prone to FPs as it reports 16 FPs across all apps by itself and 20 with ReScan. This increase is expected, as the scanner audits a larger area of the application when enhanced by our system. Nonetheless, ReScan is able to identify these injections as *potential* FPs due to ZAP reusing the same payload. Regarding true positives, in most cases ReScan effectively enhances the underlying scanner and facilitates the detection of more vulnerabilities, both for reflected and stored XSS. We also observe that the detection capabilities improve both for benchmark, and more recent applications. When considering the aggregated results per scanner for all applications, we find that w3af reports five more reflected XSS with ReScan and nine additional stored XSS. Moreover, wapiti located four more reflected and another 19 stored injections, while ZAP has an improvement of two and nine additional flaws respectively. Enemy of the State exhibited the least improvement but still located four additional stored XSS; this highlights that while ReScan is naturally dependent on the underlying scanner's capabilities, it can still effectively facilitate vulnerability detection. Overall, the standalone scanners reported six *unique* reflected and 13 stored XSS among all applications, while with ReScan they reported 10 reflected and 34 stored XSS respectively. Even in the few cases where no additional flaws were detected, the presence of ReScan does not negatively affect scanning as the same vulnerabilities were detected both with and without our system. Detecting the same flaws does not necessarily mean that the *standalone* scanner has detected all endpoints of the application or that it has sufficiently tested it; it might simply mean that while ReScan covers a larger area of the application, no other vulnerabilities exist for it to detect. To uncover more insights, we need to examine the code coverage that was achieved in each case.

Code coverage. Table III shows the precise coverage achieved by our system, as unique LoC executed on the server-side during the scan, and compares it to the coverage of each individual scanner. ReScan achieves better coverage in all cases and offers an improvement of at least 3% and at most 935%, with an average of 168%. To validate the quality of the increased coverage, we manually sampled and inspected LoC executed only by ReScan and found that in many cases ReScan-enabled runs reached and tested critical functionality that the standalone scanners did not. Indicatively, ZAP could not reach the categories' editing functionality in osCommerce, while none of the scanners could post and read draft discussions in the Vanilla app; both cases led to missing XSS flaws. We also observe that vanilla scanners reach some LoC which are not executed by ReScan. After inspecting these cases, we found that they belong to unauthenticated parts of the application, indicating that the scanner was logged out and continued as such. ReScan's authentication helper module ensured that the scanner remained authenticated throughout the scan, as intended.

Performance analysis. In Figure 3 we present the total time required to scan each application both with and without ReScan. The overhead induced by our system is considerable; however this is expected due to the numerous enhancement techniques we apply for each and every intercepted request and the fact that a full-fledged browsing environment is leveraged. In addition, the maximum scan time of one day was reached by all scanners for one of the applications (HotCRP) while in total, 15 of the 40 ReScan-enabled runs reached this limit. We detail our system's individual components', as well as total processing time per handled request in Appendix D. Most notably, event discovery can be quite expensive for applications that heavily rely on Javascript and client-side events, i.e., on average it takes one to three seconds for nine of the apps, while Prestashop required 19 seconds. Similarly, fetching ISD sinks on average took less than two seconds for five applications, while in the worst cases (Wordpress, Prestashop) it took 11 and 16 seconds respectively. Other system components, such as fetching and executing the workflow, constructing the navigation model and crafting the final HTTP response introduce negligible overhead in most cases, i.e., less than a second. Interestingly, while executing the crucially-important authentication oracle after every request might seem costly, our analysis showed that it only takes up to two seconds for 99% of requests. Overall, each request can be completed on average within three seconds for four applications and five seconds for another two, while Prestashop generally has slower response times and can take up to 21 seconds. In summary, while the performance overhead is non-negligible when compared to the standalone scanners, the significant improvements both in code coverage and vulnerability detection render this a viable and acceptable trade-off.

Prominent use cases. Next we outline interesting use cases that highlight the benefits of using our framework.

Vanilla FP. When wapiti scanned the Vanilla forums with ReScan it *incorrectly* reported an XSS vulnerability due to the payload appearing inside a `textarea` element (i.e., was not executed). However, while ReScan detected the injection attempt, due to our FP-elimination mechanism it *correctly* did not report a vulnerability.

Wordpress FN. One vulnerability is a reflected XSS stemming from the submission of a vulnerable field in an AJAX request. The AJAX response is then reflected in the same page, the payload is executed and is then dynamically removed from the DOM; thus neither wapiti, nor ZAP are able to detect their otherwise successful injection. Since ReScan is agnostic to the presence and structure of

TABLE III: Total lines of code (LoC) executed by ReScan (R), the standalone scanner (S), and common to both of them (R ∩ S).

App / Scanner	w3af			wapiti			Enemy			ZAP		
	R	R ∩ S	S	R	R ∩ S	S	R	R ∩ S	S	R	R ∩ S	S
SCARF	**662**	533	548	659	596	611	623	261	288	613	578	599
WackoPicko	**1,009**	888	907	911	692	710	873	433	452	819	684	784
Wordpress	51,612	30,779	30,805	53,974	30,862	31,134	43,731	28,908	29,266	**54,329**	33,514	34,484
osCommerce	7,056	2,066	2,074	7,179	6,947	7,140	5,194	2,067	2,067	**7,270**	6,247	6,925
Vanilla	12,247	8,073	8,137	12,138	7,936	8,717	12,404	2,477	2,479	**12,951**	8,774	9,568
PhpBB	9,803	2,321	2,330	9,942	3,069	3,091	8,225	6,780	7,018	**10,487**	4,816	5,259
Prestashop	93,361	14,544	14,709	96,712	14,916	14,926	28,209	19,062	19,062	**103,955**	10,043	10,409
Joomla	43,094	14,822	14,895	54,048	16,505	17,476	20,113	15,527	15,876	**54,711**	15,448	16,149
Drupal	80,195	26,251	28,655	**80,620**	23,290	25,105	70,998	59,998	68,236	74,428	28,272	30,291
HotCRP	**19,109**	8,772	8,777	17,737	10,517	11,415	17,063	14,871	14,918	15,647	5,463	5,509

the attempted payloads, instead relying on code execution, it is able to detect and report this missed vulnerability. This clearly highlights the need for dynamic vulnerability verification. Moreover, we note that this AJAX call requires a valid CSRF token, demonstrating the importance of our correct workflow execution.

Vanilla ISD detection. The Vanilla forums' vulnerability is a stored XSS that occurs when saving a new discussion as a draft, and is triggered when viewing the author's drafts. However, the sink of the injection (`/drafts`) is not visible anywhere on the application *before* saving the first draft and is only added to the home page afterwards. Therefore, the scanner would need to re-visit the home page and actively search for new URLs and visit them, in order for ReScan to discover the ISD link, all while before the scanner started fuzzing the source. This highlights the practicality of the background worker, which operates independently of the active scan and attempts to find such links before the scanner starts fuzzing the corresponding ISD source.

A. *Other Vulnerabilities*

While our evaluation focused on XSS, as they are the most prevalent bug among our applications and also allow for a direct comparison with recent work [25], ReScan aims to support any vulnerability type that scanners might test for. To that end, we conducted another set of experiments, where we picked known vulnerabilities from our applications and re-configured the scanners to use the corresponding auditing plugins, to assess whether our system can effectively facilitate their detection.

File upload. osCommerce suffers from an unrestricted file upload vulnerability [10], in the image-upload functionality for a new product category in the administration panel. Specifically, while the `.htaccess` file in the upload directory attempts to prevent access for a number of executable files, it does not prevent *all* of them. To uncover this vulnerability, we configured w3af (with ReScan present) to use its `file_upload` plugin and pointed it to the vulnerable upload form, which led to the successful upload of an executable file and was detected by the scanner. w3af without ReScan, however, can never reach the upload form as it cannot authenticate in the application, which also justifies its rather low coverage in Table III. While the root cause for w3af missing the vulnerability is not specific to the mechanics behind the vulnerability itself, this highlights the fact that having the necessary payload to trigger a vulnerability is only one aspect crucial to a scanner's success.

Login brute-forcing. For brute-forcing weak account credentials we opted to use Prestashop, due to its irregular login form functionality. In more detail, while logins are carried out through a regular HTML form with its `action` attribute set to the login page's URL, upon submission the form sends a request to a different endpoint; as such the scanner has no way of discovering it and successfully logging in. ReScan, correctly submits the form and follows all redirections, and manages to log into the application. To that end, we enabled w3af's *form_auth* brute-force plugin and pointed it to the administrator's login page. As expected, w3af by itself was unable to detect the correct credentials, while with ReScan it detected and reported the vulnerability.

Blind SQL injection. Since the applications in our dataset did not include any non-trivial SQL injection vulnerabilities (SQLi), we opted to install a vulnerable Wordpress plugin, namely GB Gallery Slideshow v1.5 [2]. One of the HTML forms generated by the plugin has a registered *onsubmit* event, which overrides the form's default functionality by sending an AJAX request with a completely different payload. One of the AJAX parameters is vulnerable to a blind SQLi [8], where successful exploitation is not directly visible on the landing page but is inferred based on the time required to get a response. We configured Wapiti to leverage its *blindsql* module and pointed it to the page containing the vulnerable form. As expected, wapiti without ReScan was able to fuzz the form's default structure but could not uncover the actual AJAX request that is sent when *correctly* submitting the form. In contrast, ReScan identified the AJAX request through its event discovery process and transcribed it to a static HTML form in the HTTP response so Wapiti could identify it too. Later, the scanner fuzzed the correct request, which was replayed by ReScan through the navigation model, leading to the detection of the vulnerability.

Overall, while certain ReScan components are tailored towards specific types of vulnerabilities, such as the ISD detection and FP/FN elimination for (stored) XSS, the remaining components effectively facilitate the detection of other types of vulnerabilities as they are not tied to the vulnerability itself. For instance, realistic rendering and interaction through a SotA browser, event discovery, correct workflow execution and maintaining the authenticated state are of crucial importance for two main reasons. First, all these aid scanners in discovering further application endpoints and functionalities to audit. Second, they are necessary for properly executing certain functionalities, which is a necessary prerequisite for the scanner to be able to correctly test its payloads and deploy its fuzzing strategy regardless of the vulnerability type being tested. As such, our system can effectively be applied to a large class of different vulnerabilities; by open-sourcing our code, we hope to facilitate other researchers in the field of black-box web application testing.

B. *URL Clustering*

It is critical to ensure that our URL clustering algorithm does not result in scanners missing relevant parts of the application.

mNDD threshold. To identify the optimal threshold for our mNDD metric, we performed the following experiment. For each of the applications, we manually compiled three sets of pages. The first set included pages with completely different URLs and functionalities which should *not* be clustered together, while the second set included pages with similar URLs *and* functionalities that should be clustered. Finally, the third set incorporated pages that had similar URLs, but should *not* be clustered due to different functionalities. We then proceeded to compute the mNDD score for each pair of pages within each set, where a higher value denotes different pages and a lower value indicates some similarity. For the pages that should not be clustered (1st and 3rd set), the minimum mNDD value was 0.014, setting an upper bound for the threshold. For the pages that should be grouped together, the maximum mNDD value was 0.009, indicating a lower bound for the threshold. As there is no overlap between the range of values for different and similar pages, we opt to use the lower of the two as our page similarity threshold (i.e., 0.009) to ensure that we avoid false positives where different pages that happen to have an mNDD slightly less than 0.014 are clustered. During this process we encountered a single false positive, in osCommerce, where two different pages were incorrectly clustered. This was due to the fact that the two pages were identical in structure even though they had different functionalities (adding an item and editing an existing item respectively). However, it is important to note, that this false positive is not limited to our mNDD approach, as the tree edit distance value is 0 for the regular NDD as well. We also note that while this threshold might not work for all applications, it is well-suited for most cases as our empirical analysis was conducted on a dataset that incorporates a diverse set of applications.

Correctness. To assess the correctness of our algorithm, we relied on the clustering rules that were set for each application during our main evaluation runs. Specifically, we inspected the different parameters that were clustered and proceeded to visit the corresponding pages both with the values that the scanner requested but were redirected, as well as the final redirection value too. We then observed whether the pages were in fact similar to each other or if they were incorrectly clustered together. Among all applications, this process yielded two cases of false positives. As expected, the first case was the aforementioned issue with osCommerce described during our similarity threshold experiment. The second case was for PhpBB, but only for w3af's scan. After analyzing the cause for this false positive, we deduced that it was not caused by our algorithm or the mNDD metric, but instead, was caused due to w3af's inability to maintain an authenticated session in PhpBB, even with ReScan. In more detail, PhpBB uses a randomly generated *sid* URL parameter in all administrator URLs and also sets the same value in a cookie. After logging in, w3af sent another request *without* cookies, ReScan's authentication helper re-established the session and the scanner ended up with two different values for *sid* and the cookie, effectively creating a mismatch between them and leading to unauthenticated responses. As a result, while it initially discovered the existence of post-login URLs, it could not properly request them and it kept getting an *invalid session* message in all responses, leading to the clustering of different pages. This, however, was not the case for the other two scanners, which got proper, authenticated responses for these pages and did not cluster them incorrectly.

Performance gain. To measure performance gain we further analyze wapiti's run on osCommerce as a representative case, since the application includes several similar pages that should be clustered and wapiti also takes the longest among all scanners to complete its operation. We then proceeded to re-run the scan

TABLE IV: Qualitative differences between ReScan and Black Widow.

Feature / System	**Black widow**	**ReScan**
Browser support	●	●
Navigation model	●	●
Inter-state dependencies	●	●
Event triggering	◑	●
- Handle XHR payloads	○	●
Authentication helper	◑	●
- Detect/configure credentials	○	●
- Dynamic state oracle	○	●
- Re-login	◑	●
- Retry failed edges	○	●
URL clustering	○	●
Concurrent workers	○	●

TABLE V: Detection and coverage comparison between the best run of ReScan and Black Widow for each app.

System	**Detection**				**Coverage**	
	ReScan		**Black Widow**		**ReScan**	**Black Widow**
App	R-XSS	S-XSS	R-XSS	S-XSS	LoC	
SCARF	-	**8**	-	4	**662**	593
WackoPicko	**3**	1	2	**2**	**1,009**	1,003
Wordpress	**1**	1	-	1	54,329	**62,281**
osCommerce	**3**	**16**	-	11	7,270	**12,193**
Vanilla	-	1	-	1	**12,951**	10,108
PhpBB	-	**4**	-	-	**10,487**	8,072
Prestashop	**1**	-	-	-	**103,955**	23,166
Joomla	-	-	-	-	**54,711**	50,240
Drupal	-	-	-	-	**80,620**	39,247
HotCRP	**1**	-	-	-	19,109	**23,241**

without the URL clustering module and without a maximum scan time. The scan with URL clustering enabled took 62,690 seconds (17.4 hours) while the other scan took 418,622 seconds (116.3 hours), resulting in a ∼6.7x speedup.

C. State-of-the-Art Comparison

Our system adopts a novel approach that allows it to leverage *any* underlying scanner. Nonetheless, we opt to compare it to Black Widow (BW) [25], a state-of-the-art scanner, that highlighted the need to combine features from multiple other systems and offers certain comparable features. This comparison highlights that even though BW was designed to incorporate multiple ideas from prior approaches, it was still built as a standalone tool, and thus is susceptible to the inherent limitations of a monolithic approach. In contrast, ReScan is designed to provide researchers with flexibility, allowing them to leverage the capabilities of any existing system of their choice, due to its middleware architecture.

Setup. To compare against BW, we downloaded and ran it on all applications. One minor change we made to its source code was to support user-defined credentials, as their system uses hardcoded values for *all* input fields, including usernames or emails and passwords. Moreover, we had to fix a few minor runtime exceptions that halted the tool's operation, and write a custom parser that de-duplicates the scanner's results. We stress that these changes were strictly limited to necessary modifications for the tool to execute properly and did not interfere with its overall approach or methodology. While their study ran each scanner for a maximum of eight hours in the evaluation, we opted to let BW run for up to one day.

Qualitative differences. As can be seen in Table IV, both systems use a fully-fledged browser, create a navigation model based on which they execute workflows and also uncover ISD links. While ISD detection is conceptually similar, ReScan does not have any control over when to crawl or fuzz each endpoint, highlighting the necessity of our BG worker approach. In more detail, BW always prioritizes form submissions over other edges and re-fetches *all* GET edges right before initiating the scanning phase; this results in it first fuzzying an ISD source and *then* visiting the corresponding sink, allowing for the timely detection of ISD links. In contrast, ReScan cannot make this assumption as it depends on the internals of each individual scanner; thus, we need to be more generalizable when handling ISD detection and need to account for arbitrary fuzzing and crawling orderings due to the underlying scanner design. Additionally, both systems can discover events and capture asynchronous requests and DOM changes. However, BW lacks the ability to set such requests' payloads dynamically, while ReScan leverages an internal proxy to do so after the request has left the browser. Regarding authentication, while BW can submit a login form (with hardcoded credentials), it cannot infer whether the login was successful or not. Moreover, while it can re-login to the application if needed, it only does so when presented with a login form. Applications' behavior varies and accessing an authenticated resource or exercising an authenticated functionality when logged out does not always result in a login page; thus their system will miss authenticated parts of target applications. On the other hand, ReScan automatically deduces a robust authentication oracle and consults it after the execution of *every* request and retries any operation that might have failed due to a broken session. Regarding similar pages, BW clusters pages and imposes a hard limit on how many similar pages they will test (if they share the same path but with possibly different URL query parameters) without considering the actual pages' content and functionality. This can incorrectly cluster pages that should be audited separately. Another main difference is that ReScan operates in a concurrent fashion, while BW is sequential, directly affecting its performance as seen in Figure 3. Finally, BW is a standalone tool that tests only for reflected and stored XSS, leaving a plethora of other flaws undetected. In contrast, our system operates as a generic enhancement middleware framework that can accurately replicate and potentially enhance virtually *any* test performed by scanners.

Quantitative differences. BW reported two *unique* reflected and 19 stored XSS among all applications, compared to ten and 34 detected by ReScan, as shown in Tables II and V. Excluding Drupal and Joomla, for which no vulnerabilities were detected by any scanner, in all but two of the remaining cases there is at least one ReScan-enabled scan that outperforms BW. In WackoPicko, BW manages to detect a stored XSS through a comment which needs to be previewed first and *then* posted. However, the vulnerable field is not present in the final submission form, only in the intermediary preview form. Due to this irregular structure and since scanners attack each form separately, despite ReScan correctly modeling and executing the workflow for the final submission form, scanners cannot detect the flaw as they try to fuzz that form's fields only. Regarding coverage, ReScan outperforms BW in seven out of ten applications, and overall offers an average improvement of 46% in reached LoC. For the three remaining cases where BW achieved better coverage, it was either due to the max scan time being reached by all scanners, and since BW prioritizes form submissions over other edges, it likely managed to execute more functionalities, BW visited URLs that were excluded from the other scanners, as stated in § IV, or it visited unauthenticated parts of the application due to a broken session. Nonetheless, ReScan still managed to detect *more vulnerabilities* in these cases as well. Performance-wise, as can be seen in Figure 3, BW reached the max scan time of one day in eight of the applications, highlighting the shortcomings of their sequential execution.

V. Limitations and Future Work

Categorization issues. In certain cases scanners might report a stored XSS as reflected: When ReScan appends an ISD sink in the HTTP response, the scanner will detect the vulnerability as a reflected XSS, as both the injection and its reflection occurred in the same request-response pair from the scanner's perspective. However, there is also a significant advantage to our technique, i.e., inspecting ISD sinks right after the injection in contrast to scanners' default behavior. Scanners will either visit discovered URLs at the end of the scan, looking for previous stored injections, or will try to re-inject their payloads and then inspect the URLs. These approaches, however, are not robust since a change in the application's state or content might render the detection impossible at this point. This could be due to a successful payload being overwritten by one that does not trigger the vulnerability, or the reflection page or injection point not being available anymore; directly checking ISD sinks solves this issue. Moreover, this miscategorization may also occur during scanners' regular operation, if a stored injection is reflected directly in the HTTP response. In this case, the scanner will initially classify the flaw as a reflected XSS, but later on, when checking for stored XSS it might be missed as one, due to the aforementioned reasons.

During our evaluation, we reported the discovered vulnerabilities based on their *actual* nature, despite being misclassified by the scanner. The rationale behind this decision is three-fold. First, this issue is inherent to scanners' behavior and further exacerbated by ReScan's techniques. More importantly, the vulnerable parameter has been detected, and the effort needed to patch it is the same regardless of the reported XSS type. Finally, ReScan's results provide all necessary information about each vulnerability, i.e., identifying the vulnerable parameter, the URL in which it is located, and where the injection is triggered.

Session sharing. As stated in § III, workers share the authenticated session based solely on the website's cookies.While this is sufficient for our experimental setup and application set, in practice, applications might utilize other APIs and functionalities for their state and session management (e.g., local/session storage, service workers etc). We plan to augment our session sharing among workers to support such alternative approaches as part of our future work.

False positives & negatives. ReScan aims to eliminate FPs and FNs specifically for XSS flaws. Implementing this capability for other types of vulnerabilities requires inferring *what* vulnerability is being tested in each request and *how* successful exploitations would be verified. We consider the development of such modules for different classes of flaws as part of our future work.

Overhead. Leveraging a fully-fledged browser to appropriately execute every request, and employing our numerous enhancement techniques, imposes a considerable overhead in the overall scanning time. However, due to the significant improvement in code coverage and vulnerability detection, as well as due to ReScan outperforming the current state of the art in most cases [25], we believe this to be an acceptable trade-off which renders the deployment of our system feasible. Nonetheless, we consider the exploration of additional optimization techniques an interesting future direction. For instance, ReScan could identify requests that do not require our enhancement techniques (e.g., edges with no ISD effects or that do not require precise workflow execution) and directly proxy them to the web application.

Ethical considerations. We note that all vulnerabilities detected during our evaluation have already been disclosed to the corresponding vendors by prior studies or researchers.

VI. Related Work

Web application scanning has received considerable attention from the research community through the years, as both black-box [25], [31], [19], [49], [22], [55], [54] and white- or grey-box [28], [14], [26], [30], [37], [60], [52], [29], [53], [45] techniques have been proposed and thoroughly evaluated. Moreover, several studies have carried out extensive comparisons between black-box vulnerability scanners [15], [20], [61], [58], [46], collectively agreeing that such tools suffer from certain core limitations, such as detecting and correctly modelling all injection points, replaying the necessary steps to perform an injection, or their inadequacy on persisting the authentication state. In the following paragraphs, we relate our work to the most prominent black-box approaches proposed in prior work.

As early as 2006, Kals et al. [31] designed a simple web application vulnerability scanner that visited pages, extracted HTML forms and tested for common XSS and SQL injection payloads. Later, Doupé et al. [19] highlighted the importance of taking the application's state into account for better coverage and implemented their approach in a *state-aware crawler* which, however, only considered static HTML links and forms for detecting state changes, navigating *and* clustering similar pages together. As we outlined, these features only, fall short for navigation and can also incorrectly cluster pages that offer different functionality (e.g., through input elements or buttons *outside* an HTML form). In another line of work, Duchène et al. [23], [22] inferred a control flow model for the application under test and an attack grammar to generate appropriate payloads for detecting XSS flaws; however, their approaches do not consider client-side events and require the ability to reset the application. Pellegrino et al. proposed jÄk [49], which considered client-side events towards covering a larger part of the application, but did not use a fully-fledged browser and only considered reflected XSS. More recently, Eriksson et al. [25] developed an XSS scanner that tackled some of the limitations we address in our paper. We provide a detailed comparison in §IV-C. Finally, a multitude of other works have focused on identifying specific flaws, such as client-side XSS [36], [55], [54], CSRF [16], [33] and unrestricted file uploads [35].

The common thread among all these works is that they suffer from at least one of the core limitations we highlight in this study. Moreover, they all constitute standalone tools that target a specific selection of vulnerability types. In contrast, ReScan aims to address these challenges in a scanner-agnostic manner irrespectively of the specific tests carried out by each scanner.

URL clustering has also been used in different domains, e.g., the detection of phishing pages. Such systems deduce page similarity either through *visual* analysis [41], [40], [13], [18] and comparing benign to suspicious webpages, or by utilizing URL and HTML related features that mainly focus on a page's textual contents [38], [34], [59]. Despite being successful for their respective goals, such approaches are not suitable in our context. Specifically, a visual representation of a page might not reflect the subtle yet important elements that denote different functionality, such as *hidden* buttons or input elements [39]. Moreover, two pages' textual contents might differ significantly even in conceptually similar pages (e.g., two product pages with different reviews). Fundamentally, our system aims to cluster pages that essentially offer the same *functionality* regardless of their specific contents and precise appearance.

VII. Conclusions

With web browsers and applications incorporating and supporting complex new features and functionality, vulnerability scanners that operate through raw HTTP requests are facing considerable obstacles that hinder their detection capabilities. Nonetheless, developing an alternative scanner for the modern web ecosystem that replicates all the features offered by existing scanners would require an exorbitant and infeasible amount of engineering effort. Alternatively, we have opted for a strategy that allows for both *forward* and *backward* compatibility, as our system mediates communication between applications and scanners that already exist or will be developed in the future. Apart from mediating communication with a fully-fledged modern browser, our framework also includes enhancement modules that tackle multiple limitations that affect state-of-the-art scanners. Our experimental evaluation demonstrated how our framework significantly improves the detection of vulnerabilities in both benchmark and modern web applications.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation under grants CNS-1934597, CNS-2211574, CNS-2143363. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the NSF. This work has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 883275 (HEIR), No. 883540 (PUZZLE) and No. 101021659 (SENTINEL).

References

[1] "spiderman — w3af - Open Source Web Application Security Scanner," 2013, http://w3af.org/plugins/crawl/spider_man.

[2] "GB Gallery Slideshow - WordPress plugin — WordPress.org," 2014, https://wordpress.org/plugins/gb-gallery-slideshow/.

[3] "Vega Tutorial - How to Set Up Vega to Work with Browser," 2018, https://rkhal101.github.io/_posts/WAVS/vega/vega_browser_setup.

[4] "html5lib · PyPI," 2020, https://pypi.org/project/html5lib/.

[5] "OWASP. Owasp zed attack proxy (zap)," 2020, https://www.zaproxy.org/.

[6] "MutationObserver - Web APIs — MDN," 2021, https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[7] "beautifulsoup4 · PyPI," 2022, https://pypi.org/project/beautifulsoup4/.

[8] "Blind SQL Injection — OWASP Foundation," 2022, https://owasp.org/www-community/attacks/Blind_SQL_Injection.

[9] "OWASP ZAP - Getting Started," 2022, https://www.zaproxy.org/getting-started/.

[10] "Unrestricted File Upload — OWASP Foundation," 2022, https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload.

[11] "ReScan-evaluated applications' Docker images," 2023, https://gitlab.com/kostasdrk/rescanApps.

[12] "ReScan repository," 2023, https://gitlab.com/kostasdrk/rescan.

[13] S. Abdelnabi, K. Krombholz, and M. Fritz, "Visualphishnet: Zero-day phishing website detection by visual similarity," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[14] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy*, 2008.

[15] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE Symposium on Security and Privacy*, 2010.

[16] S. Calzavara, M. Conti, R. Focardi, A. Rabitti, and G. Tolomei, "Mitch: A machine learning approach to the black-box detection of csrf vulnerabilities," in *IEEE European Symposium on Security and Privacy*, 2019.

[17] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, “mitmproxy: A free and open source interactive HTTPS proxy,” 2022, https://mitmproxy.org/.

[18] F. C. Dalgic, A. S. Bozkir, and M. Aydos, “Phish-iris: A new approach for vision based brand prediction of phishing web pages via compact visual descriptors,” *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, 2018.

[19] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, “Enemy of the state: A state-aware black-box web vulnerability scanner,” in *21st USENIX Security Symposium*, 2012.

[20] A. Doupé, M. Cova, and G. Vigna, “”why johnny can’t pentest: An analysis of black-box web vulnerability scanners”,” in *”Detection of Intrusions and Malware, and Vulnerability Assessment”*, 2010.

[21] K. Drakonakis, S. Ioannidis, and J. Polakis, “The cookie hunter: Automated black-box auditing for web authentication and authorization flaws,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[22] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, “Kameleonfuzz: Evolutionary fuzzing for black-box xss detection,” in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.

[23] F. Duchène, S. Rawat, J.-L. Richier, and R. Groz, “Ligre: Reverse-engineering of control and data flow models for black-box xss detection,” in *20th Working Conference on Reverse Engineering*, 2013.

[24] S. el Idrissi, N. Berbiche, F. Guerouate, and S. Mohamed, “Performance evaluation of web application security scanners for prevention and protection against vulnerabilities,” *International Journal of Applied Engineering Research*, 2017.

[25] B. Eriksson, G. Pellegrino, and A. Sabelfeld, “Black widow: Blackbox data-driven web scanning,” in *IEEE Symposium on Security and Privacy*, 2021.

[26] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, “Toward automated detection of logic vulnerabilities in web applications,” in *Proceedings of the 19th USENIX Conference on Security*, 2010.

[27] R. Fielding and J. Reschke, “RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” 2014, https://tools.ietf.org/html/rfc7231#section-4.2.1.

[28] W. G. Halfond, S. R. Choudhary, and A. Orso, “Penetration testing with improved input vector identification,” in *International Conference on Software Testing Verification and Validation*, 2009.

[29] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai, “Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis,” in *24th International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[30] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, “Securing web application code by static analysis and runtime protection,” in *Proceedings of the 13th International Conference on World Wide Web*, 2004.

[31] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, “Secubat: A web vulnerability scanner,” in *Proceedings of the 15th International Conference on World Wide Web*, 2006.

[32] S. Karami, P. Ilia, and J. Polakis, “Awakening the web’s sleeper agents: Misusing service workers for privacy leakage,” in *Network and Distributed System Security Symposium*, 2021.

[33] S. Khodayari and G. Pellegrino, “JAW: Studying client-side CSRF with hybrid property graphs and declarative traversals,” in *30th USENIX Security Symposium*, 2021.

[34] J. Lee, P. Ye, R. Liu, D. M. Divakaran, and M. Chan, “Building robust phishing detection system: an empirical analysis,” in *Workshop on Measurements, Attacks, and Defenses for the Web*, 2020.

[35] T. Lee, S. Wi, S. Lee, and S. Son, “FUSE: Finding file upload bugs via penetration testing,” in *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[36] S. Lekies, B. Stock, and M. Johns, “25 million flows later: Large-scale detection of dom-based xss,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[37] X. Li, W. Yan, and Y. Xue, “Sentinel: Securing database from logic flaws in web applications,” in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, 2012.

[38] Y. Li, Z. Yang, X. Chen, H. Yuan, and W. Liu, “A stacking model using url and html features for phishing webpage detection,” *Future Generation Computer Systems*, 2019.

[39] X. Lin, P. Ilia, and J. Polakis, “Fill in the blanks: Empirical analysis of the privacy threats of browser form autofill,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[40] Y. Lin, R. Liu, D. M. Divakaran, J. Y. Ng, Q. Z. Chan, Y. Lu, Y. Si, F. Zhang, and J. S. Dong, “Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[41] R. Liu, Y. Lin, X. Yang, S. H. Ng, D. M. Divakaran, and J. S. Dong, “Inferring phishing intention via webpage appearance and dynamics: A deep vision based approach,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[42] F. Marcantoni, M. Diamantaris, S. Ioannidis, and J. Polakis, “A large-scale study on the risks of the html5 webapi for mobile sensor-based attacks,” in *The World Wide Web Conference*, 2019, pp. 3063–3071.

[43] A. Mesbah, E. Bozdag, and A. van Deursen, “Crawling ajax by inferring user interface state changes,” in *Eighth International Conference on Web Engineering*, 2008.

[44] H. Nikšić, “Wget - gnu project - free software foundation,” 2020, https://www.gnu.org/software/wget/.

[45] S. Park, D. Kim, S. Jana, and S. Son, “FUGIO: Automatic exploit generation for PHP object injection vulnerabilities,” in *31st USENIX Security Symposium*, 2022.

[46] M. Parvez, P. Zavarsky, and N. Khoury, “Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities,” in *10th International Conference for Internet Technology and Secured Transactions*, 2015.

[47] M. Pawlik and N. Augsten, “Efficient computation of the tree edit distance,” *ACM Trans. Database Syst.*, vol. 40, 2015.

[48] ——, “Tree edit distance: Robust and memory-efficient,” *Information Systems*, vol. 56, 2016.

[49] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, “jäk: Using dynamic analysis to crawl and test modern web applications,” in *Research in Attacks, Intrusions, and Defenses*, 2015.

[50] D. Rethans, “Xdebug - Debugger and Profiler Tool for PHP,” 2021, https://xdebug.org/.

[51] A. Riancho, “w3af - open source web application security scanner,” 2013, http://w3af.org/.

[52] O. v. Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, “webfuzz: Grey-box fuzzing for web applications,” in *European Symposium on Research in Computer Security*, 2021.

[53] M. Shcherbakov and M. Balliu, “Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web,” in *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.

[54] M. Steffens, C. Rossow, M. Johns, and B. Stock, “Don’t trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild,” in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[55] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, “From facepalm to brain bender: Exploring client-side cross-site scripting,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[56] C. Sullo, “Nikto2 — CIRT.net,” 2022, https://cirt.net/Nikto2.

[57] N. Surribas, “Wapiti : a Free and Open-Source web-application vulnerability scanner,” 2021, https://wapiti.sourceforge.io/.

[58] L. Suto and C. San, “Analyzing the accuracy and time costs of web application security scanners,” 2010.

[59] R. Verma and K. Dyer, “On the character of phishing urls: Accurate and robust statistical learning classifiers,” in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[60] A. Vernotte, F. Lebeau, F. Dadeau, B. Legeard, F. Peureux, and F. Piat, “Efficient detection of multi-step cross-site scripting vulnerabilities,” in *10th International Conference on Information Systems Security*, 2014.

[61] M. Vieira, N. Antunes, and H. Madeira, “Using web security scanners to detect vulnerabilities in web services,” in *IEEE/IFIP International Conference on Dependable Systems Networks*, 2009.

[62] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, “Maneuvering around clouds: Bypassing cloud-based security providers,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[63] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, 1989.

Appendix

Here we provide additional details about our framework's internal workings, to further shed light on our design decisions and the technical challenges addressed by our approach.

A. *Static file extensions*

Requests towards the following file extensions are directly proxied to the web application: `js`, `json`, `css`, `crt`, `mp3`, `wav`, `wma`, `ogg`, `mkv`, `zip`, `gz`, `tar`, `xz`, `rar`, `z`, `deb`, `iso`, `csv`, `tsv`, `dat`, `txt`, `log`, `sql`, `xml`, `mdb`, `apk`, `bat`, `bin`, `exe`, `jar`, `wsf`, `fnt`, `fon`, `otf`, `ttf`, `ai`, `bmp`, `gif`, `ico`, `jpeg`, `png`, `ps`, `psd`, `svg`, `tif`, `tiff`, `cer`, `rss`, `key`, `odp`, `pps`, `ppt`, `pptx`, `c`, `class`, `cpp`, `cs`, `h`, `java`, `sh`, `swift`, `vb`, `odf`, `xlr`, `xls`, `xlsx`, `bak`, `cab`, `cfg`, `cpl`, `cur`, `dll`, `dmp`, `drv`, `icns`, `ini`, `lnk`, `msi`, `sys`, `tmp`, `3g2`, `3gp`, `avi`, `flv`, `h264`, `m4v`, `mov`, `mp4`, `mpeg`, `rm`, `swf`, `vob`, `wmv`, `doc`, `docx`, `odt`, `pdf`, `rtf`, `tex`, `wks`, `wps`, `wpd`, `woff`, `eot`, `xap`.

B. *Navigation Model Details*

Edge IDs. As mentioned in §III, each edge is assigned a unique ID consisting of its type, the destination URL, and the set of parameter names in the payload. We ignore the payload values as they are volatile and can be altered, e.g., when the scanner fuzzes a form. The payload information is required since two forms may point to the same URL, but perform different operations in the back-end, e.g., a login and a signup form that both POST data to the `/auth.php` endpoint. We need a way to distinguish the two different transitions and the edge type and destination URL alone would not suffice. Additionally, edges are annotated with any information required to replay them, e.g., for form edges we store the specific element's unique CSS selector in the page, while for events we also store the exact event type.

Mapping requests to edges. Since our system's input are raw HTTP requests, we need a way to map them to existing edges in the navigation model, so we can properly replay the correct workflow. An incoming request can initially be assigned only two of the five edge types, i.e., `GET` or `FORM`, depending on the HTTP method and whether it carries a payload. However, in practice, the request might consider a different edge type, e.g., an asynchronous POST request triggered by an event. To handle such cases, and only when we cannot locate an edge with the initial edge type, we simply change the type and check again. In more detail, a simple GET request can be mapped to a `GET`, `IFRAME`, `EVENT` or `REDIRECT` edge, since it does not include a payload. In contrast, a payload-bearing request can either be mapped to a `FORM` or an `EVENT` edge, since these are the only edge types that can transmit a payload.

Arbitrary requests. The model is constructed based on the edges the system observes in each page, i.e., links, forms, events, iframes and redirections. However, scanners are not restricted to sending these exact same requests, which we can map and replay based on the model. Instead, they can send arbitrary requests to *virtually* any endpoint of the application. For instance, they can extract the URL *example.com/user/auth/* from a form action (which is included in the navigation model) and send a request to a part of that URL (e.g., *example.com/user/*) that has not been observed by the system so far and is *not* included in the model. Depending on the request's type, we tackle this issue in two ways. For simple `GET` requests, we simply execute the request, as `GET` requests are not considered state-changing. In contrast, for arbitrary requests that include a payload (i.e., corresponding to `FORM` or `EVENT` edges) we don't have any information on the necessary workflow needed to properly replay them, which might not even exist in the first place. In this case, we employ a best-effort approach where we generate a form matching the request (same input fields, action and HTTP method) on the fly and submit it. This way we can at least get a properly rendered response through the browser's environment.

Model reuse. It is worth noting that the constructed navigation model is not scanner-specific, but rather a generic, high-level representation of the web application. In practice, this allows the reuse of existing models either when reconfiguring a scanner or performing a completely new scan with another tool, effectively limiting the costly parts of ReScan's processing to the first run on a new application.

C. *Browser Workers Implementation Details*

Request headers. One more aspect we need to consider are the HTTP request headers sent by the scanner in each request. Generally, we want to avoid sending the scanner's headers and need the browser to send its own to ensure realistic interaction (e.g., user-agent, accept-encoding etc.). However, there are a couple of exceptions to this. Firstly, any cookies sent by the scanner *must* be passed through unchanged, which we detail in §III-A. Another case in which we need to preserve the scanner's original request headers is when it tries to inject a payload through them. To do this, we observe the first incoming requests to learn the scanner's default header values and, for subsequent requests, pass through any headers that had their default values changed.

Events in workflows. Another special case involves `EVENT` edges when executing workflows. The main idea is that a client-side event that reveals further edges when triggered (e.g., other events or a form), might not be required to be triggered more than once to reveal these elements; on the contrary, triggering the event again might make these elements disappear or become inaccessible (e.g., removing a dynamically generated form). To address such cases, when encountering an `EVENT` edge, we perform a look-ahead operation on the workflow. In more detail, we check if the last edge (i.e., the one corresponding to the initial request) is already present and can be executed, in which case we simply ignore all intermediary edges. If it does not exist, we check for the second to last edge and repeat the same process, until we either find an edge to jump to and continue the execution from there, or end up on the event edge at hand and continue as normal.

CSRF tokens. When a scanner sends a request containing a CSRF token, it will include a stale token value, as it was captured in a previous request. During the execution of its workflow ReScan will acquire a fresh token value, but in order to correctly submit the request, it needs to replace the stale scanner value with the new one. To achieve this after executing the workflow, when we are ready to send the request (i.e., submit the form or trigger the event) we check each payload parameter against common token and nonce keywords and simply ignore the scanner-provided value. For forms we iteratively inspect the form's input elements statically in the DOM, while for event-originating requests we perform the inspection on-the-wire using XDriver's internal proxy. Since this is a best-effort approach based on common naming conventions for CSRF tokens, it is important to note that even if ReScan misses an actual token (i.e., incorrectly assumes it is a regular parameter) it does not negatively affect the underlying scanner, which would miss it even without our system. Moreover, scanners can typically be configured to ignore specific parameters, such as CSRF tokens; in such cases, this mechanism would essentially be idle as the scanner would not fuzz the tokens.

D. Request Processing Performance

In Figure 4, we show the total processing time required for each request handled by ReScan per application and per individual component. We note that each CDF has been calculated using the aggregated requests from *all* scanners for that application. This is due to the fact that the time required to handle each request is irrelevant to the scanner that initiated it, but heavily depends on the application's characteristics (e.g. usage and number of Javascript events, number of detected ISD sinks). Additionally, the totals that were used to calculate the different components' CDF differ, as not all of them are executed on each request. For instance, retrieving and executing a workflow is applicable to all requests. However, triggering events is only relevant to pages that include them and have not been explored before, and collecting ISD sinks is only relevant to edges for which we have detected them.

As expected, retrieving the correct workflow is negligible as it can be fetched in less than a second for $\sim$95% of the incoming requests among all applications. Executing the workflow depends on its length, i.e. how many steps are needed to correctly execute the request. For most applications it takes *up to* three seconds on average, while for Wordpress and Prestashop it takes five seconds. This is expected as these two applications have generally slower response and rendering times, due to their heavy use of Javascript. Regarding the event discovery process, on average it takes between one and three seconds for nine applications, while Prestashop needs 19 seconds. However, for 95% of requests Prestashop required up to 479 seconds, while osCommerce, Drupal, Joomla and Wordpress took up to 47, 26, 21 and 19 seconds respectively. This is expected as we perform the event discovery at least once per page, which can be quite costly for pages with numerous events. For requests that were associated with ISD sinks that needed to be fetched, on average it took less than two seconds for four applications, up to six, nine and 10 seconds for another four, 11 seconds for Wordpress and 16 for Prestashop. Crafting the final HTTP response, as well as submitting the newly discovered edges to the graph worker, can be completed within one second for all applications. Executing the authentication oracle after each request could possibly be quite time consuming. In reality, for most applications it takes less than two seconds for 99% of the requests, e.g. SCARF and Wackopicko require less than a second for almost all requests, while HotCRP and PhpBB take less than a second for 96% of them. The slowest case, as expected, was again Prestashop, which needed up to nine seconds to run the oracle for 90% of incoming requests. Finally, regarding the total processing time per request, on average each one can be completed within three seconds for four of the applications, up to five seconds for another two apps, up to eight, 10 and 14 seconds for HotCRP, PhpBB and Wordpress respectively, while Prestashop needed up to 21 seconds.

E. ReScan's API

ReScan's API endpoints, which we detail next, can be split in two categories: *passive*, which aim to provide valuable insights and information sharing between ReScan and the scanner, and *active*, which alter ReScan's behavior at runtime, according to the scanner's needs. For instance, if the scanner tests for stored XSS it can enable the ISD module, but disable it later when testing for a vulnerability that does not affect other pages, e.g., open redirects. Similarly, it might disable the authentication helper when it is not required to maintain the session and avoid running the oracle in every request, e.g., when brute-forcing for sensitive files and directories. Finally, disabling the URL clustering module, which will no longer create any new clusters nor redirect already clustered pages, can be useful if the scanner wants to perform thorough checks in all pages, e.g., harvest e-mails or registered users.

To utilize the API, the scanner simply sends its API requests to ReScan through the same port it uses to proxy the requests targeting the web application, i.e., via the same HTTP channel. ReScan then identifies these API requests and performs the requested operation. It is worth noting that different components of the system are responsible for handling different API calls. For example, retrieving the entire navigation model or a specific workflow requires communicating with the *graph worker*, checking the authentication state requires invoking a *browser worker*, while enabling or disabling modules is handled by the *orchestrator*.

Passive endpoints:

- `/graph`: Return the entire navigation model for the target application.
- `/workflow`: Given a request, return its workflow from the model.
- `/isd`: Given a request, return the detected ISD sinks.
- `/isdAll`: Return all detected ISD sinks and sources.
- `/auth`: Return if we are currently authenticated.
- `/xss`: Return all successful XSS injections so far.
- `/isClustered`: Given a URL, return if clustered and what cluster it belongs to.

Active endpoints:

- `/[enable|disable]ISD`: Enable/disable ISD detection and sink collection.
- `/[enable|disable]Auth`: Enable/disable authentication helper.
- `/[enable|disable]Events`: Enable/disable event discovery.
- `/[enable|disable]Clustering`: Enable/disable URL clustering.

F. Scanners Configuration

- **w3af.** We enabled the `web_spider` plugin for crawling, the `auth.generic` plugin for authentication and the `xss` plugin for auditing with the `persistent_xss` parameter set to true for vanilla runs and false for ReScan.
- **wapiti.** We used the `wapiti-getcookie` utility for authentication, and enabled the `xss` and `permanentxss` auditing modules, disabling the latter for ReScan.
- **Enemy of the State.** The following command was issued:

```
$ jython crawler2.py <target URL>
```

- **ZAP.** The `spider` and `ajaxSpider` plugins were used for crawling (`ajaxSpider` was disabled for ReScan) and ZAP's standard authentication module. For auditing, we enabled the `xss` module, which incorporates both reflected and stored XSS detection, while for ReScan we enabled only the `xss_reflected` module.

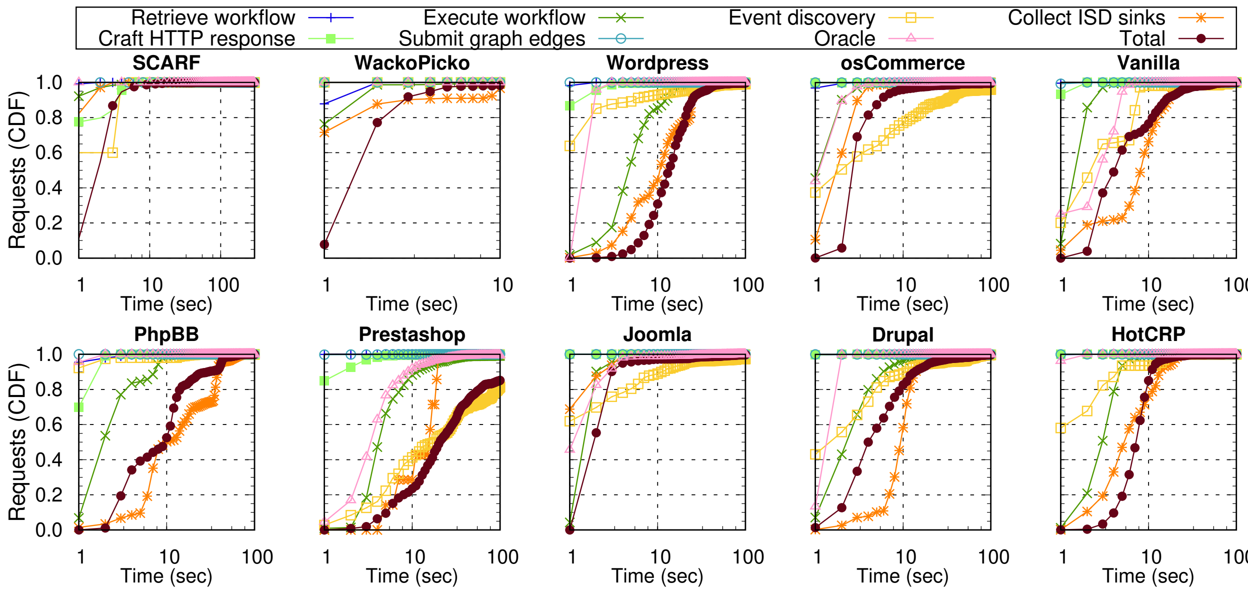

Fig. 4: Requests' CDF per application, in terms of total as well as individual components' processing time.

- **Black widow.** The following command was issued:

```
$ ./crawl.py --url <target URL>
--username <username> --passwd <password>
```

- **ReScan.** The following command was issued:

```
$ ./rescan.py --headless --workers 4
--isd --events --clustering --auth
--port <proxy port>
```