Enemy of the State:
A State-Aware Black-Box Web Vulnerability Scanner

Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna
University of California, Santa Barbara
{adoupe, cavedon, chris, vigna}@cs.ucsb.edu

Abstract

Black-box web vulnerability scanners are a popular choice for finding security vulnerabilities in web applications in an automated fashion. These tools operate in a *point-and-shoot* manner, testing any web application—regardless of the server-side language—for common security vulnerabilities. Unfortunately, black-box tools suffer from a number of limitations, particularly when interacting with complex applications that have multiple actions that can change the application's state. If a vulnerability analysis tool does not take into account changes in the web application's state, it might overlook vulnerabilities or completely miss entire portions of the web application.

We propose a novel way of inferring the web application's internal state machine *from the outside*—that is, by navigating through the web application, observing differences in output, and incrementally producing a model representing the web application's state.

We utilize the inferred state machine to drive a black-box web application vulnerability scanner. Our scanner traverses a web application's state machine to find and fuzz user-input vectors and discover security flaws. We implemented our technique in a prototype crawler and linked it to the fuzzing component from an open-source web vulnerability scanner.

We show that our state-aware black-box web vulnerability scanner is able to not only exercise more code of the web application, but also discover vulnerabilities that other vulnerability scanners miss.

1 Introduction

Web applications are the most popular way of delivering services via the Internet. A modern web application is composed of a back-end, server-side part (often written in Java or in interpreted languages such as PHP, Ruby, or Python) running on the provider's server, and a client part running in the user's web browser (implemented in JavaScript and using HTML/CSS for presentation). The two parts often communicate via HTTP over the Internet using Asynchronous JavaScript and XML (AJAX) [20].

The complexity of modern web applications, along with the many different technologies used in various abstraction layers, are the root cause of vulnerabilities in web applications. In fact, the number of reported web application vulnerabilities is growing sharply [18, 41].

The occurrence of vulnerabilities could be reduced by better education of web developers, or by the use of security-aware web application development frameworks [10, 38], which enforce separation between structure and content of input and output data. In both cases, more effort and investment in training is required, and, therefore, cost and time-to-market constraints will keep pushing for the current fast-but-insecure development model.

A complementary approach for fighting security vulnerabilities is to discover and patch bugs before malicious attackers find and exploit them. One way is to use a white-box approach, employing static analysis of the source code [4, 15, 17, 24, 28]. There are several drawbacks to a white-box approach. First, the potential applications that can be analyzed is reduced to only those applications that use the target programming language. In addition, there is the problem of substantial false positives. Finally, the source code of the application itself may be unavailable.

The other approach to discovering security vulnerabilities in web applications is by observing the application's output in response to a specific input. This method of analysis is called *black-box* testing, as the application is seen as a sealed machine with unobservable internals. Black-box approaches are able to perform large-scale analysis across a wide range of applications. While black-box approaches usually have fewer false positives than white-box approaches, black-box approaches suffer

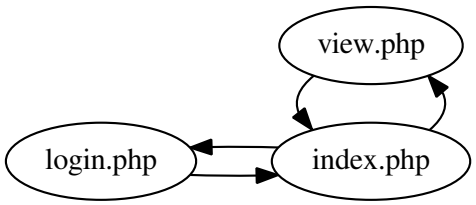

Figure 1: Navigation graph of a simple web application.

from a discoverability problem: They need to reach a page to find vulnerabilities on that page.

Classical black-box web vulnerability scanners crawl a web application to enumerate all reachable pages and then fuzz the input data (URL parameters, form values, cookies) to trigger vulnerabilities. However, this approach ignores a key aspect of modern web applications: Any request can change the state of the web application.

In the most general case, the state of the web application is any data (database, filesystem, time) that the web application uses to determine its output. Consider a forum that authenticates users, an e-commerce application where users add items to a cart, or a blog where visitors and administrators can leave comments. In all of these modern applications, the way a user interacts with the application determines the application's state.

Because a black-box web vulnerability scanner will never detect a vulnerability on a page that it does not see, scanners that ignore a web application's state will only explore and test a (likely small) fraction of the web application.

In this paper, we propose to improve the effectiveness of black-box web vulnerability scanners by increasing their capability to understand the web application's internal state. Our tool constructs a partial model of the web application's state machine in a fully-automated fashion. It then uses this model to fuzz the application in a state-aware manner, traversing more of the web application and thus discovering more vulnerabilities.

The main contributions of this paper are the following:

- A black-box technique to automatically learn a model of a web application's state.
- A novel vulnerability analysis technique that leverages the web application's state model to drive fuzzing.
- An evaluation of our technique, showing that both code coverage and effectiveness of vulnerability analysis are improved.

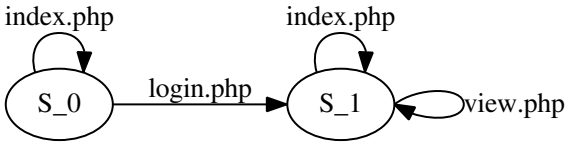

Figure 2: State machine of a simple web application.

2 Motivation

Crawling modern web applications means dealing with the web application's changing state. Previous work in detecting workflow violations [5, 11, 17, 30] focused on navigation, where a malicious user can access a page that is intended only for administrators. This unauthorized access is a violation of the developer's intended workflow of the application.

We wish to distinguish a navigation-based view of the web application, which is simply derived from crawling the web application, from the web application's internal state machine. To illustrate this important difference, we will use a small example.

Consider a simple web application that has only three pages, `index.php`, `login.php`, and `view.php`. The `view.php` page is only accessible after the `login.php` page is accessed. There is no logout functionality. A client accessing this web application might make a series of requests like the following:
⟨`index.php`, `login.php`, `index.php`, `view.php`, `index.php`, `view.php`⟩

Analyzing this series of requests from a navigation perspective creates a navigation graph, shown in Figure 1. This graph shows which page is accessible from every other page, based on the navigation trace. However, the navigation graph does not represent the information that `view.php` is only accessible after accessing `login.php`, or that `index.php` has changed after requesting `login.php` (it includes the link to `view.php`).

What we are interested in is not how to navigate the web application, but how the requests we make influence the web application's internal state machine. The simple web application described previously has the internal state machine shown in Figure 2. The web application starts with the internal state `S_0`. Arrows from a state show how a request affects the web application's internal state machine. In this example, in the initial state, `index.php` does not change the state of the application, however, `login.php` causes the state to transition from `S_0` to `S_1`. In the new state `S_1`, both `index.php` and `view.php` do not change the state of the web application.

The state machine in Figure 2 contains important information about the web application. First, it shows that `login.php` permanently changes the web application's

state, and there is no way to recover from this change. Second, it shows that the `index.php` page is seen in two different states.

Now the question becomes: "How does knowledge of the web application's state machine (or lack thereof) affect a black-box web vulnerability scanner?" The scanner's goal is to find vulnerabilities in the application, and to do so it must fuzz as many execution paths of the server-side code as possible[1]. Consider the simple application described in Figure 2. In order to fuzz as many code paths as possible, a black-box web vulnerability scanner must fuzz the `index.php` page in both states `S_0` and `S_1`, since the code execution of `index.php` can follow different code paths depending on the current state (more precisely, in state `S_1`, `index.php` includes a link to `view.php`, which is not present in `S_0`).

A black-box web vulnerability scanner can also use the web application's state machine to handle requests that change state. For example, when fuzzing the `login.php` page of the sample application, a fuzzer will try to make several requests to the page, fuzzing different parameters. However, if the first request to `login.php` changes the state of the application, all further requests to `login.php` will no longer execute along the same code path as the first one. Thus, a scanner must have knowledge of the web application's state machine to test if the state was changed, and if it was, what requests to make to return the application to the previous state before continuing the fuzzing process.

We have shown how a web application's state machine can be leveraged to improve a black-box web vulnerability scanner. Our goal is to infer, in a black-box manner, as much of the web application's state machine as possible. Using only the sequence of requests, along with the responses to those requests, we build a model of as much of the web application's state machine as possible. In the following section, we describe, at a high level, how we infer the web application's state machine. Then, in Section 4, we provide the details of our technique.

3 State-Aware Crawling

In this section, we describe our state-aware crawling approach. In Section 3.1, we describe web applications and define terms that we will use in the rest of the paper. Then, in Section 3.2, we describe the various facets of the state-aware crawling algorithm at a high level.

[1]Hereinafter, we assume that the scanner relies on fuzzer-based techniques. However, any other automated vulnerability analysis technique would benefit from our state-aware approach.

3.1 Web Applications

Before we can describe our approach to inferring a web application's state, we must first define the elements that come into play in our web application model.

A web application consists of a server component, which accepts HTTP requests. This server component can be written in any language, and could use many different means of storage (database, filesystem, memcache). After processing a request, the server sends back a response. This response encapsulates some content, typically HTML. The HTML content contains links and forms which describe how to make further requests.

Now that we have described a web application at a high level, we need to define specific terms related to web applications that we use in the rest of this paper.

- Request—The HTTP request made to the web application. Includes anything (typically in the form of HTTP headers) that is sent by the user to the web application: the HTTP Method, URL, Parameters (`GET` and `POST`), Cookies, and User-Agent.
- Response—The response sent by the server to the user. Includes the HTTP Response Code and the content (typically HTML).
- Page—The HTML page that is contained in the response from a web application.
- Link—Element of an HTML page that tells the browser how to create a subsequent request. This can be either an anchor or a form. An anchor always generates a `GET` request, but a form can generate either a `POST` or `GET` request, depending on the parameters of the form.
- State—Anything that influences the web application's server-side code execution.

3.1.1 Web Application Model

We use a *symbolic Mealy machine* [7] to model the web application as a black-box. A Mealy machine is an automaton where the input to the automaton, along with the current state, determines the output (i.e., the page produced by the response) and the next state. A Mealy machine operates on a finite alphabet of input and output symbols, while a symbolic Mealy machine uses an infinite alphabet of input and output symbols.

This model of a web application works well because the input to a web application, along with the current state of the web application, determines the output and the next state. Consider a simple e-commerce web application with the state machine show in Figure 3. In this state graph, all requests except for the ones leaving a state

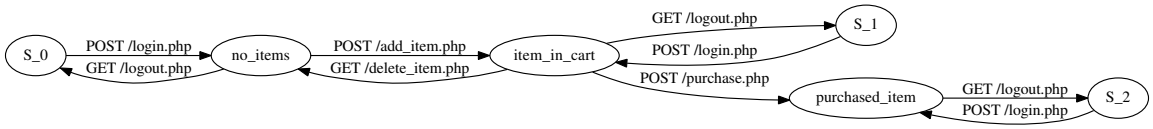

Figure 3: The state machine of a simple e-commerce application.

bring the application back to the same state. Therefore, this state graph does not show all the request that can be made to the application, only the subset of requests that change the state.

For instance, in the initial state `S_0`, there is only one request that will change the state of the application, namely `POST /login.php`. This change logs the user into the web application. From the state `no_items`, there are two requests that can change the state, `GET /logout.php` to return the user to state `S_0` and `POST /add_item.php` to add an item to the user's shopping cart.

Note that the graph shown in Figure 3 is not a strongly connected graph—that is, every state cannot be reached by every other state. In this example, purchasing an item is a permanent action, it irrecoverably changes the state (there is no link from `purchased_item` to `item_in_cart`). Another interesting aspect is that one request, `GET /logout.php`, leads to three different states. This is because once the web application's state has changed, logging out, and then back in, does not change the state of the cart.

3.2 Inferring the State Machine

Inferring a web application's state machine requires the ability to detect when the state of the web application has changed. Therefore, we start with a description of the state-change detection algorithm, then explain the other components that are required to infer the state machine.

The key insight of our state-change algorithm is the following: We detect that the state of the web application has changed when we make an identical request and get a different response. This is the only externally visible effect of a state-change: Providing the same input causes a different output.

Using this insight, our state-change detection algorithm works, at a high level, as follows: (1) Crawl the web application sequentially, making requests based on a link in the previous response. (2) Assume that the state stays the same, because there is no evidence to the contrary. (3) If we make a request identical to a previous request and get a different response, then we assume that some request since the last identical request changed the state of the web application.

The intuition here is that a Mealy machine will, when given the same input in the same state, produce the same output. Therefore, if we send the same request and get a different output, the state must have changed. By detecting the web application's state changes only using inputs and outputs, we are agnostic with respect to both *what* constitutes the state information and *where* the state information is located. In this way, we are more generic than approaches that only consider the database to hold the state of the application, when in fact, the local file system or even memory could hold part of the web application's state.

The state-change detection algorithm allows us to infer when the web application's state has changed, yet four other techniques are necessary to infer a state machine: the clustering of similar pages, the identification of state-changing requests, the collapsing of similar states, and navigating.

Clustering similar pages. We want to group together pages that are similar, for two reasons: To handle infinite sections of web applications that are generated from the same code (e.g., the pages of a calendar) and to detect when a response has changed.

Before we can cluster pages, we model them using the links (anchors and forms) present on the page. The intuition here is that the links describe *how* the user can interact with the web application. Therefore, changes to what a user can do (new or missing links) indicate when the state of the web application has changed. Also, infinite sections of a web application will share the same link structure and will cluster together.

With our page model, we cluster pages together based on their link structure. Pages that are in different clusters are considered different. The details of this approach are described in Section 4.1.

Determining the state-changing request. The state-change detection algorithm only says that the state has changed, however we need to determine *which* request actually changed the state. When we detect a state change, we have a temporal list of requests with identical requests at the start and end. One of the requests in this list changed the state. We use a heuristic to determine which request changed the state. This heuristic favors newer requests over older requests, `POST` requests over `GET` requests, and requests that have previously changed

the state over those that have never changed the state. The details are described in Section 4.2.

Collapsing similar states. The state-change detection algorithm detects only when the state has changed, however, we need to understand if we returned to a previous state. This is necessary because if we detect a state change, we want to know if this is a state we have previously seen or a brand new state. We reduce this problem to a graph coloring problem, where the nodes are the states and an edge between two nodes means that the states cannot be the same. We add edges to this graph by using the requests and responses, along with rules to determine when two states cannot be the same. After the graph is colored, states that are the same color are collapsed into the same state. Details of this state-merging technique are provided in Section 4.3.

Navigating. We have two strategies for crawling the web application.

First, we always try to pick a link in the last response. The rational behind choosing a link in the last response is that we emulate a user browsing the web application. In this way, we are able to handle multi-step processes, such as previewing a comment before it is committed.

Second, for each state, we make requests that are the least likely to change the state of the web application. The intuition here is that we want to first see as much of a state as possible, without accidentally changing the state, in case the state change is permanent. Full details of how we crawl the web application are provided in Section 4.4

4 Technical Details

Inferring a web application's state machine requires concretely defining aspects such as page clustering or navigation. However, we wish to stress that this is one implementation of the state machine inference algorithm and it may not be optimal.

4.1 Clustering Similar Pages

Our reason for grouping similar pages together is twofold: Prevent infinite scanning of the website by grouping the "infinite" areas together and detect when the state has changed by comparing page responses in an efficient manner.

4.1.1 Page Model

The output of a web application is usually an HTML document (it can actually be any arbitrary content, but we only consider HTML content and HTTP redirects). An HTML page is composed of navigational information (anchors and forms) and user-readable content. For our state-change detection algorithm, we are not interested in changes to the content, but rather to changes in the navigation structure. We focus on navigation changes because the links on a page define how a user can interact with the application, thus, when the links change, the web application's state has changed.

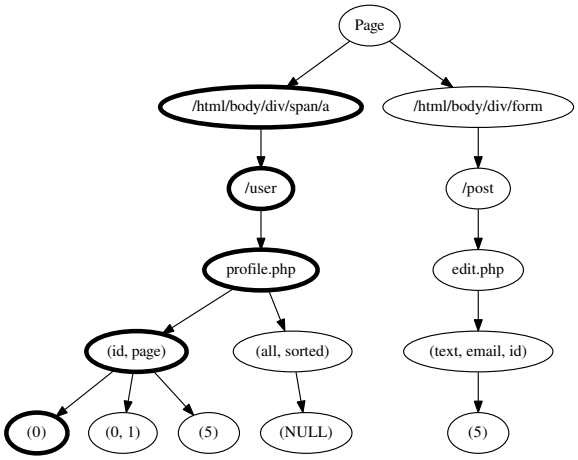

Figure 4: Representation of a page's link vectors stored in a prefix tree. There are five links present on this tree, as evidenced by the number of leaf nodes.

Therefore, we model a page by composing all the anchors and forms. First, every anchor and form is transformed into a vector constructed as follows:

$$\langle dompath,\ action,\ params,\ values \rangle$$

where:

- *dompath* is the DOM (*Document Object Model*) path of the HTML link (anchor or form);
- *action* is a list where each element is from the `href` (for anchors) or `action` (for forms) attribute split by '/';
- *params* is the (potentially empty) set of parameter names of the form or anchor;
- *values* is the set of values assigned to the parameters listed in *params*.

For instance, an anchor tag with the `href` attribute of `/user/profile.php?id=0&page` might have the following link vector:

⟨/html/body/div/span/a, /user, profile.php, (id, page), (0)⟩

All link vectors of a page are then stored in a prefix tree. This prefix tree is the model of the page. A prefix tree for a simple page with five links is shown in Figure 4. The link vector previously described is highlighted in bold in Figure 4.

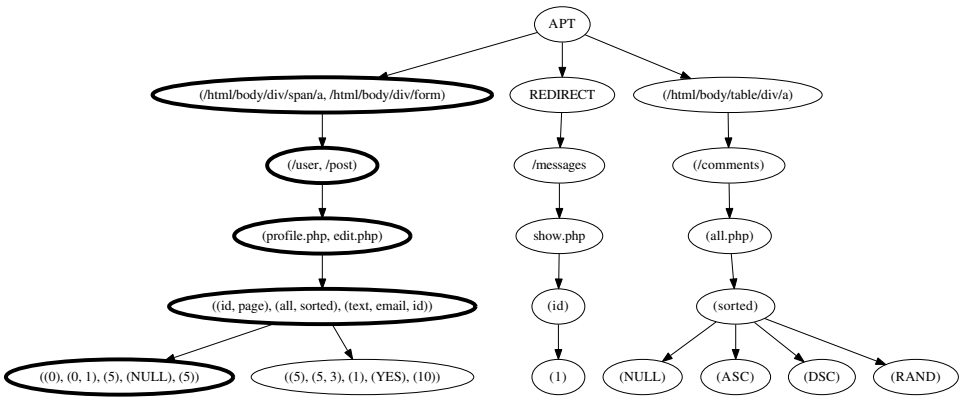

Figure 5: Abstract Page Tree. Every page's link vector is stored in this prefix tree. There are seven pages in this tree. The page link vector from Figure 4 is highlighted in bold.

HTTP redirects are handled as a special case, where the only element is a special *redirect* element having the target URL as the value of the *location* attribute.

4.1.2 Page Clustering

To cluster pages, we use a simple but efficient algorithm. As described in the previous section, the model of a page is a prefix tree representing all the links contained in the page.

These prefix trees are translated into vectors, where every element of this vector is the set of all nodes of a given level of the prefix tree, starting from the root. At this point, all pages are represented by a *page link vector*. For example, Figure 4 has the following page link vector:

$$\langle (\text{/html/body/div/span/a, /html/body/div/form}),\ (\text{/user, /post}),\ (\text{profile.php, edit.php}),\ ((\text{id, page}), (\text{all, sorted}), (\text{text, email, id})),\ ((0), (0, 1), (5), (NULL), (5))\rangle$$

The page link vectors for all pages are then stored in another prefix tree, called the *Abstract Page Tree* (APT). In this way, pages are mapped to a leaf of the tree. Pages which are mapped to the same leaf have identical page link vectors and are considered to be the same page. Figure 5 shows an APT with seven pages. The page from Figure 4 is bold in Figure 5.

However, we want to cluster together pages whose page link vectors do not match exactly, but are similar (e.g., shopping cart pages with a different number of elements in the cart). A measure of the similarity between two pages is how many elements from the beginning of their link vectors are the same between the two pages. From the APT perspective, the higher the number of ancestors two pages (leaves) share, the closer they are.

Therefore, we create clusters of similar pages by selecting a node in the APT and merging into one cluster, called an *Abstract Page*, all the leaves in the corresponding subtree. The criteria for deciding whether to cluster a subtree of depth n from the root is the following:

- The number of leaves is greater than the median number of leaves of all its siblings (including itself); in this way, we cluster only subtrees which have a larger-than-usual number of leaves.
- There are at least $f(n)$ leaves in the subtree, where $f(n)$ is inversely related to n. The intuition is that the fewer ancestors a subtree has in common (the higher on the prefix tree it is), the more pages it must have to cluster them together. We have found that the function $f(n) = 8(1 + \frac{1}{n+1})$ works well by experimental analysis on a large corpus of web pages.
- The pages share the same $dompath$ and the first element of the $action$ list of the page link vector; in this way, all the pages that are clustered together share the same link structure with potentially different parameters and values.

4.2 Determine the State-Changing Request

When a state change is detected, we must determine which request actually changed the web application's state. Recall that we detect a state change when we make a request that is identical to a previous request, yet has different output. At this point, we have a list of all the requests made between the latest request R and the request R' closest in time to R such that R is identical to R'. We use a heuristic to determine which request in this list changed the web application's state, choosing the request i between R' and R which maximizes the function:

$$\text{score}(n_{i,transition}, n_{i,seen}, distance_i)$$

where:

- $n_{i,transition}$ is the number of times the request caused a state transition;
- $n_{i,seen}$ is the number of times the request has been made;
- $distance_i$ is how many requests have been made between request R and request i.

The function *score* is defined as:

$$score(n_{i,transition}, n_{i,seen}, distance_i) = 1 - (1 - \frac{n_{i,transition}+1}{n_{i,seen}+1})^2 + \frac{\text{BOOST}_i}{distance_i+1}$$

BOOST_i is .2 for `POST` requests and .1 for `GET` requests.

We construct the *score* function to capture two properties of web applications:

1. A POST request is more likely to change the state than a GET request. This is suggested by the HTTP specification, and *score* captures this intuition with BOOST_i.
2. Resistant to errors. Because we cannot prove that the selected request changed the state, we need to be resistant to errors. That is why *score* contains the ratio of $n_{i,transition}$ to $n_{i,seen}$. In this way, if we accidentally choose the wrong state-changing request once, but then, later, make that request many times without changing the state, we are less likely to choose it as a state-changing request.

4.3 Collapsing Similar States

Running the state detection algorithm on a series of requests and responses will tell us when the state has changed. At this point, we consider each state unique. This initial state assignment, though, is not optimal, because even if we encounter a state that we have seen in the past, we are marking it as new. For example, in the case of a sequence of login and logout actions, we are actually flipping between two states, instead of entering a new state at every login/logout. Therefore, we need to minimize the number of different states and collapse states that are actually the same.

The problem of state allocation can be seen as a graph-coloring problem on a non-planar graph [27]. Let each state be a node in the graph G. Let two nodes a and b be connected by an edge (meaning that the states cannot be the same) if either of the following conditions holds:

1. If a request R was made when the web application was in states a and b and results in pages in different clusters. The intuition is that two states cannot be the same if we make an identical request in each state yet receive a different response.
2. The two states a and b have no pages in common. The idea is to err on the conservative side, thus we require that two states share a page before collapsing the states into one.

After adding the edges to the graph by following the previous rules, G is colored. States assigned the same color are considered the same state.

To color the nodes of G, we employ a custom greedy algorithm. Every node has a unique identifier, which is the incremental number of the state as we see it in the request-response list. The nodes are ordered by identifier, and we assign the color to each node in a sequential way, using the highest color available (i.e., not used by its neighbors), or a new color if none is available.

This way of coloring the nodes works very well for state allocation because it takes into account the temporal locality of states: In particular, we attempt to assign the highest available color because it is more likely for a state to be the same as a recently seen state rather than one seen at the beginning of crawling.

There is one final rule that we need to add after the graph is colored. This rules captures an observation about transitioning between states: If a request, R, transitions the web application from state a_1 to state b, yet, later when the web application is in state a_2, R transitions the web application to state c, then a_1 and a_2 cannot be the same state. Therefore, we add an edge from a_1 to a_2 and redo the graph coloring.

We continue enforcing this rule until no additional edges are added. The algorithm is guaranteed to converge because only new edges are added at every step, and no edges are ever removed.

At the end of the iteration, the graph coloring output will determine the final state allocation—all nodes with the same color represent the same state (even if seen at different stages during the web application crawling process).

4.4 Navigating

Typical black-box web vulnerability scanners make concurrent HTTP requests to a web application to increase performance. However, as we have shown, an HTTP request can influence the web application's state, and, in this case, all other requests would occur in the new state. Also, some actions require a multi-step, sequential process, such as adding items to a shopping cart before purchasing them. Finally, a user of the web application does not browse a web application in this parallel fashion, thus, developers assume that the users will browse sequentially.

```
def fuzz_state_changing( fuzz_request ):
  make_request( fuzz_request )
  if state_has_changed():
    if state_is_reversible():
      make_requests_to_revert_state()
      if not back_in_previous_state():
        reset_and_put_in_previous_state()
    else:
      reset_and_put_in_previous_state()
```

Listing 1: Psuedocode for fuzzing state-changing request.

Our scanner navigates a web application by mimicking a user browsing the web application sequentially. Browsing sequentially not only allows us to follow the developer's intended path through the web application, but it enables us to detect *which* requests changed the web application's state.

Thus, a state-aware crawler must navigate the application sequentially. No concurrent requests are made, and only anchors and forms present in the last visited page are used to determine the next request. In the case of a page with no outgoing links we go back to the initial page.

Whenever the latest page does not contain unvisited links, the crawler will choose a path from the current page towards another page already seen that contains links that have not yet been visited. If there is no path from the current page to anywhere, we go back to the initial page. The criteria for choosing this path is based on the following intuitions:

- We want to explore as much of the current state as possible before changing the state, therefore we select links that are less likely to cause a state transition.

- When going from the current page to a page with an unvisited link, we will repeat requests. Therefore, we should choose a path that contains links that we have visited infrequently. This give us more information about the current state.

The exact algorithm we employ is Dijkstra Shortest Path Algorithm [14] with custom edge length. This edge length increases with the number of times we have previously visited that link. Finally, the edge length increases with how likely the link is to cause a state change.

5 State-Aware Fuzzing

After we crawl the web application, our system has inferred, as much as possible, the web application's state machine. We use the state machine information, along with the list of request–responses made by the crawler, to drive a state-aware fuzzing of the web application, looking for security vulnerabilities.

To fuzz the application in a state-aware manner, we need the ability to reset the web application to the initial state (the state when we started crawling). We do not use this ability when crawling, only when fuzzing. It is necessary to reset the application when we are fuzzing an irreversible state-changing request. Using the reset functionality, we are able to recover from these irreversible state changes.

Adding the ability to reset the web application does not break the black-box model of the web application. Resetting requires no knowledge of the web application, and can be easily performed by running the web application in a virtual machine.

Our state-aware fuzzing starts by resetting the web application to the initial state. Then we go through the requests that the crawler made, starting with the initial request. If the request does not change the state, then we fuzz the request as a typical black-box scanner. However, if the request is state-changing, we follow the algorithm shown in Listing 1. The algorithm is simple: We make the request, and if the state has changed, traverse the inferred state machine to find a series of requests to transition the web application to the previous state. If this does not exist, or does not work, then we reset the web application to the initial state, and make all the previous requests that the crawler made. This ensures that the web application is in the proper state before continuing to fuzz.

Our state-aware fuzzing approach can use *any* fuzzing component. In our implementation, we used the fuzzing plugins of an open-source scanner, w3af [37]. The fuzzing plugins take an HTTP request and generate variations on that request looking for different vulnerabilities. Our state-aware fuzzing makes those requests while checking that the state does not unintentionally change.

6 Evaluation

As shown in previous research [16], fairly evaluating black-box web vulnerability scanners is difficult. The most important, at least to end users, metric for comparing black-box web vulnerability scanners is true vulnerabilities discovered. Comparing two scanners that discover different vulnerabilities is nearly impossible.

There are two other metrics that we use to evaluate black-box web vulnerability scanners:

- False Positives. The number of spurious vulnerabilities that a black-box web vulnerability scanner reports. This measures the accuracy of the scanner. False positives are a serious problem for the end user of the scanner—if the false positives are

Application	Description	Version	Lines of Code
Gallery	Photo hosting.	3.0.2	26,622
PhpBB v2	Discussion forum.	2.0.4	16,034
PhpBB v3	Discussion forum.	3.0.10	110,186
SCARF	Stanford conference and research forum.	2007-02-27	798
Vanilla Forums	Discussion forum.	2.0.17.10	43,880
WackoPicko v2	Intentionally vulnerable web application.	2.0	900
WordPress v2	Blogging platform.	2.0	17,995
WordPress v3	Blogging platform.	3.2.1	71,698

Table 1: Applications that we ran the crawlers against to measure vulnerabilities discovered and code coverage.

high, the user must manually inspect each vulnerability reported to determine the validity. This requires a security-conscious user to evaluate the reports. Moreover, false positives erode the user's trust in the tool and make the user less likely to use it in the future.

- Code Coverage. The percentage of the web application's code that the black-box web vulnerability scanner executes while it crawls and fuzzes the application. This measures how effective the scanner is in exercising the functionality of the web application. Moreover, code coverage is an excellent metric for another reason: A black-box web vulnerability scanner, by nature, cannot find a vulnerability along a code path that it does not execute. Therefore, greater code coverage means that a scanner has the potential to discover more vulnerabilities. Note that this is orthogonal to fuzzing capability: A fuzzer—no matter how effective—will never be able to discover a vulnerability on a code path that it does not execute.

We use both the metrics previously described in our evaluation. However, our main focus is on code coverage. This is because a scanner with greater code coverage will be able to discover more vulnerabilities in the web application.

However, code coverage is not a perfect metric. Evaluating raw code coverage percentage numbers can be misleading. Ten percent code coverage of an application could be horrible or excellent depending on how much functionality the application exposes. Some code may be intended only for installation, may be only for administrators, or is simply dead code and cannot be executed. Therefore, comparing code coverage normalized to a baseline is more informative, and we use this in our evaluation.

6.1 Experiments

We evaluated our approach by running our state-aware-scanner along with three other vulnerability scanners against eight web applications. These web applications range in size, complexity, and functionality. In the rest of this section, we describe the web applications, the black-box web vulnerability scanners, and the methodology we used to validate our approach.

Scanner	Description	Language	Version
wget	GNU command-line website downloader.	C	1.12
w3af	Web Application Attack and Audit Framework.	Python	1.0-stable
skipfish	Open-source, high-performance vulnerability scanner.	C	2.03b
state-aware-scanner	Our state-aware vulnerability scanner.	Python	1.0

Table 2: Black-box web vulnerability scanners that we compared.

6.1.1 Web Applications

Table 1 provides an overview of the web applications used with a short description, a version number, and lines of executable PHP code for each application. Because our approach assumes that the web application's state changes only via requests from the user, we made slight code modifications to three web applications to reduce the influence of external, non-user driven, forces, such as time. Please refer to Appendix A for a detailed description of each application and what was changed.

6.1.2 Black-Box Web Vulnerability Scanners

This section describes the black-box web vulnerability scanners that were compared against our approach, along with the configuration or settings that were used. Table 2 contains a short description of each scanner, the scanner's programming language, and the version number. Appendix B shows the exact configuration that was used for each scanner.

wget is a free and open-source application that is used to download files from a web application. While not a vulnerability scanner, wget is a crawler that will make all possible `GET` requests it can find. Thus, it provides an excellent baseline because vulnerability scanners make `POST` requests as well as `GET` requests and should discover more of the application than wget.

wget is launched with the following options: recursive, download everything, and ignore robots.txt.

w3af is an open-source black-box web vulnerability scanner which has numerous fuzzing modules. We enabled the blindSqli, eval, localFileInclude, osCommanding, remoteFileInclude, sqli, and xss fuzzing plugins.

skipfish is an open-source black-box web vulnerability scanner whose focus is on high speed and high performance. Skipfish epitomizes the "shotgun" approach, and boasts about making more than 2,000 requests per second to a web application on a LAN. Skipfish also attempts to guess, via a dictionary or brute-force, directory names. We disabled this behavior to be fair to the other scanners, because we do not want to test the ability to guess a hidden directory, but how a scanner crawls a web application.

state-aware-scanner is our state-aware black-box vulnerability scanner. We use HtmlUnit [19] to issue the HTTP requests and render the HTML responses. After crawling and building the state-graph, we utilize the fuzzing plugins from w3af to generate fuzzing requests. Thus, any improvement in code coverage of our crawler over w3af is due to our state-aware crawling, since the fuzzing components are identical.

6.1.3 Methodology

We ran each black-box web vulnerability scanner against a distinct, yet identical, copy of each web application. We ran all tests on our local cloud [34].

Gallery, WordPress v2, and WordPress v3 do not require an account to interact with the website, thus each scanner is simply told to scan the test application.

For the remaining applications (PhpBB v2, PhpBB v3, SCARF, Vanilla Forums, and WackoPicko v2), it is difficult to fairly determine how much information to give the scanners. Our approach only requires a username/password for the application, and by its nature will discover the requests that log the user out, and recover from them. However, other scanners do not have this capability.

Thus, it is reasonable to test all scanners with the same level of information that we give our scanner. However, the other scanners lack the ability to provide a username and password. Therefore, we did the next best thing: For those applications that require a user account, we log into the application and save the cookie file. We then instruct the scanner to use this cookie file while scanning the web application.

While we could do more for the scanners, like preventing them from issuing the logout request for each application, we believe that our approach strikes a fair compromise and allows each scanner to decide how to crawl the site. Preventing the scanners from logging out of the application also limits the amount of the application they will see, as they will never see the web application from a guest's perspective.

6.2 Results

Table 3 shows the results of each of the black-box web vulnerability scanners against each web application. The column "% over Baseline" displays the percentage of code coverage improvement of the scanner against the wget baseline, while the column "Vulnerabilities" shows total number of reported vulnerabilities, true positives, unique true positives among the scanners, and false positives.

The prototype implementation of our state-aware-scanner had the best code coverage for every application. This verifies the validity of our algorithm: Understanding state is necessary to better exercise a web application.

Figure 6 visually displays the code coverage percent improvement over wget. The most important thing to take from these results is the improvement state-aware-scanner has over w3af. Because we use the fuzzing component of w3af, the only difference is in our state-aware crawling. The results show that this gives state-aware-scanner an increase in code coverage from as little as half a percent to 140.71 percent.

Our crawler discovered three unique vulnerabilities, one each in PhpBB v2, SCARF, and WackoPicko v2. The SCARF vulnerability is simply a XSS injection on the comment form. w3af logged itself out before fuzzing the comment page. skipfish filed the vulnerable page under "Response varies randomly, skipping checks." However, the content of this page does not vary randomly, it varies because skipfish is altering it. This *random* categorization also prevents skipfish from detecting the simple XSS vulnerability on WackoPicko v2's guestbook. This result shows that a scanner needs to understand the web application's internal state to properly decide *why* a page's content is changing.

Skipfish was able to discover 15 vulnerabilities in Vanilla Forums. This is impressive, however, 14 stem from a XSS injection via the referer header on an error page. Thus, even though these 14 vulnerabilities are on different pages, it is the same root cause.

Surprisingly, our scanner produced less false positives than w3af. All of w3af's false positives were due to faulty timing detection of SQL injection and OS com-

Scanner	Application	% over Baseline	Vulnerabilities			
			Reported	True	Unique	False
state-aware-scanner	Gallery	16.20%	0	0	0	0
w3af	Gallery	15.77%	3	0	0	3
skipfish	Gallery	10.96%	0	0	0	0
wget	Gallery	0%				
state-aware-scanner	PhpBB v2	38.34%	4	3	1	1
skipfish	PhpBB v2	5.10%	3	2	0	1
w3af	PhpBB v2	1.04%	5	1	0	4
wget	PhpBB v2	0%				
state-aware-scanner	PhpBB v3	115.45%	0	0	0	0
skipfish	PhpBB v3	60.21%	2	0	0	2
w3af	PhpBB v3	16.16%	0	0	0	0
wget	PhpBB v3	0%				
state-aware-scanner	SCARF	67.03%	1	1	1	0
skipfish	SCARF	55.66%	0	0	0	0
w3af	SCARF	21.55%	0	0	0	0
wget	SCARF	0%				
state-aware-scanner	Vanilla Forums	30.89%	0	0	0	0
w3af	Vanilla Forums	1.06%	0	0	0	0
wget	Vanilla Forums	0%				
skipfish	Vanilla Forums	-2.32%	17	15	2	2
state-aware-scanner	WackoPicko v2	241.86%	5	5	1	0
skipfish	WackoPicko v2	194.77%	4	3	1	1
w3af	WackoPicko v2	101.15%	5	5	1	0
wget	WackoPicko v2	0%				
state-aware-scanner	WordPress v2	14.49%	0	0	0	0
w3af	WordPress v2	12.49%	0	0	0	0
wget	WordPress v2	0%				
skipfish	WordPress v2	-18.34%	1	0	0	1
state-aware-scanner	WordPress v3	9.84%	0	0	0	0
w3af	WordPress v3	9.23%	3	0	0	3
skipfish	WordPress v3	3.89%	1	0	0	1
wget	WordPress v3	0%				

Table 3: Results of each of the black-box web vulnerability scanners against each application. The table is sorted by the percent increase in code coverage over the baseline scanner, wget.

manding. We believe that using HtmlUnit prevented our scanner from detecting these spurious vulnerabilities, even though we use the same fuzzing component as w3af.

Finally, our approach inferred the state machines of the evaluated applications. These state machines are very complex in the large applications. This complexity is because modern, large, application have many actions which modify the state. For instance, in WackoPicko v2, a user can log in, add items to their cart, comment on pictures, delete items from their cart, log out of the application, register as a new user, comment as this new user, upload a picture, and purchase items. All of these actions interact to form a complex state machine. The state machine our scanner inferred captures this complex series of state changes. The inferred WackoPicko v2 state machine is presented in Figure 7.

7 Limitations

Although dynamic page generation via JavaScript is supported by our crawler as allowed by the HtmlUnit framework [19], proper AJAX support is not implemented. This means that our prototype executes JavaScript when the page loads, but does not execute AJAX calls when clicking on links.

Nevertheless, our approach could be extended to handle AJAX requests. In fact, any interaction with the web application always contains a request and response, however the content of the response is no longer an HTML page. Thus, we could extend our notion of a "page" to typical response content of AJAX calls, such as JSON or XML. Another way to handle AJAX would be to follow a Crawljax [33] approach and covert the dynamic AJAX calls into static pages.

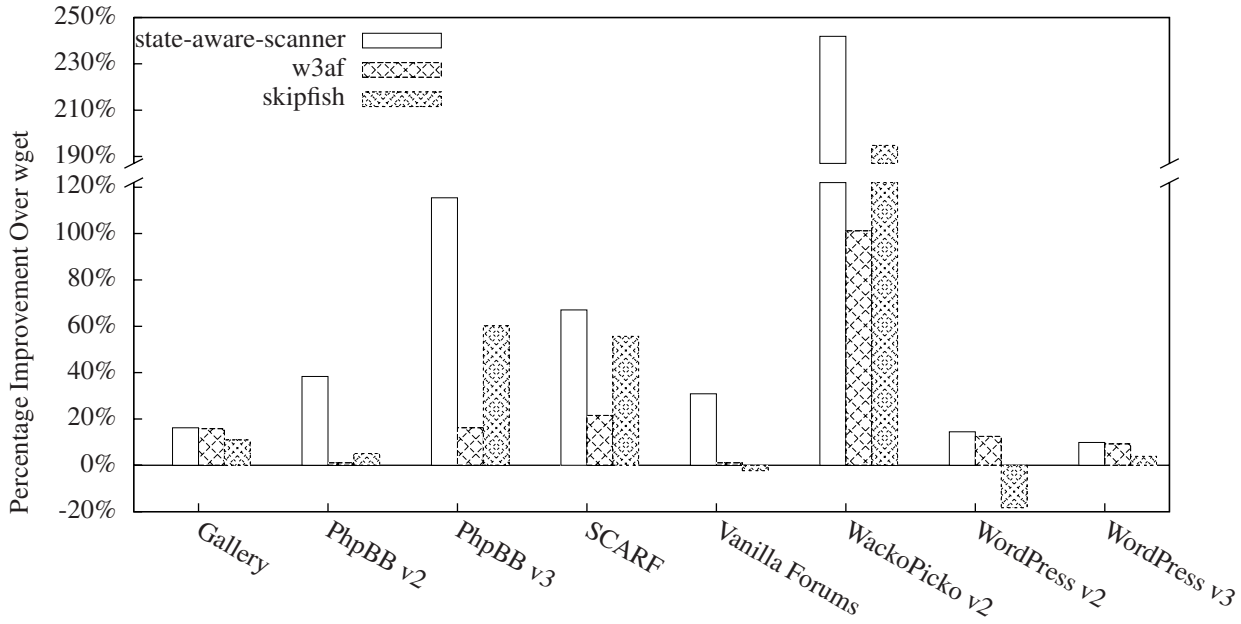

Figure 6: Visual representation of the percentage increase of code coverage over the baseline scanner, wget. Important to note is the gain our scanner, state-aware-scanner, has over w3af, because the only difference is our state-aware crawling. The *y*-axis range is broken to reduce the distortion of the WackoPicko v2 results.

Another limitation of our approach is that our scanner cannot be used against a web application being accessed by other users (i.e., a public web application), because the other users may influence the state of the application (e.g., add a comment on a guestbook) and confuse our state change detection algorithm.

8 Related Work

Automatic or semi-automatic web application vulnerability scanning has been a hot topic in research for many years because of its relevance and its complexity.

Huang et al. developed a tool (WAVES) for assessing web application security with which we share many points [24]. Similarly to us, they have a scanner for finding the entry points in the web application by mimicking the behavior of a web browser. They employ a learning mechanism to sensibly fill web form fields and allow deep crawling of pages behind forms. Attempts to discover vulnerabilities are carried out by submitting the same form multiple times with valid, invalid, and faulty inputs, and comparing the result pages. Differently from WAVES, we are using the knowledge gathered by the first-phase scanner to help the fuzzer detect the effect of a given input. Moreover, our first-phase scanner aims not only at finding relevant entry-points, but rather at building a complete state-aware navigational map of the web application.

A number of tools have been developed to try to automatically discover vulnerabilities in web applications, produced as academic prototypes [4,17,22,25,28,29,31], as open-source projects [8,9,37], or as commercial products [1,23,26,35].

Multiple projects [6,16,42,43] tackled the task of evaluating the effectiveness of popular black-box scanners (in some cases also called *point-and-shoot* scanners). The common theme in their results is a relevant discrepancy in vulnerabilities found across scanners, along with low accuracy. Authors of these evaluations acknowledge the difficulties and challenges of the task [21, 43]. In particular, we highlighted how more deep crawling and reverse engineering capabilities of web applications are needed in black-box scanners, and we also developed the *WackoPicko* web application which contains known vulnerabilities [16]. Similarly, Bau et al. investigated the presence of room for research in this area, and found improvement is needed, in particular for detecting second-order XSS and SQL injection attacks [6].

Reverse engineering of web applications has not been widely explored in the literature, to our knowledge. Some approaches [13] perform static analysis on the code to create UML diagrams of the application.

Static analysis, in fact, is the technique mostly employed for automatic vulnerability detection, often combined with dynamic analysis.

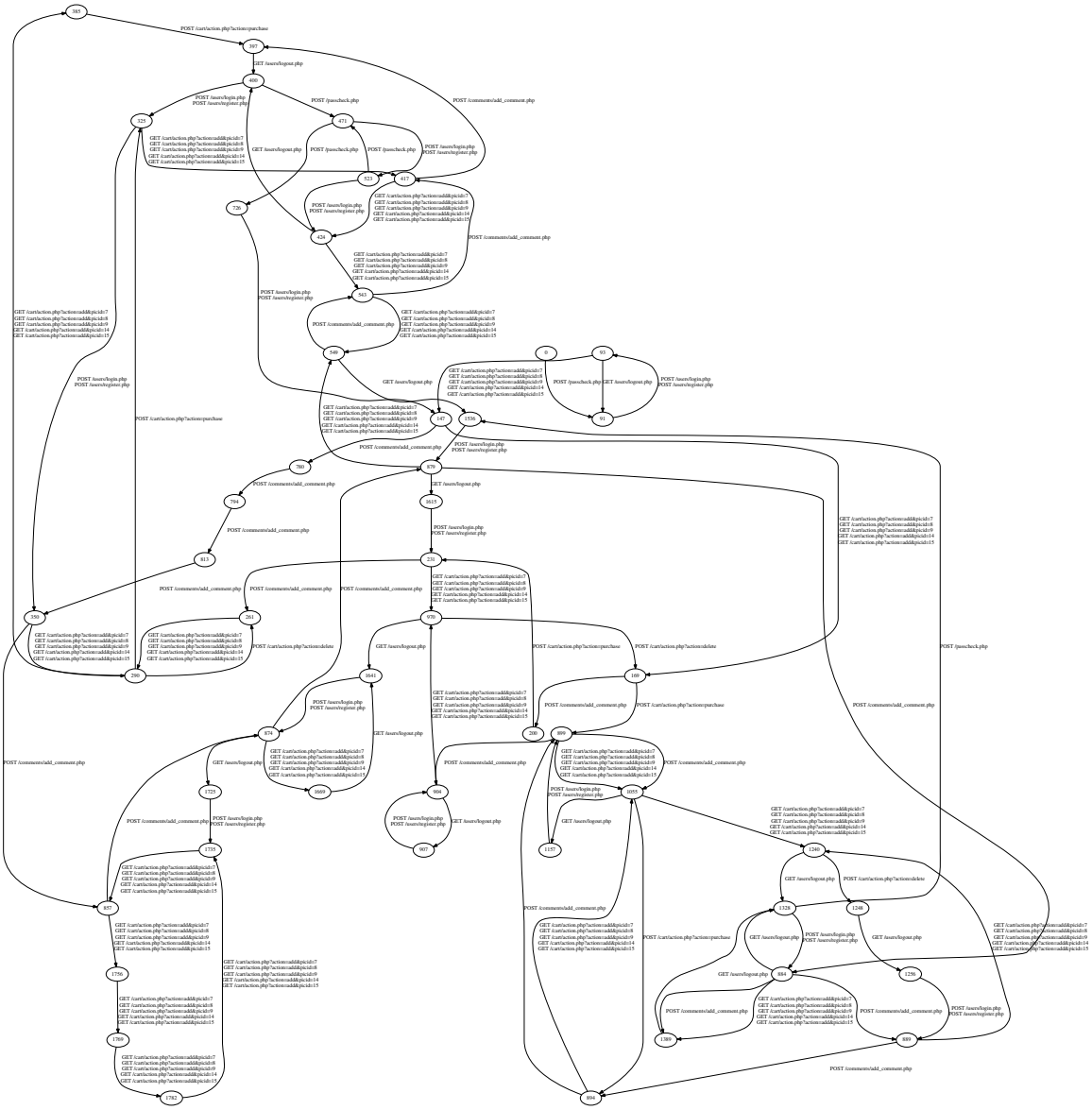

Figure 7: State machine that state-aware-scanner inferred for WackoPicko v2.

Halfond et al. developed a traditional black-box vulnerability scanner, but improved its result by leveraging a static analysis technique to better identify input vectors [22].

Pixy [28] employed static analysis with taint propagation in order to detect SQL injection, XSS and shell command injection, while *Saner* [4] used sound static analysis to detect failures in sanitization routines. Saner also takes advantage of a second phase of dynamic analysis to reduce false positives. Similarly, *WebSSARI* [25] also employed static analysis for detecting injection vulnerabilities, but, in addition, it proposed a technique for runtime instrumentation of the web application through the insertion of proper sanitization routines.

Felmetsger et al. investigated an approach for detecting a different type of vulnerability (some categories of logic flaws) by combining execution traces and symbolic model checking [17]. Similar approaches are also used for generic bug finding (in fact, vulnerabilities could be considered a subset of the general bug category). Csallner et al. employ dynamic traces for bug finding and for dynamic verification of the alerts generated by the static analysis phase [12]. Artzi et al., on the other hand, use symbolic execution and model checking for finding general bugs in web applications [3].

On a completely separate track, efforts to improve web application security push the developers toward writing secure code in the first place. Security experts are ty-

ing to achieve this goal by either educating the developers [40] or designing frameworks which prohibit the use of bad programming practices and enforce some security constraints in the code. Robertson and Vigna developed a strongly-typed framework which statically enforces separation between structure and content of a web page, preventing XSS and SQL injection [38]. Also Chong et al. designed their language for developers to build web applications with strong confidentiality and integrity guarantees, by means of compile-time and runtime checks [10].

Alternatively, consequences of vulnerabilities in web applications can be mitigated by trying to prevent the attacks before they reach some potentially vulnerable code, like, for example, in the already mentioned *WebSSARI* [25] work. A different approach for blocking attacks is followed by Scott and Sharp, who developed a language for specifying a security policy for the web application; a gateway will then enforce these policies [39].

Another interesting research track deals with the problem of how to explore web pages behind forms, also called the *hidden web* [36]. McAllister et al. monitor user interactions with a web application to collect sensible values for HTML form submission and generate test cases that can be replayed to increase code coverage [32]. Although not targeted to security goals, the work of Raghavan and Garcia-Molina is relevant for our project for their contribution in classification of different types of dynamic content and for their novel approach for automatically filling forms by deducing the domain of form fields [36]. Raghavan and Garcia-Molina carried out further research in this direction, by reconstructing complex and hierarchical query interfaces exposed by web applications.

Moreover, Amalfitano et al. started tackling the problem of reverse engineering the state machine of client-side AJAX code, which will help in finding the web application server-side entry points and in better understating complex and hierarchical query interfaces [2].

Finally, we need to mention the work by Berg et al. in reversing state machines into a *Symbolic Mealy Machine* (SMM) model [7]. Their approach for reversing machines cannot be directly applied to our case because of the infeasibility of fully exploring all pages for all the states, even for a small subset of the possible states. Nevertheless, the model they propose for a SMM fits our needs.

9 Conclusion

We have described a novel approach to inferring, as much as possible, a web application's internal state machine. We leveraged the state machine to drive the state-aware fuzzing of web applications. Using this approach, our crawler is able to crawl—and thus fuzz—more of the web application than a classical state-agnostic crawler. We believe our approach to detecting state change by differences in output for an identical response is valid and should be adopted by all black-box tools that wish to understand the web application's internal state machine.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under Grant N000141210165, the National Science Foundation (NSF) under grant CNS-1116967, and by Secure Business Austria.

References

[1] ACUNETIX. Acunetix Web Vulnerbility Scanner. `http://www.acunetix.com/`.

[2] AMALFITANO, D., FASOLINO, A., AND TRAMONTANA, P. Reverse Engineering Finite State Machines from Rich Internet Applications. In *2008 15th Working Conference on Reverse Engineering* (2008), IEEE, pp. 69–73.

[3] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering* (2010).

[4] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy* (2008), IEEE, pp. 387–401.

[5] BALZAROTTI, D., COVA, M., FELMETSGER, V., AND VIGNA, G. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[6] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 332–345.

[7] BERG, T., JONSSON, B., AND RAFFELT, H. Regular Inference for State Machines using Domains with Equality Tests. In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering* (2008), Springer-Verlag, pp. 317–331.

[8] BYRNE, D. Grendel-Scan. `http://www.grendel-scan.com/`.

[9] CHINOTEC TECHNOLOGIES. Paros. `http://www.parosproxy.org/`.

[10] CHONG, S., VIKRAM, K., AND MYERS, A. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (2007), USENIX Association, p. 1.

[11] COVA, M., BALZAROTTI, D., FELMETSGER, V., AND VIGNA, G. Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2007)* (2007), pp. 63–86.

[12] CSALLNER, C., SMARAGDAKIS, Y., AND XIE, T. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM) 17*, 2 (2008), 1–37.

[13] DI LUCCA, G., FASOLINO, A., PACE, F., TRAMONTANA, P., AND DE CARLINI, U. WARE: a tool for the reverse engineering of Web applications. In *Sixth European Conference on Software Maintenance and Reengineering, 2002. Proceedings* (2002), pp. 241–250.

[14] DIJKSTRA, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1* (1959), 269–271.

[15] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)* (Chicago, IL, October 2011).

[16] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2010)* (2010), Springer, pp. 111–131.

[17] FELMETSGER, V., CAVEDON, L., KRUEGEL, C., AND VIGNA, G. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium* (Washington, DC, August 2010).

[18] FOSSI, M. Symantec Global Internet Security Threat Report. Tech. rep., Symantec, April 2009. Volume XIV.

[19] GARGOYLE SOFTWARE INC. HtmlUnit. `http://htmlunit.sourceforge.net/`.

[20] GARRETT, J. J. Ajax: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, Feb. 2005.

[21] GROSSMAN, J. Challenges of Automated Web Application Scanning. Blackhat Windows 2004, 2004.

[22] HALFOND, W., CHOUDHARY, S., AND ORSO, A. Penetration testing with improved input vector identification. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on* (2009), IEEE, pp. 346–355.

[23] HP. WebInspect. `https://download.hpsmartupdate.com/webinspect/`.

[24] HUANG, Y.-W., HUANG, S.-K., LIN, T.-P., AND TSAI, C.-H. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), WWW '03, ACM, pp. 148–159.

[25] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 40–52.

[26] IBM. AppScan. `http://www-01.ibm.com/software/awdtools/appscan/`.

[27] JENSEN, T. R., AND TOFT, B. *Graph Coloring Problems*. Wiley-Interscience Series on Discrete Mathematics and Optimization. Wiley, 1994.

[28] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security 18*, 5 (2010), 861–907.

[29] KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIC, N. Secubat: a Web Vulnerability Scanner. In *Proceedings of the 15th international conference on World Wide Web* (2006), ACM, pp. 247–256.

[30] LI, X., AND XUE, Y. BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2011)* (Orlando, FL, December 2011).

[31] LI, X., YAN, W., AND XUE, Y. SENTINEL: Securing Database from Logic Flaws in Web Applications. In *CODASPY* (2012), pp. 25–36.

[32] MCALLISTER, S., KIRDA, E., AND KRUEGEL, C. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Recent Advances in Intrusion Detection* (2008), Springer, pp. 191–210.

[33] MESBAH, A., BOZDAG, E., AND VAN DEURSEN, A. Crawling AJAX by Inferring User Interface State Changes. In *Web Engineering, 2008. ICWE '08. Eighth International Conference on* (july 2008), pp. 122 –134.

[34] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The Eucalyptus Open-Source Cloud-Computing System. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on* (may 2009), pp. 124 –131.

[35] PORTSWIGGER. Burp Proxy. `http://www.portswigger.net/burp/`.

[36] RAGHAVAN, S., AND GARCIA-MOLINA, H. Crawling the hidden web. In *Proceedings of the International Conference on Very Large Data Bases* (2001), Citeseer, pp. 129–138.

[37] RIANCHO, A. w3af – Web Application Attack and Audit Framework. `http://w3af.sourceforge.net/`.

[38] ROBERTSON, W., AND VIGNA, G. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium* (Montreal, Quebec CA, September 2009).

[39] SCOTT, D., AND SHARP, R. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web* (2002), ACM, pp. 396–407.

[40] SPI DYNAMICS. Complete Web Application Security: Phase 1 – Building Web Application Security into Your Development Process. SPI Dynamics Whitepaper, 2002.

[41] STEVE, C., AND MARTIN, R. Vulnerability Type Distributions in CVE. *Mitre report, May* (2007).

[42] SUTO, L. Analyzing the Accuracy and Time Costs of Web Application Security Scanners, 2010.

[43] VIEIRA, M., ANTUNES, N., AND MADEIRA, H. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on* (2009), IEEE, pp. 566–571.

A Web Applications

This section describes the web applications along with the functionality against which we ran the black-box web vulnerability scanner.

Gallery is an open-source photo hosting application. The administrator can upload photos and organize them into albums. Guests can then view and comment on the uploaded photos. Gallery has AJAX functionality but gracefully degrades (is fully functional) without JavaScript. No modifications were made to the application.

PhpBB v2 is an open-source forum software. It allows registered users to perform many actions such as create new threads, comment on threads, and message other users. Version 2 is notorious for the amount of security vulnerabilities it contains [6], and we included it for this reason. We modified it to remove the "recently online" section on pages, because this section is based on time.

PhpBB v3 is the latest version of the popular open-source forum software. It is a complete rewrite from Version 2, but retains much of the same functionality. Similar to PhpBB v2, we removed the "recently online" section, because it is time-based.

SCARF, the Stanford Conference And Research Forum, is an open-source conference management system. The administrator can upload papers, and registered users can comment on the uploaded papers. We included this application because it was used by previous research [5, 11, 30, 31]. No modifications were made to this application.

Vanilla Forums is an open-source forum software similar in functionality to PhpBB. Registered users can create new threads, comment on threads, bookmark interesting threads, and send a message to another user. Vanilla Forums is unique in our test set in that it uses the path to pass parameters in a URL, whereas all other applications pass parameters using the query part of the URL. For instance, a specific user's profile is `GET /profile/scanner1`, while a discussion thread is located at `GET /discussion/1/how-to-scan`. Vanilla Forums also makes extensive use of AJAX, and does not gracefully degrade without JavaScript. For instance, with JavaScript disabled, posting a comment returns a JSON object that contains the success or failure of the comment posting, instead of an HTML response. We modified Vanilla Forums by setting an XSRF token that it used to a constant value.

WackoPicko v2 is an open-source intentionally vulnerable web application which was originally created to evaluate many black-box web vulnerability scanners [16]. A registered user can upload pictures, comment on other user's pictures, and purchase another user's picture. Version 2 contains minor tweaks from the original paper, but no additional functionality.

WordPress v2 is an open-source blogging platform. An administrator can create blog posts, where guests can leave comments. No changes were made to this application.

WordPress v3 is an up-to-date version of the open-source blogging platform. Just like the previous version, administrators can create blog posts, while a guest can comment on blog posts. No changes were made to this application.

B Scanner Configuration

The following describes the exact settings that were used to run each of the evaluated scanners.

- wget is run in the following way:
```
wget -rp -w 0 --waitretry=0 -nd
  --delete-after --execute robots=off
```
- w3af settings:
```
misc-settings
set maxThreads 0
back
plugins
discovery webSpider
audit blindSqli, eval,
  localFileInclude, osCommanding,
  remoteFileInclude, sqli, xss
```
- skipfish is run in the following way:
```
skipfish -u -LV -W /dev/null -m 10
```