Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities

Erik Trickel⋆, Fabio Pagani†, Chang Zhu⋆, Lukas Dresel†,
Giovanni Vigna†, Christopher Kruegel†, Ruoyu Wang⋆, Tiffany Bao⋆, Yan Shoshitaishvili⋆, Adam Doupé⋆

⋆Arizona State University
{etrickel, czhu62, fishw, yans, tbao, doupe}@asu.edu
†University of California, Santa Barbara
{pagani, lukas.dresel, vigna, chris}@cs.ucsb.edu

***Abstract*— Black-box web application vulnerability scanners attempt to automatically identify vulnerabilities in web applications without access to the source code. However, they do so by using a manually curated list of vulnerability-inducing inputs, which significantly reduces the ability of a black-box scanner to explore the web application's input space and which can cause false negatives. In addition, black-box scanners must attempt to infer that a vulnerability was triggered, which causes false positives.**

To overcome these limitations, we propose Witcher, a novel web vulnerability discovery framework that is inspired by grey-box coverage-guided fuzzing. Witcher implements the concept of *fault escalation* to detect both SQL and command injection vulnerabilities. Additionally, Witcher captures coverage information and creates output-derived input guidance to focus the input generation and, therefore, to increase the state-space exploration of the web application. On a dataset of 18 web applications written in PHP, Python, Node.js, Java, Ruby, and C, 13 of which had known vulnerabilities, Witcher was able to find 23 of the 36 known vulnerabilities (64%), and additionally found 67 previously unknown vulnerabilities, 4 of which received CVE numbers. In our experiments, Witcher outperformed state of the art scanners both in terms of number of vulnerabilities found, but also in terms of coverage of web applications.

1. Introduction

Web application vulnerabilities are showing no signs of waning as the number of web application keeps increasing and the supported frameworks keep diversifying. These web vulnerabilities, such as SQL injections, can be catastrophic to the developers of the web application, the companies running the web application, and the end-users who visit and store their data on the website application.

Due to the number and diversity of web applications, it is critical to create automatic techniques that discover web vulnerabilities. Prior work has proposed different crawling and detection techniques, which utilizes one of the following approaches: white-box [1], [2], [3], [4], black-box [5], [6], [7], [8], and grey-box [9], [10]. However, these approaches are limited in their applicability to web application language, vulnerability type, or application inputs.

White-box static analysis tools [1], [2], [3], [4] rely on analyzing the web application's source code which is not always available. Moreover, white-box tools typically model the semantics of the specific language, which makes them language-specific, and thus applying those tools to new languages or frameworks require significant effort.

Black-box web application vulnerability scanners [5], [6], [7], [8] do not require source code and can analyze any web application—regardless of the web application's programming language. These tools generate legitimate web application inputs to explore the application and then attempt to infer the existence of vulnerabilities by sending input designed to trigger a vulnerability to the web application. The vulnerability-inducing inputs, however, are significantly constrained as they originate from manually curated strings or templates based on expert heuristics for vulnerability types [11]. As a consequence, black-box scanners will miss vulnerabilities triggered by inputs that are outside the pre-configured strings and templates. In essence, using hard-code vulnerability inducing inputs significantly reduces the ability of a black-box scanner to explore the web application's input space, thus introducing false negatives.

Even worse, black-box scanners can only infer vulnerabilities based on the output of the web application. Such inference can be error-prone. For example, consider a web application that returns an HTTP 500 status code (which denotes an internal server error). Existing black-box scanners such as Burp [5] use this error code to decide if their vulnerability-inducing input successfully triggered a vulnerability—in the case of a black-box scanner looking for a SQL Injection vulnerability, an HTTP 500 error can indicate that the input caused an SQL error. However, such an error can be caused by other, unrelated issues, such as an implementation bug rather than a security vulnerability. Therefore, inferring a vulnerability from the outside introduces false positives.

Some recent work has introduced the concept of grey-box fuzzers for automatically testing web applications [9], [10]. These tools use coverage information to guide the generation of inputs. These tools have had some success; however, the approaches target only a single language, do not detect SQL or command injection vulnerabilities, and are closed source [9], and are relatively slow [10].

In this paper, we propose Witcher, a novel web vulnerability discovery framework that is inspired by grey-box coverage-guided fuzzing. Our idea is to explore the web application's input space (without solely relying on hard-coded heuristics) by using execution coverage information to efficiently guide the generation of random inputs.

The application of grey-box coverage-guided fuzzing to web vulnerabilities faces a number of challenges. On a high level, the challenges to web fuzzing arise because the web application code—the *target under test* that contains the vulnerabilities of interest—is not the entire *execution object* but is instead a small subcomponent. When fuzzing a binary, the entire binary is both the *execution object* and the *target under test* (i.e., the security analyst is analyzing whether a vulnerability exists anywhere in binary). However, when fuzzing web applications the *execution object* contains three components: the web server, which parses the HTTP request; the web application runtime environment, which uses the input from the web server to generate a response; and the data storage and local executor, which the web application's logic uses to complete the request. For most web applications, the web application runtime environment breaks down into two subcomponents: the web application code and the code's execution environment (e.g., the interpreter or virtual environment). The web application code, a subcomponent of the web application runtime environment, is the *target under test* that contains the web application's logic and the vulnerabilities. The multi-component aspect and the other non-target components create several of the challenges that impede the use of grey-box coverage guided fuzzing to discover web vulnerabilities.

Detecting the input that triggers a web application vulnerability. Detecting whether an input triggers a vulnerability requires a tool to reason about the system being in a vulnerable program state. When detecting memory corruption vulnerabilities, traditional binary fuzzing uses a segmentation fault as an indication that input sent to the binary transitioned the system into a vulnerable program state. Current black-box scanner approaches use heuristics to infer that a given input triggers a vulnerable program state. Therefore, a key challenge is to create an approach that can identify when input to a web application leads the web application in a vulnerable program state.

Generating feasible inputs for end-to-end execution. As the execution object is composed of a web server and web application code, a successful input must satisfy both components (i.e., be a valid HTTP request for the web server and also include the necessary input parameters for the web application logic). While, in theory, a random input generation scheme will eventually produce feasible inputs, it is critical to design an approach that generates inputs that are both syntactically and semantically valid for the target web application, thus fuzzing effectively.

Collecting effective web application coverage. A strength of grey-box coverage-guided fuzzing for binary applications is that the fuzzer only keeps randomly generated inputs that exercise *new* code of the application and collecting this *coverage* information is a critical part of modern fuzzing [12], [13]. One possible approach for collecting coverage for web applications is to insert instrumentation into the web application. However, such an approach is not generally applicable to all web application, does not scale, and requires source code, which is not always available. A scalable and web application-independent approach is necessary to address web application coverage accounting.

Mutating inputs effectively. Similar to binary fuzzing, the mutation strategy of a grey-box coverage-guided fuzzer is also important to fuzz web applications effectively. However, little research has been done to study the mutation strategy for web applications. Therefore, we need to create mutation strategies that can generate high-quality new inputs and increase the fuzzing effectiveness.

Witcher is designed to tackle the prior four challenges. It does not require the source code of individual web applications, and we show that effective code coverage is possible with only 1–5 lines of changes to the language's interpreter. This change can then be used for *any* web application that runs on the interpreter. To demonstrate our approach, we implement Witcher support for web applications written in PHP, Python, Node.js, Java, Ruby, and C. For each of these languages, Witcher is able to detect both SQL injection and command injection vulnerabilities.

To demonstrate Witcher's advantages over the current state-of-the-art, we perform a multi-faceted evaluation. We compare different configurations of Witcher, enabling and disabling different features to demonstrate the features' efficacy. We evaluate Witcher on 13 web application with known vulnerabilities and five modern web applications with no known vulnerabilities. Overall, Witcher found 90 vulnerabilities in total, 67 of which were previously unknown. We then compare Witcher's code coverage and vulnerability discovery to Burp [5], a commercial black-box web vulnerability scanner, on nine of the PHP web applications. Last, we compare Witcher's code coverage to the recently published black-box vulnerabilty scanner Black Widow [14] and the grey-box scanner WebFuzz [10].

In summary, we make the following contributions:

- We create a set of techniques that address the challenges of applying grey-box coverage-guided fuzzing to web applications, and we propose a new framework that enables coverage-guided fuzzing on web applications.
- We develop Witcher, a grey-box web application vulnerability *fuzzer* that can discover multiple types of vulnerabilities from different web applications. Witcher automatically analyzes server-side binary and interpreted web applications written in PHP, JavaScript, Python, Java, Ruby, and C and detects SQL injection, command injection, and memory corruption vulnerabilities (only in C-based CGI binaries).
- We evaluate Witcher to understand the specific impacts achieved by our various techniques, the effectiveness of the approach on real-world web applications, and its applicability to the analysis of non-traditional targets such as IoT devices. In our evaluation, Witcher identified 23 out of 36 known vulnerabilities, which outperforms the state-of-the-art web vulnerability discovery tool. Moreover, in all but one web application, Witcher reached more lines of code than the state-of-the-art scanners Black Widow and WebFuzz. Witcher also identified 67 previously unknown vulnerabilities, which we are in the process of disclosing to the relevant parties.

To support open science and future researchers in the field, we will open source our Witcher prototype, our dataset of web applications, and the results of our experiment upon publication of this paper.

2. Background

Before we discuss the details of Witcher, we first introduce web application and injection vulnerabilities, and then provide

a high-level overview of automated application testing and coverage-guided fuzzing.

2.1. Web Applications and Vulnerabilities

Typically, a web application runs on a web server and interacts with its clients over a network. A client accesses the web application by sending an HTTP request to the web server, which parses and routes the request to the web application. The web application takes the input, performs the appropriate actions, and responds to the request. In this architecture, the web server acts as a gateway to the web application and a web application can be written in any language. Witcher accesses web application resources using either HTTP requests or direct Custom Gateway Interface (CGI) requests.

HTTP Requests. HTTP is a stateless client–server protocol used by web servers [15]. An HTTP request consists of a request line, zero or more header fields, and an optional message body.

Although they are not limited to it[1], web applications typically accept user input through the Cookie header (used for establishing stateful-requests), URL query parameters (ampersand-delimited list of name=value pairs), and the HTTP body (ampersand-delimited list of name=value pairs). For simplicity, we refer to all of the methods for transmitting user input (the headers, the URL query string, and the HTTP body) as HTTP parameters or simply *parameters*.

CGI Requests. The Custom Gateway Interface (CGI) enables a web server to directly invoke executable programs by translating an HTTP request into a CGI request (where aspects of the HTTP request are accessible via environment variables and standard input) [16]. Although many web applications replaced CGI with FastCGI, Apache Modules, and NSAPI plugins [17], CGI applications are still extensively used in embedded devices, such as routers and web cameras [18].

Injection Vulnerabilities. Injection vulnerabilities are an instance of code and data mixing [19], and they occur when a web application sends unsanitized user data to an external parser, such as the shell to execute commands or a database to execute a SQL query. A malicious adversary can exploit such a vulnerability by supplying user input that tricks the external parser into mis-interpreting the user-supplied data as code, thus altering the semantics of the parsing.

In a SQL injection vulnerability, an attacker sends a properly formatted payload with SQL code in their input, which is sent to the database as an SQL query. When the database executes the query, it also executes the attacker's injected SQL code. Similarly, in a command injection vulnerability, an attacker creates a payload that causes additional shell commands to execute.

2.2. Motivating Example

Consider the PHP web application in Listing 1 in the Appendix, which we created based on patterns that exist in real-world web applications and CVEs (described in § 5.2). Depending on the page's purpose, it offers the user different form fields. For additions to the database, it includes pname and ptype. For updates to the database, it includes pid and ptype. However, in this example, the update functionality was removed from the web application front-end but was left in the server side PHP. As a result, the client interface does not give any hint about the update functionality, which makes it unlikely for a black-box vulnerability scanner to trigger the latent PHP update code.

The code in Listing 1 contains three SQL injection vulnerabilities that the commercial black-box vulnerability scanner, Burp, does not detect. The first vulnerability exists in the add functionality. A successful attack requires a change to occur in the first half of the ptype field, which is used in the SQL statement without being sanitized. The second vulnerability occurs in the latent PHP update code. For an attacker to exercise the vulnerability, they must discover the update action and use the color portion of the ptype field to exploit the vulnerability. The last vulnerability requires the use of the add and update functionality because the update code requires the pid to exist in the database but does not enforce any limitations on the format of the pid. Thus, an attacker inserts the payload into the pid field in the database and then triggers an update to exploit the third vulnerability.

A black-box vulnerability scanner will most likely not find any of the three vulnerabilities. It is unlikely to find the first vulnerability because to reach it the ptype variable must contain an underscore, which does not exist in the scanner's predefined list of payloads. Next, black-box scanners are unlikely to find the other vulnerabilities because the client interface does not include the value necessary to trigger the update.

Nevertheless, Witcher finds all three of the vulnerabilities automatically. Witcher finds the first vulnerability by mutating valid input to include the underscore and values that will result in a malformed SQL statement. On parsing the malformed SQL, Witcher detects the vulnerability. Witcher finds the other two vulnerabilities because during the fuzzing process it will mutate act's value to 'u' and mark the input as interesting because act=u resulted in a previously unseen program state. Witcher then concentrates on the interesting input, which will cause Witcher to trigger the second vulnerability by adding a malformed version of ptype and the third vulnerability by using a malformed pid value that was stored into the pid column using the 'a' action. Although the third vulnerability requires the application to enter a particular state, Witcher does not analyze the application state; instead, Witcher triggers the vulnerability because the database maintains the proper state between requests.[2]

2.3. Automated Application Testing

Automated application testing falls into one of three categories, which vary depending on how much access the testing technique has to the application: black-box, white-box, and grey-box. In black-box testing, the testing runs without access to the internals of the target application [13]. As a result, black-box testing focuses only on the inputs and outputs of the application [20]. For example, a black-box web vulnerability scanner, such as Burp or Skipfish, works from outside a web application to find new inputs [21].

1. A web application may parse any aspect of the HTTP request for user input.

2. Unlike most binary fuzzing targets where each execution is a blank slate, the database preserves state between executions.

On the other end of the spectrum, white-box tools generate inputs by analyzing the source code of the application with the goal of better understanding the application's semantics [13]. Some examples of white-box analysis include symbolic execution and taint tracking [22], [23], [24]. White-box tools have access to the target application's source. Thus, white-box tools can reason about the internal structure as well as the operation of the application and can evaluate operation without being limited to paths that can be reached during execution; however, they are focused on a particular programming language and often suffer from false positives.

Grey-box testing blurs the line between white- and black-box testing as it runs with limited access to the application. The testing application uses a less-intensive form of either static or dynamic analysis. For example, coverage-guided mutational fuzzing uses either static or dynamic instrumentation to gather coverage information, which is used to identify input that exercises new execution paths in a program (thus breaking the purely black-box approach).

2.4. Coverage-Guided Fuzzing

Fuzzers automatically test applications by inputting test cases and causing the target application to enter different program states. When the fuzzer starts, it receives a set of input *seeds* that it places into a test case queue. The fuzzer then derives new test cases from those in the queue.

To derive a test case from those in the queue, the fuzzer mutates the test case using a variety of mutation strategies. For example, American Fuzzy Lop (AFL) uses deterministic mutation strategies such as bit flipping, integer arithmetic, and dictionary insertion [25]. In addition, AFL uses random strategies such as random splicing and insertion of data from a user-supplied dictionary. After mutating the input, the fuzzer sends the altered input to the target application.

For coverage-guided fuzzing, the fuzzer captures coverage data that approximates the program states to guide test case selection. The fuzzer captures coverage data that approximates the program states that is far less complete than an execution trace. The instrumentation approximates the program states because it is too processing intensive for a fuzzer to capture and analyze a complete execution trace for each execution. The fuzzer obtains coverage data through either static or dynamic instrumentation. For static instrumentation, an analyst compiles the target application's source code with a modified compiler. For dynamic instrumentation, a dynamic instrumentation tool (e.g., Pin) or a emulator modified to provide coverage data (e.g., QEMU-user) produces coverage information during execution [26], [27].

A coverage guided fuzzer saves a test case when it deems the test case as interesting. The fuzzer tags a test case as interesting when it causes the program to reach a new location or causes the application to emit a fatal signal, such as a segmentation fault, which often means the application entered a vulnerable state.

3. Challenges

Inherent challenges exist in creating a grey-box coverage-guided web application vulnerability fuzzer. We group these challenges into those that *enable* automated analysis and those that *augment* the exploration of the input space.

3.1. Enabling Fuzzing of Web Applications

Enabling the automated analysis of web applications requires the fuzzer to generate input that will reach the target application and to detect the existence of a vulnerability.

1. **How to detect web injection vulnerabilities?** A fuzzer's goal is to identify when a test case causes the program to enter a vulnerable program state. Typically, the types of faults generated by SQL and command injection vulnerabilities do not culminate in an error signal that a fuzzer can detect and they often occur in a separate process (e.g., the data storage layer). Therefore, we must develop a new approach that will enable the fuzzer to detect SQL and command injection vulnerabilities.
2. **How does the system generate a test case that will exercise an end-to-end execution of the web application?** Web applications require the test cases to match a semi-structured format to pass both the syntax checks of the web server *and* the semantics of the web application. In contrast, mutational fuzzers generate high-entropy random data that does not effectively explore the input space of applications.
   Without enforcing some structure on the test cases, the fuzzer will not be able to explore the state space of the web application. First, if the test case fails to meet the HTTP request format, then the test case will not reach the target web application because the web server will reject it. (see § 2.1). Second, the test case must include the parameters expected by the target application. Without the parameter variable names, a reasonable exploration of the target's input space is impossible because a fuzzer would generate billions of test cases to randomly guess a single variable name of only a few characters.

3.2. Augmenting Fuzzing for Web Injection Vulnerabilities

Even if a fuzzer meets the prior challenges to enable fuzzing for web applications, adding those features is not sufficient to efficiently explore the target's input space and discover the vulnerabilities. Analysis of applications using a coverage-guided mutational fuzzer is a computationally intensive task and despite numerous resources its use often results in only a portion of the input space being explored. For web applications, this problem is even worse because fuzzers do not receive execution trace information from the targeted web application code and the fuzzer does not effectively mutate the parameter and values.

1. **How to effectively collect coverage of the web application?** Coverage-guided fuzzers require instrumentation of the target application to gather coverage information. However, in the case of fuzzing applications written in interpreted languages, limited tools exist that allow instrumentation of the target web application. Instead, the fuzzer instruments the interpreter's runtime binary—not the target web application code. As a result, the coverage information reflects the interpreter's code and not the target web application, thus causing the fuzzer to focus on exploring the runtime interpreter's code instead of the

web application's code. Although coverage of the interpreter will change with alterations to the web application's execution, a large portion of the coverage data is irrelevant noise that obfuscates the target web application's coverage information. Therefore, to facilitate exploration of a web application the instrumentation must only report the target web application's execution.

2. **How to effectively mutate test cases?** Although efficient for generating test cases for binary application input, the mutation strategies used by fuzzers focus on the creation of test cases with high-entropy that require no context. However, these high-entropy test cases are less effective for the exploration of web application's input state space. Even if the high-entropy test cases are properly formed HTTP requests, the test cases lack efficacy in testing web applications because they fail to take advantage of the contextual information available to the client (e.g., the variable names found in the HTML form fields). Therefore, it is necessary to create new mutation strategies that incorporate the proper format and exploits the context offered by the web application's client interface.

4. Witcher's Design

Witcher is a grey-box web vulnerability scanner that uses a coverage-guided mutation fuzzer to drive the automated exploration of web applications. Witcher is categorized as a grey-box fuzzer because, similar to traditional coverage-guided fuzzing, it relies on coverage data to identify interesting test cases. Other than instrumenting an interpreter for coverage data, Witcher does not perform any analysis on the source code; thus, Witcher can operate without any access to the source because it can run a byte-code version of a web application. As much of the binary fuzzing research uses AFL as a starting point, we chose to use AFL as the base for demonstrating the efficacy of the Witcher framework.

Witcher solves the challenges impeding the use of coverage-guided mutation fuzzing (described in § 3) using five additional components. To enable fuzzing for web injection vulnerabilities, Witcher implements the Fault Escalator, the HTTP Harness, and the Request Crawler (the blue components in Figure 1). To augment fuzzing for web injection vulnerabilities, Witcher implements the Coverage Accountant and the HTTP Mutator (the green components in Figure 1).

4.1. Enabling Fuzzing for SQL and Command Injection Vulnerabilities

4.1.1. Fault Escalator. For a program to be free of vulnerabilities it must be impossible for user-supplied input to transition the program to a vulnerable program state, thus by identifying when this vulnerable program state occurs a scanner can detect a vulnerability in the target application. In traditional binary fuzzing, the vulnerable state results from a memory corruption vulnerability and binary fuzzers detect the transition to a vulnerable state by detecting a segmentation fault signal [13].

We leverage this insight and expand the concept to allow the fuzzer to detect when a program transitions to a vulnerable program state resulting from a SQL or command injection vulnerability. SQL and command injection vulnerabilities occur when user input causes an external parser (shell command parsing for command injection and SQL parsing for SQL injection) to interpret the user input data as code. For example, a SQL injection vulnerability occurs when attacker-controlled input alters the syntax of a SQL query. In a well-formed SQL query, user-controlled input cannot alter the syntax of a SQL query. As a result, we can view a syntax error thrown by an external parser as analogous to the segmentation fault signal that results from a memory corruption vulnerability. This correlation forms the basis behind Fault Escalator: if attacker controlled input causes a syntax error in the external parser, then an attacker can alter the command, and it is more likely than not that an exploitable vulnerability exists. Thus, when the parsing error occurs, Fault Escalator escalates the error to a segmentation fault, which notifies the fuzzer that the current test case caused a vulnerable program state. For example, imagine a PHP application that executes mysqli_query($con,"SELECT ID from tbl where ID=". $_GET['id']) and the fuzzer sets id=1', which results in a malformed SQL statement due to the single quote. When the page executes the SQL statement, the SQL parser will return a parsing error, which is intercepted by Fault Escalator and escalated to a segmentation fault that is detected by the fuzzer.

If an application uses unsanitized input to create a SQL statement or a shell command, then the stochastic input generated by the fuzzer is likely to result in a parsing error. 2Although not every input generated by a fuzzer will cause a SQL syntax error in a vulnerable query, given the stochastic nature of the fuzzer, it is unlikely that an vulnerable query will fail to result in a SQL parsing error during a fuzzing session. This is also confirmed by our experiments: none of the vulnerabilities that Witcher missed are related to false negatives in Fault Escalator.

Command Injection Escalation. For command injection, Witcher implements fault escalation using dash's command parser. The program dash is the Debian Almquist shell, which is designed to be POSIX-compliant and as small as possible. Dash replaces /bin/sh on most Linux systems [28]. Linux uses /bin/sh, and its smaller replacement dash, when an application executes a shell command. For example, a PHP script using exec(), system(), or passthru(), or a Node.js script using exec()[3] send their command to /bin/sh, which means that dash parses and runs the command. Witcher's version of dash (3 lines of code difference from the original) escalates a parsing error to a segmentation fault. Thus, if the application uses unsanitized user input to create a SQL or shell command, then the random data input by the fuzzer into the web application will result in a parsing error and trigger a segmentation fault.

SQL Injection Escalation. Witcher's Fault Escalator implements SQL injection escalation for MySQL and PostgreSQL using a technique similar to command injection escalation. To catch the syntax error, Witcher uses LD_PRELOAD to hook the libc function recv(), which is used to communicate with the database. Whenever any response from the database contains a SQL syntax error message, Witcher triggers a segmentation fault.

Fault Escalation is not Limited to Syntax Errors. Although the fault escalation techniques for SQL and command injection detection rely on the existence of a syntax error, the concept of

3. Node.js's spawn() method does not use /bin/sh.

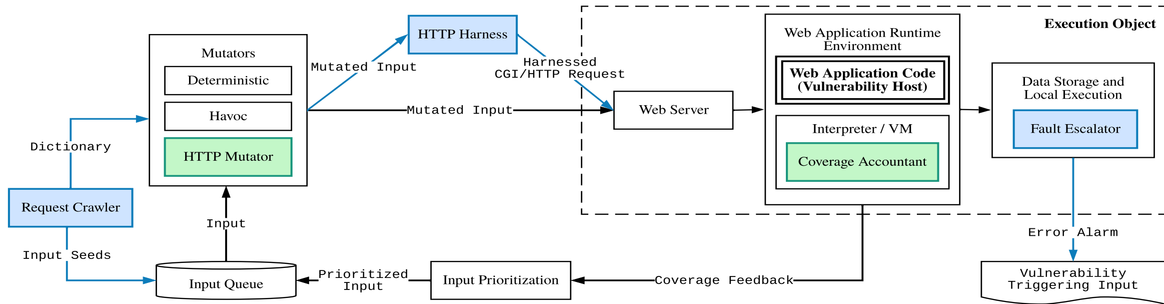

Figure 1: Overview of Witcher. Witcher's components with a blue border are enabling (i.e., necessary to fuzz a web application). Witcher's components with a green border are augmenting and enhance the fuzzer's performance on web applications.

fault escalation applies any type of warning, error, or pattern. For example, Witcher might handle file inclusion by overriding libc's open function and escalating an error when the filename parameter contains non-ascii values.

Memory Corruption Vulnerabilities. Witcher detects memory corruption vulnerabilities in CGI binary applications without the aid of fault escalation. This occurs because the fuzzer inherently detects memory corruption vulnerabilities when executing a binary application due to the segmentation fault triggered by the input.

Bugs and Vulnerabilities. Similar to a segmentation fault, the occurrence of a syntax error in a SQL statement or shell command resulting from user input signifies a bug that should be fixed and is highly likely to be vulnerable. In our evaluations, the occurrence of a syntax error signified a problem with the validation or sanitization of user input, which often meant the existence of SQL or command injection vulnerability.

However, it is possible that, due to constraints on user input, an attacker is unable to leverage the syntax error to exploit the SQL injection or command injection. For example, a web application that restricts an unsanitized parameter to be only one character, might not represent an exploitable vulnerability, but rather a bug. For this reason, in this paper, we will label any escalated fault as either a vulnerability or a bug, depending on whether we confirmed the vulnerability was *exploitable* or *not*.

Cross-site Scripting Vulnerabilities. The fault escalation technique leverages the randomness generated by the fuzzer to identify critical vulnerabilities in the *server environment*. Unfortunately, cross-site scripting vulnerabilities do not readily fall into this category (browsers are very forgiving in their parsing of HTML and therefore it can be difficult to reliably and quickly detect an cross-site scripting in HTML). As a result, we choose to focus on command injection and SQL injection. Moreover, SQL and command injection vulnerabilities represent a class of vulnerabilities that mutation-based fuzzers could not readily detect prior to our work.

4.1.2. Request Crawler. The Request Crawler (Reqr) operates as a black-box crawler that automatically discovers HTTP requests and parameters. Reqr extracts HTTP requests from all types of web applications including web applications that rely heavily on client-side JavaScript to render the web application's interface, links, forms, submissions, and requests (e.g., Rconfig, Juice Shop, and WebGoat in § 5).

Reqr operates similar to black-box vulnerability scanners: it is given an entry point URL and optionally valid login credentials and the login URL. Reqr uses the Node.js library Puppeteer (an API used to control Chromium) to simulate user actions and capture requests [29]. After Reqr starts, it will login to the web application (if required) and load the entry point. Once a page is loaded, Reqr statically analyzes the rendered HTML to identify the HTML elements that create HTTP requests or HTTP parameters, such as a, form, input, select, and textarea. Next, Reqr listens for HTTP requests while simulating user events (e.g., mouse clicks, entering values into form fields, and scrolling the page) both systematically and randomly. Reqr systematically fires the events by targeting every HTML element that accepts user events. In addition, Reqr randomly fires user input events (e.g., clicks, form fills, scrolling, and typing) using the Gremlins testing tool [30].

When Reqr completes, it creates a file containing all the request information. Witcher uses the request information to create the fuzzer's seeds and to build the fuzzer's dictionary.

4.1.3. Request Harnesses. Witcher's HTTP harnesses translates fuzzer generated inputs into valid requests. Due to the different execution models, Witcher has a different harness design for PHP and CGI binaries than it does for Python, Node.js, Java, and QEMU-based binaries. For PHP and CGI web applications, Witcher translates from the fuzzer input format into a CGI request. For Python, Node.js, Java, and QEMU-based binaries Witcher translates fuzzer's input into an HTTP request (see § 2.1).

CGI Harness. PHP (via php-cgi) and CGI binaries use the same harness because both rely on a CGI request and the invoked endpoint runs to completion once invoked. For PHP and CGI binaries, the HTTP harness uses LD_PRELOAD to create a fork server that starts the interpreter or the binary just before it processes the input. The harness receives each new input from the fuzzer, translates the input into a CGI request, and then transmits the request into a newly forked process.

HTTP Request Harness. Witcher fuzzes the other interpreted languages and the QEMU-based web applications through their associated web server by leveraging an HTTP request harness. The HTTP request harness decouples the fuzzer from the targeted platform and enables the fuzzer to work on applications that it does not automatically support. For example, AFL cannot fuzz a Node.js web application that uses Express because the application runs indefinitely waiting for new requests and is multi-threaded.

The HTTP request harness creates a bridge between AFL and the web server to leverage the web server's interface to

the web application. The HTTP harness includes its own fork server that increases the request submission throughput. The harness receives input from the fuzzer, it translates this input into a well-formed HTTP request, and sends the HTTP request to the web server. Last, when Fault Escalator detects a SQL statement or shell command that causes a syntax error, Fault Escalator sends a segmentation fault to the HTTP harness process, which the fuzzer automatically detects.

Translating Fuzzing Input into a Request. Both the CGI Harness and the HTTP Request Harness act as translators between the fuzzer and the web application. Witcher automatically creates seeds for the fuzzer that follow a null-terminator delimited format. The seeds include fields for cookies, query parameters, post variables, and other header values. As a result, the fuzzer creates test cases based on the format, which the harness then translates into the appropriate request type.

In addition to handling the fuzzer's input, the harness sets a few other parameters for the output request. The harness keeps the request path static for each instance of the fuzzer, which means Witcher fuzzes a single URL at a time. In addition, the harness adds any session cookies, query variables, or post variables that are necessary for the web application to operate correctly. For example, most of the endpoints in the OpenEMR web application require a valid login session. As a result, prior to invoking the fuzzer, Witcher inputs valid login credentials to generate a valid session cookie, which the harness includes in every request.

4.2. Augmenting Fuzzing for Web Injection Vulnerabilities

4.2.1. Coverage Accountant. Witcher's Coverage Accountant (inside the Interpreter block on the right-side of Figure 1) provides byte-code execution coverage information to the fuzzer for the interpreted languages PHP, JavaScript, Python, and Java as well as web applications that can be executed using QEMU-user or QEMU-system. Witcher uses the Coverage Account because trying to fuzz a web application by instrumenting the interpreter results in a significant amount of noise. For example, when we used AFL's standard approach of instrumenting the interpreter for a simple web page that had six unique paths the fuzzer reported that it found over a thousand unique paths. The discrepancy occurs because by instrumenting the interpreter the fuzzer focuses on test cases that alter the interpreter's execution paths; however, changing the interpreter's execution path does not usually translate to the target web application. Even though many of the paths identified by the fuzzer do not provide additional coverage of the web application code, the fuzzer stores and attempts to mutate each of the test cases because they changed the execution of the interpreter. The increased number of *equivalent* test cases prevents the fuzzer from making meaningful progress exploring the target web application. Therefore, Witcher created the Coverage Accountant to more accurately capture the web application's execution paths.

Interpreter Instrumentation. Despite the different interpreter architectures, the instrumentation of the byte-code is similar between them. The interpreter reads the source file and translates the code into byte-code instructions. Next, the interpreter executes the instruction.

During the execution of an instruction, the augmented interpreter calls Witcher's code coverage library function. Witcher's library function receives the line number, opcode, and parameters of the current byte-code instruction. Witcher then updates the fuzzer's coverage information using the line number and opcode of the current and prior instructions.

Witcher's interpreter instrumentation targets the web application. In Listing 1, the code has six visible paths plus several latent paths that occur within the functions $\$_GET()$, $\mathrm{mysqli_query}()$, and $\mathrm{uniqid}()$. Thus, with Witcher's PHP instrumentation the fuzzer will find six paths it deems unique.

CGI Binaries. In addition to interpreted languages, Witcher supports fuzzing CGI binaries. For CGI binaries, Witcher uses AFL's instrumentation when the binary's source code is available. When its source code is unavailable, Witcher's fuzzer uses dynamic instrumentation via QEMU [31]. Although the QEMU-user modifications for instrumentation are already included with AFL, Witcher makes additional modifications to QEMU-user to enable fault escalation. For QEMU-system, Witcher's modifications target the data structures used to store QEMU's intermediate language, which is processed similarly to the byte-code used by the interpreted languages.

Beyond AFL. Witcher uses AFL as the coverage guided mutational fuzzer; however, the Witcher framework can incorporate more advanced fuzzers. If a new fuzzer uses an improved technique for instrumentation, such as PTrix [32], or mutation, such as Tfuzz [33] or AFL++ [34], then Witcher can incorporate the fuzzing tool while still employing the web crawling and fault escalation to detect a wider set of vulnerabilities than either of those tools could do alone.

4.2.2. HTTP-specific Input Mutations. We modified AFL by adding two new mutation stages that focus on manipulating HTTP parameters. The purpose of these mutations is to inject parameters into the inputs more quickly than standard AFL and to share/swap values at the variable level instead of treating the parameters as a mere sequence of unstructured bytes. In effect, the mutators reduce and modulate AFL's entropy in a way that is more consistent with the syntax and semantics of web applications.

HTTP Parameter Mutator. The HTTP Parameter Mutator cross-pollinates unique parameter name and values between the interesting test cases stored in the fuzzer's queue. Witcher fuzzes one URL endpoint at a time; however, an interdependency often exists between the variables of different test cases. By cross-pollinating the parameters, the fuzzer provides targeted test cases that are more likely to trigger new execution paths than random byte mutations. For example, in Listing 1 if a test case contains $\mathrm{act{=}a}$ and another contains $\mathrm{ptype{=}dog_red}$, then by combining them, the fuzzer would reach the vulnerable code.

HTTP Dictionary Mutator. The HTTP dictionary mutator decreases the number of executions necessary to pair the current input with the variables in the dictionary. Many endpoints serve multiple purposes, as a result, an endpoint may have several requests that use different HTTP variables. For a given endpoint, Witcher places all the HTTP variables discovered by Reqr into

the fuzzing dictionary. The HTTP dictionary mutator takes advantage of the contextually similar variables by mixing and matching them with the current request. The HTTP dictionary mutator does this by randomly selecting one to ten variables from the dictionary and adds them to the current test case.

5. Evaluation

In this section, we aim to answer the following research questions through the evaluation of Witcher:

RQ1. How effective are Witcher's augmentation techniques at exploring the web application and identifying vulnerabilities? Do both augmentation techniques contribute to fuzzing (§ 5.1)?

RQ2. How effective is Witcher at identifying vulnerabilities in web applications (§ 5.2)?

RQ3. How does Witcher's code coverage and vulnerability discovery compare to a commercial black-box vulnerability scanner and cutting-edge vulnerability scanners (§ 5.3)?

5.1. Witcher Augmentation Techniques Evaluation

To better understand the impact of Witcher's augmentation features on web application fuzzing, we evaluate Witcher with different configurations and test them on two data samples. The first is a microtest using 10 self-created PHP scripts, and the second is OpenEMR, a real-world web application.

Recall that we designed two fuzzing augmentation techniques: coverage accountant and HTTP mutator. In this experiment, we used Witcher with four different configurations:

AFLR does not have coverage accountant or HTTP mutator. This configuration is meant to be a baseline against Witcher with fuzzing augmentation.

AFLHR has HTTP mutation yet does not have coverage accountant.

WiCR has coverage accountant yet does not have HTTP mutator.

WiCHR has both coverage accountant and HTTP mutator.

5.1.1. Microtest Evaluations. In the microtest evaluation, we ran each configuration on a set of ten PHP scripts designed to test the capabilities of Witcher. Each of the scripts includes a single path that reaches an injection vulnerability. The evaluation of a script with a particular configuration ran until either the target injection was reached or four hours elapsed, whichever occurred first.

The dictionary simulated the output generated by Reqr and it included the parameters used by each of the scripts, plus 100 unrelated parameters to simulate unused variables. Each script and configuration were run five times to stabilize the results.

Each of the microtests targeted the functionality of Witcher's components or added additional difficulty. The first set of scripts (post-2, post-5, post-10, get-5, and cookie-5) follow the same general format that tests Witcher's ability to input the type of variable under test. For example, post-2 (Listing 2 in the Appendix) executes a SQL statement that directly concatenates the value returned by $_GET['vul'] (i.e., an unsanitized value) when the functions isset($_POST['nv1']) and isset($_POST['nv2']) both return true. As a result to pass the test, the fuzzer must provide the post variables nv1 and nv2 and the URL parameter vul that contains a value that will trigger a SQL parsing error.

The next set of scripts test Witcher's ability to provide specific variable and values. To reach the vulnerable SQL statement in select-3, the variables and values were provided in the dictionary (as though they were harvested by the crawler) because they were provided in the user interface via the <select> tags. equals-1, equals-3, and loop-10 tests, the values necessary to reach the vulnerable SQL are not provided in the user interface; thus, the fuzzer, must discover the values. equals-1 (Listing 3 in the Appendix) executes the vulnerable SQL statement that concatenates the unsanitized input variable $_GET['vul'] when $_GET['nv1'] == "YYYY", the necessary value YYYY was not provided in the dictionary. Similarly, in equals-3, the fuzzer must discover three unknown values to reach the vulnerable statement. loop-10 (Listing 4 in the Appendix) evaluates the input using a for loop to perform a byte-by-byte comparison instead of using == to compare the entire string, which provides the fuzzer some breadcrumbs to discover the unknown value and reach the vulnerable statement. The last test is similar to equals-1 except the fuzzer is provided the necessary value but not the variable name. findvar-1 (Listing 5 in the Appendix) executes the vulnerable statement when isset($_POST['ao3']); however, a03 is not provided in the seeds or dictionary. Excerpts from some of the microtest scripts are available in the appendix. Table 1 shows the overall results for the microtests. Based on the result, we see that AFLR failed to find any of vulnerabilities. It performed poorly because the additional noise from placing the instrumentation in the interpreter greatly reduced the number of cycles through all inputs, which limited the number of dictionary values it explored.

On the other hand, WiCHR performed the best. WiCHR reached the vulnerability a total of 34 times. However, WiCHR was unable to find the vulnerability in 3 of the looping tests because AFL gives less precedence to coverage that contains repeated instructions. Therefore, both augmentation techniques are helpful to increase the effectiveness of web vulnerability discovery, and thus Witcher will include the two techniques in subsequent evaluations.

We used the Mann Whitney U-test to verify that the differences between the configurations were statistically significant [35]. Because we opted to run until first crash or timeout, we used the sum of elapsed time per trial to calculate the differences between the configurations. The WiCHR configuration took the least amount of time to run on every trial and the improvement versus the other configurations was statistically significant under the Mann Whitney U-test.

5.1.2. OpenEMR Evaluations. To evaluate the performance of Witcher's configurations on a real-world web application, we performed a second comparative evaluation using OpenEMR version 5.0.1.7. We used Reqr to identify the application's URLs and input variables.

Next, Witcher fuzzed each of the URLs in five independent trials using each configuration. We excluded AFLR because of its poor performance in the microtest evaluation; thus, we evaluated the remaining 3 configurations AFLHR, WiCR, and

TABLE 1: *Microtest Comparative Evaluation Results. The values represent the number of crashes reached after five trials that were up to four-hours in duration.*

Microtest	AFLR	AFLHR	WiCR	WiCHR
post-2	0	**5**	**5**	**5**
post-5	0	**5**	**5**	**5**
post-10	0	**2**	**5**	**5**
get-5	0	**4**	**5**	**5**
cookie-5	0	**4**	**5**	**5**
equal-1	0	0	**1**	**2**
equal-3	0	0	0	0
findvar-1	0	0	0	0
loop-10	0	0	0	**2**
select-3	0	**4**	**5**	**5**

WiCHR. We initialized the database and sessions at the start of each trial to aid consistency from run-to-run.

To perform the evaluation, we gathered PHP code coverage data to use in the Mann-Whitney test. We used Xdebug, a PHP extension, to extract PHP code coverage information [36]. Next, we calculated the total lines visited for all scripts using a particular configuration and trial. With the total lines visited, we then compared the configurations using the results from each trial.

Table 7 in the Appendix shows the results: the total lines of code reached using each of the different configurations in each trial. WiCHR consistently executed the most lines of code followed by WiCR and then AFLHR. The differences in performance between the feature sets was statistically significant: the Mann-Whitney U-Test resulted in a p-value of 0.01208.

Table 7 also shows the vulnerabilities discovered for each trial and configuration. All the feature sets found vulnerabilities on each trial; however, both WiCR and WiCHR performed significantly better than AFLHR. WiCHR identified the most vulnerabilities on each trial.

5.2. Witcher Evaluation

Based on the results of the feature comparison shown in § 5.1, we selected the WiCHR configuration to compare Witcher with other web scanning tools. We used as an evaluation dataset a diverse set of web applications written in different languages and running on different platforms: some that have known vulnerabilities and some that were up-to-date with no known injection vulnerabilities. In this evaluation, we manually confirmed each vulnerability by verifying whether the vulnerability was exploitable or not. Excluding the interesting bugs, all the remaining command and SQL injection vulnerabilities were severe because they give an attacker the capability to destroy, alter, and exfiltrate data [37]. For the command injection vulnerabilities, we verified the application executed an arbitrary shell command. For SQL injections, we automatically exploited the vulnerabilities by providing the crash information from Witcher to SQLMap, which gained full control over the database or could execute arbitrary SQL functions.

For the known vulnerable applications we used a set of eight PHP applications, five firmware images (binaries where the source is likely written in C, and the platform is ARM, MIPSEL, and MIPSEB), one Java, one Python, and one Node.js application with a combined total of 36 known vulnerabilities. We searched

TABLE 2: *The known vulnerabilities in each web application, the amount that Witcher found, missed, and previously unknown vulnerabilities that Witcher discovered.* * *Witcher found one input where the user controls a parameter to* execve*, however we could not determine if it was exploitable so we consider this a bug rather than a vulnerability (as discussed in § 4.1.1).*

Application Description	Known Vulnerabilities			Unknown Vulnerabilities
	Existing	Found	Missed	Found
Doctor Appt. Sys.	1	1	0	3
Hosp. Mgmt.	5	5	0	43
Login Mgmt.	1	1	0	5
OpenEMR	5	1	4	5
rConfig	2	0	2	11
WackoPicko	3	2	1	0
D-Link 645	1	0	1	0*
D-Link 823G	1	1	0	0
D-Link 823G	1	1	0	0
D-Link 825	1	0	1	0
Tenda AC9	1	0	1	0
FlaskBB	0	0	0	0
Juice Shop	2	2	0	0
osCommerce	0	0	0	0
phpBB	0	0	0	0
Threadded	0	0	0	0
WebGoat	12	9	3	0
Total	36	23	13	67

for public CVEs of SQL injection and command injection vulnerabilities that had working exploits (so that we could verify the existence of the vulnerability), and this resulted in: Doctor Appointment, Login Management, Hospital Management, and rConfig. We selected WackoPicko, OpenEMR, and Juice Shop because of their known vulnerabilities and use in prior research (see Table 10 in the appendix for prior work that used the same web applications for their evaluation).

We also selected five firmware targets to demonstrate Witcher's ability to fuzz on non-interpreted web applications. We chose D-Link's 825, 823G version 1.0.2B03, 823G version 1.0.2B05, and 645 as well as the Tenda AC9 because the firmware's web server runs in the QEMU emulator, they each have known CVEs, and their CVEs included working exploit scripts. Table 6 shows the known vulnerabilities in all the applications, along with the CVE number (if known) and the vulnerability type.

We also selected up to date versions of web applications used in the evaluation of prior work to ensure that Witcher would fuzz the latest versions of web applications. In particular, we choose phpBB, osCommerce, and Wordpress, each of which were evaluated in four or more prior publications, and we also added Thredded, a Ruby on Rails web application.

The name of the 18 web applications used in this evaluation are summarized in Table 10 in the Appendix, along with the language or platform of the web application, the release date of the version of the web application tested if known (the oldest was released in 2014), the version, the number of stars on GitHub for the web application (as an estimate of the popularity of the web application), the number of Google results for a custom Google Dork (link to dork given in reference) if the web application's source is not on GitHub (as another way to estimate real-world usage), the lines of code of the web application, and if this web application was used in prior research.

To run this evaluation, we created Docker containers for each of the web applications, started the web application, and ran Witcher. Witcher's configuration included the entry URL, identification of the login page, the associated credentials, and a selector for the form field. We limited Witcher's crawler to run for four hours, while we fuzzed each URL with two or more input variables for 20 minutes, As a result, the total run time varied depending on the number of endpoints identified by the crawler.

The overview of the results for this evaluation are shown in Table 2. Witcher successfully crawled and fuzzed all of the web applications, ultimately finding a total of 90 unique vulnerabilities of which 67 were previously unknown. All discovered vulnerabilities were from web applications that had known vulnerabilities (i.e., Witcher did not discover previously unknown vulnerabilities in the latest versions of Thredded, phpBB, osCommerce, or Wordpress).

Witcher discovered 23 of the 36 (63.9%) known vulnerabilities; however, Witcher missed 13 (36.1%) vulnerabilities. Table 6 shows the detailed results of exactly which known vulnerabilities were found or missed, along with a brief description of why. In particular, eight vulnerabilities were missed because the crawler was unable to find the URL. Some URLs were not discovered by the crawler because the application required a specific series of steps, such as selecting a patient in OpenEMR (this is the known problem of exploring stateful web application [20], [14]). The crawler also missed URLs when the URL was not included in the web application's user interface, such as in the case of a backdoor URL in the Tenda AC9 firmware. In the WebGoat application, the crawler missed two vulnerabilities due to a bug in the implementation that caused the webserver to unexpectedly crash and another because the HTTP harness does not currently support the HTTP `PUT` method. It is common for dynamic analysis tools to have a higher false negative rate; nevertheless Witcher's false negative rate of 36.1% is lower than the rates reported in other publications with 47% [38] and 60% [21]. Although Witcher did not find any memory corruption vulnerabilities during the evaluation, Witcher can detect them because a memory corruption vulnerability will often result in a segmentation fault.

As shown in Table 2, Witcher found 67 previously unknown vulnerabilities (65 SQL and 2 command injections). While we plan to responsibly disclosing the unique vulnerabilities that are still relevant and undiscovered, we have already received unpublished CVEs for the OpenEMR SQL vulnerabilities: CVE-2020-11754, CVE-2020-11755, CVE-2020-11756, and CVE-2020-11757.

In addition to the vulnerabilities that Witcher reported, Witcher reported two false positives and three bugs. However, the bugs were interesting because they demonstrate the potential of using high-entropy input for testing web applications.

5.3. Grey-box and black-box comparison

Now that we evaluated the effectiveness of Witcher at identifying vulnerabilities in web application in § 5.2, we compare Witcher, a grey-box web application vulnerability fuzzer, against the state-of-the-art commercial black-box web application vulnerability scanner Burp [5], the data-driven web application crawler Black Widow [14], and the recently published grey-box crawler and fuzzer WebFuzz [10]. We choose the Black Widow and WebFuzz scanners because of their recency and performance. For example, Black Widow outperformed six other open-source web-vulnerability scanners (Arachni [39], Enemy of the State [40], Skipfish [41], jÄk [42], w3af [43], and ZAP [8]). Although we had hoped to compare Witcher's NodeJS fuzzing against Back-REST, the authors were unable to share the tool due to proprietary concerns [9]. We limited the evaluations to nine of the applications (shown in Table 10) that were written in PHP, so that we could collect code coverage using the method described in § 5.1.2.

TABLE 3: Results of vulnerabilities discovered Burp and Witcher: the number of vulnerabilities found by Burp (solo), BurpPlus Witcher, and Witcher. Number in () indicates the unique vulnerabilities found by this configuration.

Application	Burp (solo)	BurpPlus Witcher	Witcher
Doctor Appt. Sys.	2 (0)	3 (0)	**3** (0)
Hosp. Mgmt.	13 (0)	13 (0)	**43 (30)**
Login Mgmt.	1 (0)	1 (0)	**6 (5)**
OpenEMR	0 (0)	0 (0)	**5 (5)**
osCommerce	0 (0)	0 (0)	0 (0)
phpBB	0 (0)	0 (0)	0 (0)
rConfig	0 (0)	0 (0)	**11 (11)**
WackoPicko	1 (0)	1 (0)	**2 (1)**
Wordpress	0 (0)	0 (0)	0 (0)
	17 (0)	18 (0)	**70 (52)**

Burp Evaluation. To compare our approach with Burp, we evaluate how much code of the target web application is executed and how many vulnerabilities are discovered. Because Burp has its own crawling components, we compare against Burp in two different configurations: (1) Burp (solo) with no changes, where Burp crawls the web application itself, and (2) BurpPlus Witcher, where we provide Burp with the requests derived from Witcher's crawler. Therefore, the comparison of the results between BurpPlus Witcher and Witcher will not be related to the differences between the crawlers, but to the differences in input generated for the applications.

We configured Burp's scan in the same way for both Burp (solo) and BurpPlus Witcher. When we configured each scan, we chose the built-in configuration for the most complete crawl and the maximum audit coverage, and the most complete crawl was limited to five hours by default. Burp's audit (i.e., finding vulnerabilities) did not have a timeout option and ran until completion.

One of the differences between Burp and Witcher is that Burp rotates URLs and does not focus on a single target URL at a time, instead, it moves through all the URLs multiple times, which increases the likelihood of discovering new parts of a web application due to the state changes caused by another page. However, Witcher focuses on a single page at a time, which means it is less likely to trigger multi-page states.

We summarized the results of this experiment in in Table 4 and Figure 2. In the code coverage results, Witcher executed more lines of code in every application over Burp (solo) and BurpPlus Witcher. Witcher increased code coverage by more than 100% for four of the applications. One surprising result is the phpBB testcase, where the code coverage for BurpPlus Witcher was 52.3% worse than Burp (solo). This was the only experiment where BurpPlus Witcher reached the crawling timeout threshold. As a result, BurpPlus Witcher had fewer end points to investigate, which resulted in fewer lines covered.

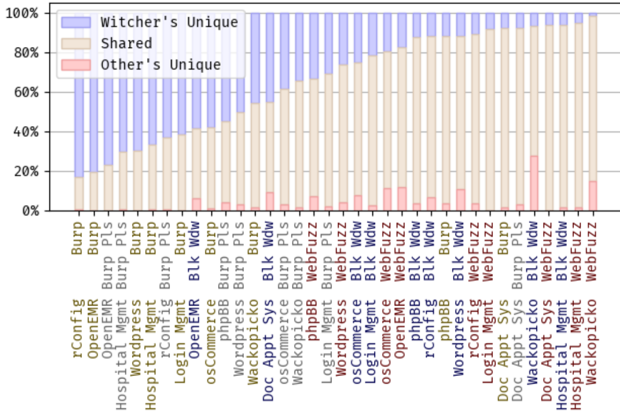

Figure 2: Each column in the stacked bar chart compares the lines found to another tool in an application. Each bar shows a percentage of the total lines found for the tool.

As shown in the vulnerability results in Table 3, Witcher discovered 70 vulnerabilities of which 52 were unique between the three configurations. Witcher found the most vulnerabilities for six of the nine applications.

Black Widow and WebFuzz Evaluation. For the Black Widow and WebFuzz evaluations, we only compare the unique lines of code executed because Black Widow and WebFuzz target XSS vulnerabilities while Witcher targets SQL and command injection vulnerabilities. We seeded WebFuzz with the HTTP requests discovered by Witcher in the crawling stage to focus the evaluation on the fuzzers. Due to the nature of Black Widow's crawler, we were unable to seed Black Widow with the same seeds. However, we did add a username and password parameter so that Black Widow would use an existing account.

Black Widow and WebFuzz interweave the execution of different pages while testing a web application. By interweaving execution, the tools may trigger and fuzz new application states that rely on the interdependence between two web pages. For example, the tool may add an item to a cart on the the product page and checkout on the shopping cart page. Currently, Witcher focuses on a single page at a time and is less likely trigger these interdependent states. In addition, Black Widow uses state monitoring to discover new application states.

Witcher's speed and mutation strategy outperformed the other scanners' URL interleaving and state monitoring. As shown in Table 4 and Figure 2, Witcher outperformed Black Widow and WebFuzz by finding more unique lines of code on all the applications except WackoPicko. On WackoPicko, the tools found additional lines of code in the shopping cart functionality. By interweaving the crawling and fuzzing, the tools were able to induce a new state in the shopping cart that exposed an otherwise hidden URL.

Vulnerability Target Bias. In the next evaluation, we tested whether different vulnerability targets may introduce result altering bias into the evaluation that unfairly benefited Witcher in the prior code coverage evaluation. Black Widow and WebFuzz target XSS vulnerabilities; whereas, Witcher targets SQL and command injection vulnerabilities. Black Widow and WebFuzz form valid pre-defined XSS payloads to detect an XSS vulnerability. Although Witcher does not generate an exploit payload—it inputs random bytes and swaps variables that will likely trigger a fault escalation—we cannot guarantee that implicit command and SQL injection driven assumptions did not influence Witcher. As a result, these different payloads and assumptions may introduce coverage bias, which makes the comparison between the tools less equivalent.

To evaluate the bias, we took an approach inspired by the comparison in Enemy of the State [20] where they added the w3af testing component to the state-aware-crawler to control for the vulnerability detection. In our evaluation, we simulated the same control by combining Black Widow's and WebFuzz's code coverage with the code coverage generated by BurpPlus (with only SQL auditing enabled and loading Burp with Witcher's URLs).

Combining BurpPlus's SQL only results with Black Widow and WebFuzz did not alter the outcome of any comparison to Witcher making it less likely that the different vulnerability targets unfairly benefited Witcher. In Table 5, BurpPlus added lines covered to most of the web applications for both scanners. However, the additional lines did not change the outcome of the comparisons; moreover, the percent increase (amount of change / total lines covered) was less than 3.3% for all the applications (see Table 11). Thus, for the chosen web applications, it is unlikely that a vulnerability target bias impacted the results.

Performance. To understand the cost in terms of requests per second, we compared the number of requests per second made by Witcher, WebFuzz, Black Widow, and Burp. In the evaluation, we executed Witcher (one core), Burp (max of ten concurrent requests), WebFuzz (a single worker), and Black Widow (one core) for eight hours on each of the PHP web applications.

Table 9 in the Appendix shows that Witcher sent the most requests per second for every web application. Witcher averaged 142.1 requests per second while WebFuzz averaged 22.5 req/s, Black Widow averaged 1.6 req/s, and Burp averaged 6.8 req/s. Although Witcher outperformed the other tools in code coverage, it issued six times the requests made by the next fastest tool; however, the coverage was not six times better. Thus, by applying a hybrid approach Witcher would likely improve coverage despite the potential cost to the requests per second.

6. Discussion

Witcher's use of fault escalation, dynamic request crawling, request harnessing, direct instrumentation, and HTTP-specific input mutations provides a framework for the effective application of coverage-guided mutational fuzzing to web applications. Our evaluation showed the effectiveness of the Witcher components, the ability to identify known and previously unknown vulnerabilities. In addition, we compared Witcher with Burp, Black Widow, and WebFuzz. Witcher outperformed the other tools, finding more vulnerabilities than Burp and covering more of the web applications than Burp, Black Widow, and WebFuzz.

Grey-box versus White-box versus Black-box Scanners. The implementation of grey-box fuzzing for web applications requires less effort to implement than its white-box counterparts. Witcher requires inserting a few lines of code into the target runtime for the language. However, a white-box tool models

TABLE 4: Results of PHP lines of code coverage between Witcher, Burp, BurpPlus, Black Widow, and WebFuzz. Each scanner is compared against Witcher. The $W\setminus B$ column shows the unique lines discovered by Witcher. The $W\cap B$ shows the lines found by Witcher and the other scanner. The $B\setminus W$ column shows the unique lines found by the other tool. If Witcher has the most unique lines, the value is ***green****. If the other tool has the most unique lines then the value is in* ***orange*** *.*

Application	Burp (solo)			BurpPlus Witcher			Black Widow			WebFuzz		
	$W\setminus B$	$W\cap B$	$B\setminus W$	$W\setminus B$	$W\cap B$	$B\setminus W$	$W\setminus B$	$W\cap B$	$B\setminus W$	$W\setminus B$	$W\cap B$	$B\setminus W$
Doctor Appt. Sys.	**34**	386	6	**34**	386	13	**209**	211	43	**74**	1,067	10
Hospital Mgmt	**971**	471	8	**1,021**	421	8	**92**	1,350	24	**164**	3,162	54
Login Mgmt	**104**	64	0	**53**	115	3	**37**	131	4	**27**	492	0
OpenEMR	**32,878**	7,859	7	**31,428**	9,309	40	**25,273**	15,464	2583	**25,237**	107,917	17,852
osCommerce	**5,733**	4,024	90	**3,890**	5,867	277	**2,657**	7,100	798	**4,147**	22,635	3,622
phpBB	**3,148**	22,183	851	**14,482**	10,849	1001	**3,210**	22,121	879	**15,483**	35,480	6,544
rConfig	**2,960**	592	15	**2,263**	1,289	15	**458**	3,094	239	**301**	2,259	30
WackoPicko	**343**	399	10	**258**	484	10	72	670	**283**	50	2,415	**392**
Wordpress	**37,308**	15,987	50	**27,823**	25,472	1723	**7,036**	46,259	6482	**41,239**	109,076	5,935

TABLE 5: This table compares the increase in code coverage introduced by combining BurpPlus's code coverage data to Black Widow's and WebFuzz's code coverage. It shows the unique lines found by Witcher and the scanner in the first two columns. In the third column, it shows the increase in the scanner's coverage over the results show in Table 4. A more detailed table is available in the Appendix at Table 11

Application	Witcher v. Black Widow			Witcher v. WebFuzz		
	$W\setminus BW+$	$(BW+)\setminus W$	Inc.	$W\setminus WF+$	$(WF+)\setminus W$	Inc.
Doctor Appt.	**206**	43	0	**74**	10	0
Hosp. Mgmt.	**88**	24	0	**71**	60	6
Login	**19**	7	3	**3**	0	0
OpenEMR	**25,117**	2,606	23	**22,473**	19,915	2,063
OSCommerce	**2,497**	863	65	**3,833**	3,622	0
phpBB	**2,967**	1,159	280	**15,133**	7,508	964
rConfig	**426**	254	15	**104**	62	32
WackoPicko	66	**285**	2	50	**407**	15
Wordpress	**6,966**	6,733	251	**37,401**	10,365	4,430

the semantics of the language. The semantics differ for each language; thus, significant effort is required to initially implement a semantic-driven white-box approach to a different language. For instance, Pixy [44] did not support object-oriented features of PHP, which limits its applicability to modern PHP. Moreover, white-box tools often fail when analyzing real-world code. However, we tested Witcher and grey-box fuzzing using multiple real-world targets, languages, and architectures. Although black-box vulnerability scanners are typically language agnostic, grey-box fuzzers out-performed a commercial black-box tool and two state-of-the-art vulnerability scanners.

6.1. Limitations

The current Witcher prototype is limited to discovering SQL injection and command injection vulnerabilities. While these two vulnerability classes represent high-severity vulnerabilities, there are other web vulnerabilities such as cross-site scripting, path traversal, local file inclusion, or remote code evaluation that Witcher does not currently detect.

Another limitation of Witcher is that it can only detect *reflected* injection vulnerabilities—that is, injection vulnerabilities where the untrusted user input flows unsanitized to a sensitive sink during one HTTP request. This is in contrast to second-order vulnerabilities, such as stored SQL injection, where the untrusted user input is *safely* stored by the web application on the initial HTTP request, where it finally flows unsanitized to a sensitive sink while processing a subsequent HTTP request. Although Witcher might be able to detect the vulnerability using Fault Escalator, it would not be able to reason about what input actually caused the vulnerability.

A related limitation is that Witcher does not reason about web application state. A key limitation of the Witcher prototype is that it fuzzes one URL at a time, which does not allow it to reason about or understand multi-state actions in the web application. However, Witcher is able to induce some application states between requests because the web application's database maintains state. Perhaps the techniques proposed in prior work to understand web application state [20], [14] could be applied to Witcher.

6.2. Future Work

While Witcher worked well in the evaluations, we see several potential improvements. Witcher could benefit more automation of the initial setup and configuration. Witcher would also benefit from simultaneous crawling and fuzzing that shares results and interweaves the execution of different URLs.

Witcher can be improved to detect other types of vulnerabilities. Witcher could be augmented to detect local file inclusion and path traversal vulnerabilities by (1) creating a honeypot directory (witchers-honey/) in each directory of the web application and (2) adding a detector that escalates when a new file is detected in the honeypot directory. Witcher could include XSS vulnerability detection likely at the cost of performance by using a JavaScript engine to render and detect the XSS using WebFuzz's technique.

7. Related Work

Recently, three grey-box fuzzing tools have emerged in the literature BackREST [9], WebFuzz [10], and Cefuzz [45]. Witcher is distinguishable from the tools because Witcher supports multiple languages, while BackREST only supports Node.js and WebFuzz and Cefuzz only support PHP. With respect to BackREST, Witcher is more robust because Witcher handles full web applications whereas BackREST focuses on exercising REST APIs. In addition, Witcher is open source; however, BackREST and Cefuzz are closed source and BackREST is unavailable for testing or evaluation. Witcher also differs from WebFuzz because Witcher uses a compiled fuzzer to generate inputs whereas WebFuzz's fuzzer is written in Python, which improves

the requests per second Witcher can make. Next, Witcher tracks execution by adding the instrumentation to the interpreter; whereas, WebFuzz directly modifies the web application's scripts and nearly doubles the size of the scripts. Lastly, Witcher targets command and SQL injection vulnerabilities whereas BackREST and WebFuzz target Cross-site Scripting vulnerabilities and Cefuzz targets remote code execution vulnerabilities.

Several black-box fuzzers exists for fuzzing web applications such as Burp [5], Acunetix Web Vulnerability Scanner [6], IBM AppScan [7], OWASP Zap [8], and Skipfish [41] Arachni [39], Enemy of the State [40], jÄk [42], w3af [43], and Black Widow [14]. Each the the scanners detect injection and other common web vulnerabilities [46]. Most of the tools "fuzz" using predefined heuristics or user-defined rules [47]. However, unlike the black-box tools, Witcher relies execution instrumentation to guide the input generation and fault escalation to detect an injection vulnerability.

AFL CGI Wrapper enables the fuzzing of CGI binaries by receiving input via standard input and translating it into a CGI request [48]. Although the initial inspiration of Witcher's CGI harness came from the AFL CGI Wrapper, it only detects memory corruption vulnerabilities does not identify injection vulnerabilities and it lacks the input generation capabilities of Witcher.

The tool μ4SQLi automatically produces inputs that lead to harmful SQL statements and bypasses application firewalls [49]. μ4SQLi starts with legitimate input and mutates the values using a predefined group of mutation operators that are meant to build new types of SQL injection payloads. Similar to Witcher, μ4SQLi uses a database proxy to monitor the network traffic between the database and the web server so that it detects whether the SQL statement is harmful. μ4SQLi differs from Witcher because it does not rely on any execution instrumentation to guide input generation and it does not detect command injection vulnerabilities.

KameleonFuzz is a black-box web application fuzzer that detects XSS vulnerabilities [50]. KameleonFuzz attempts to uses an attack grammar and variable mutations to generate XSS payloads along with valid input, which it then submits to the site, and then detects whether the payload landed. It guides the mutations based on a fit score, which is calculated after the input is submitted. KamelonFuzz differs from Witcher because it does not use any execution information to guide the input generation phase and it does not use detect SQL or command injection vulnerabilities.

RESTler and Pythia automatically test REST APIs that have interfaces defined using Sparrow. RESTler uses grammar-based fuzzing and static analysis of API specifications to automatically test REST APIs [47]. Pythia, which builds on RESTler, adds coverage-guided fuzzing and learning-based mutations [51]. However, these tools focus bugs instead of vulnerabilities and only work on Sparrow documented REST apis, unlike Witcher, which works on web applications and detects SQL and command injection vulnerabilities.

Another recent tool, HYDRA, uses user weighting and output monitoring to guide the targeted generation of injection payloads [52]. HYDRA uses context changes to detect the injection vulnerabilities. However, Witcher uses fault escalation to detect injection vulnerabilities. In addition, HYDRA requires more interaction with the user for initialization of inputs and context weighting.

SRFuzzer fuzzes the web interface of router-based IoT devices to detect memory corruption and command injection vulnerabilities. SRFuzzer uses a browser-based crawler to gather the HTTP variables. SRFuzzer then fuzzes the variables and monitors by testing the device's responsiveness and listening for reverse connections made by crafted command injection payloads.

Although not the primary contribution of Witcher, its device fuzzing offers features not included in SRFuzzer. SRFuzzer and Witcher use similar techniques to identify the HTTP variables. However, Witcher detects the vulnerability from inside the device (which means that the vulnerability trigger does not need to be a syntactically correct command injection). Moreover, SRFuzzer's scaling is limited by the use of the physical device; while Witcher uses an emulated version of the firmware and scales to fuzz in parallel. Finally, Witcher uses instrumentation of the binary to guide fuzzing, while SRFuzzer does not.

Rampart detects denial-of-service attacks using a PHP plugin to measure the execution time of user-created functions to detect anomalous execution performance, which indicates a denial-of-service attack [53]. Witcher's instruction instrumentation is more fine-grained than user-created functions because user-created functions often contain multiple lines of code. Thus, Witcher provides the fuzzer with additional information to guide its analysis than produced by Rampart.

Eriksson et al. created the vulnerability scanner Black Widow [14]. Black Widow uses navigation modeling, traversing, and inter-state dependencies to scan web applications. Witcher uses fault escalation to find SQL and command injection vulnerabilities, but Black Widow is limited to XSS vulnerabilities. Based on the evaluation results, the two approach while producing similar results also seem to activate different portions of the code. Thus, integrating the Witcher and Black Widow approaches will likely result in a more effective tool.

Several fuzzers exist that propose different methods for mutating context-free grammars or other types of structured data [54], [55], [56], [57], [58]. We plan to investigate their performance on web applications in future work.

8. Conclusion

In this paper, we propose Witcher, a novel web application vulnerability discovery platform that is generalizable to web languages without hard-coded heuristics for testing inputs. Witcher is inspired by coverage-guided mutational fuzzing. To bridge the gap between coverage-guided mutational fuzzing and web application vulnerabilities, we design multiple techniques in Witcher that generate both syntactically-valid and semantically-correct inputs and detect injection vulnerabilities. In our evaluation, we observed that Witcher is able to find both known and unknown web vulnerabilities effectively. Witcher is the first step toward the development of a web application *fuzzer*, as opposed to vulnerability scanners, and we believe this approach is a promising path forward to automatically identifying vulnerabilities in web applications.

Acknowledgements: This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541 and by the Defense Advanced Research Projects Agency (DARPA) under contracts HR001118C0060 and FA8750-19C-0003.

References

[1] 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, pp. 25–36.

[2] N. Jovanovic, C. Kruegel, and E. Kirda, "Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications," *Journal of Computer Security*, vol. 18, no. 5, pp. 861–907, 2010.

[3] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward Automated Detection of Logic Vulnerabilities in Web Applications," in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 143–160.

[4] 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 *Thirteenth International World Wide Web Conference Proceedings*, 2004, pp. 40–52.

[5] PortSwigger, "Burp Suite. Application Security Testing," https://portswigger.net/burp, 2020.

[6] "Acunetix Web Vulnerability Scanner," https://www.acunetix.com/, 2020.

[7] "IBM/HCL AppScan," https://www.hcltechsw.com/appscan, 2020.

[8] "OWASP Zed Attack Proxy," https://www.zaproxy.org/, 2020.

[9] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Backrest: A model-based feedback-driven greybox fuzzer for web applications," *arXiv preprint arXiv:2108.08455*, 2021.

[10] 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*. Springer, 2021, pp. 152–172.

[11] E. Bazzoli, C. Criscione, F. Maggi, and S. Zanero, "XSS PEEKER: Dissecting the XSS exploitation techniques and fuzzing mechanisms of blackbox web application scanners," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2016.

[12] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[13] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[14] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," *proceedings of IEEE SSP 2021*, 2021.

[15] "HTTP/1.1 Message Syntax and Routing," https://tools.ietf.org/html/rfc7230, 2020.

[16] "HTTP State Management Mechanism," https://tools.ietf.org/html/rfc3875, 2020.

[17] "Common Gateway Interface," https://en.wikipedia.org/wiki/Common_Gateway_Interface, 2020.

[18] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, vol. 16, 2016, pp. 1–16.

[19] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, November 2013.

[20] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the State: A State-Aware Black-Box Vulnerability Scanner," in *Proceedings of the 21st Symposium on USENIX Security*, Bellevue, WA, August 2012.

[21] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 111–131.

[22] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[23] B. Cui, F. Wang, Y. Hao, and X. Chen, "Whirlingfuzzwork: a taint-analysis-based api in-memory fuzzing framework," *Soft Computing*, vol. 21, no. 12, pp. 3401–3414, 2017.

[24] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 818–825.

[25] "American Fuzzy Lop," https://github.com/google/AFL, 2020.

[26] "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2020.

[27] "High-performance binary-only instrumentation for afl-fuzz," https://github.com/google/AFL/blob/master/qemu_mode/README.qemu, 2020.

[28] V. Gite, "What is Dash (/bin/dash) Shell?" https://www.cyberciti.biz/faq/debian-ubuntu-linux-binbash-vs-bindash-vs-binshshell/, 2020.

[29] "Puppeteer - Node.js library that provides high-level API access to Chrome and Chromium," https://github.com/puppeteer/puppeteer, 2021.

[30] "Gremlins - Monkey Testing Library for Web Apps and Node.js," https://github.com/marmelab/gremlins.js, 2020.

[31] "QEMU," https://qemu.org, 2020.

[32] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[33] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[34] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[35] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[36] Derick Rethans, "Xdebug: A Debugger and Profiling Tool for PHP," https://xdebug.org, 2020.

[37] "OWASP, SQL Injection," https://owasp.org/www-community/attacks/SQL_Injection#:~:text=TheseverityofSQLInjection,Injectionahighimpactseverity, 2022.

[38] R. Mohammed, "Assessment of web scanner tools," *International Journal of Computer Applications*, vol. 133, no. 5, pp. 1–4, 2016.

[39] "Arachni - Web Application Security Scanner Framework," https://www.arachni-scanner.com/, 2021.

[40] 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 ({USENIX} Security 12)*, 2012, pp. 523–538.

[41] "Skipfish: Web Application Security Scanner," https://github.com/spinkham/skipfish, 2020.

[42] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 295–316.

[43] "w3af - Open Source Web Application Security Scanner ," http://w3af.org/, 2021.

[44] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 6–pp.

[45] J. Zhao, Y. Lu, K. Zhu, Z. Chen, and H. Huang, "Cefuzz: An directed fuzzing framework for php rce vulnerability," *Electronics*, vol. 11, no. 5, p. 758, 2022.

[46] M. Qasaimeh, A. Shamlawi, and T. Khairallah, "Black box evaluation of web application scanners: Standards mapping approach," *Journal of Theoretical and Applied Information Technology*, vol. 22, 07 2018.

[47] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Rest-ler: automatic intelligent rest api fuzzing," *arXiv preprint arXiv:1806.09739*, 2018.

[48] Floyd Fuh, "AFL CGI Wrapper," https://github.com/floyd-fuh/afl-cgi-wrapper, 2020.

[49] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: an input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 259–269.

[50] 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, pp. 37–48.

[51] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.

[52] M. Leithner, B. Garn, and D. E. Simos, "Hydra: Feedback-driven black-box exploitation of injection vulnerabilities," *Information and Software Technology*, p. 106703, 2021.

[53] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 393–410.

[54] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[55] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[56] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[57] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021.

[58] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[59] Z. McGee and S. Acharya, "Security analysis of openemr," in *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2019, pp. 2655–2660.

[60] F. Akowuah, J. Lake, X. Yuan, E. Nuakoh, and H. Yu, "Testing the security vulnerabilities of openemr 4.1. 1: a case study," *Journal of Computing Sciences in Colleges*, vol. 30, no. 3, pp. 26–35, 2015.

[61] G. Deepa, P. S. Thilagam, A. Praseed, and A. R. Pais, "Detlogic: A black-box approach for detecting logic vulnerabilities in web applications," *Journal of Network and Computer Applications*, vol. 109, pp. 89–109, 2018.

[62] D. Esposito, M. Rennhard, L. Ruf, and A. Wagner, "Exploiting the potential of web application vulnerability scanning," in *ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22-26 July 2018*. IARIA, 2018, pp. 22–29.

[63] N. Khoury, P. Zavarsky, D. Lindskog, and R. Ruhl, "Testing and assessing web vulnerability scanners for persistent sql injection attacks," in *proceedings of the first international workshop on security and privacy preserving in e-societies*, 2011, pp. 12–18.

[64] X. Li and Y. Xue, "Block: a black-box approach for detection of state violation attacks towards web applications," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 247–256.

[65] "Google Search for Uses of Login System," https://tinyurl.com/doctorappointmentsystem, 2021.

[66] "Google Search for Uses of Login System," https://tinyurl.com/usermanagementsystem, 2021.

[67] "Google Search for Uses of Hospital Management System," https://tinyurl.com/hospitalmanagementsystemuses, 2021.

[68] S. Micheelsen and B. Thalmann, "A static analysis tool for detecting security vulnerabilities in python web applications," 2016.

[69] B. Jabiyev, O. Mirzaei, A. Kharraz, and E. Kirda, "Preventing server-side request forgery attacks," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1626–1635.

[70] G. Pellegrino and D. Balzarotti, "Toward black-box detection of logic flaws in web applications." in *NDSS*, 2014.

[71] S. Gupta and B. B. Gupta, "Php-sensor: a prototype method to discover workflow violation and xss vulnerabilities in php web applications," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[72] 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*. IEEE, 2010, pp. 332–345.

9. Appendix

TABLE 6: Known vulnerabilities and results from Witcher's evaluation.

Application Description	Name	Type	Results	Reason Missed
OpenEMR	CVE-2019-17197	SQL	Missed	Crawler Missed
	CVE-2019-14529	SQL	Missed	Crawler Missed
	CVE-2019-16404	SQL	Missed	Crawler Missed
	CVE-2018-17181	SQL	Missed	Crawler Missed
	CVE-2018-17179	SQL	Missed	Crawler Missed
WackoPicko	login.php	SQL	Found	
	passcheck.php	Command	Found	
	similar.php	Stored SQL	Missed	Failed to Recall
Doctor Appt.	CVE-2020-29283	SQL	Found	
Login Mgmt.	CVE-2020-25952	SQL	Found	
rConfig	CVE-2019-16662	Command	Missed	
	CVE-2019-16663	Command	Missed	Crawler Missed
Hosp. Mgmt.	CVE-2020-5192	SQL	Found	
D-Link 825	CVE-2020-10213	Command	Missed	Crawler Missed
D-Link 823G	CVE-2018-17787	Command	Found	
D-Link 823G	CVE-2019-15530	Command	Missed	Did not trigger
D-Link 645	CVE-2015-2051	Command	Missed	Did not trigger
Tenda AC9	CVE-2018-16334	Command	Missed	Crawler Missed
Juice Shop	login	SQL	Found	
	search	SQL	Found	
WebGoat	attack2	SQL	Found	
	attack3	SQL	Found	
	attack4	SQL	Found	
	attack5a	SQL	Found	
	attack5b	SQL	Missed	Java Inst Bug
	attack8	SQL	Found	
	attack9	SQL	Found	
	attack10	SQL	Missed	Java Inst Bug
	Adv/attack6a	SQL	Found	
	Adv/challenge	SQL	Missed	PUT not Supported

TABLE 7: OpenEMR Results. This table shows the lines of code covered and the vulnerabilities discovered for each of the five trials.

	AFLHR		WiCR		WiCHR	
	Lines	Vulns	Lines	Vulns	Lines	Vulns
Trial 1	23,113	2	29,723	7	30,714	8
Trial 2	23,142	2	29,082	5	30,777	8
Trial 3	23,011	1	29,543	6	30,935	9
Trial 4	23,111	2	29,105	6	30,833	8
Trial 5	23,220	3	29,160	6	30,800	8

TABLE 8: This table shows the difference between code triggered by Witcher and WebFuzz + BurpPlus (with SQL auditing enabled). Although it did change the results by as much as 4,000 lines of code, the change did not change the outcome of the comparison.

Application	$W \setminus (WF+)$	$W \cap (WF+)$	$(WF+) \setminus W$	Inc.	% Inc.
Doctor Appt. Sys.	74	1067	10	0	0.0%
Hosp Mgmt.	71	3,255	60	6	0.2%
Login Mgmt.	3	516	0	0	0.0%
OpenEMR	22,473	109,511	19,915	2,063	1.4%
OSCommerce	3,833	22,608	3,622	0	0.0%
phpBB	15,133	35,730	7,508	964	1.7%
rConfig	104	2,456	62	32	1.2%
WackoPicko	50	2,415	407	15	0.5%
Wordpress	37,401	87,610	10,365	4,430	3.3%

TABLE 9: The requests per second of Witcher and WebFuzz on the PHP applications used in the evaluation.

Applications	Witcher Req/s	WebFuzz Req/s	Black Widow Req/s	Burp Req/s
Doctor Appt. Sys.	539.4	43.4	3.4	7.3
Hosp. Mgmt. Sys.	327.4	26.6	3.0	5.0
Login Mgmt.	180.9	112.2	0.9	13.3
OpenEMR	15.3	1.5	1.7	5.1
osCommerce	22.2	4.7	0.7	2.4
phpBB	14.7	1.5	0.7	1.1
rConfig	52.7	10.1	3.3	3.3
WackoPicko	101.9	2.2	0.2	23.5
Wordpress	24.4	0.1	0.5	0.1
Average	142.1	22.5	1.6	6.8

Listing 1: Example PHP code with three SQL injections that the commercial black-box scanner Burp does not find.

```
1  ...
2  <? $pid = $_GET['pid']; $act = $_GET["act"]; ?>
3  <input name="pid" value="<?= $pid ?>">
4  <input name="pname" value="<?=get_name($pid)?>">
5  <select name="ptype">
6  <option value="dog_red">Red Dog</option>
7  <option value="dog_grey">Grey Dog</option>
8  </select>
9  <input type="hidden" name="act" value="a"/>
10 <?php
11 $pname = $_GET["pname"];
12 $inp = explode('_', $_GET["ptype"]);
13 $tab=$inp[0]; $c = $inp[1];
14 $pid = isset($pid) ? $pid : uniqid();
15 if (count($inp) >= 2 && $act == "a") {
16   $pid   = $conn->real_escape_string($pid);
17   $pname = $conn->real_escape_string($pname);
18   $c = $conn->real_escape_string($c);
19   $sql  = "INSERT into {$tab} (id, pname, color)";
20   $sql .= " VALUES ('{$pid}','{$pname}','{$c}')";
21   $ret  = mysqli_query($conn, $sql);
22 } else if (count($inp) >= 2 && $act == "u"){ //TBD
23   if (get_name($pid) != null){
24     $sql = "UPDATE dog SET color= '{$c}' ";
25     $sql .= "WHERE id = '{$pid}'";
26     $ret = mysqli_query($conn, $sql);
27         ...
```

Listing 2: Excerpt from Post-2 in the microtest. The code is simliar for Post-5, Post-10, Cookie-5, and Get-5 scripts.

```
1  ...
2    if(isset($_POST['nv1'])) {
3      if(isset($_POST['nv2'])) {
4        $ret=mysqli_query($con,"SELECT * FROM tbl
5                         WHERE ID='$_GET['vul']'");
6  ...
```

Listing 3: Excerpt from Equals-1 microtest.

```
1  ...
2  if($_GET['nv1'] == "YYYY") {
3    $ret=mysqli_query($con,"SELECT * FROM tbl
4                     WHERE ID='$_GET['vul']'");
5  ...
```

Listing 4: Excerpt from Loop microtest.

```
1  ...
2  for($i=0; $i < strlen($teststr); $i++){
3    if ($i < $nv1_len){
4      if ($teststr[$i] == $nv1[$i]){
5      } else {
6        $all_match = FALSE;
7          break;
8      }
```

```
9      } else {
10         $all_match = FALSE;
11         break;
12     }
13  }
14  if ($all_match){
15     $ret=mysqli_query($con,"SELECT * FROM tbl
16                            WHERE ID='$_GET['vul']'");
17  } ...
```

Listing 5: Excerpt from FindVar microtest.

```
1  ...
2  if(isset($_POST['ao3'])) {
3     if($_POST['ao3'] == "add") {
4        $ret=mysqli_query($con,"SELECT * FROM tbl
5                               WHERE ID='$_GET['vul']'");
6  ...
```

TABLE 10: Web applications used in the evaluation.

Application	Lang. or Platform	Release Date	Ver.	GitHub Stars	Google Results	Lines of Code	Prior Research
OpenEMR	PHP 7	2018	5.0.1.7	1.6k ★		9,443	[59], [60]
WackoPicko	PHP 5	2018	1.0	265 ★		2,510	[14], [61], [40], [62], [63], [64]
Doctor Appointment Booking System	PHP 7	2020	1.0	n/a	≈10 [65]	3,981	-
User Login Management System	PHP 7	2020	2.1	n/a	≈3 [66]	1,490	-
rConfig	PHP 7	2018	3.9.2	80 ★		48,405	-
Hospital Management System	PHP 7	2019	4.0	n/a	≈100 [67]	9,443	-
D-Link DIR-823G	C/MIPSEL	2018	1.0.3.B3	n/a		1,585,157	-
D-Link DIR-823G	C/MIPSEL	2018	1.0.2.B5	n/a		1,569,829	-
D-Link DIR-645	C/ARM	2014	1.0.4.B12	n/a		465,324	-
D-Link DIR-825	C/MIPSEB	2015	1.2.10.B1	n/a		542,992	-
Tenda AC9	C/ARM	2018	15.03.05.19	n/a		982,880	-
WebGoat	Java	2020	8.10	3.9k ★		14,761	[50]
FlaskBB	Python	2018	2.0.2	2.0k ★		14,534	[68]
Juice Shop	Node.js	2020	8.1.0	242 ★		26,221	[62], [69]
Thredded	Ruby/Rails	2021	16.16	1.3k ★		4,426	-
phpBB	PHP 7	2021	3.3.3	1.4k ★		318,104	[42], [14], [40], [50]
osCommerce	PHP 7	2017	2.3.4.1	272 ★		44,355	[70], [14], [61], [71], [64]
Wordpress	PHP 7	2021	5.7.1	15k ★		253,183	[42], [14], [40], [50], [72]

TABLE 11: This table shows the difference between the code triggered by Witcher and Black Widow + BurpPlus (with SQL auditing enabled). Although it did change the results by as much as 280 lines of code, none of the additions changed the outcome of the comparison.

Application	Witcher v. Black Widow					Witcher v. WebFuzz				
	$W \setminus BW+$	$W \cap BW+$	$(BW+) \setminus W$	Inc.	% Inc.	$W \setminus WF+$	$W \cap BW+$	$(WF+) \setminus W$	Inc.	% Inc.
Doctor Appt.	**206**	258	43	0	0.00%	**74**	1,067	10	0	0.00%
Hosp. Mgmt.	**88**	1,386	24	0	0.00%	**71**	3,255	60	6	0.18%
Login	**19**	149	7	3	1.71%	**3**	516	0	0	0.00%
OpenEMR	**25,117**	77,499	2606	23	0.02%	**22,473**	109,511	19915	2063	1.36%
OSCommerce	**2,497**	8,947	863	65	0.53%	**3,833**	22,608	3622	0	0.00%
phpBB	**2,967**	38,529	1159	280	0.66%	**15,133**	35,730	7508	964	1.65%
rConfig	**426**	6,206	254	15	0.22%	**104**	2,456	62	32	1.22%
WackoPicko	66	676	**285**	2	0.19%	50	2,415	**407**	15	0.52%
Wordpress	**6,966**	71,927	6733	251	0.29%	**37,401**	87,610	10365	4430	3.27%