Thesis Dissertation

AN INSTRUMENTATION APPROACH TO WEB FUZZING

Orpheas van Rooij

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Jan 2020

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

An Instrumentation Approach to Web Fuzzing

Orpheas van Rooij

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

Jan 2020

Acknowledgements

First and foremost, I would like to thank my supervisor Dr. Elias Athanasopoulos for his guidance and support throughout this project, and for encouraging us to strive higher. To my fellow project members Marcos Charalambous and Demetris Kaizer, even though we had some disagreements, thank you for your help and contribution. This experience has taught me valuable lessons concerning web security, application development, and team-work skills.

I would like to also thank my professors here in the University of Cyprus that have taught me invaluable knowledge and have further sparked my interest in the field of computer science.

Lastly, a big thank you to my family and friends that have provided love and support throughout this journey.

Abstract

The advent of web applications has been the key driving force for the success of the web and internet as a whole. Numerous security critical web applications such as Internet banking sites are available making the need for robust security tools that can uncover vulnerabilities in them evermore important.

Web Fuzzers are a de facto tool for finding web vulnerabilities but to the best of our knowledge, coverage-based web fuzzers have yet to be developed regardless of their notable successes in native application fuzzing.

In this thesis we introduce a library for instrumenting web applications to enhance the performance of Web fuzzers and provide the first coverage-based grey-boxed fuzzer named *webFuzz* developed specifically for detecting Reflective Cross Site Scripting (RXSS) bugs. With the feedback received from the instrumented applications, *webFuzz* can cleverly choose its next fuzzing input based on the previous coverage observed and thus fuzz more of its intended logic.

We evaluate our solution in terms of instrumentation overhead, code coverage, throughput and vulnerability detection. We show that the instrumentation can provide a significant enhancement to web fuzzers, as *webFuzz* can successfully discover RXSS bugs faster than the prominent black-box fuzzer *wFuzz*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the advent of the World Wide Web in 1989, the internet has observed an exponential growth in its users, with 2019 seeing around 6,700 new users (new broadband subscriptions) every hour [46]. Web applications have been a key driving force for this success, as through their ability to generate dynamic content they can provide live finance management, over-the-globe communication, interactive media and many more capabilities.

Web applications usually follow a multi-tiered client-server architecture that incorporates a web-server for serving static files, an application server to generate dynamic content and a back-end database system for data management and access. They posses an extremely heterogeneous nature as they can be executed in different web servers and browsers, and are made up of various components each possibly developed in a different programming language and model [14].

All though the heterogeneous nature of them makes vulnerability testing a challenging task, the overwhelmingly large number of web users makes the task ever-more important. The severities of web vulnerabilities have been realised by both commercial and governmental bodies and their remedies have been incorporated into major standards such as Health Insurance Portability and Accountability Act (HIPAA) and Payment Card Industry Data Security Standard (PCIDSS) [8].

Despite this fact, the research community has given comparatively less attention to web security tools, especially in the area of fuzzing. Currently, most if not all security tools that target web applications only exist in two flavors namely black- and white-box approaches.

Black-box fuzzers are unaware of the internal program structure, that is, their target is a black-box, no feedback other than what is directly observable is provided. One of their main advantages is their low overhead which allows them to exercise the Program Under Test (PUT) with millions of inputs. In this way their chances of triggering a bug increases. On the other hand, their lack of knowledge on the program's structure comes with a cost. *AFL* Fuzzer has shown that its feedback loop that uses previously generated interesting inputs to built new test cases from them is a key idea for successful bug discovery [6, 52], Black-box fuzzers though lack the ability to make sound judgements on what is considered interesting input [27, 41]. As a result they either do not retain generated inputs for further mutation [21, 24, 37] or the heuristic used to identify favorable inputs is insufficient as it can only rely on what is observable in the response of the PUT [16]. This limits the effectiveness of black-box solutions.

On the other side of the scale exist the white-box solutions that require access to source code and rely heavily on static-analysis. Most of these approaches will utilize constraint-solving and a combination of symbolic and concolic execution [3, 4] to identify vulnerable code statements. All though sophisticated in their approach, their inherent limitation lies in the computationally demanding constraint solver. For instance, [3, 4], utilise static analysis to perform a mapping between source variables such as URL parameters to sink statements, that is, server-side code statements (such as the `print` command) that output the source back to the client. Creating this source-sink pair link and identifying whether sanitisation happens is the key ingredient for exposing a vulnerability. For each associated pair to be created though, lies an expensive constraint solving operation, and the magnitude of this problem only increases with the number of sources and sinks. To get a scale on this problem, *AflGO*, a directed grey-box fuzzer, can uncover the *Heartbleed* vulnerability within less than 20 minutes. A white-box fuzzer that employs constraint solving such as the tool *Katch* struggles to identify the bug even after 24 hours of trying [11].

Coverage-based grey box fuzzing comes with an ideal comprise between sophistication and scalability. Instead of statically analysing the source code, it relies on input mutations and lightweight coverage feedback to explore the restricted input-space within a limited time frame. Given that the input-space can be enormous, possibly infinite, a fuzzer can leverage the instrumentation feedback to deduce if an input is interesting and thus built upon it new test cases. Ways of defining an interesting input can be if it explores new unobserved code paths or if it exercises the business logic of the application and does not fail early on in the input-format checks. In this way, it maximises code coverage, whether that it on a global or function level [11], and thus increasing the chances of triggering a vulnerability.

Notable fuzzers such as *AFL* [52] and *libfuzz* [42] have pioneered the technique of coverage-based grey-box fuzzing in native applications, with both of them having found thousands of vulnerabilities in security critical projects such as *PHP, Python, Linux Kernel, Mozilla Firefox, OpenSSL, OpenSSH, Nginx* [12, 51].

In this thesis, we design, implement and evaluate a coverage-based fuzzing solution for web applications aimed at detecting Reflective Cross-Site scripting vulnerabilities. Named *webFuzz*, it features input-mutation, coverage-aware request ranking and an in-built crawler. Additionally we have developed *webInstr*, the instrumentation tool counterpart that statically augments the program under test (PUT) with stub code that provides to *webFuzz* coverage statistics about the current request. This thesis focuses on the instrumentation side of our approach, while still analysing the core concepts and the evaluation of *webFuzz*.

Our main contributions are:

1. We design and implement an instrumentation library named *webInstr*. Instrumentation is applied on the AST level of *PHP-based* web applications. We provide *3* instrumentation policies, with different granularity and overheads. Our library is designed to be easily extendable for custom use-cases.

2. We design and implement the first coverage based fuzzer designed to detect Cross-Site scripting vulnerabilities. We further evaluate *webFuzz* in terms of coverage, throughput and efficiency in finding unknown bugs. *webFuzz* manages to explore 27% and 21.5% of the entire *Drupal* and *WordPress* code respectively, which both consist of over half a million source lines of code (SLOC). To further evaluate the capabilities of webFuzz we compare our results with the prominent web-application fuzzer, *wFuzz*. In terms of vulnerabilities detected, it finds the most RXSS vulnerabilities (31 with *wFuzz* having found 29) for a fuzzing session that lasts 65 hours.

3. To foster further research in the field, we release all of our contributions, namely the toolchain for instrumenting PHP applications and the actual fuzzer as open source.

The remainder of this thesis is structured as follows. Chapter 2 discusses the core concepts that will be used throughout the thesis. Chapter 3 analyses the instrumentation policies and mechanisms of *webInstr*, elaborates on how instrumentation is leveraged by the fuzzer and finally discusses the mutation and vulnerability detection mechanisms of *webFuzz*. Chapter 4 follows, in which it dives deeper into the implementation details of *webInstr* and on the main design choices of *webFuzz*. The following Chapter 5 covers the evaluation of our two tools, and lastly related and future work concerning both tools is discussed.

Chapter 2

Background

In order to have a detailed understanding of this thesis, certain key background information must be discussed. In this section we firstly discuss the methods for defining code coverage, we then describe the different approaches to fuzzing and lastly give a trivial example showing an RXSS vulnerability which is the main type of web vulnerability that our web Fuzzer *webFuzz* aims to detect. References for the reader to dive further into these topics are also provided.

2.1 Basic Blocks and Flow Graphs

It is helpful to split the code to basic blocks and create flow graphs from them that visualise the possible control flows in the program. As shown below, a basic block has two requirements [2]. Simply put, it is the maximal block of consecutive statements that are always executed together in one batch. There exists a simple algorithm to identify leader statements which mark the beginning of a new basic block and the end of the preceding block [2]. The three criteria to identify such statements are also stated below.

Requirements of a basic block

The two requirements for a set of statements to be considered a basic-block.

1. The entry point of the basic block is only through its first statement (the leader statement). There can not be any jump targets that are in the middle of a block.
2. The exit point of the block is the last statement in it. That is, as soon as the leader statement is executed, all the other statements in the block are guaranteed to be executed sequentially as well without the control halting or branching elsewhere in the code (except in the presence of an exception where control is transferred to an

exception handler or the program aborts. For simplicity purposes this scenario is not taken into account).

Identifying leader statements

There are three rules to determine leader instructions and these are stated below.

1. The first statement to be executed in a program is a leader.
2. Any statement that immediately follows a conditional or unconditional jump is a leader. That is, the first statement inside a control statement's body is a leader.
3. Any statement that is the target of a conditional or unconditional jump is a leader. This can be the first statement following the end of a control statement's body, but can also be the conditional expression that checks if control should continue inside or after the loop's body.

Control Flow Graphs

A Control Flow Graph (CFG), is a directed, possibly cyclic graph where all the nodes are basic blocks and all the edges are the possible transitions between the basic blocks. Two pseudo-nodes are also added that indicate the entry and exit node of the program. Every basic block that the program can start with has an edge with the Entry node, and every basic block that can terminate the program has an edge with the Exit node. Control flow graphs are useful in optimising the register allocation process during code generation [2], in probe-pruning for instrumentation optimisation (Section 7) and in many more use-cases.

2.2 Coverage Criteria

There are different kinds of instrumentation policies with each providing different measures on the code coverage of an execution. They can be quantified according to whether they measure basic blocks, control flow edges, execution paths and more. In this section we describe their differences. We also note that our instrumentation tool *webInstr* can provide both Node, Edge and Path coverage feedback.

Basic Block Coverage

Basic Block coverage also known as Node (nodes in a CFG), statement or line coverage, measures the amount of unique blocks executed during an execution [5]. Since basic blocks are non-branching sequences of code, it is equivalent to measuring the total unique

Definition 1 An arbitrary function `foo`.

```
function FOO($x, y$)
    $numerator \leftarrow x$                                    ▷ Block A
    $denominator \leftarrow y$                                  ▷ Block A
    if $numerator < denominator$ then                           ▷ Block A
        $numerator \leftarrow y$                                ▷ Block B
        $denominator \leftarrow x$                              ▷ Block B
    end if
    return $numerator/denominator$                              ▷ Block C
end function
```

statements executed. This approach provides the least granularity in comparison with the other methods described below as the ordering of the blocks executed is not accounted for. For instance, in Definition 1, an execution of `foo(2,4)` is sufficient for a 100% coverage score as Block A, B and C are executed. The execution path that flows from block A directly to block C (not taken case) is not accounted though.

Branch Coverage

Similarly in Branch Coverage, which is also called Edge coverage, to reach complete coverage all the edges in the CFG of an application must be visited. This means that every possible sequence of two consecutive blocks must be executed, meaning all branch decisions must be exercised [5,14]. For this reason, in Definition 1 we need an execution such as `foo(4,2)` in addition to `foo(2,4)` to reach 100% as it takes into account the Not Taken case. Complete Branch coverage also implies complete Basic Block coverage, as to visit all the edges of a graph one must visit all its nodes as well.

Path Coverage

Path Coverage measures the number of unique paths traversed in the CFG out of all possible paths. Since cyclic graphs can produce infinitely many paths, this method is not very applicable unless a variant of it that poses additional restrictions is used such as Specified Path Coverage and Prime Path Coverage [5]. Nevertheless determining the CFG path traversed by an execution can be useful on its own, even without knowledge of the upper limit (total paths).

Other Criteria

There are other more advanced methods to measure code coverage which will not be analysed in this thesis. We nevertheless briefly state some of them here for the sake of completeness.

Conditional Coverage provides stronger guarantees than Edge coverage as it requires that every possible combination of truth values for the clause in every conditional branch is evaluated.

Data-Flow Coverage measures how many of the data-flow paths between each definition and use of a variable are traversed. More specifically, it measures for each definition of a variable x, how many of the possible paths from the definition to a use statement that accesses the value of x were executed. It is further divided to more stringent definitions such as *all def-use*, *all uses*, *all defs* and more [14].

2.3 Fuzzing

Fuzzing is a software testing technique that relies on bombarding a program under test (PUT) with a large number of malicious or unexpected inputs, generated in some automated fashion, in hopes of triggering an underlying bug. The bugs found can have security implications, degrade the quality of a service, open the gate for denial of service attacks or perform any other undesired behavior [44]. In this thesis we use fuzzing in the scope of finding security critical flaws, more specifically in detecting RXSS bugs in web applications.

The input generation technique can be generation based, or mutation based. Generation based approaches use a provided attack-grammar that specifies the general payload format for triggering a specific vulnerability [16], or use a file-format structure grammar to guide the input generation process. On the other hand, mutation-based approaches start form a provided initial-seed (e.g. default application inputs) and perform a series of mutations like insertion of random characters or syntax-related tokens to create new test cases. Both approaches can use previously generated inputs to further built upon them new test cases. Hybrid approaches that perform mutation-based input generation with the constraints of adhering to a known grammar also exist [27].

Fuzzing is further characterised in relation to its awareness of the program structure into black-, grey- (coverage-guided), white-box fuzzers.

2.4 Reflective Cross-Site Scripting

In the OWASP Top 10 web vulnerabilities list that represents a consensus of the most critical security risks in web applications [26], Cross-Site Scripting (XSS) is listed as number 7.

XSS flaws occur when an application includes untrusted input data in its HTML responses without validating or escaping them first. As a result these untrusted data can get executed which can in turn hijack the browser, deface the web site, redirect the user to dangerous sites and many other attacks. Some XSS types include Reflected (aka Non-Persistent or Type II), Stored (Persistent or Type I) and DOM-based(Type-0) [19].

Reflected XSS (RXSS) vulnerabilities arise when input data from a request is reflected back to the application's immediate response whereas in Stored XSS vulnerabilities the malicious payload is firstly stored in an intermediate repository (such a database) and is only later reflected by another request.

As an example to better grasp what a server-side RXSS bug typically looks like, let us assume a web application provides a login page with two input fields: username and password. A naive server-side PHP implementation for this is shown in Listing 2.1.

```
1   <?php
2   $username = $_POST['username'];
3   $passwd = $_POST['password'];
4   $user = search_username_in_db($username);
5
6   if (! $user) {
7       echo 'Error' . $username . 'was␣not␣found.';
8   } else {
9       if (match_password($user, $passwd)) {
10          redirect_to_app($user);
11      } else {
12          echo 'Wrong␣Password';
13      }
14  }
15  ?>
```

Listing 2.1: Vulnerable code that handles login form submissions.

The source of the bug is on line 7 where the error message *'the $username was not found'* is displayed. Because the `$username` variable receives no sanitization, an attacker can inject a malicious payload in this variable using the `$_POST['username']` parameter and then be outputted to the response unmodified. The HTML parser in the client would then freely interpret it according to its content thus executing the XSS payload.

Exploit: A victim is fooled into submitting a form located in an attacker controlled website. This malicious form is designed to trigger the vulnerability found in the above login form. As soon as the form is submitted the vulnerable login page is opened with the XSS script executed in it. If the victim now tries to login, the XSS script can easily send the credentials to the attacker as well. This is a probable scenario, as the page that was opened is the original page with the identical layout and with the possible HTTPS padlock icon in the address bar. The XSS script can also hide the error message of line 7, thus ensuring its stealthiness even more.

Chapter 3

Architecture

Our instrumentation tool can provide coverage in the form of Node, Edge and Path coverage. In this section we elaborate on how it achieves this, while also describe certain key concepts of *webFuzz* like how it leverages the instrumentation feedback, the mutation process, and its vulnerability detection mechanism.

3.1 webInstr

3.1.1 Overview

Our instrumentation tool, *webInstr*, is focused on instrumenting PHP applications. We have additionally created a preliminary version that instruments *HACK* applications to demonstrate the general applicability of our approach. Since there is only a limited fraction of HACK applications released as open source, we further focus on the PHP implementation of our approach.

In contrast to other known instrumentation frameworks such as *sancov* [23] and *bcov* [10] that target the Intermediate Representation (IR) and the native code respectively, our tool works on the Abstract Syntax Tree (AST) level. Working with the bytecode representation of the PHP language is also possible as other similar libraries have shown [36] and it left as a future goal. Our approach parses the AST of each PHP source file, adds instrumentation code to it and reverts it back to source code form. This work flow is achieved using the *PHP-Parser* [32] library, a stable fully featured AST modification library actively developed by a PHP core developer. Section 4 analyses the procedure of identifying basic blocks whereas this section focuses on how we can measure node, edge and path coverage once the basic blocks have been identified.

3.1.2 Edge Coverage

Inspired by *AFL*'s method of providing edge coverage information, we have adapted this approach to web applications. Listing 3.1 shows the instrumented version of a function `foo`. At the beginning of every basic block, a unique randomly generated number (the basic block's label) is XORed with the label of the previously visited block. The result of this operation represents the edge label. Since large numbers are used to represent the basic block labels, we can assume that clashes between identical edge labels are minimal.

AFL goes a step further and stores additional bookkeeping information such as hit counts that measure how many times an edge has been executed in a single run. This functionality has also been implemented by our instrumentation tool. The edge label is used as index in the global 'map' array where the counters for each edge are stored. For every increment in the edge hit-count, we must first check that the entry is initialised (using the coalesce operator) and only then increment the counter.

The last statement in the stub code performs a right bitwise shifting on the current basic block label and stores the result as the label of the previously visited block. The shifting is needed to avoid cases where a label is XORed with itself thus giving zero as a result. This can happen for instance, with simple loops that do no contain control statements in their body [52].

The super-global variable of PHP (`GLOBALS`) allows us to have access to the instrumentation data structures anywhere in the code, regardless of scope.

```php
<?php
function foo(int $x, int $y) {

    # BEGIN instrumentation code for basic block A
    # $____key represents Edge CallSite->A
    $____key = BASIC_BLOCK_A_LABEL ^ $GLOBALS["____instr"]["prev"];
    # check that counter is initialised first
    $GLOBALS["____instr"]["map"][$____key] ??= 0;
    $GLOBALS["____instr"]["map"][$____key] += 1;
    $GLOBALS["____instr"]["prev"] = BASIC_BLOCK_A_LABEL >> 1;
    # END instrumentation code

    $z = 0;

    if ($y == 0) {
        # BEGIN instrumentation code for basic block B
        # this represents Edge A->B
        $____key = BASIC_BLOCK_B_LABEL ^ $GLOBALS["____instr"]["prev"];
        $GLOBALS["____instr"]["map"][$____key] ??= 0;
        $GLOBALS["____instr"]["map"][$____key] += 1;
        $GLOBALS["____instr"]["prev"] = BASIC_BLOCK_B_LABEL >> 1;
        # END instrumentation code

        $z = 1
    }

    # BEGIN instrumentation code for basic block C
    # this represents Edge A->C or B->C
    $____key = BASIC_BLOCK_C_LABEL ^ $GLOBALS["____instr"]["prev"];
    $GLOBALS["____instr"]["map"][$____key] ??= 0;
    $GLOBALS["____instr"]["map"][$____key] += 1;
    $GLOBALS["____instr"]["prev"] = BASIC_BLOCK_C_LABEL >> 1;
    # END instrumentation code

    $result = $x / ($y + $z);
    return $result;
}
```

Listing 3.1: Example of an instrumented function using edge coverage policy.

3.1.3 Node Coverage

The aforementioned approach for estimating edge coverage comes with its costs. Depending on the code structure of the targeted application (whether it contains many small control statements), the instrumentation overhead can be significant. The impact on the application's response time is more thoroughly analysed in Section 5. In the Node Coverage policy the instrumentation code is lighter as we are only concerned with which basic blocks were triggered regardless of the flow between basic blocks. The instrumentation code inserted at each block is shown in Listing 3.2. Just like in Edge Coverage policy, the hit-count of each basic block are stored.

```php
<?php
function divide(int $x, int $y) {
    # BEGIN instrumentation code for Block A
    $GLOBALS["____instr"]["map"][BASIC_BLOCK_A_LABEL] ??= 0; # initialisation guard
    $GLOBALS["____instr"]["map"][BASIC_BLOCK_A_LABEL] += 1;
    # END instrumentation code

    return $x/$y;
}
```

Listing 3.2: Example of the stub code inserted in a basic block for estimating node coverage.

3.1.4 Path Coverage

Similarly, path coverage is implemented by keeping track of which basic block is currently being executed. Listing 3.3 shows the basic block stub code inserted with this policy. Using PHP's syntax sugar for inserting an element in an array (the empty square brackets), no array index information need to be retrieved.

All though requiring the smallest stub code out of all three methods, this policy can introduce heavy memory overheads. The order of the basic-blocks executed must be kept in the array, and thus executions that contain multiple loop iterations can increase the memory usage significantly.

```php
<?php
function divide(int $x, int $y) {
    # BEGIN instrumentation code for Block A
    $GLOBALS["____instr"]["map"][] = BASIC_BLOCK_A_LABEL;
    # END instrumentation code

    return $x/$y;
}
```

Listing 3.3: Example of the stub code inserted in a basic block for estimating node coverage.

3.1.5 Coverage Report Output

By leveraging the rich set of features PHP provides to us, we can output the 'map' array on program exit in an easy and cost-friendly manner. We have implemented three different output formats, namely using HTTP headers, regular files and shared-memory regions. In general we have observed that the header output method has the largest overhead out of all three methods but has the advantage of being self-contained. There are no external files involved and it does not require sharing the same address space.

As seen in Listing 3.4, every PHP source file is prepended with stub code that initialises the instrumentation data structures and registers a function to be called upon script exit. This stub code will be the first statements executed during an execution, and will only be called once.

This particular function will write the 'map' array to a regular file. The version for the shared-memory output is identical except that the `fopen` call is replaced with a `shmop_open` and the `fwrite` call is replaced with a `shmop_write`. For the header output method, the `fwrite` call is replaced with a `header` call and an additional `ob_start(null,0,0)` call is required to enable response output buffering so as to avoid sending the HTTP body before all the HTTP headers have been written.

```php
<?php
if (!array_key_exists("____instr", $GLOBALS)) {
    $GLOBALS["____instr"]["map"] = array();
    $GLOBALS["____instr"]["prev"] = 0;
    function ____instr_write_map()
    {
        $f = fopen("/var/instr/map." . $_SERVER["HTTP_REQ_ID"], "w+");
        foreach ($GLOBALS["____instr"]["map"] as $k => $v) {
            fwrite($f, $k . "-" . $v . "\n");
        }
        fclose($f);
    }
    register_shutdown_function("____instr_write_map");
}
```

Listing 3.4: The instrumentation header code that is inserted at the beginning of each file.

3.2 webFuzz

This section begins with the high-level view of *webFuzz* delving into how it is structured and its work flow. A thorough discussion on how the coverage feedback is leveraged by the fuzzer is provided next. We end this section by briefly analysing the type of input mutations it implements and the web vulnerabilities it can detect.

3.2.1 High-Level Conceptual View

Figure 3.1 shows the UML Class diagram of *webFuzz*. It has been designed in an Object-Oriented way with each class focused on a single task thus adhering to the Single Responsibility Principle (SRP).

The `Fuzzer` class is the main link between the classes where it combines their functionalities to provide a fuzzing session. It consists of multiple workers, with each responsible in sending, receiving and analysing requests in an endless fashion.

The elementary class `Request` is utilized by all the other classes and represents one request. A request can be complete (its response has been received and analysed) or incomplete (has not been sent to the PUT yet).

The `RequestQueue` class handles the storing of the most *favorable* complete requests while providing efficient insertion and retrieval operations.

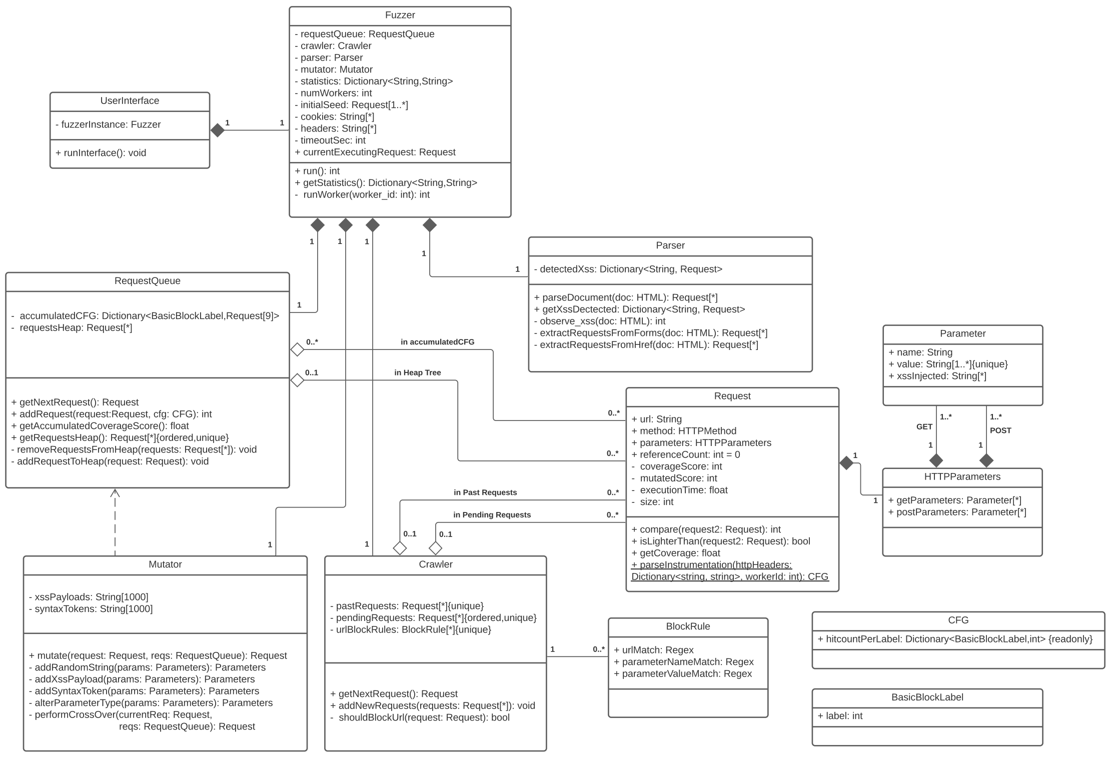
The `Crawler` class handles the bookkeeping of *all* complete requests so as to avoid revisiting the same links. It further keeps track of incomplete requests that are awaiting execution and manages URL blocking.

The `Mutator` class is responsible in creating mutated versions of previous completed requests by modifying their *GET* and *POST* parameters.

The `Parser` class handles the *HTML* document analysis which consists of checking for XSS vulnerabilities and scanning for new URLs.

Workflow

Initially a worker will either fetch a new, incomplete request that is provided by the crawler (`getNextRequest` in `Crawler` class) or will mutate the most favorable complete request which is provided by the `getNextRequest` from the `RequestQueue` class. Currently *webFuzz* will strictly prioritise new unvisited requests over mutated existing ones. Its HTML response is then analysed using `parseDocument` from the `Parser` class.



Fuzzer
- requestQueue: RequestQueue
- crawler: Crawler
- parser: Parser
- mutator: Mutator
- statistics: Dictionary<String,String>
- numWorkers: int
- initialSeed: Request[1..*]
- cookies: String[*]
- headers: String[*]
- timeoutSec: int
+ currentExecutingRequest: Request
+ run(): int
+ getStatistics(): Dictionary<String,String>
- runWorker(worker_id: int): int
UserInterface
- fuzzerInstance: Fuzzer
+ runInterface(): void
Parser
- detectedXss: Dictionary<String, Request>
+ parseDocument(doc: HTML): Request[*]
+ getXssDectected: Dictionary<String, Request>
- observe_xss(doc: HTML): int
- extractRequestsFromForms(doc: HTML): Request[*]
- extractRequestsFromHref(doc: HTML): Request[*]
RequestQueue
- accumulatedCFG: Dictionary<BasicBlockLabel,Request[9]>
- requestsHeap: Request[*]
+ getNextRequest(): Request
+ addRequest(request:Request, cfg: CFG): int
+ getAccumulatedCoverageScore(): float
+ getRequestsHeap(): Request[*]{ordered,unique}
- removeRequestsFromHeap(requests: Request[*]): void
- addRequestToHeap(request: Request): void
in accumulatedCFG
in Heap Tree
Parameter
+ name: String
+ value: String[1..*]{unique}
+ xssInjected: String[*]
GET
POST
Request
+ url: String
+ method: HTTPMethod
+ parameters: HTTPParameters
+ referenceCount: int = 0
- coverageScore: int
- mutatedScore: int
- executionTime: float
- size: int
+ compare(request2: Request): int
+ isLighterThan(request2: Request): bool
+ getCoverage: float
+ parseInstrumentation(httpHeaders: Dictionary<string, string>, workerId: int): CFG
HTTPParameters
+ getParameters: Parameter[*]
+ postParameters: Parameter[*]
in Past Requests
in Pending Requests
Mutator
- xssPayloads: String[1000]
- syntaxTokens: String[1000]
+ mutate(request: Request, reqs: RequestQueue): Request
- addRandomString(params: Parameters): Parameters
- addXssPayload(params: Parameters): Parameters
- addSyntaxToken(params: Parameters): Parameters
- alterParameterType(params: Parameters): Parameters
- performCrossOver(currentReq: Request, reqs: RequestQueue): Request
Crawler
- pastRequests: Request[*]{unique}
- pendingRequests: Request[*]{ordered,unique}
- urlBlockRules: BlockRule[*]{unique}
+ getNextRequest(): Request
+ addNewRequests(requests: Request[*]): void
- shouldBlockUrl(request: Request): bool
BlockRule
+ urlMatch: Regex
+ parameterNameMatch: Regex
+ parameterValueMatch: Regex
CFG
+ hitcountPerLabel: Dictionary<BasicBlockLabel,int> {readonly}
BasicBlockLabel
+ label: int

Figure 3.1: A simplified UML Class Diagram of *webFuzz*. Note that almost all classes have a dependency relationship with the `Request` class. For readability purposes this is not shown.

In here, the document is checked for RXSS vulnerabilities and any new unseen URLs from `form` and `anchor` elements are also extracted. These new links are fed back to the crawler using `addNewRequests` in the `Crawler` class. Depending on the type of instrumentation, the coverage feedback will either be in the HTTP header, an external file, or in a shared-memory region. The static method `parseInstrumentation` in `Request` class is responsible for reading the feedback and returning a `CFG` dictionary that holds all the triggered labels by the current request together with their hit-counts. Whether this request will be stored for future mutation is handled by the `addRequest` in the `RequestQueue` class. The decision process is thoroughly discussed in Section 3.2.2. From this point onwards the cycle repeats endlessly.

3.2.2 Coverage Feedback

webFuzz leverages the instrumentation feedback in two ways. First, it can reduce its memory footprint by grouping similar requests and storing only the most favorable from each group. In this way, it only stores just enough requests to cover every single label it has observed. As it has been explained earlier, a label can either be a CFG edge or a basic block. Secondly, by knowing the code paths a request triggers, it can prioritise requests that have explored code paths not previously observed or that exercise large parts of the code. In fact, *webFuzz* computes a weighted rank for each request using the aforementioned metrics and a few other such as the request's response time. At its core, *webFuzz* will utilize two data structures to achieve these goals: a dictionary of all the labels it has observed and a Heap Tree of all the favorable requests in a semi-ordered fashion. Using these two structures which are stored in `RequestQueue` class, it can decide whether a new request will be kept or discarded, and which is the most favorable request.

Discarding Requests

Since many requests do not trigger new labels, an algorithm is needed to compare such similar requests so as to store only the most favorable. Algorithm 2 shows the *AFL*-inspired algorithm that determines if a new request will be kept or discarded. The coverage feedback of a new request is checked against a dictionary (`accumulatedCFG`) that holds information about all the labels that have been observed so far together with the requests that have triggered it. Each entry in the dictionary is further split to *9* buckets, with each bucket corresponding to the different number of times the label was executed in a single run. The use of buckets aims to distinguish requests that have executed a CFG node or edge only a few times versus many more [52].

When the hit-count of a label returned by a new request falls in a bucket that has already been observed, there will be a clash for the same bucket in the same entry in the dictionary. The two requests will be compared in terms of execution time and request size and the lightest node will get the entry.

Each `Request` object has an additional `referenceCount` attribute that holds how many entries it contains in this dictionary. A reference count of zero means that for each label the request triggered, there exists a lighter request that has also triggered it.

As soon as a request has no longer entries in this dictionary it will be removed as well from the Heap structure that is described next.

Request Ranking

As soon as a request has successfully acquired at least one entry in the dictionary it will then be inserted to the internal min-Heap tree that contains all the favorable, complete requests. For the insertion to work, a method of comparing two requests is needed. This is provided by `compare` in `Request` class. It calculates a weighted difference score of the two requests based on their attributes. The metrics it uses are listed below. Note that a *(+)* symbol indicates higher values are better while the opposite applies to *(-)*.

- **Coverage Score (+)**: Total number of labels it has triggered

Algorithm 2 Adding a new request. `addRequest` method in `RequestQueue` class

```
function ADDREQUEST($accumulatedCFG, newRequest, coverageFeedback$)
    for ($blockLabel, hitCount$) in $coverageFeedback$ do
        $bucket \leftarrow 0$                                    ▷ Calculate bucket number from hit count
        if $hitCount \geq 256$ then
            $bucket \leftarrow 8$
        else
            $bucket \leftarrow ceiling(log2(bucket))$
        end if

        $pendingRemoval \leftarrow \{\}$
        $existingRequest \leftarrow accumulatedCFG[blockLabel][bucket]$

        if $existingRequest == \emptyset$ then                  ▷ Has this bucket been hit before ?
            $accumulatedCFG[blockLabel][bucket] \leftarrow newRequest$
            $newRequest.refCount \leftarrow newRequest.refCount + 1$

        else if $newRequest.isLighterThan(existingRequest)$ then
            $accumulatedCFG[blockLabel][bucket] \leftarrow newRequest$
            $newRequest.refCount \leftarrow newRequest.refCount + 1$
            $existingRequest.refCount \leftarrow existingRequest.refCount - 1$

            if $existingRequest.refCount == 0$ then
                $pendingRemoval.insert(existingRequest)$
            end if
        end if
    end for

    if $newRequest.refCount == 0$ then
        return $-1$                                              ▷ request is discarded
    end if

    $removeRequestsFromHeap(pendingRemoval)$      ▷ Replaced requests are discarded
    $addRequestToHeap(newRequest)$                          ▷ New request is kept

    return $size(pendingRemoval)$          ▷ Number of previous requests it has removed
end function
```

- **Mutated Score (+)**: Crude approximation on how much the execution trace differed from that of its parent request (the request that it got mutated from)
- **Execution Time (-)**: Round-trip time of the request
- **Size (-)**: Total number of characters in the URL and in the POST parameters
- **Picked Score (-)**: How many times it has been picked for further mutation. This ensures that all requests in the Heap will eventually be fuzzed.

3.2.3 Mutations

Mutation is a necessary step in the fuzzing process in order to maximize the number of paths explored and to trigger bugs lying in vulnerable pieces of code. The choice of mutation functions is both a challenging and empirical task. Aggressive mutating functions can destroy much of the input data structure which will result in the test case failing early on during program execution. On the other hand, too conservative mutations may not be enough to trigger new control flows [50].

Currently five mutation functions are supported but *webFuzz* can easily be extended to support custom GET or POST parameter mutations. The mutation functions it employs are as follows:

- Injection of real-life XSS payloads found in the wild into GET or POST parameters
- Mixing GET or POST parameters from other favourable requests (in Evolutionary Algorithms this is similar to Crossover)
- Insertion of a randomly generated payload (can be a string or an integer) into a parameter
- Insertion of HTML, PHP or JavaScript syntax tokens into a parameter
- Altering the type of a parameter (from an Array to a String and vice versa)

Conversely to many fuzzers that employ malicious payload generation via the use of specific attack grammar [16, 41], *webFuzz* has taken a mutation-based approach [34] where starting with the default input values of an application (e.g. specified by the value attribute in a HTML input element), real-life XSS payloads, random strings or specific HTML, JavaScript and PHP syntax tokens are prepended and appended to them. Some parameters may also get randomly opted out from the mutation process. This can be useful in cases where certain parameters need to remain unchanged for certain areas of the program to execute.

Although arrays in URL strings are not clearly defined in RFCs and their format is more framework specific, a number of web applications rely on them or are even oblivious to their existence. For this purpose we have added an input type altering mutation, where an input parameter that is expected to be parsed as a string in the web application is transformed into an array or vice versa. Web applications that do not guard against unexpected types could fall victims of unintended code execution behavior.

Lastly, as the incorporation of evolutionary algorithms in the test case creation process has been widely used in fuzzers to optimize the solution searching process [16,34,41,52], *webFuzz* will also mix GET or POST parameters from different favorable requests to generate new inputs. Conversely to how evolutionary algorithms specify, this crossing over of input is not defined as a necessary step in each new input creation but can happen with a medium probability.

3.2.4 Vulnerability Detection

webFuzz is currently designed to detect Reflective Cross Site Scripting bugs. Additionally detecting Stored Cross-site Scripting vulnerabilities and endpoints susceptible to Denial of Service (DoS) attacks should not require heavy changes and will be implemented in the future. To detect RXSS vulnerabilities we use the method of string searching for the injected malicious payload in the returned HTML response. This method is the most performing in terms of speed but can induce false positives as the location of the payload in the HTML response is not accounted for. One example of this would be if the XSS payload is returned unsanitized inside an HTML element's attribute. If the web application correctly sanitizes any quotes found in the XSS payload then the payload will not be executable. Listing 3.5, shows this difference. In the first two `form` elements, due to correct sanitization of double quote characters and the location of the JavaScript code, the `alert` message will not be shown. In the third form, the double quote character found in the XSS payload is not sanitized and will thus be executed. Plans on improving the accuracy of our XSS detector are discussed in Section 7.

```
<html>
    <body>
        <form class="><script>alert(1)</script>" method="GET">
            <input name='submit' type='submit'>
        </form>

        <form class="&quot;><script>alert(1)</script>" method="GET">
            <input name='submit' type='submit'>
        </form>

        <form class="\"><script>alert(1)</script>"␣method="GET">
␣␣␣␣␣␣␣␣␣␣<input␣name='submit'␣type='submit'>
␣␣␣␣␣␣</form>
␣␣␣</body>
```

```
</html>
```

Listing 3.5: How *JavaScript* code inside *HTML* attributes is threated by the *HTML* parser. In the first two `form` elements, the payload `<script>alert(1);</script>` will not be executed. But in the third form, the quote character is not properly sanitised and so an alert message will be displayed.

Chapter 4

Implementation

4.1 Instrumentation

To implement *webInstr*, we have used the library *PHP-Parser* that works on the AST level of the code. The library offers multiple features such as providing an interface to traverse and modify the tree, ability to convert the tree back to PHP source code, and fully supports PHP version *5*, *7* and *8*. In this section we will dive deeper in the implementation details of *webInstr*, showing how we can use the interface provided by the PHP-Parser library to implement our tool.

4.1.1 PHP-Parser

PHP-Parser is designed to provide an easy to use mechanism to parse, analyse and modify the AST of individual PHP source files. This process happens on the file-level, meaning the AST produced contains the statements and expressions of a single file only. Listing 4.2 shows the resulting AST parsed from the ReLu function as defined in Listing 4.1. In the AST, the statements are the nodes (type starting with `Stmt_`), and the edges between the nodes are defined in attributes like `stmts`, `elseifs`, `else`. These attributes themselves hold an array of statements that define the body of the enclosing statement. Expressions are not nodes in the tree but properties of a node statement. This means that a depth-first traversal of the tree would begin with the top-level statements (as defined by the outermost `array()`) and recursively jump to each statement, followed by its body. Expressions can be retrieved only by inspecting the properties of a node. The tree is self-contained and can thus be transformed back to source code without breaking functionality in the original code (except perhaps code formatting). For instance, metadata such as passing by reference or by value, and type-hints are provided in the properties of each associated statement. In addition, each token can easily be differentiated by inspecting its type.

```php
<?php

function reLu(float $x):float {
    if ($x <= 0)
        return 0;
    else
        return $x;
}
```

Listing 4.1: Definition of the ReLu function in PHP for demonstration purposes.

```
<?php

array(
    0: Stmt_Function(
        attrGroups: array()
        byRef: false
        name: Identifier(name: reLu)
        params: array(
            0: Param(
                attrGroups: array()
                flags: 0
                type: Identifier(name: float)
                byRef: false
                variadic: false
                var: Expr_Variable(name: x)
                default: null
            )
        )
        returnType: Identifier(name: float)
        stmts: array(
            0: Stmt_If(
                cond: Expr_BinaryOp_SmallerOrEqual(
                    left: Expr_Variable(name: x)
                    right: Scalar_LNumber(value: 0)
                )
                stmts: array(
                    0: Stmt_Return(
                        expr: Scalar_LNumber(value: 0)
                    )
                )
                elseifs: array()
                else: Stmt_Else(
                    stmts: array(
                        0: Stmt_Return(
                            expr: Expr_Variable(name: x)
                        )
                    )
                )
            )
        )
    )
)
```

Listing 4.2: The resulting AST converted from the ReLu function defined in Listing 4.1

The library also provides methods to traverse the tree and modify it in place. This functionality is provided by a visitor class that a user can extend to specify the desired changes to the tree. This class provides four methods to implement, namely `beforeTraverse`, `afterTraverse`, `enterNode` and `leaveNode`. The first two methods define what changes should happen to the top-level statements before and after the tree traversal process respectively. The latter two methods define the changes on individual nodes (statements). `enterNode` method is called for each node at the point of entry to its branch in the AST and `leaveNode` is called when leaving this branch rooted at this node [30].

4.1.2 webInstr

The general concept of *webInstr* is that it identifies basic-blocks in the code during the AST traversal process and inserts stub code at the beginning of each block. In addition, it will prepend each source file with extra stub code that registers a shutdown function to be called upon script exit. This function is responsible in outputting the coverage information gathered from the execution.

webInstr has been designed to be extendable, where custom basic block stub code and custom output methods can easily be used. Figure 4.1 shows the hierarchy of the classes that *webInstr* is composed of. An abstract base class (`BasicBlockVisitorAbstract`) is implemented that identifies the basic blocks and provides abstract methods to extend that define the actual stub code to be inserted.

Base Class

The base class of *webInstr* is named `BasicBlockVisitorAbstract` and is a Visitor class that extends the `NodeVisitorAbstract` class provided by *PHP-Parser*. This class is responsible in identifying the basic blocks from the AST while the stub code to be inserted is left to the deriving classes to provide.

As mentioned in Section 2.1, there are 3 rules that determine leader statements and thus identify the boundaries between the basic blocks. For the sake of performance our approach identifies leader statements as being only (i) the first statement *inside* a control statement's body (Rule 2) and (ii) the first statement that follows *after* a control statement's body (Rule 3). This method does not strictly adhere to the definition of a basic block, as clause expressions are not instrumented.

Since Rule 3 specifies that the target statement of a jump is a leader, the clause expression in loop statements is a basic block itself, while the start of its body another. This is because in loops, control can jump directly to the clause expression using either a `continue` statement or after the last statement in the loop is executed. By this logic, and as Rule 3 specifies, the clause expression in a loop statement would be a jump target and

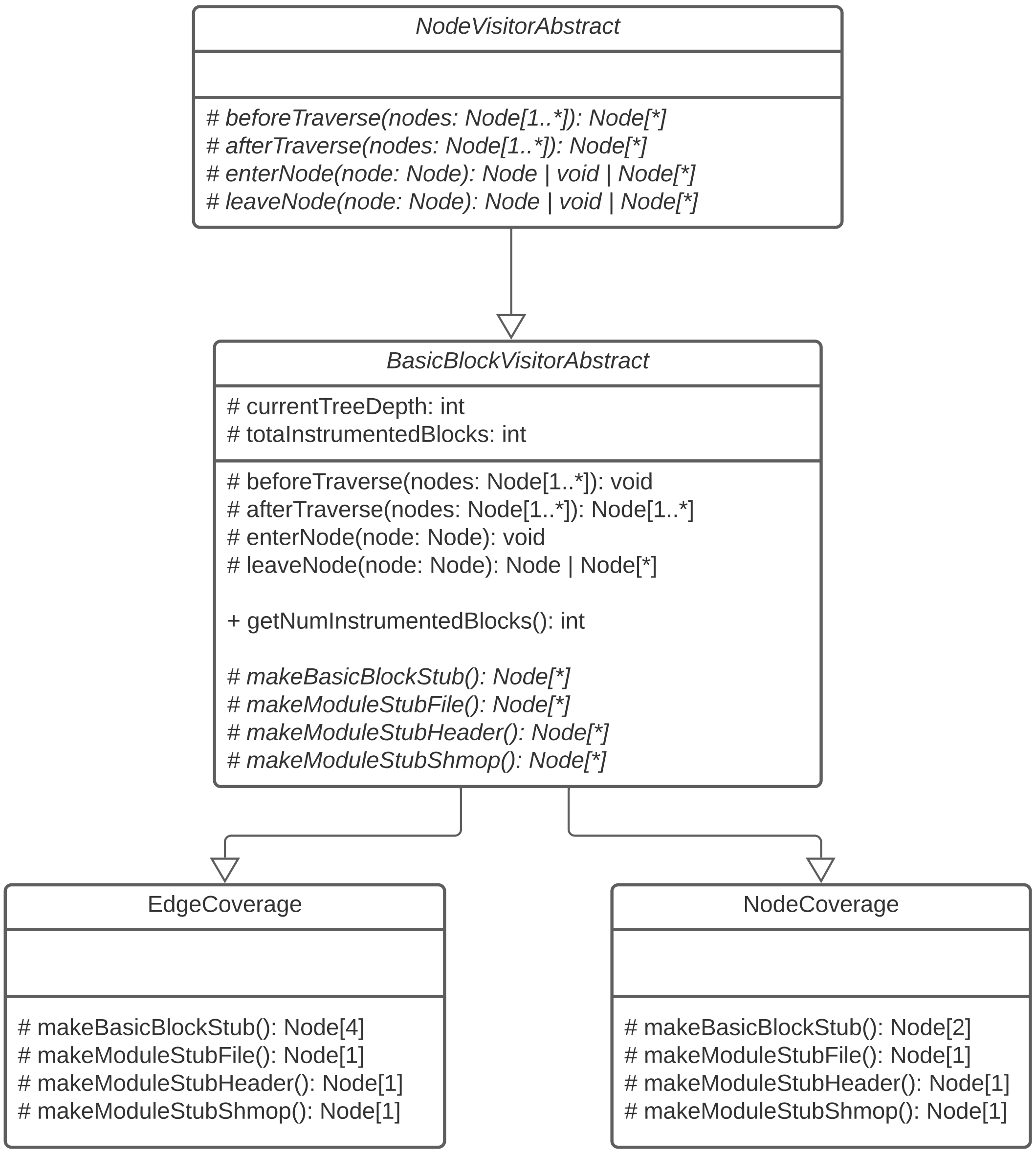

NodeVisitorAbstract
# beforeTraverse(nodes: Node[1..*]): Node[*]
# afterTraverse(nodes: Node[1..*]): Node[*]
# enterNode(node: Node): Node | void | Node[*]
# leaveNode(node: Node): Node | void | Node[*]
BasicBlockVisitorAbstract
# currentTreeDepth: int
# totalInstrumentedBlocks: int
# beforeTraverse(nodes: Node[1..*]): void
# afterTraverse(nodes: Node[1..*]): Node[1..*]
# enterNode(node: Node): void
# leaveNode(node: Node): Node | Node[*]
+ getNumInstrumentedBlocks(): int
# makeBasicBlockStub(): Node[*]
# makeModuleStubFile(): Node[*]
# makeModuleStubHeader(): Node[*]
# makeModuleStubShmop(): Node[*]
EdgeCoverage
# makeBasicBlockStub(): Node[4]
# makeModuleStubFile(): Node[1]
# makeModuleStubHeader(): Node[1]
# makeModuleStubShmop(): Node[1]
NodeCoverage
# makeBasicBlockStub(): Node[2]
# makeModuleStubFile(): Node[1]
# makeModuleStubHeader(): Node[1]
# makeModuleStubShmop(): Node[1]

Figure 4.1: The UML Class Diagram for *webInstr*. Note the italic font which specifies that it is defined as abstract.

thus a separate basic block. Some information is lost when not accounting this scenario, such as control flows produced by statements such as `break` and `continue` inside loops can in some cases not be differentiated.

More importantly, multiple blocks can be missed by not instrumenting the clause expression, as it can be composed of multiple basic blocks itself. An expression such as `Condition1 && Condition2 && Condition3`, can cause the control flow to branch at the location of every `&&` operator. This is because PHP evaluates clauses in a lazy manner. Thus each logical operator in a clause expression corresponds to one possible jump statement and thus to one leader statement.

Nevertheless, to identify the control statements and add the needed stub code, the leaveNode method has been implemented. Identification is done by inspecting the type of each node visited during the traversal process. For every control statement identified, it will prepend the array in its stmts property with nodes (the stub code) generated using the abstract method makeBasicBlockStub. This handles the case (i) as mentioned before. In addition, the leaveNode method will return an array of nodes instead of a single one. This array will contain the modified control statement, and new nodes generated using makeBasicBlock again. In this way, *PHP-Parser* knows that the extra nodes added should be placed after this control statement. This handles the case (ii).

Certain statement types such as the Stmt_Else must be handled differently. Since they share the same ending target as their enclosing Stmt_If statement, stub code should only be added at the beginning of their body. Case (ii) would already be handled when visiting their enclosing Stmt_If. Similar reasoning happens for the switch-case statements.

The class also implements the afterTraverse method, in which it finds the right location between the top-level statements of a script to insert the initialisation stub code. The actual stub code is provided using the abstract methods makeModuleStub* for each output method selected.

In PHP, if a script uses Declare or Namespace statements, then these must be placed at the beginning of the script before all other statements. Our implementation guards against this scenario, and will prepend each source file with initialisation code only after such statements. The overhead that these initialisation statements impose is also minimal. Because they are added as top-level statements, they will be executed at most once per file instrumented. This can happen when a file is directly executed (i.e. not imported by another file) or during the first call of include or require where files are imported to scope (included files are only loaded once).

Lastly, the implementation of enterNode and beforeTraverse nodes is rather simple, as they only keep track of the current tree depth during the traversal process. The depth is provided using the attribute currentTreeDepth.

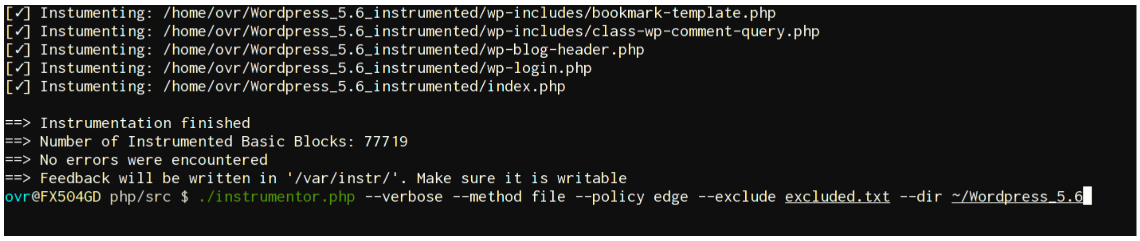
Deriving Classes

There are four methods that these classes must implement. Firstly, the makeBasicBlockStub method in which it generates code to be inserted at the beginning of each basic block. Section 3.1.2 has shown the stub code inserted in each basic-block for the edge instrumentation policy. The rest three methods generate the header code for each output method possible. Section 3.1.5 again showed a possible implementation for the file output method.

In general, these methods should initialise the data structures used by the basic block stub code, and further register a shutdown function to output the coverage feedback on script exit.

CLI Interface

Figure 4.2 shows the result of a successful instrumentation of *WordPress*. The CLI Interface of *webInstr* is responsible in parsing the command-line arguments, providing features such directory/file exclusion, and finally initialising and running the derived Visitor classes.



```
[✓] Instumenting: /home/ovr/Wordpress_5.6_instrumented/wp-includes/bookmark-template.php
[✓] Instumenting: /home/ovr/Wordpress_5.6_instrumented/wp-includes/class-wp-comment-query.php
[✓] Instumenting: /home/ovr/Wordpress_5.6_instrumented/wp-blog-header.php
[✓] Instumenting: /home/ovr/Wordpress_5.6_instrumented/wp-login.php
[✓] Instumenting: /home/ovr/Wordpress_5.6_instrumented/index.php

==> Instrumentation finished
==> Number of Instrumented Basic Blocks: 77719
==> No errors were encountered
==> Feedback will be written in '/var/instr/'. Make sure it is writable
ovr@FX504GD php/src $ ./instrumentor.php --verbose --method file --policy edge --exclude excluded.txt --dir ~/Wordpress_5.6
```

Figure 4.2: Snapshot of *webInstr* after having successfully instrumented *WordPress*.

4.2 Fuzzer

This section discusses some of the implementation choices for *webFuzz* and the reasoning behind them. Thorough analysis of its implementation is not provided as it is outside the scope of this thesis.

4.2.1 Design Choices

Language of Choice

webFuzz is written in Python 3 due to its ease of use, rapid development, high code readability and conciseness and a delegation of concerns from the programmer such as memory management.

Since the Python language itself is only a specification, there exist multiple implementations of it. *webFuzz* runs in both *CPython* (the reference implementation) and *PyPy*. *PyPy* generally runs Python code faster but with a higher memory footprint than *CPython*, as it uses Just-In-Time compilation rather than an interpreter [38]. We have not noticed substantial speed improvements with *PyPy* though, possibly because it is optimized for CPU heavy tasks and not I/O heavy such as our task [35].

Alternative implementations such as *Jython* and *IronPython* are not applicable due to their lack of support for Python 3 and their higher overhead with respect to *CPython*.

These implementations are more suitable in situations where Python needs to be interoperated with other languages such as Java and .NET languages correspondingly [35].

Concurrency

Since sending HTTP requests is an I/O heavy process, utilizing asynchronous execution, multi-threading or multi-processing can offer substantial speed improvements. While waiting for the response from the web server, we could process the response of other requests thus the CPU can remain busy throughout the fuzzing session.

webFuzz relies on asynchronous execution rather that multi-threading or multi-processing to provide concurrency largely due to the idiosyncrasies of Python. In its most prevalent implementations (*CPython* and *PyPy*), each thread must acquire a special mutex named the Global Interpreter Lock (GIL), before it can actually execute Python bytecode. This is needed because the internal memory management unit is not thread-safe [33]. As a result of this, parallel execution of threads is not possible, as at any one instance, only one thread can execute, the thread that holds the Lock. For this reason, multi-threading approaches would not offer substantial performance improvements over asynchronous execution (which is single-threaded).

In addition, multi-threading and multi-processing approaches utilize multiple processor cores, and so the effect of race-conditions is present. The GIL mutex only provides thread-safety on the internal operations of the interpreter/JIT engine and not on the application level. Since *webFuzz* uses multiple data-structures to achieve its goals, fine-grained locking solutions would need to be implemented to protect them. This would in turn reduce the maintainability of the code while also require more developing time in implementation and debugging.

If on the other hand asynchronous execution is used (as provided in Python by the `asyncio` library), no locks would be required. This is because concurrent tasks run on a single processor, and `await` statements are not used during writes to the shared data-structures. A context-switch between tasks can only happen while waiting for a HTTP request and while reading the HTTP response from the socket. Any modification to the shared data-structures is guaranteed to succeed with no interruption.

4.2.2 Maintainability

Test Suite

The use of a Test Driven Development approach inspires confidence in the correctness of the code and reduces debugging time. Python only catches type errors during run-time thus an even bigger need for code testing exists as Test suites need to catch both type and application logic errors. For this reason, we have developed a Test Suite which is composed of Unit Tests that target each module that *webFuzz* is made of. Our test suite it is not yet complete but we plan on maximising its code coverage in the future.

Type-Hints

Given the benefits that dynamic typing provides such as polymorphism (e.g. via duck typing), and rapid development, it has proven to be a highly cherished feature by developers. This of course does not come without its disadvantages, as the lack of static type checking and type information in the code reduces the readability and maintainability in the long run [25]. As the need for type information in Python became apparent, PEP 484 [48] introduced the concept of type hints i.e. type annotations for functions and variables that are accompanied at their declaration but are not enforced nor are used internally by the Python run-time engine. These type annotations are only hints for programmers to get a better idea on the functionality of a function or method. Additionally external Python tools can perform static type analysis using these hints thus provide static type checking for dynamic languages such as Python. For this reason, we have extensively annotated *webFuzz* with type hints to improve its maintainability and readability.

Chapter 5

Evaluation

To effectively evaluate our two tools, we identified four key metrics that our experiments should measure. These are the instrumentation overhead, code coverage, throughput and vulnerability detection. For each measurement we raised a research question that we aim to answer from the results. Our research questions are as follows:

RQ1 How much overhead does the instrumentation pose to an application and how does this relate with different software design patterns and project sizes?

RQ2 Does our approach of coverage-guided input selection and mutation achieve high code coverage? Do we still notice an increase in code coverage after the initial crawling process is finished (i.e. are the input mutations effective in triggering new paths)?

RQ3 What is the combined overhead of our solution (instrumentation and coverage processing) in terms of throughput, and how does this compare with black-box approaches?

RQ4 Can *webFuzz* detect more RXSS bugs within a given time frame in comparison with other fuzzers?

To answer *RQ1-2* we applied *webInstr* and *webFuzz* on *4* well known PHP projects. In *RQ1* we measured how different page response times compare in the instrumented and uninstrumented versions of the same application. Section 5.1 discusses the methodology used to obtain our results and the analysis of our findings. For *RQ2* we have measured how the accumulated code coverage develops in these *4* applications, for a fuzzing session lasting at least 17 hours in each. The experiment is thoroughly analysed in Section 5.2.

For *RQ3-4*, we selected one of the most prominent open-source black-box fuzzers, *wFuzz* [24], to compare *webFuzz* with. Our test subjects are *2* prevalent Content Management Systems (CMS) namely *WordPress* and *Drupal*. Sections 5.3, 5.4 focus on answering these questions.

All the tests were executed on two Ubuntu 18.04 LTS Linux machines both possessing a quad-core Intel® Xeon® W-2104 Processor @3.20 GHz and 64GB of RAM. The web server used for our experiments is *Nginx 1.19.6* and for the database back-end we use *MariaDB 10.5.8*, with InnoDB engine the primary storage engine used by all the web applications tested. In total, we have spend around 450 computational hours in running our experiments.

5.1 Instrumentation Overhead

To evaluate the overhead that the instrumentation introduces, we have selected *4* PHP projects from GitHub's trending PHP project list and compared the response time between identical instrumented and uninstrumented versions. The projects are *Mautic 3.2.1* a marketing automation application, *Drupal 9.0.6* a CMS web application, *WordPress 5.6* also a CMS web application, and *Firefly-III 5.4.6* a personal finance management application. All these projects are listed as open-source and they consist of 803000, 707000, 647000 and 533000 PHP lines of code respectively (including external libraries).

Methodology

The methodology followed to generate the data firstly consisted of producing two identical versions of each application. To ensure this, the same installation steps were taken with no further interactions (e.g. submitting forms, and altering the application state). The only difference in the installation procedure is the name of the database used. One of this versions was additionally instrumented using the edge coverage policy.

We then crawled each project using the uninstrumented version and selected a set of URLs to represent each one. Each link had to return a HTTP code 200 for both versions for it to be selected. The number of links for each project varied, with *WordPress* consisting of *65* URLs, *Drupal* with *60*, *Firefly-III* with *23*, and finally *Mautic* with *80* URLs. This is due to their different project sizes, features, and their difficulty in crawling them (e.g. input validating forms blocked further crawling).

For each URL, we measured the time taken to send and receive its response (together with reading the response body) in both the instrumented and uninstrumented version. To further eliminate measurements errors, the response time of each link was calculated using the following procedure. We firstly send the link *20* times to ensure that the *opcache* [28] had compiled and cached the source code into bytecode, and we then took the average response time from *10* runs. This was to ensure that external factors that could affect the final result such as context switches in the *PHP/Zend* engine and database latencies were minimized.

Finally, we calculated the overhead factor of each link, and then computed the minimum, average and overhead factor out of all the links in each project. The overhead factor, was simply calculated by dividing the response times produced from the two versions.

Table 5.1 shows the results of our experiment. Each column represents one PHP project, and each row states the minimum, average and maximum overhead factor.

Analysis

	Mautic 3.2.1	Drupal 9.0.6	Firefly-III 5.4.6	WordPress 5.6
Basic-Blocks	227000	128000	102000	77700
Min. Overhead	1.27x	1.71x	1.81x	1.06x
Avg. Overhead	2.92x	1.85x	2.19x	2.51x
Max. Overhead	3.51x	2.06x	3.37x	3.89x

Table 5.1: Instrumentation overhead factor in *4* PHP Web applications. The first row shows the total number of instrumented basic-blocks in each project. The following rows show the minimum, average and maximum overhead observed from all the URLs in each project.

The results of the experiment show that the instrumentation overhead is significant in all four projects. *Mautic* which has the largest number of instrumented blocks also has the highest average overhead, with it being about *3* times slower than the uninstrumented version. *Drupal* on the other hand, all though holding the second largest number in instrumented blocks, has the least average and maximum overhead. This result may suggest that the software architecture design may play a bigger role in the instrumentation overhead in comparison to the basic-block count. *Drupal* uses a Presentation-Abstraction-Control (PAC) model whereas *Mautic* and *Firefly-III* (based on *Symphony* and *Laravel* web frameworks) are largely MVC based. Both *Mautic* and *Firefly-III*, all though possessing largely different block counts, incurred high overheads, with both having higher average and maximum overhead factors than *Drupal*.

On the other hand, the semi-structured nature of *WordPress*, which uses an Event-Based model but with a less rigid software design pattern than the other three, produced the largest variations in overhead.

Our data though, are not enough to produce any certain conclusions on the relation between basic-block count, software design pattern and the instrumentation overhead. Nevertheless, all four projects noticed substantial decrease in performance with *webInstr*. Section 7 discusses possible improvements that could dramatically decrease this overhead.

5.2 Code Coverage

Measuring code coverage can be an important metric to evaluate a fuzzer, as higher code coverages may entail higher chances in triggering vulnerabilities. This can be explained by the fact that to trigger a given bug, the fuzzer must first be able to reach the associated code path that the bug lies. In addition, code coverage can provide feedback on the effectiveness of the mutation functions employed i.e. whether they can trigger new code paths.

Methodology

For this experiment we have used the same *4* PHP applications specified in Section 5.1. All *4* applications were instrumented using a hybrid node-edge coverage policy. This policy simply provides both node and edge coverage information. The reason for this choice is only to provide an accurate means to measure code coverage as a percentage of the total code coverage possible. If we instrumented the application using only the edge policy, we would not know the number of all possible edges (as that requires CFG generation) and thus the coverage percentage would not be available. Internally *webFuzz* utilizes only the edge information to drive its decision process (as explained in Section 3.2.2) and the node coverage is only used to measure the code coverage percentage.

Analysis

Figure 5.1 shows how the accumulated code coverage progresses with time in the *4* applications. *webFuzz* managed to trigger 27% and 21.5% of the total basic blocks in *Drupal* and *WordPress* respectively, while for both *Firefly-III* and *Mautic*, the code coverage was lower than 10%. With *WordPress* the code coverage is seen to increase even after 50 hours of fuzzing. This is a good sign that the mutation functions employed are effective enough to trigger new code paths at least in *WordPress*. *Drupal* also notices increases in code coverage late in the fuzzing session, but at a much smaller degree.

There are a number of reasons that can explain the low coverages observed. Firstly, the code coverage percentage can be considered an underestimation for both *Mautic* and *Firefly-III*, as instrumentation was applied to both the project's source code, and to its external libraries. Since more than 50% of the total SLOC in these two projects resided in the external libraries, it is possible that a large number of the instrumented basic blocks cannot be executed. This can happen for instance, if features provided by an external library are not used by these projects. The fact that both libraries are implemented on top of feature-dense web frameworks such as *Symphony* and *Laravel* indicates that this may indeed be the case. A preliminary evaluation for this hypothesis on *Mautic* shows that

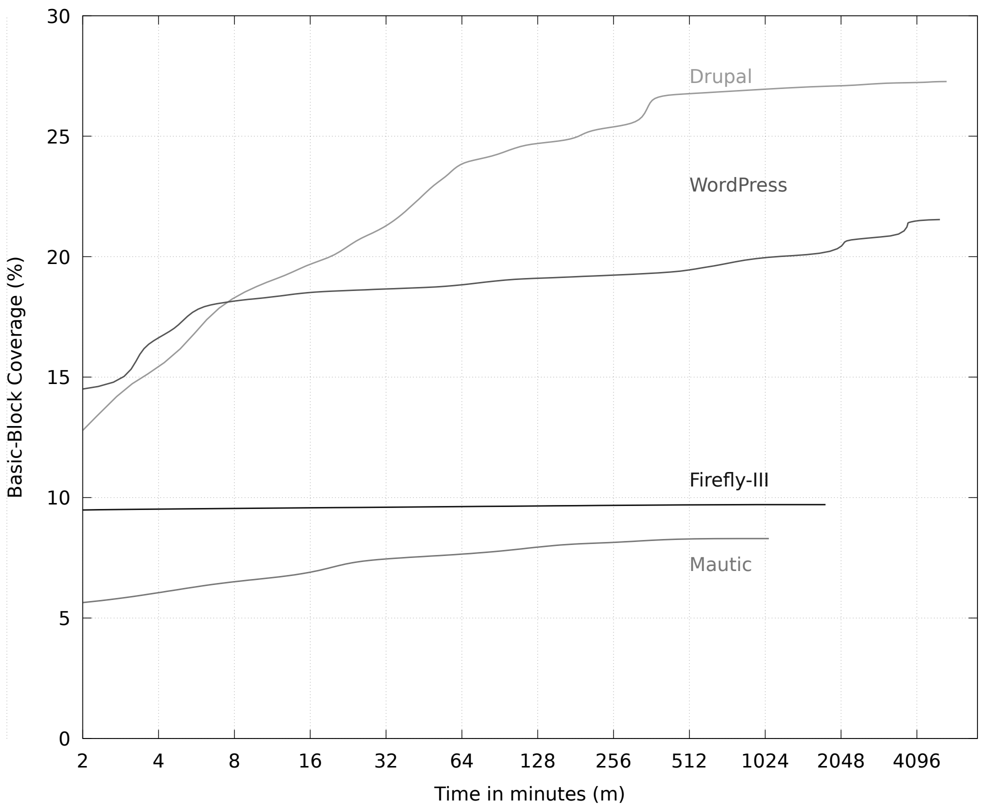


Figure 5.1: Accumulated Basic-Block coverage in *4* web applications using *webFuzz*.

by excluding external libraries the instrumented basic-block count is reduced by 73% and the code coverage increases up to 19% in just 20 minutes of fuzzing.

Perhaps more importantly, both applications perform input validation in their HTML forms, and default input values are most of the times not provided. As a result, *webFuzz* struggles to conceive valid inputs for a large number of forms, and thus code coverage suffers. This can be seen for instance with *Firefly-III*, where the first *2* minutes contributed for the 9.6% whereas the rest 1000 minutes only provided 0.2%.

Mautic had the shortest fuzzing session as it contained multiple I/O heavy links that stalled the fuzzing process considerably. After *512* minutes in the session, the throughput fell to almost *0* (with response times as high as *20* seconds) and so the session was terminated early. More experiments need to be run with *Mautic* to identify appropriate weight values in request ranking metrics (as seen in Section 3.2.2) and thus avoid fixations on I/O heavy links.

5.3 Throughput

One reason for the effectiveness of fuzzers in discovering vulnerabilities is their ability to test millions of inputs in a short time frame. It is thus of paramount importance that the coverage-guided approach to fuzzing does not severely degrade the fuzzing throughput. For this reason we have conducted an experiment to test the relative difference in throughput (requests/sec) between the black-box fuzzer *wFuzz* and *webFuzz*.

Methodology

Two identical versions of *WordPress* and *Drupal* were used for this experiment, with the only difference that one version was additionally instrumented using the edge coverage policy. In addition, *wFuzz* was extended to include a complete crawling functionality and was instructed to fuzz each URL target *5000* times before moving on to the next target. More details on the exact modifications done are discussed in Section 5.5.1. Lastly, we specified *wFuzz* to use *10* concurrent workers while *webFuzz* used only *8* to account for the additional server load introduced by the instrumentation.

Analysis

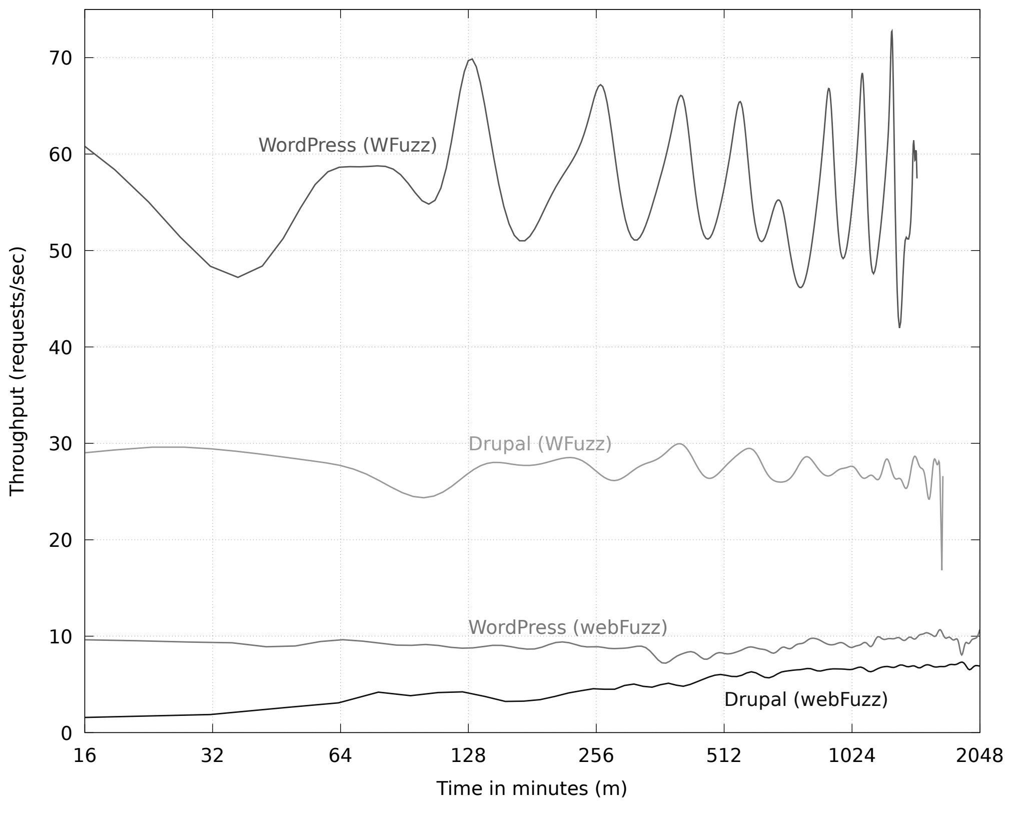


Figure 5.2: Throughput (request/sec) of *webFuzz* and *wFuzz* in Drupal and Wordpress. *webFuzz* is seen to be around *3* to *5* times slower.

As seen in Figure 5.2, the black-box fuzzer *wFuzz* is about *3* times faster than *webFuzz* in the case of *Drupal* and *5* times faster in *WordPress*.

In the case of *Drupal*, the instrumentation overhead alone introduces a factor of *2x* in the response time (as seen from Section 5.1). The post-processing steps taken by *webFuzz* for each request, together with any implementation differences between the two fuzzers added an additional *1x* factor to the overall overhead. With *WordPress*, the performance difference between the two fuzzers is even more accentuated as it introduces an additional factor of *2x* (if we deduct the average instrumentation overhead).

One reason for the performance difference in the two fuzzers is from the additional post-processing steps taken by *webFuzz* together with its live crawler functionality. Firstly *webFuzz* employs HTML parsing for each request in order to extract new URLs from anchor and form elements. These new links will then be filtered so as to identify only the unique new links (that were not already sent in the past). *wFuzz* on the other hand does not feature a live crawler and only applies simple string searching to identify XSS bugs in the returned document. In addition, *webFuzz* needs to analyse the coverage feedback and update the relative data structures as described in Section 3.2.2.

Another possible reason is that *wFuzz* can exploit the *opcache*, and the server and database resources more effectively than *webFuzz*. Because its work-flow consists of sending the same request *5000* before moving on to new fuzz targets, caching mechanisms in all levels will have a higher hit ratio than in *webFuzz*. For instance, the pages in InnoDB buffer pool [40] (caches table indexes, data and more) have a higher chance of being located in the CPU caches. The *opcache* in the *Zend* engine is more likely to receive a CPU cache hit for a needed bytecode. On the other hand, *webFuzz* changes its fuzz target rapidly (on average every 3-4 requests), thus the cache hit-ratio will suffer. We note here, that Page Faults (due to a full main-memory) should not have been a limiting factor as both caching mechanisms mentioned and the RAM of the Linux machines had enough available free space throughout the fuzzing session.

Ideas on how to optimise *webFuzz*'s internal mechanisms are discussed in Section 7.

5.4 Vulnerability Detection

The most crucial test for *webFuzz* is whether it can detect more RXSS vulnerabilities in a limited time frame than other black-box fuzzers. RXSS is a common web vulnerability [26], but apart from non-descriptive records in publicly known vulnerability databases such as CVE [13], there does not exist a standard test suite to evaluate web penetration tools [15]. A search for the term "XSS" in CVE produces 16500 results at the time of writing but these vulnerabilities are scattered across different projects and versions. To further complicate matters, each vulnerability is stated only as a reference guide and reproducible steps are not provided to preserve confidentiality. As a result identifying a project with sufficient RXSS vulnerabilities is a challenging task.

Nevertheless, conducting an evaluation on real-life vulnerable projects is possible as [3,4] have showed. To evaluate their PHP web vulnerability detection tools, they managed to identify *15* inactive or out-dated, known to be vulnerable PHP projects. The two tools combined managed to detect a total of *95* RXSS bugs spanning across these *15* projects, with a vulnerable application named *osCommerce 2.3.3* possessing the most RXSS bugs (*42* bugs). The disadvantage of this approach though, is that the bugs per project ratio is rather small (on average around *4* for the *15* projects). This would in turn mean that numerous experiments would need to be conducted to cover multiple projects, and with each project possibly requiring multiple repetitions. In addition, the actual number of RXSS bugs present in each application is not known, thus the detection capability of a fuzzer as a percentage of the total bugs present cannot be known.

For this reasons, we instead decided to create our own artificial RXSS bug injection tool, that can inject reasonably realistic RXSS bugs in the scale of hundreds in just a single project. Section 5.5.2 briefly describes the workings of our bug injection tool named *Centaur*. In total, we have injected 150 triggerable RXSS bugs in *WordPress* and have compared *webFuzz* and *wFuzz* on their detection capabilities in this modified *WordPress* CMS. Section 5.4 illustrates and discusses our findings.

Methodology

The methodology used to evaluate the two fuzzers in terms of RXSS detections firstly consisted of running the two fuzzers independently for 65 hours against two copies of the artificially bugged *WordPress* application. The two copies were identical with the only difference that one version was instrumented using the edge coverage policy. As it has been already mentioned, in total *150* artificial RXSS bugs were injected in each copy.

```php
<?php

if ( $_POST['v1'] % 10 === MAGIC_NUMBER % 10) {
   if ( $_POST['v2'] % 100 === MAGIC_NUMBER % 100) {
      # ...
      # more IF statements follow
      # depending on the number of digits in the Magic number
      # ...

         if ( $_POST['v1'] === MAGIC_NUMBER) {
            echo $_POST['v2'];
         }
   }
}
```

Listing 5.1: The Bug Template used in *Centaur* for this experiment. The two *POST* variables *v1, v2* are only provided for demonstration purposes. In actuality, the parameter names are retrieved from the URL that is currently being examined.

The bug template used with *Centaur* can be seen in Listing 5.1. It consists of a series of nested `if` statements, that lead to a vulnerable sink (the echo statement in line 11). It utilizes two URL parameters that are found from the link being examined (see Section 5.5.2 for more details). For a fuzzer to trigger the bug, it would need to guess the magic number and place it in variable *v1* and then further include an XSS payload in variable *v2*. To evaluate the coverage-guided mechanism of *webFuzz* we have further placed a series of nested `if` statements before the final `if` that leads to the bug. In this way, when *webFuzz* correctly guesses one digit from the magic number, it can receive coverage feedback (as this bug template is also instrumented), and thus prioritise the request that triggered it.

Analysis

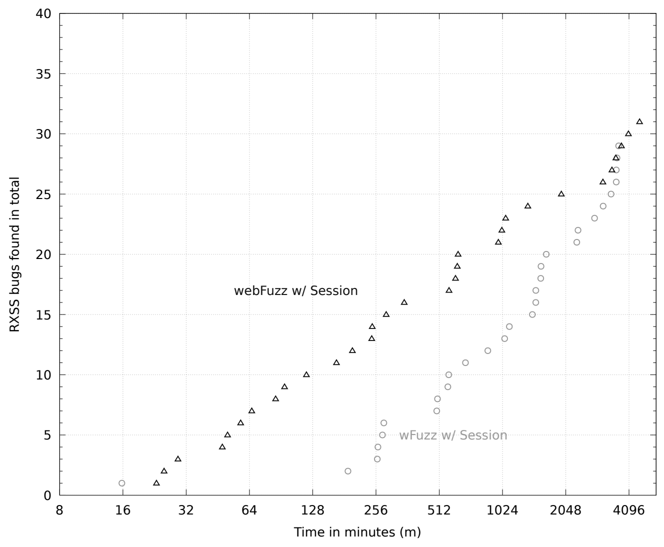


Figure 5.3: Artificial RXSS bug detections with *webFuzz* and *wFuzz*. *webFuzz* manages to uncover bugs faster than *wFuzz* from early on the fuzzing process.

The results of our 65-hour experiment comparing the fuzzers *webFuzz* and *wFuzz* in terms of artificial RXSS bugs detected is shown in Figure 5.3. Throughout the run, *webFuzz* is on the lead with their difference slowly decreasing over time. By leveraging the instrumentation feedback, *webFuzz* could find the artificial bugs faster than *wFuzz*'s brute force approach. Since correctly guessing a digit of a magic number–situated in a vulnerable payload–will trigger a new CFG edge, *webFuzz* will detect this change and will thus prioritize the request that causes it. With this method, the finding of a magic number is done incrementally, one correct digit at a time which is faster than guessing the whole number all at once (*wFuzz*'s approach). As a real-world analogy, each digit of the magic number can represent one correct mutation that gets us closer to the vulnerable sink.

5.5 External tools

This section discusses the external tools that have been used in some of our experiments. We firstly discuss how *wFuzz* was extended to better fit our needs, and then briefly describe the artificial RXSS bug injection tool we have used in the experiment for *RQ4*.

5.5.1 WFuzz

wFuzz is a black-box web application fuzzer, designed to work as a standalone tool and as an external library to further develop state-full fuzzers i.e. that exploit knowledge gained from previous requests. In its standalone form, it works in a state-less fashion, that is, previous test cases are not used in the succeeding test case generation process, and has the ability to fuzz HTTP headers, cookies, GET and POST parameters and more. With over 10 thousand monthly downloads [18], it is considered an effective web penetration tool.

It is an appropriate candidate to compare *webFuzz* with in our experiments for *RQ3-4*, as it (i) it is considered a black-box fuzzer, and (ii) can identify RXSS bugs.

wFuzz offers crawler-like functionality, that parses URLs found in HTML responses and enqueues them in its list of fuzzing targets. This functionality is limited though to certain types of URLs (such as from `href` attributes and `meta` HTML tags) as it does not employ full HTML parsing. For instance, HTML forms together with their `input` elements are omitted. To make a fair comparison between the two tools, we decided to extend its crawler functionality using *webFuzz*'s own crawler-related methods. Our aim is to evaluate the coverage-guided input mutation of *webFuzz* and not the crawling effectiveness.

In addition, since *wFuzz* requires explicit definition of the fuzzing payloads, we have further instructed *wFuzz* to use random strings and numbers, HTML, PHP, JavaScript syntax tokens and real-life XSS payloads similar to what *webFuzz* uses.

The resulting work-flow of *wFuzz* is as follows. First we run a crawler to identify the entry points (links) of the targeted web application. When the crawler finishes, we feed this list of fuzz targets to *wFuzz* and further instructed it to fuzz each URL target for a predefined number of times (*5000*) before moving on to another target. The links in the list are fuzzed repeatedly until a *SIGINT* signal is received.

5.5.2 Centaur

Our artificial RXSS bug injection tool named *Centaur*, relies on the coverage feedback as given by *webInstr* to find the appropriate location to inject a bug. It uses the node coverage instrumentation policy in which the actual executed basic blocks are given. It additionally utilizes a crawler to select entry points (URLs) in which to target. A simplified working of the algorithm is shown in Algorithm 3. The tool firstly crawls a targeted web application to identify its entry points. For each URL it finds, it runs the request and its execution trace showing the executed basic blocks is retrieved. The instrumentation policy additionally provides the depth that each block is located, i.e. in how many control statements a basic block is nested in. Given a basic block selection policy, it then iteratively selects a basic block, injects an RXSS bug given a bug template, and lastly tests that the bug is triggerable. This process it repeated a number of times for each link. We note here that the basic-block selection policy can be a user-defined method, such as random selection or based upon the block's depth. In addition the Bug Template is again user-provided, where each template requires a different format of XSS payload for it to be triggered. For instance, one template may require that the XSS payload contains quote characters for it to be executed whereas another require the `<script>` tag.

Algorithm 3 Simplified work-flow of *Centaur*

```
function BUGINJECTION($application, blockSelectionPolicy, bugTemplate$)
    $injectedBugs \leftarrow \{\}$
    $links \leftarrow crawl(application)$                              ▷ retrieve the entry points
    for $url$ in $links$ do
        $executionTrace \leftarrow run(url)$                         ▷ get the coverage feedback
        for $i \leftarrow 1$ to $n$ do                                 ▷ try inserting $n$ bugs per url
            $basicBlock \leftarrow selectBlock(executionTrace, blockSelectionPolicy)$
            $bug \leftarrow createBug(url.parameters, bugTemplate)$
            $injectBug(basicBlock, bug, application)$            ▷ inject bug in source code
            if $isTriggerable(url, bug)$ then                         ▷ can bug be triggered
                $injectedBugs.insert(url, bug)$
            else                               ▷ Remove the injected bug from application
                $revertChanges(basicBlock, bug, application)$
            end if
        end for
    end for
    return $injectedBugs$
end function
```

Chapter 6

Related Work

This section discusses the related work concerning instrumentation tools for measuring code coverage and in RXSS vulnerability detection scanners. In general, most research has been focused in applications written in native code while for web applications, specifically written in *PHP* which is the leading web development language, relatively little attention has been given by the research community.

6.1 Instrumentation

Native Applications

bcov provides instrumentation on the binary level for *x86_64 ELF* binaries without any compiler support [10]. It extends the data and code segment of the binary to accommodate the instrumentation data structures and code. Through the use of trampolines–a common technique found in native code instrumentation–once the execution flow reaches the beginning of a basic block, it is momentarily redirected to the injected code segment which records that the block has been observed. The control-flow will then be redirected back to its original course while making sure the program state (e.g. the register contents) are at their original state before the trampoline jump.

Similarly, Tikir et al. (2002) with their extensions developed for *dyninst*, provide dynamic, runtime instrumentation to binaries [45]. With the use of trampolines which are injected to specific areas called probes (carefully selected basic-blocks using optimisation techniques) and features like instrumentation code deletion they manage to reduce the instrumentation overhead significantly.

insTrim on the other hand is implemented as an *LLVM pass*, and provides an efficient instrumentation alternative to fuzzers such as *AFL* [20]. Their work achieves higher performance than current AFL instrumentation techniques, which are *InstrRand* (instrumenting a random subset of blocks) and *InstrAll*(instrumenting all blocks).

PHP Applications

Instrumentation tools for *PHP* applications do exist but with different use-cases. For instance, *XDebug*, is used for debugging and profiling purposes and utilizes instrumentation to provide node, edge and path coverage information back to the user [36]. Because it is designed as a debugging tool, primarily used for observing the code coverage in unit tests, it introduces overheads in the scale of *6x-120x* depending on the application and coverage policy used. Similarly, *PHPDBG*, a debugger shipped with *PHP* itself can provide node code coverage information but with similar overheads to that of *XDebug*. *pcov* on the other hand, is a faster alternative for measuring code coverage. Its overhead is comparable with that of *webInstr* but can only provide node coverage information [29]. All three tools are written as PHP extensions, that is, non-portable libraries written in the *C* language that directly interact with the API exposed by the underlying *PHP/Zend* Engine. They can be statically compiled inside the PHP binary itself or be dynamically loaded at runtime.

6.2 Web Fuzzing

Concerning black-box fuzzing, multiple tools have been developed throughout the years, which range from free open-source to subscription-based enterprise focused solutions. There also exists an extensive body of literature that analyses their effectiveness [8, 15, 22, 43, 47]. Doupé et al. report in their analysis of eleven prominent black-box vulnerability scanners, that automated software testing suffers from issues such as insufficient support for prevalent technologies such as *JavaScript* and unsophisticated crawlers. They conclude that plenty of research and progress still needs to be made [15]. More recent studies such as [22], show that many black-box fuzzers still suffer from the same issues. The vulnerability detection rates vary depending on the tool, the Program Under Test, and the amount of user configuration supplied. In general the detection rates are below 50% [8, 15, 22]

While most of the black-box fuzzers have been developed outside of academic scope, certain black-box fuzzers are more research aimed [16, 21]. *KameleonFuzz* for instance, is a black-box fuzzer that can detect reflected and stored XSS vulnerabilities [16]. It relies upon the pre-existing *LigRE* tool for producing a control+taint flow model from the targeted web application, and further builds upon on it by providing malicious input generation and double-taint inference. To generate test cases it uses a genetic algorithm and an attack grammar for each reflection context i.e. the location of the reflected input such as inside an HTML attribute or inside an HTML element. Its double-taint inference algorithm can accurately distinguish True and False positive test cases while also providing a means for ranking a test case (its fitness score). All though it requires manual specification of attack-grammars to achieve good performance, their approach has nevertheless been successful in finding XSS vulnerabilities.

There also exist multiple subscription-based, commercial web vulnerability scanners such as *Burb*, *Acunetix*, *AppScan*. These tools are highly feature-dense, can detect a large variety of web vulnerabilities including XSS and provide cloud-based monitoring. Their associated costs reflect this fact, as their business model is aimed towards enterprises.

Concerning web applications written in the *PHP* language, to our knowledge there exists one open-source project related to coverage-based fuzzing. Named *PHP-Fuzzer* [31], it is developed by the same author of *PHP-Parser* library that we internally use to implement *webInstr*. The instrumentation policy it uses is similar to our Edge coverage implementation with additional basic block stub inserted in clause expressions of control statements. Just like *webInstr*, no optimisation techniques such as probe pruning (discussed in Section 7) have been implemented. On the other hand, it relies on run-time instrumentation, where a file is instrumented as soon as it is included by a script to scope. Since the fuzzer is aimed at detecting run-time errors and not web vulnerabilities such as XSS, its feedback loop happens on the function-level instead on the HTTP level. Similar to *libfuzz* [42], the targeted application is an entry-point function, and the fuzzer is coupled with the function under test in one long running process. For this reason, it can benefit from techniques such as run-time instrumentation. For web fuzzing though where a server-client interaction is needed, run-time instrumentation would bring unnecessary overheads as instrumentation needs to be re-injected for each new request (unless inter-request caching is used which is not the case for *PHP-Fuzzer*).

Lastly, there also exist white-box web penetration tools that do not fall in the fuzzing category. For instance tools such [3, 4], rely on static analysis and constraint solving to identify vulnerable source-sink pairs. Backes et al. in their PHP aimed tool, rely on Code Property Graphs and on modeling of vulnerabilities as graph traversals [7]. They do not repeatedly test the PUT with inputs generated in an automated fashion but on the other hand carefully craft a bug largery using static analysis.

Chapter 7

Future Work

7.1 Instrumentation

As it can be seen from Section 5, the instrumentation overhead imposed to the targeted web application substantially increases the execution time, with Wordpress increasing it almost three-fold. It is thus essential to explore ways to reduce its performance footprint on the PUT.

Multiple research papers have explored innovative ways to decrease this overhead, with primary focus in native application instrumentation [1, 10, 20, 45]. In this section, we discuss how certain key ideas can be adapted to *webInstr*. We will first explore the path differentiation problem and how solving this can reduce the number of instrumented basic blocks significantly and then briefly outline other future improvements.

7.1.1 Path differentiation

The path differentiation problem also referenced as *probe pruning* in literature consists of finding the minimum set of basic blocks in a CFG, where instrumenting only these blocks still provides us enough information to distinguish all executions from each other.

Consider the CFG in Figure 7.1 of the function `bar` as defined in Definition 4. Only marking the basic blocks v_2, v_3 and v_4 can provide us the same information as instrumenting all the blocks in the function.

One possible approach to identify the minimal set of blocks involves generating the Pre-Dominator and Post-Dominator trees from a CFG [45]. Using just the Pre-Dominator tree can produce acceptable non-optimal solutions. The nodes in a Pre-Dominator tree are defined as the basic blocks just like in a CFG. For a directed edge to exist from node u to node v, then all execution paths that lead to basic block v must pass through basic block u first. If such a directed edge exists, then node u is said to pre-dominate node v. This would also mean that nodes pre-dominating node u also pre-dominate node v.

Definition 4 An arbitrary function `bar`

```
function BAR($x, y$)
    $result \leftarrow 0$                                  ▷ Block 0
    if $x > y$ then                                        ▷ Block 0
        $x \leftarrow x * 2$                               ▷ Block 1
        if $x \leq y$ then                                 ▷ Block 1
            $result \leftarrow x$                          ▷ Block 2
        else
            $result \leftarrow 2 * y$                      ▷ Block 3
        end if
    end if
    $result \leftarrow result * result$                    ▷ Block 4
    return $result$                                        ▷ Block 4
end function
```

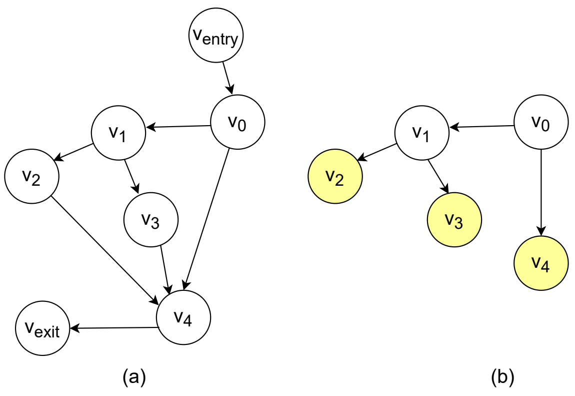


Figure 7.1: The CFG of function `bar` is seen in graph *a* and the corresponding pre-dominator tree from *a* is seen at *b*. Only nodes (basic blocks) v_2, v_3, v_4 need to be instrumented (the leaves of tree *b*).

In the case that the CFG is acyclic, a –possibly non-optimal– solution to the problem involves only selecting the leaf nodes of the Pre-Dominator tree. As seen in Figure 7.1 selecting the leaf nodes would actually give us the optimal solution in this case. On the other hand, cycles in the CFG create execution paths that cannot be differentiated if only the leaf nodes are selected. To overcome this problem, one solution is to additionally mark each node that has an outgoing edge in the CFG headed to a node that it does not pre-dominate [45].

Such work to optimise the instrumentation process has been implemented by [10, 20, 45]. Tikir et al. (2002) using on-demand instrumentation and Dominator Trees have observed reduction in blocks instrumented ranging from 42% to 79% [45]. Hsu et al. (2018) with their tool named *InsTrim* report up to 80% fewer basic blocks instrumented

[20]. Similarly, Khandra et al. (2020) based on the instrumentation policy used, their tool *bcov* instruments on average 46% and 30% out of the total basic blocks as identified on the binary level [10].

Adopting this approach to *webInstr* would require an accurate CFG generation tool. Backes et al. (2017), developed a static-analysis based tool that uses graph traversals on Code Property Graphs (CPG) [49] to identify code patterns and source-sink flows that expose vulnerabilities [7]. As one element of CPG are the Control flow graphs, they have showed that generating inter-procedural Control flow graphs (function-level CFG connected using their call sites) is possible in *PHP* all though with some caveats. Due to the dynamically typed nature of *PHP* and its lack of a strong type system that can infer an object's type statically, mapping each function and method call site to its declaration is not always trivial. For this reason, their inter-procedural CFG accuracy ranged around 88% [7].

We can infer that solving the path differentiation problem to an acceptable degree is possible in *PHP*, if we utilize the CFG generation tool provided by [7]. A simpler approach could also ignore call-sites and apply this optimization technique only on function-level CFG. This would still reduce the basic-block count but not to the same extend if we use inter-procedural CFG. Future work can expand on this and develop an efficient version of *webInstr*.

It is worthy of noting that such optimisations would introduce additional overhead in the application that receives the reduced coverage feedback. Additional book-keeping information and post-processing steps are needed. Each label (basic block) received in the execution trace must be assigned with its weight value, thus the need for additional data structures to hold such information. For instance, the weight of node v_3 in an execution trace where only v_3 is received has value of *4*, indicating that the path exercises four blocks, namely [v_0, v_1, v_3, v_4].

7.1.2 Additional Improvements

It can be useful for *webInstr* to provide an option to dump coverage data on demand. For instance, fuzzers usually kill long duration requests so as to avoid stalling. With on-demand dumping the coverage data would not be lost. Such feature would also not be hard to implement as it can be done by registering a signal handler for a user signal such `SIGUSR1` or `SIGUSR2`.

Concerning the basic block stub code, as seen in Listing 3.1, an initialisation check (using the coalesce operator) happens before every counter incrementation. Because *PHP* does not offer any substitute data structure or special instruction that can solve this problem with a single instruction the overhead adds up. On the other hand, simply recording whether an edge has been executed without keeping hit-counts can be implemented using a single instruction. For this reason more evaluation is needed to weigh the benefits of keeping counters for each edge. Migrating the instrumentation library as an internal PHP extension can possibly offer more performance improvements as well.

Lastly, extending the support to other server-side languages should be possible as long as some method of working with their AST, bytecode or native code exist. We have already experimented with instrumenting HACK programs on the AST level using the HHAST [17] library with success although more work needs to be done for it to be considered a complete instrumentation method.

7.2 Fuzzer

One of our main future goals is to improve our in-built crawler. Weak crawlers that cannot uncover JavaScript generated URLs, or are unable to pass through simple input validation barriers substantially hinder the performance of a fuzzer. Our evaluation showed that this was indeed the case for *Mautic* and *Firefly-III*. The problem of unsophisticated crawlers is a general problem affecting all security tools that target web applications. Due to the complex and heterogeneous nature of web applications, effectively crawling deep into the application structure while still ensuring genericness can be an almost impossible task to do [15,43].

Certain ideas can be adopted from existing work that tackles this problem. For instance, Alhuzali et al. (2018) in their tool *NAVEX* use a combination of static and dynamic analysis to maximise the explored state of an application. They utilise concolic execution, constraint solving and execution tracing. More specifically, in order to crawl deep into an application, it must find valid inputs to *HTML* forms that comply with the input restrictions (such as type and length) enforced by both the client-side *JavaScript* and server-side code. To tackle this, they create input constraints to be fed to a solver using the form's *HTML* attributes and from a hybrid concrete-symbolic execution of JavaScript client-side code. In addition, to pass the server-side input checks, they will additionally monitor the server's execution trace for accesses to external databases, changes to superglobal variables and more, to identify whether a request is successful. They resort to server-side concolic execution once again to find the right set of inputs [4]. All though heavy use of concolic execution can hinder the performance of the fuzzer, we can still utilize some HTML and JavaScript static analysis as such mechanisms would only need to be run *once* per JavaScript file and form element throughout the fuzzing session.

To decrease the number of false positives that occur during XSS detection, we plan on improving our current string searching method by taking into consideration the location of the payload in the document as well. Such work will allow us to detect cross-site scripting bugs that do not get triggered due to the HTML's <script> tag but on the contrary due to HTML attributes such as onload and onmouseclick. Additionally, any request that is flagged as vulnerable, can be be tested on a real browser environment to observe if it can truly be considered a true positive. The Selenium library can help automate this task.

We can additionally add support for cookie and in general HTTP header fuzzing since these can also be input sources in RXSS bugs. Many black-box fuzzers and white-box scanners do provide such options such as [7,24,37].

Regarding *webFuzz*'s overhead in the response post-processing step, certain performance critical code sections can be ported to *Cython*. *Cython* provides the ability to compile certain Python modules to *C* extensions via the use of C-like type annotations [9]. In this way, CPU heavy algorithms such as the `addRequest` method in Algorithm 2 that rely on Heap insertion and sorting can be statically compiled and embedded as C-extensions to the *CPython* runtime.

As a long-term goal we are also planning on introducing netmap [39], a tool to effectively bypass the Operating System's expensive network stack and allow the client and server to communicate efficiently.

Chapter 8

Conclusion

This thesis has explored the topics of web application instrumentation and its utilization in coverage-based web fuzzers. We have developed two tools *webInstr* and *webFuzz* respectively, and have evaluated them using a number of metrics.

Currently the instrumentation overhead introduced by *webInstr* is significant (around *2.5x*), but judging from other related work concerning code coverage measurement in *PHP* applications, it compares perfectly well. Nevertheless, there is plenty of room for improvement. For instance, we can implement probe pruning techniques as utilized by the majority of native application instrumentation tools and thus reduce the number of instrumented basic blocks up to 80% less.

Our fuzzer *webFuzz* managed to outsmart the prominent black-box fuzzer *wFuzz*, as in our experiments it could find more RXSS bugs and with a faster rate. The overall overhead of the coverage-based approach though introduced overheads in the range of *2x-5x*. In some input-validation heavy applications, it has also been unsuccessful in exercising much of their application logic. As it has been discussed in Section 7 there is still plenty of work to be done in improving the fuzzer, such as detecting input validation rules enforced by JavaScript, uncovering JavaScript generated URLs, optimizing the work-flow and internals of the fuzzer, and improving the accuracy of our XSS detector.

Bibliography

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–34, 1994.

[2] A. Aho, M. Lam, J. Ullman, and R. Sethi. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011.

[3] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652, 2016.

[4] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[5] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[6] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[7] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*, pages 334–349. IEEE, 2017.

[8] 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*, pages 332–345. IEEE, 2010.

[9] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, 2011.

[10] M. A. Ben Khadra, D. Stoffel, and W. Kunz. Efficient binary-level coverage analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1153–1164, 2020.

[11] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[12] O. Chang, A. Arya, K. Serebryany, and J. Armour. Oss-fuzz: Five months later, and rewarding projects, 2017. `https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html`.

[13] T. M. Corporation. Common vulnerabilities and exposures (cve), 2020.

[14] G. A. Di Lucca and A. R. Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[15] 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*, pages 111–131. Springer, 2010.

[16] 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*, pages 37–48, 2014.

[17] Facebook. Hhast. `https://github.com/hhvm/hhast`.

[18] Flynn. Pypi stats: wfuzz, 2020. `https://pypistats.org/packages/wfuzz`.

[19] A. Hoffman. *Web Application Security: Exploitation and Countermeasures for Modern Web Applications*. O'Reilly Media, 2020.

[20] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[21] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256, 2006.

[22] R. F. Khalil. *Why Johnny Still Can't Pentest: A Comparative Analysis of Open-source Black-box Web Vulnerability Scanners*. PhD thesis, Université d'Ottawa/University of Ottawa, 2018.

[23] LLVM. Sanitizer coverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`.

[24] X. Mendez. Wfuzz - the web fuzzer, 2011. `https://github.com/xmendez/wfuzz`.

[25] N. Milojkovic, M. Ghafari, and O. Nierstrasz. Exploiting type hints in method argument names to improve lightweight type inference. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 77–87, 2017.

[26] OWASP. Owasp top ten web application security risks, 2017. `https://owasp.org/www-project-top-ten/`.

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

[28] PHP. Opcache. `https://www.php.net/manual/en/intro.opcache.php`.

[29] PHP.WATCH. Php code coverage tools, 2020. `https://php.watch/articles/php-code-coverage-comparison`.

[30] N. Popov. Documentation on the visitor class in php-parser. `https://github.com/nikic/PHP-Parser/blob/master/doc/component/Walking_the_AST.markdown`.

[31] N. Popov. Php fuzzer. `https://github.com/nikic/PHP-Fuzzer`.

[32] N. Popov. Php parser. `https://github.com/nikic/PHP-Parser`.

[33] Python. Global interpreter lock, 2020. `https://docs.python.org/3/glossary.html#term-global-interpreter-lock`.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[35] J. M. Redondo and F. Ortin. A comprehensive evaluation of common python implementations. *IEEE Software*, 32(4):76–84, 2015.

[36] D. Rethans. Xdebug - code coverage, 2020. `https://xdebug.org/docs/code_coverage`.

[37] A. Riancho. w3af-web application attack and audit framework, 2011. `http://w3af.org/`.

[38] A. Rigo and S. Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, 2006.

[39] L. Rizzo and M. Landi. Netmap: Memory mapped access to network devices. *SIGCOMM Comput. Commun. Rev.*, 41(4):422–423, Aug. 2011.

[40] B. Schwartz, P. Zaitsev, and V. Tkachenko. *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.", 2012.

[41] S. M. Seal. Optimizing web application fuzzing with genetic algorithms and language theory. Master's thesis, 2016.

[42] K. Serebryany. libfuzzer–a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[43] L. Suto. Analyzing the accuracy and time costs of web application security scanners. *San Francisco, February*, 2010.

[44] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[45] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):86–96, 2002.

[46] I. T. Union. Itu world telecommunication/ict indicators database 2019. 2019.

[47] F. van der Loo. Comparison of penetration testing tools for web applications. Master's thesis, Master's thesis, University of Radboud, Netherlands, 2011.

[48] L. van Rossum, Lehtosalo. Pep484, 2014. `https://www.python.org/dev/peps/pep-0484/`.

[49] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[50] M. Zalewski. Binary fuzzing strategies: what works, what doesn't, aug 2014. `https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html`.

[51] M. Zalewski. American fuzzy lop trophy case, 2015.

[52] M. Zalewski. More about afl - afl 2.53b documentation, 2019. `https://afl-1.readthedocs.io/en/latest/about_afl.html`.