webFuzz: Grey-Box Fuzzing for Web Applications

Orpheas van Rooij(✉), Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos

University of Cyprus, Nicosia, Cyprus
{ovan-r01,mchara01,dkaize01,mpapae04,eliasathan}@cs.ucy.ac.cy

Abstract. Fuzzing is significantly evolved in analysing native code, but web applications, invariably, have received limited attention until now. This paper designs, implements and evaluates webFuzz, a *gray-box fuzzing* prototype for discovering vulnerabilities in web applications.

webFuzz is successful in leveraging instrumentation for detecting cross-site scripting (XSS) vulnerabilities, as well as covering more code faster than black-box fuzzers. In particular, webFuzz has discovered *one* zero-day vulnerability in WordPress, a leading CMS platform, and *five* in an online commerce application named CE-Phoenix.

Moreover, in order to systematically evaluate webFuzz, and similar tools, we provide the first attempt for automatically synthesizing reflective cross-site scripting (RXSS) vulnerabilities in vanilla web applications.

1 Introduction

Automated software testing, or fuzzing, is an established technique for analyzing the behaviour of applications, and recently has been focused, among others, in finding unknown vulnerabilities in programs. The drive to discover bugs in software through an automated process has progressed with the introduction of American Fuzzy Lop (AFL) [47], a state-of-the-art fuzzer that leverages the coverage feedback from the *instrumented* target program. In creating this *feedback loop*, fuzzers can significantly improve their performance by determining whether an input is interesting, namely, it triggers a new code path, and use that input to produce other test cases.

Although automated software testing has become a burgeoning field of research, it still has a long way to go, especially for web applications [17]. On the other hand, the proliferation of the web attracts many more malicious attacks on web applications. This predicates a strong need for the development of automated vulnerabilities scanners that target web applications as well as for automated vulnerabilities injection tools to evaluate the former.

Traditionally, fuzzers come under three categories; black-, white- or grey-box depending on the level of awareness they have of the target program's structure. Black-box fuzzers are unaware of the internal program structure, that is, their

E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12972, pp. 152–172, 2021.
https://doi.org/10.1007/978-3-030-88418-5_8

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 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 is a key idea for successful bug discovery [13,47]. Black-box fuzzers though lack the ability to make sound judgements on what is considered interesting input [35,39]. As a result they either do not retain generated inputs for further mutation or the heuristic used to identify favorable inputs is insufficient as it can only rely on what is observable in the application response. This limits the effectiveness of black-box solutions.

On the other end of the scale exist the white-box solutions that require access to source code and rely heavily on static-analysis. Most of these approaches utilize constraint-solving and a combination of symbolic and concolic execution [3,4,6] to identify vulnerable code statements. Although sophisticated in their approach, their inherent limitation lies in the computationally demanding constraint solver. For instance, tools such as Chainsaw [3] and Navex [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 along the way is the key ingredient in 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.

Coverage-based grey box fuzzing comes at an ideal compromise between sophistication and scalability. Instead of statically analysing the source code, it relies on input mutations and lightweight coverage feedback to explore the input-space within a limited time frame. Given that the input-space can be enormous, a fuzzer can leverage the instrumentation feedback to identify interesting inputs and built new test cases from them. 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 is on a global or function level [12], thus increasing the chances of triggering a vulnerability.

In this paper, we design, implement and evaluate a coverage-based *grey-box* fuzzing solution for web applications aimed at detecting reflective and stored cross-site scripting vulnerabilities. For native applications, instrumentation is carried out at the intermediate representation of the application's code (e.g., at the LLVM's IR), when the source code is available, or directly to the binary [47]. For web applications, instrumentation is challenging, since (a) several different frameworks are used to realize web applications, (b) applications are executed through a web server and (c) there is no standard intermediate representation of web code. webFuzz applies all instrumentation at the abstract-syntax tree layer of PHP applications. Therefore, our instrumentation can cover a significant

amount of available web code, while it is generic enough – labeling basic blocks, collecting feedback, and embedding the feedback to a shared resource can all be translated to other web languages without facing many challenges.

Evaluating fuzzing is another challenging task [28], since migrating known vulnerabilities to existing software, in order to test the capabilities of the fuzzer in finding bugs, can be a tedious process [32]. Thus, for evaluating webFuzz, but also other fuzzers for web applications, we develop a methodology for automatically injecting bugs in web applications written in PHP. Our methodology is inspired by LAVA [15] and targets web applications instead of native code. Injecting vulnerabilities in web code, again, is challenging, since important tools used for analyzing native code and injecting vulnerabilities (e.g., taint-tracking and information-flow frameworks) are not available for web applications. To overcome this lack of available tools, our vulnerability injection methodology leverages the instrumentation infrastructure we use for building webFuzz, in the first place.

1.1 Contributions

In this paper, we make the following contributions.

1. We design, implement and evaluate webFuzz, a prototype grey-box fuzzer created for discovering vulnerabilities in web applications. We thoroughly evaluate webFuzz in terms of efficiency in finding unknown bugs, of code coverage and throughput. Indicatively, webFuzz can cover about 27% of WordPress (almost half a million LoCs of PHP), in 1.4 days (2,000 min) of fuzzing. It has additionally managed to uncover *1* zero-day RXSS bug in the latest version of WordPress, and *5* zero-day XSS bugs in an active commerce application named CE-Phoenix. Compared to Wfuzz, a prominent open-source black-box fuzzer, webFuzz also finds more real-life and artificial XSS bugs.
2. We design and implement a methodology for automated bug injection in web applications written in PHP. Our bug-injection methodology is *not only* essential for evaluating webFuzz but also vital for the progression of further research in vulnerability finding for web software.
3. To foster further research in the field, we release all of our contributions, namely the toolchain for instrumenting PHP applications, the actual fuzzer, and the toolchain for injecting bugs in web applications, as open-source. The source code can be found at: https://bitbucket.org/srecgrp/webfuzz-fuzzer

2 webFuzz

This section describes the architecture of webFuzz. We begin by discussing how webFuzz instruments a target application before analysis, and then elaborate on how a fuzzing analysis works by expanding on how inputs are mutated, responses are analyzed and vulnerabilities are detected. We also discuss how we minimize fuzz targets in order to favor the ones that lead eventually to better code coverage.

2.1 Instrumentation

webFuzz instruments PHP applications in the Abstract Syntax Tree (AST) level of the code to provide coverage information in the form of *basic block* or *branch coverage* [5]. To achieve this, it parses the AST of each PHP source file using the PHP-Parser [36] library, it identifies basic blocks during the AST traversal process, and lastly appends to each block identified extra stub code. In order to output the coverage feedback at the end of the execution, stub code is also inserted at the beginning of every source file. We elaborate more on this mechanism, below.

Instrumentation Level. Basic blocks are maximal sequences of consecutive statements that are always executed together (i.e., contain no branching statements) [2]. In a Control Flow Graph (CFG), the basic blocks correspond to the nodes in the graph. In order to measure code coverage, it is sufficient to identify the executed blocks and their order of execution. Identification of the blocks happens during the AST traversal process, in which the type of each statement encountered is inspected and modified accordingly. webFuzz identifies the beginning of a basic block as:

- the first statement in a function definition;
- the first statement in a control statement;
- the exit statement in a control statement.

Control statements are all conditional, looping and exception handling constructs. Although expressions with logical operators can also be composed out of multiple basic blocks due to short-circuit evaluation, for the sake of performance webFuzz does not instrument them.

Coverage. Basic-block coverage provides the least granularity as it only measures which basic blocks get executed irrespective of their order. Branch coverage enhances the accuracy by measuring the pairs of consecutive blocks executed. The latter method is also known as *edge coverage* method, as one pair corresponds to an edge in the CFG of a program. We further focus on the implementation of the most complex case, namely the edge coverage.

Inspired by the AFL [47] method of providing edge coverage information, we have adapted a similar approach for web applications. Listing 1.1 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. The edge label is used as index in the global `map` array where the counter for the edge is incremented.

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 not contain control statements in their body [47]. 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"];
    $GLOBALS["_instr"]["map"][$_key] += 1;
    $GLOBALS["_instr"]["prev"] = BASIC_BLOCK_A_LABEL >> 1;
    # END instrumentation code

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

        $z = 1
    }
    # BEGIN instrumentation code for basic block C
    # $_key represents Edge A->C or B->C
    $_key = BASIC_BLOCK_C_LABEL ^ $GLOBALS["_instr"]["prev"];
    ...
```

Listing 1.1. Sample of an instrumented function `foo` for measuring edge coverage.

Feedback. Coverage information is reported at the end of program execution. To achieve this, we prepend to every source file additional header stub code that will be the first statements executed in a program. We additionally utilize the inbuilt function `register_shutdown_function`, where a custom function is specified that will automatically get called at the end of the program. In this header stub, a function is registered that will write the resulting `map` array to a file, HTTP headers or shared memory region and the instrumentation data structures are initialised. Listing 1.2 shows the header stub code for outputting coverage information in a file. webFuzz will place this stub after any `Namespace` and `Declare` statements present in the source file, as PHP dictates that these must be the top-most statements. The header stub follows that will only be called once during program execution (guarded by the enclosing `if`) and the remaining statements in the source file come next.

```php
<?php
# Namespace and Declare statements of source file
if (! array_key_exists('_instr', $GLOBALS)) {
    $GLOBALS['_instr']['map'] = array();
    $GLOBALS['_instr']['prev'] = 0;
    function instr_out() {
        $f = fopen("/var/instr/map." . $_SERVER["HTTP_REQ_ID"], "w+");
        foreach ($GLOBALS["_instr"]["map"] as $edge => $count) {
            fwrite($f, $edge . "-" . $count . "\n");
        }
        fclose($f);
    }
    register_shutdown_function('instr_out');
}
# Remaining source file statements follow
...
```

Listing 1.2. The instrumentation header stub inserted at the beginning of each source file. This particular stub will write the coverage report in a file.

2.2 Fuzzing Analysis

webFuzz is a mutation based fuzzer that employs incremental test case creation guided by the coverage feedback it receives from the instrumented web application. It additionally features a dynamic builtin crawler, a proxy for intercepting browser requests, easy session cookie retrieval and a low false-positive XSS detector. We expand on all of these aspects below.

Workflow. A fuzzing session consists of multiple workers continuously sending requests and analysing their responses. The high-level process that each worker performs is as follows. Initially, webFuzz fetches an unvisited GET or POST request that has been uncovered by the builtin crawler. If no such request exists, webFuzz will create a new request by mutating the most favorable previous one. It then sends the request back to the web application and reads its HTTP response and coverage feedback. Furthermore, webFuzz parses the HTML of the response to uncover new links from `anchor` and `form` elements and scans the document for XSS vulnerabilities. Finally, if the request is favorable (deduced from its coverage feedback) webFuzz computes its rank and stores it for future mutation and fuzzing.

The fuzzing session can be run as an authenticated user by spawning a browser window for the user to login and for webFuzz to retrieve the session cookies.

Mutation. Mutating inputs 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 [46]. Conversely to many fuzzers that employ malicious payload generation via the use of a specific attack grammar [18,39], webFuzz takes a mutation-based approach [37]. It starts with the default input values of an application (e.g., specified by the value attribute in a HTML input element), and performs a series of mutations on them. Currently five mutation functions are employed which modify the `GET` and/or `POST` parameters of a request. They are as follows.

- Insertion of real-life XSS payloads found in the wild;
- Mixing `GET` or `POST` parameters from other favourable requests (in evolutionary algorithms this is similar to *crossover*);
- Insertion of randomly generated strings;
- Insertion of HTML, PHP or JavaScript syntax tokens;
- Altering the type of a parameter (from an `Array` to a `String` and vice versa).

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.

Proxy Server. Due to the heavy utilization of client-side code in most web applications, multiple URL links are generated on the fly by the browser's JavaScript engine instead of being present in the HTML response of the server. Since webFuzz only searches for new URL links in the HTML response and does not execute any client-side code, such JavaScript generated URLs will be missed from the fuzzing process. To solve this issue, webFuzz provides an option to initiate a proxied session before the fuzzing process begins, where the web application loads in a web browser environment and the user is given the ability to exercise the functionality of the web application manually by submitting new requests. As soon as the proxied web browser session is closed by the user, webFuzz retrieves all the URL links send during this session, adds them as possible fuzz targets and begins the fuzzing process.

Dynamic Crawling Functionality. Every HTML response received from the web applications is parsed and analysed in order to effectively crawl the whole application. HTML parsing is performed using the lenient *html5lib* [24] library which adheres to the HTML specification thus ensures similar results with that of web browsers. Using the parsed result, webFuzz can dynamically extract new fuzz targets from links in `anchor` and `form` elements, while also retrieve inputs from `input`, `textarea` and `option` elements. The crawler module additionally filters out any previously observed links to avoid crawling the same links repeatedly.

Vulnerability Detection. webFuzz is currently designed to detect stored and reflective cross-site scripting bugs produced by faulty server-side code. To ensure a low false positive rate webFuzz utilizes lightweight JavaScript code analysis. To identify whether a link is vulnerable, JavaScript code present in the HTML responses is parsed to its AST representation using the *esprima* [19] library. As every HTML document is parsed, identifying the executable HTML elements and attributes is trivial. webFuzz will look for code in the following locations in the HTML.

- Link attributes (e.g., `href`) that start with the *javascript:* label;
- Executable attributes that start with the *on* prefix (e.g. `onclick`);
- Script elements.

The XSS payloads webFuzz injects to `GET` and `POST` parameters are designed to call the JavaScript `alert` function with a unique keyword (having a fixed prefix) as input. The goal for the detector is to find the corresponding `alert` function call during the AST traversal process. If such a function call exists, it can infer that the XSS payload is executable, thus proving that the responsible link is vulnerable.

Additionally, in order to pinpoint which link triggers a found vulnerability, webFuzz will search in the request history and look for the request that contained the unique keyword present in the executable `alert` function call. Since this request history bookkeeping is a memory intensive process, webFuzz provides an option to limit the history size up to a maximum size.

Culling the Corpus. The majority of the mutations performed on requests are unsuccessful at triggering new code paths. It is thus essential to shrink the

corpus of fuzz targets and store only the most favorable. In this way, webFuzz can reduce its memory footprint and ensure test case diversity.

Algorithm 1. Algorithm to decide whether a new request will be kept for future fuzzing

```
function AddRequest(hashTable, heapTree, newRequest)
    for (label, hitCount) in newRequest.coverage do
        bucket ← floor(log2(max(hitCount, 128)))

        existingRequest ← hashTable[blockLabel][bucket]
        if existingRequest == ∅ then
            hashTable[blockLabel][bucket] ← newRequest

        else if newRequest is lighter than existingRequest then
            hashTable[blockLabel][bucket] ← newRequest

            if existingRequest ∉ hashTable then
                remove existingRequest from HeapTree
            end if
        end if
    end for
    if newRequest ∈ hashTable then
        add newRequest to HeapTree
    end if
end function
```

Algorithm 1 shows the *AFL*-inspired algorithm that determines if a new request is kept for future fuzzing or is discarded. The coverage feedback of a new request is checked against a hash table that holds information about all the instrumentation points (labels) that have been observed. Each entry in the table is split to *8* buckets, with each bucket corresponding to the number of times the label was executed in a single run. At each label-bucket entry, the lightest request that triggered this combination is stored. The use of buckets aims to distinguish requests that have executed a label a few times versus triggering it many more times [47].

When the hit-count of a label falls in a bucket that has already been observed, there is a clash for the same bucket in the same entry in the dictionary. The two requests are compared in terms of execution time and request size and the lightest of the two gets the entry.

As soon as a request has no longer entries in the table it is removed from the fuzzing session. This is done by removing it from an internal heap tree that stores the available fuzz targets in a semi-ordered fashion.

On the other hand, if a request has successfully acquired at least one entry in the hash table, it is inserted to a heap tree that contains all the favorable requests. This tree is consulted every time webFuzz has run out of new unvisited

links, and thus requests the most favorable previous request to mutate. To rank requests we calculate a weighted difference score 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;
- **Mutated Score (+)**: approximation on the difference of code coverage with its parent request (the request it got mutated from);
- **Sinks Present (+)**: whether injected GET or POST parameters managed to find their way in the HTML response;
- **Execution Time (-)**: round-trip time of the request;
- **Size (-)**: total number of characters in the `GET` and `POST` parameters;
- **Picked Score (-)**: the number of times it has been picked for further mutation (this ensures that all requests in the heap tree will eventually be fuzzed).

3 Bug Injection

In this section we discuss a technique for injecting synthetic bugs in PHP. Such synthetic bugs can be useful not only for evaluating webFuzz, but also other bug-finding techniques for web applications. Our methodology is inspired by LAVA [15], a tool which is widely used to inject bugs into native code. Web vulnerabilities are different from memory-corruption ones, however the underlying mechanics of LAVA are still useful. In practice, we can inject hundreds of common web vulnerabilities, such as reflective cross-site scripting, and file inclusion, in a reasonable time period.

The artificial bugs should have several characteristics. They must be easy to inject, be plenty, span throughout the code, and be triggerable using only a limited set of inputs [15]. The vulnerability should also be injected in appropriate places by adding new code or modifying existing one. Lastly each injected bug should come with an input that serves as an existence proof.

3.1 Analysis and Injection

Using our custom instrumentation toolkit, we analyze the PHP source code dynamically to identify potential basic blocks and input parameters suitable for bug injection. Static analysis techniques using control flow and data flow graphs is also an option, however it would be more challenging due to the dynamically typed nature of PHP.

```
<?php
if ($_POST['v1'] == 2) {
    # Basic Block A
    if ($_POST['v2'] == 3) {
        # Basic Block B
        ...
    } else {
        # Basic Block C
        ...
    }
} else {
    // Basic Block D
    ...
}
?>
```

Listing 1.3. Example of PHP code where different set of inputs trigger different blocks.

In PHP, users' input is largely given using the *super-global* arrays the language provides, such as `_POST`, `_GET` and `_REQUEST`. By definition these variables contain the users' input unmodified. Discovering input variables that do not alter the execution path taken by a request is useful, as those variables can be used to synthesize an artificial bug.

For instance, in Listing 1.3 we present a possible pattern that can occur inside a PHP program. Specifically, variable `_POST['v2']` can be used to synthesize a bug in basic block D as its value is not used inside the branch decisions to reach the block. On the contrary, in basic blocks A, B and C, the variable `_POST['v1']` cannot be used to synthesize a bug inside these blocks, as any changes to the variable value would cause the blocks to not get executed. For this reason, finding the right input parameters and basic blocks to inject a bug into requires analyzing the branch decisions that reach it.

Due to the complexity in performing such analysis, our tool instead relies on a random, trial and error technique to determine the right input and basic block pairs to inject a bug. The bug injection process is as follows. The tool firstly crawls the web application and finds all the unique URLs. For each URL, it will extract the input fields from HTML forms and URL parameters, together with all the basic blocks triggered from this request. Using some block selection criteria (e.g. basic block nested level), it selects one of the blocks reported and it inserts a *candidate* bug in its location in the source code. The same request is sent to the web application with the input parameters modified to trigger the candidate bug, and if the bug is triggerable it is then considered a working artificial bug. This can only happen if the input parameters used in the artificial bug still trigger the execution path to the selected basic block.

3.2 Bug Template

Each candidate bug is created from a bug template that represents its general structure. The template can consist of one to three input fields from either `GET` or `POST` parameters. An example of a template is shown in Listing 1.4.

```
<?php

if ( ($_POST['v1'] % 10) == (MAGIC % 10) ) {
    if ( ($_POST['v1'] % 100) == (MAGIC % 100) ) {
        if ( ($_POST['v1'] % 1000) == (MAGIC % 1000) ) {
            # Vulnerable Sink
            # Examples:
            # 1) echo $_POST['v2']
            # 2) print '<div class="' . $_POST['v2'] . '"></div>'
            # 2) include $_POST['v2']
        }
    }
}
```

Listing 1.4. Construction of a bug using two variables. The exploiting input must contain the magic value in v1 and the bug payload in v2.

The *MAGIC* value in Listing 1.4 is a random number that needs to be guessed to execute the bug. This magic number must be placed in POST variable `v1` and the bug payload in variable `v2`.

The format of the template is designed to mimic how a real-world bug may be hidden inside a deeply nested block. The stacked `if` statements also help in the process of guessing the magic number by rewarding with a new basic block (i.e. increase in code coverage) every time a correct digit is found. A bug finding tool that has access to information on the executed basic blocks can identify which inputs are closer to executing an artificial bug.

The aim of the stacked `if` approach is similar to the *value profile* feature provided by libfuzzer [40], with the difference that libfuzzer inherently supports instrumenting comparison instructions whereas webFuzz requires explicit unrolling of the comparison instruction as a series of stacked `if` statements.

The vulnerability the sink aims to introduce ranges from cross-site scripting bugs due to unsafe use of `echo` and `print` statements to `include` statements for file inclusion bugs. A plethora of different sink formats are also available which can further test the bug finding tool's ability in generating the correct bug payload format. Some examples for cross-site scripting bugs include unsafe code inside JavaScript block and unsafe code inside HTML attributes.

Using our bug injection tool, we have added numerous triggerable bugs in popular web applications, such as WordPress. In Sect. 4 we evaluate webFuzz's performance in finding these bugs.

4 Evaluation

In this section we evaluate webFuzz and the bug injection methodology. Our evaluation aims to answer the following research questions (RQ).

- **RQ1** 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)?
- **RQ2** What is the combined overhead of our solution (instrumentation and webFuzz processing) in terms of throughput, and how does this compare with black-box approaches?

– **RQ3** Can webFuzz detect more artificial and real-life XSS bugs in comparison to other fuzzers within a limited time frame?

For *RQ1* we measure how the accumulated code coverage develops in our test subjects for a fuzzing session lasting 35 h in each. The experiment is thoroughly analysed in Sect. 4.1. For *RQ2-3*, we compare webFuzz with Wfuzz [31], one of the most prominent open-source black-box fuzzers, and we present the results in Sects. 4.2 and 4.3. Finally, we use the following web applications for all experiments: (1) CE-Phoenix 1.0.5 (e-commerce application), (2) Joomla 3.7.0 (CMS), (3) WackoPicko (a web application with known vulnerabilities), (4) WeBid 1.2.1 (online auctions application), and (5) WordPress 4.6.19 (CMS).

All experiments are executed on four identical Ubuntu 18.04 LTS Linux machines that possess a quad-core Intel Xeon W-2104 Processor @3.20 GHz and 64 GB of RAM. In total, we have spend around *1000* computational hours in running our experiments.

The amount of manual effort needed in setting up the experiments varies depending on the application. Applications such as WeBid and CE-Phoenix are more susceptible to damage caused by sending unexpected input and thus all sensitive endpoints need to be firstly identified and blocked from the fuzzing process. Additionally, webFuzz has a number of tunable parameters such as the weight of each attribute when calculating the request's rank and the weight of each mutation strategy. Although webFuzz comes with sane defaults for these values, we have tweaked these parameters in each of our experiments to increase the throughput and code coverage but at the cost of more manual effort.

4.1 Code Coverage

Code coverage is an important metric for a fuzzer, as higher coverage entails higher chances of triggering vulnerabilities. Thus is justifiable as to trigger a given bug the fuzzer must be able to reach the associated code path where the bug lies. In addition, code coverage can provide feedback on the effectiveness of the mutation functions employed and, in particular, whether they trigger new code paths.

Methodology. All *five* web applications are instrumented using the hybrid node-edge method, that provides coverage information for both the basic-blocks and the pairs of consecutive blocks (edges) executed. Since the actual number of all possible edges is not known (as that requires CFG generation), we can use the block count to get an estimate on the code coverage as a percentage. In addition, we run the proxy feature as described in Sect. 2.2 in all *five* projects to include the JavaScript generated URLs in the list of fuzzing targets.

Analysis. Figure 1 shows how the accumulated code coverage progresses with time in the five applications. webFuzz managed to trigger more than 20% of the total basic blocks in all five applications, with CE-Phoenix managing to reach code coverage as high as 34% and with WackoPicko reaching 41%. The lenient and sometimes non-existent input validation rules with CE-Phoenix makes the

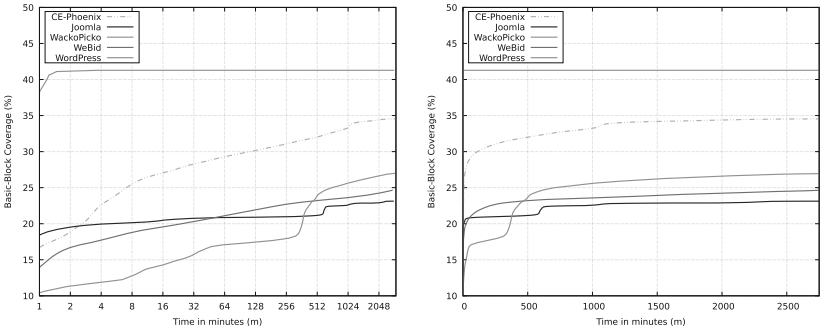

Fig. 1. Accumulated Basic-Block coverage achieved with webFuzz in *five* web applications. The figure on the left shows the coverage in logarithmic time scale, and on the right in linear time scale.

fuzzing process particularly effective as seen from the high code coverage. webFuzz can mutate form inputs with much freedom without requiring the input to adhere to stringent validation rules. The minimalistic nature of the WackoPicko app on the other hand, allowed webFuzz to quickly crawl the application within the first few minutes, while the input mutations only slightly increased the code coverage with time.

With WordPress, at around 350 min in the fuzzing session the mutation process kicks in as webFuzz finished the initial crawling process. The code coverage is seen to take a steep increase, indicating that the mutation functions employed are largely effective in triggering new code paths. A similar pattern can be observed with Joomla, albeit on a smaller scale. Long after the initial crawling finished, at 600 min in, the mutation functions manage to trigger multiple new basic blocks increasing the code coverage by 1%.

The code coverages reached by webFuzz is comparable to that of native application fuzzers such as AFL, AFL-Fast, TortoiseFuzz, FairFuzz and MOPT. Wang et al. have measured the code coverage of seven native fuzzers in 12 applications in their evaluation of TortoiseFuzz [44]. Their work showed that the average coverage reached for a application ranged from 4.55% ± 1.26 up to 76% ± 0.3 and with an average coverage throughout all fuzzers and applications being at 27.4%. webFuzz in the five test subjects tested reached 32.8% on average.

4.2 Throughput

One reason for the effectiveness of fuzzers in discovering vulnerabilities is their ability to test millions of inputs in a short time frame. For this reason we conduct an experiment to test the difference in throughput (requests/sec) between the black-box fuzzer Wfuzz and webFuzz.

Methodology. We measure the throughput of the two fuzzers during fuzzing sessions lasting 1 h for each web application. We further measure how the

throughput varies with different worker counts, i.e., number of parallel connections used. webFuzz uses the instrumented whereas Wfuzz uses the original versions of the web applications. In addition, because Wfuzz requires explicit definition of the fuzzing payloads to use, we have instructed it to generate XSS payloads in a similar fashion to webFuzz. Lastly, due to Wfuzz's limited crawling functionality, we assist the fuzzer by crawling each web application using our tools. We then instruct Wfuzz to fuzz each link in the crawler's result in a round-robin fashion.

Table 1. Average and Maximum throughout achieved with webFuzz and Wfuzz using different worker counts. Each cell states the throughput (requests/sec) at the particular scenario.

Fuzzer workers	webFuzz						Wfuzz					
	4		8		16		4		8		16	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
CE-Phoenix	83	141	70	183	62	182	133	392	137	446	133	453
Joomla	36	50	46	74	45	74	272	896	61	174	63	250
WackoPicko	52	260	45	170	128	152	480	976	491	1008	496	1004
WeBid	29	75	26	89	18	81	143	418	43	57	27	30
WordPress	8	33	8	34	10	36	17	142	72	194	62	176

Analysis. In Table 1 we see the average and maximum throughput achieved by the two fuzzers. Depending on the size of the web application, the bottleneck factor that limits the throughput is either the web server or the fuzzer. With WeBid, a relatively small project, we notice with both fuzzers a slight decrease in throughput when increasing the number of workers. The bottleneck in both situations is the fuzzer, where due to their single-core nature, the core is utilized at the fullest with just 4 worker counts, thus not benefiting from higher worker counts. On the other hand, WordPress benefits from higher worker counts due to its large project size. The average round-trip time is relatively high, thus both fuzzers can benefit from more workers as more requests are sent in parallel and the fuzzing process does not stall.

In general, the introduction of instrumentation, HTML parsing, and coverage analysis takes a toll on the performance, as Wfuzz is seen to reach throughputs up to ten times higher than webFuzz. Section 4.3 shows that this overhead is outweighed by the improved detection speed of webFuzz.

4.3 Vulnerability Detection

The crucial test for webFuzz is whether it can outperform black-box fuzzers in detecting XSS vulnerabilities. To test this, we fuzz the set of web applications using webFuzz and Wfuzz and we compare the findings. To further evaluate

our bug injection tool, we artificially inject XSS vulnerabilities in the *five* web applications and measure the number of artificial bugs found by both fuzzers.

Methodology. We fuzz every web application for at least 50 hours with each fuzzer. We again assist Wfuzz in crawling and payload selection as described in Sect. 4.2.

Table 2. Number of real-life (known from CVE records) and artificial XSS bugs found with webFuzz and Wfuzz. Each cell states the number of bugs found in respect to the total bugs present. The zero-day bugs found by each fuzzer are additionally stated.

Fuzzer	webFuzz			Wfuzz		
Bug type	Zero-day	Real-life	Artificial	Zero-day	Real-life	Artificial
CE-Phoenix	5	14/15	23/541	1	13/15	2/541
Joomla	0	1/32	2/64	0	0/32	0/64
WackoPicko	0	3/3	2/48	0	3/3	0/48
WeBid	1	7/7	3/72	1	7/7	0/72
WordPress	1	2/4	3/241	0	1/4	3/241

Analysis. Table 2 shows the results of our experiments. webFuzz outperforms Wfuzz in all five applications. webFuzz additionally manages to find *zero-day* XSS bugs in the latest versions of WordPress and CE-Phoenix. We have reported all found vulnerabilities to their developers, they have acknowledged our findings, and we have been awarded $500 from WordPress for our responsible disclosure. The official WordPress bug report is Report 1103740 and will be publicly disclosed as soon as a bug fix is available.

Beginning with CE-Phoenix, both fuzzers manage to uncover multiple bugs. In total webFuzz found *14* real XSS bugs and *5* zero-day bugs which were not listed in CVE records. Interestingly, webFuzz has found more XSS bugs in this project than Wfuzz. One reason for this lies in the request ranking mechanism employed by webFuzz that prioritizes requests that have high code coverage and contain sinks. As a result, the vulnerable URLs receive more fuzzing time than in the round-robin approach used by Wfuzz.

The input validations and heavy utilization of client-side JavaScript code in Joomla has proven to be an obstacle in the fuzzing process as webFuzz found just one out of the 32 real XSS bugs present.

In the WackoPicko application, both fuzzers manage to find all three XSS vulnerabilities that it contained. The required XSS payload structure is relatively simple so the main feature needed to find the bugs is a good crawler. For this reason, if Wfuzz was not extended with additional crawler functionality, it would not be able to find the multi-step XSS present in the application.

Continuing with WeBid, this inactive project was known to contain *7* XSS vulnerabilities from the CVE [14] records on it. Since the complexity of the real

bugs present in this project is relatively simple – required simple payload format and little crawling – both fuzzers manage to find all known real-life XSS bugs and an additional unlisted XSS bug.

With WordPress, we enable four third-party plugins that each contained one XSS vulnerability. webFuzz uncovers *two* out of the *four* plugin bugs, as the other two bugs require complex JSON formatted XSS payload or a GET parameter not present in the HTML responses.

The *zero-day* RXSS bug in WordPress consisted of finding a vulnerable GET parameter in a link and inserting a specially formatted payload to effectively bypass any sanitization steps. Two features employed by webFuzz have accelerated the finding of this bug, which are the source-sink identification and the executed basic block count. Firstly, by analysing the HTML response and finding out that the GET parameter is outputted in the response, webFuzz prioritized the request using its request ranking mechanism. Additionally, the needed payload format exercised more code paths in the input parsing process in comparison to other types of payload formats which meant that webFuzz would be rewarded with extra basic blocks when the right format was guessed. More specifically, in order for the payload to avoid getting sanitized, it had to contain a leading hashtag character which the input parsing code would mistakenly treat it as a URL fragment. When the URL fragment parsing code got executed though, webFuzz received in its coverage feedback more basic blocks. These two features allowed webFuzz to guess the right XSS payload faster as the request with the vulnerable GET parameter containing a leading hashtag character scored higher.

Concerning the artificial RXSS bugs, webFuzz also finds more bugs than Wfuzz due to the different XSS payload creation mechanisms. As it has been described in Sect. 3.2, an artificial bug requires that a fuzzer finds the correct XSS payload format and also guesses the right magic number. Since correctly guessing a digit from the magic number triggers a new instrumentation point, webFuzz detects this change and prioritizes 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 Wfuzz's blind fuzzing approach. As a real-world analogy for this process, each digit of the magic number can represent one correct mutation that gets us closer to the vulnerable sink.

5 Limitations

Concerning the Bug Injection tool, our prototype is currently limited to injecting surface-level bugs that rely on magic byte guessing to increase their complexity. Because the algorithm works by crawling the web application and semi-randomly picking an executed basic block to inject a vulnerability to, the resulting bugs do not rely on a complex internal application state for a bug to be triggered. More work needs to be done to improve the bug injection algorithm by enhancing it with means to monitor the application's state (located in the document cookies and in the database) and thus be able to expand its bug types to vulnerabilities that rely on a series of dependent requests.

Limitations in our fuzzer also exist, with the main limitation being the inability for webFuzz to fuzz Single Page Applications (SPA). These types of web applications heavily rely on JavaScript and the server responses are usually not in HTML format. Since webFuzz will not execute client-side JavaScript code, which in SPA applications is fully responsible for the HTML document creation and rendering, our fuzzer will not be applicable in such situations.

6 Future Work

As it can be seen from Sect. 4.2, the instrumentation overhead and webFuzz's request processing introduces high overheads. Multiple research papers have explored innovative ways to decrease the instrumentation overhead [1,10,43]. One common solution is to perform probe pruning which consists of finding the minimum set of basic blocks, where instrumenting only these blocks still provides us enough information to distinguish all executions from each other, i.e., no coverage information is lost. To perform this, CFG analysis must be performed which is possible in PHP [7].

There are also plans to extend our detection capabilities to SQL injection (SQLi) attacks which occur when untrusted data are sent to an interpreter or database as part of a query. To achieve this, SQLi related attack payloads found in the wild can be used as the base of the payload. To detect an SQLi vulnerability, webFuzz can scan the HTML response for database related keywords or for database hangs which can occur when a call to the database `SLEEP` command successfully slips through the query.

Finally, we are also planning on introducing netmap [38], a tool to effectively bypass the operating system's expensive network stack and allow the client and server to communicate faster.

7 Related Work

Native Fuzzing. Fuzzing has been perceived through several techniques and algorithms over the years, initially with fuzzing native applications. There exist the Black-box fuzzers [23,41,45] that are unaware of the fuzz target's internals, and the white- and grey-box fuzzers that leverage *program analysis* and *instrumentation* respectively to obtain feedback concerning the inputs' precision in discovering unseen paths [21,22,37,42,47] Additionally directed-based fuzzers use the coverage feedback to direct the fuzzer towards particular execution paths [21].

Web-app Fuzzing. Even though a huge effort is directed toward building fuzzers with the aim to weed out vulnerabilities in *native code*, little attention has been given to web application bugs. Tools currently available that target web application vulnerabilities behave predominantly in a black-box fashion that are subject to limitations due to limited knowledge of the application's state [9,17]. Some examples are: *Enemy of the State* [16], a black-box fuzzer that attempts

to recreate the web application's state machine from the application responses and uses it to drive the fuzzing session. and *KameleonFuzz* [18], an evolutionary algorithm based black-box fuzzer.

There have been attempts to overcome the shortcomings of black-box techniques. White-box approaches for instance, utilize access to the web application's source code to create test cases intelligently [3,4,7,8,25–27,29,30]. Artzi et al. [6] developed a tool for discovering web application vulnerabilities by collecting information about the target extracted through concrete and symbolic execution. Another tool combining static and dynamic analysis is Saner [8] which tries to identify any sanitization processes that do not work as expected to, resulting in allowing attackers to introduce exploits. Similarly other work [3,4], rely on static analysis and constraint solving to identify vulnerable source-sink pairs that contain insufficient sanitization. Backes et al. in their PHP aimed tool on the other hand, rely on Code Property Graphs and on modeling of vulnerabilities as graph traversals [7].

Contrary to the above research work for identifying web vulnerabilities, our technique adopts the grey-box approach. webFuzz *instruments* the fuzz target in order to create a feedback loop.

Vulnerability Injection. When evaluating automated vulnerability scanners, there is this great need of ground truth corpora, programs that have realistic vulnerabilities in known places. An example of such effort is Juliet [11], a suite that consists of thousands of small programs in C/C++ and Java, that contain various vulnerabilities (e.g., buffer overflows, NULL pointer dereference). Another example of such suite is BugBox [33], a vulnerability corpus for PHP web applications. However, these examples are pre-defined sets of vulnerable programs, that while being helpful for evaluating vulnerability scanners, they cannot simulate real world scenarios because of their small size. In contrast, automated bug injection tools can simulate real world scenarios because they are capable of injecting bugs in real-world programs. Main example of such tool and the inspiration of our automated bug injection tool is LAVA [15] which can automatically synthesize and inject thousands of bugs in native code programs. Some other examples include SolidiFI [20], an automated vulnerability injection tool targeted for evaluating smart contracts and EvilCoder [34], a framework that finds and modifies potentially vulnerable source code.

8 Conclusion

In this paper we presented webFuzz, the first *grey-box* fuzzer for discovering vulnerabilities in web applications. webFuzz applies instrumentation on the target web application for creating a *feedback loop* and utilizing it in order to increase code coverage. Consequently it increases the number of potential vulnerabilities found.

Acknowledgements. We thank the anonymous reviewers for helping us to improve the final version of this paper. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct), No. 830929 (CyberSec4Europe) and No. 101007673 (RESPECT).

References

1. Agrawal, H.: Dominators, super blocks, and program coverage. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 25–34 (1994)
2. Aho, A., Lam, M., Ullman, J., Sethi, R.: Compilers: Principles, Techniques, and Tools. Pearson Education (2011). https://books.google.com.cy/books?id=NTIrAAAAQBAJ
3. Alhuzali, A., Eshete, B., Gjomemo, R., Venkatakrishnan, V.: Chainsaw: chained automated workflow-based exploit generation. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 641–652 (2016)
4. Alhuzali, A., Gjomemo, R., Eshete, B., Venkatakrishnan, V.: NAVEX: precise and scalable exploit generation for dynamic web applications. In: 27th USENIX Security Symposium (2018)
5. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
6. Artzi, S., et al.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Trans. Softw. Eng. **36**, 474–494 (2010)
7. Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F.: Efficient and flexible discovery of PHP application vulnerabilities. In: 2017 IEEE European Symposium on Security And Privacy (EuroS&P), pp. 334–349. IEEE (2017)
8. Balzarotti, D., et al.: Saner: composing static and dynamic analysis to validate sanitization in web applications. In: 2008 IEEE Symposium on Security and Privacy (SP 2008) (2008)
9. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: automated black-box web application vulnerability testing. In: 2010 IEEE Symposium on Security and Privacy (2010)
10. Ben Khadra, M.A., Stoffel, D., Kunz, W.: 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, pp. 1153–1164 (2020)
11. Black, P.E., Black, P.E.: Juliet 1.3 test suite: changes from 1.2. US Department of Commerce, National Institute of Standards and Technology (2018)
12. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2329–2344 (2017)
13. Cornelius Aschermann et al.: REDQUEEN: fuzzing with input-to-state correspondence. In: NDSS, vol. 19, pp. 1–15 (2019)
14. Corporation, T.M.: Common vulnerabilities and exposures (CVE) (2020). https://cve.mitre.org/
15. Dolan-Gavitt, B., et al.: LAVA: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP). IEEE (2016)

16. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: a state-aware black-box web vulnerability scanner. In: 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, pp. 523–538. USENIX Association, August 2012. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe
17. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 111–131. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14215-4_7
18. Duchene, F., Rawat, S., Richier, J.L., Groz, R.: KameleonFuzz: evolutionary fuzzing for black-box XSS detection. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY 2014, New York, NY, USA, p. 3748. Association for Computing Machinery (2014). https://doi.org/10.1145/2557547.2557550
19. Germán Méndez Bravoi, A.H.: esprima-python (2017). https://github.com/Kronuz/esprima-python
20. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. arXiv preprint arXiv:2005.11613 (2020)
21. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, New York, NY, USA, pp. 213–223. Association for Computing Machinery (2005). https://doi.org/10.1145/1065010.1065036
22. Godefroid, P., Levin, M.Y., Molnar, D.: SAGE: whitebox fuzzing for security testing. Queue (2012)
23. Householder, A.D., Foote, J.M.: Probability-based parameter selection for black-box fuzz testing, Technical report. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst. (2012)
24. James Graham, S.S.: html5lib-python (2007). https://github.com/html5lib/html5lib-python
25. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), pp. 6-pp. IEEE (2006)
26. Jovanovic, N., Kruegel, C., Kirda, E.: Precise alias analysis for static detection of web application vulnerabilities. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS 2006, New York, NY, USA, pp. 27–36. Association for Computing Machinery (2006). https://doi.org/10.1145/1134744.1134751
27. Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 199–209 (2009)
28. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, New York, NY, USA, pp. 2123–2138. Association for Computing Machinery (2018). https://doi.org/10.1145/3243734.3243804
29. Medeiros, I., Neves, N., Correia, M.: DEKANT: a static analysis tool that learns to detect web application vulnerabilities. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 1–11 (2016)

30. Medeiros, I., Neves, N.F., Correia, M.: Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In: Proceedings of the 23rd International Conference on World Wide Web, WWW 2014, pp. 63–74, New York, NY, USA. Association for Computing Machinery (2014). https://doi.org/10.1145/2566486.2568024
31. Mendez, X.: Wfuzz - the web fuzzer (2011). https://github.com/xmendez/wfuzz
32. Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G.: Understanding the reproducibility of crowd-reported security vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD. pp. 919–936. USENIX Association, August 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/mu
33. Nilson, G., Wills, K., Stuckman, J., Purtilo, J.: BugBox: a vulnerability corpus for PHP web applications. In: 6th Workshop on Cyber Security Experimentation and Test (CSET 13). USENIX Association, Washington, D.C., August 2013. https://www.usenix.org/conference/cset13/workshop-program/presentation/nilson
34. Pewny, J., Holz, T.: EvilCoder: automated bug insertion. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, New York, NY, USA, p. 214225. Association for Computing Machinery (2016). https://doi.org/10.1145/2991079.2991103
35. Pham, V.T., Böhme, M., Santosa, A.E., Caciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. IEEE Trans. Softw. Eng. (2019)
36. Popov, N.: PHP parser. https://github.com/nikic/PHP-Parser
37. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17, pp. 1–14 (2017)
38. Rizzo, L., Landi, M.: Netmap: Memory mapped access to network devices. SIGCOMM Comput. Commun. Rev. **41**(4), 422–423 (2011). https://doi.org/10.1145/2043164.2018500
39. Seal, S.M.: Optimizing web application fuzzing with genetic algorithms and language Theory. Master's thesis, Wake Forest University (2016)
40. Serebryany, K.: Libfuzzer-a library for coverage-guided fuzz testing (2015). https://llvm.org/docs/LibFuzzer.html
41. Sparks, S., Embleton, S., Cunningham, R., Zou, C.: Automated vulnerability analysis: leveraging control flow for evolutionary input crafting. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 477–486 (2007)
42. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, vol. 16, pp. 1–16 (2016)
43. Tikir, M.M., Hollingsworth, J.K.: Efficient instrumentation for code coverage testing. ACM SIGSOFT Softw. Eng. Notes **27**(4), 86–96 (2002)
44. Wang, Y., et al.: Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: NDSS (2020)
45. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 511–522 (2013)
46. Zalewski, M.: Binary fuzzing strategies: what works, what doesn't, August 2014. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html
47. Zalewski, M.: More about AFL - AFL 2.53b documentation (2019). https://afl-1.readthedocs.io/en/latest/about_afl.html