# Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities

Emre Güler[1], Sergej Schumilo[1], Moritz Schloegel[2], Nils Bars[2], Philipp Görz[2], Xinyi Xu[2], Cemal Kaygusuz[1], and Thorsten Holz[2]

[1]Ruhr University Bochum
[2]CISPA Helmholtz Center for Information Security

## Abstract

Server-side web applications are still predominantly implemented in the PHP programming language. Even nowadays, PHP-based web applications are plagued by many different types of security vulnerabilities, ranging from SQL injection to file inclusion and remote code execution. Automated security testing methods typically focus on static analysis and taint analysis. These methods are highly dependent on accurate modeling of the PHP language and often suffer from (potentially many) false positive alerts. Interestingly, dynamic testing techniques such as *fuzzing* have not gained acceptance in web applications testing, even though they avoid these common pitfalls and were rapidly adopted in other domains, e. g., for testing native applications written in C/C++.

In this paper, we present ATROPOS, a snapshot-based, feedback-driven fuzzing method tailored for PHP-based web applications. Our approach considers the challenges associated with web applications, such as maintaining session state and generating highly structured inputs. Moreover, we propose a feedback mechanism to automatically infer the key-value structure used by web applications. Combined with eight new bug oracles, each covering a common class of vulnerabilities in server-side web applications, ATROPOS is the first approach to fuzz web applications effectively and efficiently. Our evaluation shows that ATROPOS significantly outperforms the current state of the art in web application testing. In particular, it finds, on average, at least 32% more bugs, while not reporting a single false positive on different test suites. When analyzing real-world web applications, we identify seven previously unknown vulnerabilities that can be exploited even by unauthenticated users.

## 1 Introduction

The number of web applications we interact with and rely upon daily is constantly growing. With 77.6%, a large majority of the top ten million most visited websites still rely on PHP as the language for their server-side applications [62]. Prominent examples include *Wikipedia*, *Etsy*, *WordPress*, *Baidu*, and *Tumblr*. Given the widespread adoption and continued popularity of the PHP language, security testing is critical to identify potential vulnerabilities as early as possible.

Given the typical size of web applications, manual source code audits prove impractical. Instead, state-of-the-art approaches typically rely on static analysis techniques [6, 12, 13, 15, 21]. By accurately modeling the PHP language, particularly its built-in functions, and applying a taint analysis-based approach, these tools aim to identify a wide range of vulnerabilities ranging from SQL injection to file inclusion and remote code execution. Despite all efforts to modeling the PHP language faithfully, existing methods often suffer from (potentially many) false positives. We have found empirically that this observation still holds for modern approaches (see Section 5.2). Noteworthy, this problem is not specific to PHP, but plagues most static analysis efforts [30, 54]. A high number of false positives is counterproductive, misleading developers to ignore actual findings or to waste their time chasing non-existent vulnerabilities [24]. Furthermore, static analysis is typically unsuitable for effectively producing test cases that assist developers in debugging the problem at hand [7].

Instead of trying to avoid false positives by improving the model's accuracy, we can explore other approaches. In testing native applications, a dynamic approach to uncovering faults has recently proven particularly effective: *feedback-driven fuzzing* [4, 16, 20, 67]. This method provides a simple but fast method for testing a variety of (slightly mutated) inputs against a target program and uncovers inputs that lead to actual program faults. The crashing inputs provide the developer with a primitive that allows reproducibilty and aids in debugging.

Intuitively, fuzzing web applications may sound like a promising approach. Unfortunately, modern fuzzers are optimized for a specific type of program and rely on several properties that are not available in the context of web applications. First, their targets expose a relatively simple interface (e. g., *stdin* or files), whereas a web application expects input from the web server, which in turn receives its input from a web browser via HTTP(s). Second, typical fuzz targets do

not maintain an extensive state and can be reset by simply recreating the process (which contributes to the fuzzers' main advantage, their performance), e. g., using `fork()`. In contrast, web applications maintain a lot of state, as the web server, browser, sessions, databases, and similar aspects need to be considered. Third, standard fuzzing targets operate on byte-stream oriented inputs, i. e., flipping bits and bytes often succeeds in exploring new program behavior. Instead, web applications are text-oriented and expect highly structured input, usually containing developer-defined identifiers. These challenges make classical fuzzing approaches inefficient for testing web applications. Even worse, fuzzers are typically unable to detect when a server-side fault has been triggered, as they only check for crashes caused by memory access violations. However, typical web application bugs, such as SQL injections, server-side request forgery (SSRF), or command injection, do not crash the interpreter. So even if the fuzzer successfully triggers a bug, it would not recognize it as such.

While a few web fuzzing approaches have been proposed in previous work, they fail to solve the outlined challenges and have a limited scope. WEBFUZZ [46] uses coverage-guided feedback but can only detect *client-side* stored and reflective cross-site scripting (XSS) vulnerabilities. Similarly, CEFUZZ [69] is limited to remote code execution and command injection vulnerabilities. Instead of fuzzing the web application directly, both tools send HTTP requests to a web server, which forwards them to the actual web application. This performance limitation is shared by virtually all tools focusing on dynamically testing web applications, including those used in industry such as WFUZZ [31]. This tool is an advanced blackbox fuzzer capable of sending and mutating payloads specified by a human domain expert. However, it is not fully automated, lacks awareness of the state maintained by the web application, has no access to coverage information, and generally has no bug oracles for server-side vulnerabilities. Concurrent to our work, Trickel et al. proposed WITCHER [58], a coverage-guided fuzzer that implements fault escalation to detect SQL and command injection vulnerabilities. Similar to other grey-box fuzzers, the state space is explored by tracking code coverage. The output of the web application is analyzed to guide the mutation process. However, their approach is limited to just two types of vulnerabilities.

In this work, we present a fuzzing method capable of effectively and efficiently testing web applications for different types of security vulnerabilities. Our approach is specifically designed to account for the *stateful* nature of web applications by using snapshots. Furthermore, we propose a novel feedback mechanism tailored to web applications that allows our fuzzer to generate inputs bypassing the shallow parsing stages of the tested web application and effectively exploring deeper program parts. Last, we introduce eight new bug oracles, each capable of detecting a particular category of server-side PHP web application bugs. We implemented a prototype of the proposed approach in a tool called ATROPOS. The evaluation demonstrates that our dynamic approach significantly outperforms static analysis approaches: We find 32% more bugs than the best performing static analysis approach while at the same time reporting zero false positives on the test suites. In terms of coverage, we cover on average between 50% and 230% more code compared to WEBFUZZ and WFUZZ, respectively. In summary, we make the following main contributions:

- We present ATROPOS, a novel feedback-guided, snapshot-based fuzzing method for web applications that can detect eight types of server-side vulnerabilities.
- We introduce a novel feedback mechanism that extracts relevant runtime information directly from the interpreter, which informs the random mutator. As our evaluation shows, this approach can achieve deeper code coverage than state-of-the-art web application fuzzers.
- We present eight new bug oracles that can efficiently detect different types of server-side vulnerabilities, featuring high detection rates and few false positives. Moreover, our approach does not require heavyweight program analysis techniques or complex instrumentations.

To foster further research in this area, we open-source our implementation of ATROPOS at https://github.com/cispa-syssec/atropos-legacy. An extended version of this work is available as technical report [19].

## 2 Challenges

Traditional fuzzing approaches [3, 8, 16, 20, 65, 67] that have proven effective for languages that compile to native binaries are not directly applicable to server-side web applications written in interpreted languages such as PHP. In particular, we identify three main challenges that must be addressed to enable efficient and effective fuzzing of such applications.

### 2.1 Challenge 1: Complex Interface

The first challenge is the complex interfaces of server-side web applications. Instead of passing an input stream via `stdin` or files—as with most native binaries—we need to replicate the environment provided by the web server for the interpreter that is executing the target application (i. e., the web application that is accessed with a web browser). More specifically, we need to replace the web browser and the HTTP web server with an agent that directly communicates with the interpreter, reducing unnecessary overhead.

Once the fuzzer can pass inputs to the web application, it needs to generate *meaningful* inputs to the target's application logic. A full specification of the interface is usually not available, and hence we cannot use grammar-based fuzzing approaches. Similarly, random byte sequences, as generated by traditional fuzzing methods, are unlikely to match the input of the web application and are therefore rejected in the early parsing stages. Typically, web applications use semantically complex identifiers that are set by developers. For instance, when

submitting a form, each field is assigned a string descriptor set by the developer (e. g., *password*). Thus, these descriptors carry a semantic meaning that humans can use, but hinders traditional fuzzing approaches: To pass this form, the fuzzer must generate a string *password*=*value*, where *password* is the correct identifier associated with the form, and *value* must be a valid value for this field. Such semantic tokens (in the form of key-value pairs) are not only used to retrieve data from forms but also for session attributes (e. g., cookies), URL parameters, or JSON inputs sent to REST APIs. Noteworthy, this problem is a form of the magic byte problem [4, 8, 42], where the fuzzer must set particular bytes to a specific value to pass a check. Previous research suggested two common solutions: generating a dictionary before the fuzzing process starts or using LAF-INTEL-style [26] instrumentation that provides feedback even on partial solutions, i. e., an input of "*passw*" would give positive feedback indicating that the fuzzer is getting closer to the correct solution. While both solutions are helpful, they are limited to static strings and fail for strings generated at runtime. Another common obstacle with the second solution is the number of intermediate stepping stones that are generated (i. e., inputs added to the corpus) for each single comparison, which can overwhelm the seed scheduler for large applications with many such comparisons.

## 2.2 Challenge 2: Stateful Environment

Modern web applications maintain an extensive state for a user session. Usually, this starts with authenticating the user to access protected or user-specific resources. Moreover, web applications heavily rely on databases or persistent storage to maintain state. This leads to two problems: First, the state is scattered across many components, which we must take into account. Second, the persistent nature of the state impacts any subsequent execution (e. g., deleting a file or database entry will prevent future inputs from accessing it).

There are two approaches to dealing with the state problem. The naive way is to ignore the state for most interactions with the web application, e. g., creating a new entry in the database. For critical operations that affect later runs and are difficult to revert (e. g., logging out or deleting data), a human domain expert must identify the appropriate code and place it on a block list. This approach is used by WEBFUZZ [46]. However, this requires a human expert and does not allow for testing the complete functionality of the web application. Additionally, overlooked interactions may silently impede the fuzzing progress. The second approach for dealing with state is to track all changes (e. g., additions to the database) and implement special logic to eventually revert all changes (e. g., by deleting the new data from the database). However, doing so for each input is a potentially costly procedure.

## 2.3 Challenge 3: Bug Oracles

Unlike common fuzzing approaches that test memory-unsafe programs, application bugs in interpreted languages such as PHP or JavaScript typically do not manifest as memory safety violations and thus cannot be observed using standard bug oracles that rely on crash signals. Instead, the following eight bug classes represent typical security challenges for PHP:

**1) SQL Injection:** This bug class occurs when unsanitized input is used in an SQL query, allowing an attacker to execute arbitrary SQL commands (e. g., to extract or modify sensitive information in a database) [39].

**2) Remote Code Execution:** A bug allowing an attacker to inject and execute arbitrary PHP code; this happens, for example, when the attacker controls the input to `eval()` [34].

**3) Remote Command Execution:** Also called *command injection*, this enables the adversary to execute arbitrary *shell* commands on the server [35].

**4) Local and Remote File Inclusion:** To include another file (e. g., a module), PHP provides multiple directives. A local or remote file inclusion vulnerability allows attackers to parse and execute arbitrary local files or remote resources as PHP code and, thereby, gain remote code execution [11].

**5) PHP Object Injection:** PHP data and objects can be converted to a stream of bytes via `serialize()` and converted back via `unserialize()`. If the web application allows users to manipulate the stream of bytes ending up in the latter function, arbitrary PHP objects can be injected. This introduces a variety of other vulnerabilities, such as SQL injections or remote code execution [13, 36, 41].

**6) Server-side Request Forgery (SSRF):** PHP provides many functions to access remote resources, e. g., to query a REST API. Gaining control of the destination used to access such resources allows an attacker to forge requests in the name of the web server. This allows bypassing countermeasures such as firewalls that normally isolate the server's private network from the Internet [37].

**7) Arbitrary File Read and Write:** These bugs enable an attacker to read or write arbitrary files on the web server.

**8) File Upload:** Many web applications allow users to upload files, e. g., to set a profile picture. However, if the uploaded file(-name) is not properly sanitized or checked against an allowlist, potentially malicious files can be uploaded, which may ultimately lead to the execution of arbitrary code (e. g., by uploading PHP files) [40].

In summary, web applications are challenging to test for traditional fuzzing approaches because they exhibit a complex interface, maintain an extensive state, and contain software faults that traditional bug oracles cannot detect.

# 3 Design

To address the challenges outlined above, we present the design of ATROPOS, a novel fuzzing method for testing web

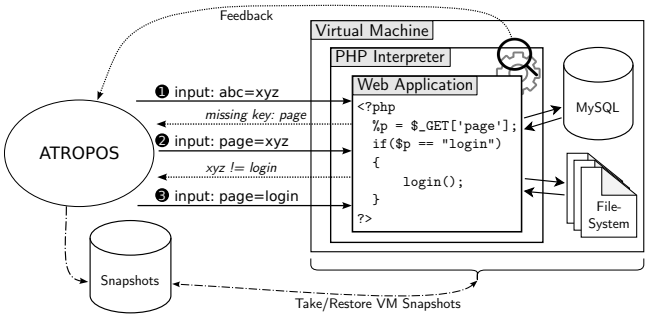


Figure 1: High-level overview of ATROPOS's architecture.

applications. We focus on PHP-based web applications, but the techniques described below can be applied to other types of web frameworks as well.

## 3.1 Architecture Overview

Figure 1 provides a high-level overview of ATROPOS' architecture. Generally speaking, we want to fuzz a web application that consists of multiple processes, i. e., the PHP interpreter executes the application and interacts with a database, the filesystem, and potentially other components. All components are highly dependent on their state and environment, which requires them to run in an isolated system (e. g., a virtual machine), to allow snapshots to be taken and state to be reset.

To guide the fuzzing process, we instrument the PHP interpreter to provide coverage feedback and introspection capabilities. Internally, ATROPOS works similar to fuzzers such as AFL++ [16]. In fact, ATROPOS uses the same algorithm, called *explore* [16], as AFL++ for seed selection and prioritization. We also include its typical byte-oriented mutations, such as bitflips. To enable fuzzing of web applications, we propose a number of changes that address the unique challenges in the web context. First, as a web application can consist of multiple PHP files—all known a priori to the fuzzer—ATROPOS selects one randomly for each fuzzing iteration. Second, each fuzzing input can contain more than one *request*, allowing the fuzzer to run two or more files sequentially in one fuzzing iteration. As each request to a web application is highly structured, ATROPOS features a custom mutator reflecting the key-value-oriented structure of inputs and executes the target and all associated processes in a VM so that it can efficiently restore the entire environment via a fast full-system snapshot mechanism. In addition, we design eight custom bug oracles tailored to server-side vulnerabilities. ATROPOS explicitly avoids costly operations such as static analysis or taint analysis to maintain high throughput, and its bug oracles are designed to keep the number of false positives low.

## 3.2 Advanced Feedback Mechanisms

A web application is usually executed by an interpreter (e. g., a PHP interpreter), which receives its input from the web server, which in turn receives its input from the web browser. To reduce overhead and increase testing performance, we replace the web browser and the web server with the FastCGI [43] interface as a more direct means of communication. However, PHP web applications still expect *highly structured* inputs because they make heavy use of semantic tokens in the form of key-value pairs. Traditional fuzzing tools such as AFL [67] apply random mutations to the input. In contrast, our design is based on a web application-aware representation of the input and involves a number of techniques to generate meaningful key-value pairs, e. g., we identify parts of the input structure that the web application expects and provide them to the fuzzer in the form of a dictionary, which can be used to precisely mutate specific keys and values. The following techniques constitute our PHP web application-specific advanced feedback mechanisms that complement the fuzzer's coverage feedback.

### 3.2.1 Inferring Application-specific Keys

When the web application receives a request, several global maps (e. g., `$_GET`, `$_POST`, or `$_SERVER`) that can be accessed by the web application are populated according to the web browser's request. Thus, our fuzzer must also populate the keys that are accessed by the web application. These are usually complex semantic tokens that a fuzzer is unlikely to generate randomly by chance. To solve this problem, we take advantage of having full control over the execution environment: ATROPOS hooks into the PHP interpreter's process of accessing these global maps. When it observes a new access, our hook provides the accessed key as feedback to the fuzzer. This allows ATROPOS to set the expected key for the next fuzzing iteration. For example, consider input ❶ in Figure 1: While processing our initial random fuzzing input `abc=xyz`, the web application accesses key `page`, and our hook reports it to the fuzzer as a missing key. This way, ATROPOS can set the key in our subsequent request ❷. Notably, this works for all keys, even those that are dynamically generated at runtime.

### 3.2.2 Inferring Expected Values

To test deeper parts of the application, the fuzzer not only need the correct key, but also a *specific value*, e. g., `page=login`. Our fuzzer can generically infer all values that the web application uses during runtime. Traditionally, this can be done in several ways: (1) taint analysis, (2) symbolic execution, or (3) heuristic-based techniques. The former two techniques suffer from state explosion and significant overhead and are comparably costly to implement. Thus, in our design, we use heuristics similar to input-to-state correspondence [4] but tailored to the domain of web applications. The intuitive insight is that in many cases (parts of) the input is directly compared

with a specific value. As the input to the web application is based on strings, we can hook all string comparison functions in the PHP interpreter, e. g., `zend_string_equal_val`. When ATROPOS encounters a string comparison, our hook passes the information to the fuzzer as feedback. Then, ATROPOS can randomly replace occurrences of "wrong" values with the expected one. In Figure 1, the `if` clause in line 3 of the web application compares our random input, `xyz`, to the expected value, `login`. Receiving the correct value as feedback, the fuzzer can replace `xyz` with `login` and thus send the valid input `page=login` in the third fuzzing iteration, ❸, to unlock deeper parts of the application's logic.

We emphasize that this inference process is not limited to full string comparisons but also covers partial comparisons. Empirically, we observed that ATROPOS was able to manipulate highly-structured input values like JSON and even generate a valid HTTP header from scratch when the parsing itself was implemented in PHP, thus providing relevant information via these advanced feedback mechanisms. Notably, this approach even works for CSRF tokens, since ATROPOS uses full system snapshots, such that CSRF tokens are deterministic.

While the input-to-state correspondence approach seems to work well in practice, more extreme changes to the input, such as cryptographic hashing and base64 encoding, are indeed challenging. However, this is true for fuzzing in general [4,42]. For example, when the input is hashed before being compared to a SHA256 hash, it is not possible for ATROPOS to solve this check. While it would certainly be possible to add support for encodings such as BASE64, we have yet to see a need for this in practice. Less extreme input transformations such as STRTOUPPER or STR_REPLACE should be solvable, as the resulting runtime strings are extracted and used during input generation.

ATROPOS uses an abstract representation of these key-value pairs for fuzzing purposes, which are only converted to HTTP (or FastCGI) input in the form of `a=b&c=d` in the last step when feeding it into PHP. We emphasize that the difficulty is not in generating and maintaining the HTTP input format, but in deriving the correct key-value pairs.

#### 3.2.3 Inferring Values for Regular Expressions

A direct comparison (of parts) of the input with a string does not cover all scenarios. In practice, web applications often check whether an input matches a regular expression, e. g., to verify whether an input is a valid e-mail address or phone number (see Listing 1). In these cases, ATROPOS uses the mechanism outlined above and pass the observed regular expression as feedback to the fuzzer. The fuzzer can then derive a random string conforming to this regular expression with the help of existing techniques (e. g., we use XEGER [10] in our prototype).

```
1 if(preg_match('/^\S+@\S+\.\S+$/', $_GET['email'])
2     echo "address looks correct";
```

Listing 1: The regex is observed when executed and then passed back as feedback for the fuzzer.

```
1 <form action="login.php">
2     <input name="user" />
3     <input name="passwd" type="password"/>
4 </form>
```

Listing 2: Example for an HTML login form.

#### 3.2.4 Inferring Keys from HTML

As an optimization technique, ATROPOS parses the HTML output generated by the application to extract potentially useful keys or key-value pairs. For example, the login form in Listing 2 would yield the keys `user` and `passwd`. This is a simple form of crawling, we do not explicitly request further pages.

This step is optional and a pure optimization step: Our generic identification mechanism can potentially find all keys and values used but requires multiple fuzzing iterations, as the web application must access the key or value at least once to trigger the feedback. Using this "crawling", the fuzzer can discover all*static* keys embedded in the HTML output in a single pass, but not the dynamically generated keys. For the latter, it still needs the regular identification mechanism presented above.

#### 3.2.5 Performance Overhead

Since collecting and returning this feedback information to the fuzzer is costly, ATROPOS executes the dynamic advanced feedback mechanisms only *once* for each new input that yielded new coverage. As a result, this is a one-time cost per seed file with negligible overhead compared to the whole fuzzing campaign.

In summary, this web application-specific feedback mechanism enables us to effectively and efficiently explore deeper states of the application under test. Next, we need to address the extensive state the web application maintains.

### 3.3 Stateful Environment

To address the need for the web application to maintain a complex state (e. g., sessions, database, filesystem, etc.) without compromising fuzzing efficiency, we run the web application in an isolated environment suitable for snapshots and restores. This differs from other web application fuzzers like WEBFUZZ [46], which manually harness the targeted application, e. g., by preventing the fuzzer from executing a code region related to logging out the user. Such an approach has three main drawbacks. First, it requires a human domain expert to harness the target. Second, the harnessing is error-prone: Skipping code may alter the functionality of the web application or

render bugs invisible to the fuzzer. Third, the lack of isolation between executions leads to situations where the executed code changes the filesystem (e. g., modifies files), alters the database (e. g., changes or deletes entries), or mutates the user session (e. g., a user can change their e-mail address, log out, or log in as a different user). All these actions have profound effects on subsequent executions and may make triggered bugs non-reproducible. Even running the same input twice may result in different behavior, making the entire process highly non-deterministic.

If the fuzzer instead runs the web application in an isolated environment, it can use fast snapshots optimized for fuzzing to conveniently recover the entire system state after processing each input. This way, the web application's functionality is not limited, and we maintain a high fuzzing performance. Moreover, the bugs found are guaranteed to be reproducible. Snapshots ensure that not only the file system is restored, but also user sessions and even memory, including the state of the database. As neither the database nor any applications running in the background have to be restarted after a restore, ATROPOS avoids a long initialization phase at startup. To allow for fast snapshot restores of the filesystem and all running processes, we build ATROPOS on top of NYX [47]. As a base strategy, ATROPOS takes a snapshot after initialization of all assets, such as the database, and restore to this snapshot after every fuzzing iteration, i. e., after the web application processed the fuzzing input or reached the one second timeout while doing so.

Note that some bugs might require the chaining of multiple inputs *without* resetting the state between executions, e. g., a bug that occurs only when the database has thousands of entries, requiring many inputs to first populate the database before some input can then trigger the bug. A fuzzer that always resets the state will not be able to detect this bug. This can be addressed by not resetting the state after every input but only after every $n$ inputs. However, even a single fuzzing input can already contain multiple requests to the web application, allowing our fuzzer to find such vulnerabilities in principle. The more complex the bugs are and the more state must be accumulated, the less likely the fuzzer is to craft an input triggering the bug. In general, finding bugs depending on a complex state is challenging.

In summary, running the web application in an isolated environment allows us to use snapshots to skip costly initialization and ensure reproducibility of bugs. As a last step, we need to address the discrepancy between traditional bug oracles and typical server-side web vulnerabilities.

## 3.4 Bug Oracles Beyond Memory Corruption

Many types of software faults in web applications do not manifest themselves in memory safety violations and hence in a crash that a fuzzer can recognize as a bug. Based on this insight, we propose a set of custom bug oracles to identify eight common bugs in server-side web applications.

**Principles.** At their core, our proposed heuristics rely on two generic conditions to identify vulnerabilities. (i) A potentially unsafe function shows unusual or suspicious behavior, often producing a warning or an error. (ii) This behavior was triggered by an attacker-controlled input. If *both* conditions are met, the bug oracle reports a found vulnerability for specific PHP functions. Our underlying insight is that a majority of common bugs are caused by a few critical functions, e. g., `mysqli_query()` or `unserialize()`. By instrumenting these functions in the PHP interpreter, we can convert them into bug oracles sensitive to a particular class of bugs.

Instead of *modeling* the whole PHP program (i. e., for static analysis), we instrument only critical sinks and *execute* the web application with different inputs. In contrast to taint analysis, we do not need to track the user input but instead monitor the sink for suspicious behavior, which is likely to be caused by user input. In Listing 3a, the fuzzer may generate an input that is passed to `mysqli_query()`, resulting in an invalid SQL query. This will raise a syntax error and is a strong indication that user input is passed unsanitized as part of a query (which could make the web application vulnerable), since properly sanitized input is unlikely to break the query syntax. Empirically, we find that unsanitized random inputs, such as generated by a fuzzer, are likely to provoke a PHP error or warning since many security sensitive functions require structured inputs. While this meets our first condition (*suspicious function behavior*), it does not necessarily imply that the *attacker* (i. e., fuzzer) has control over the query and, thus, could exploit the bug. Using our lightweight inference process outlined in Section 3.2.2, ATROPOS can insert a special string into the input and observe if this string appears in the query. If it does, we conclude that the attacker controls the query, making it vulnerable to SQL injection. With both criteria, (i) and (ii), met, we report a vulnerability. We discuss potential shortcomings of our heuristics in Section 6.

**Novelty.** While techniques using errors as feedback for manual code reviews have been known for a long time, to the best of our knowledge, we are the first to combine this with a lightweight inference of checking whether attacker-controlled input arrives at the particular sink in an automated fashion. To do so efficiently, we avoid known but costly techniques such as taint analysis and use a form of input-to-state correspondence tailored to web applications. Concurrent to our work, WITCHER [58] proposes the concept of fault escalation to turn SQL and command injection vulnerabilities into a signal a fuzzer can detect. Compared to their work, we provide oracles for eight bug types and locate our bug oracles within the PHP interpreter. In doing so, we sense bugs even when the tested web application disables printing of errors or warnings.

**Bug triggers.** To speed up the process of finding inputs that trigger errors, ATROPOS contains a list of potential *bug trigger strings*, e. g., a string containing different quotation marks that are often required to break out of strings in an

```php
// a) SQL injection vulnerability
$id = $_GET['id'];
$query = "SELECT name FROM users WHERE id='$id';";
$result = mysqli_query($mysql,  $query)

// b) remote code execution vulnerability
eval('$var = "'. $_COOKIE['a'] . '";');

// c) remote command execution vulnerability
shell_exec('ping  ' . $_POST['ip']);

// d) local / remote file inclusion vulnerabilities
include("lib/" . $_GET['page']); // LFI
include($_GET['page']); // RFI

// e) PHP object injection vulnerability
unserialize($_COOKIE['session']);

// f) SSRF vulnerability
file_get_contents("http://".$_POST['host']);

// g) arbitrary read vulnerability
$file = $_POST['file'];
echo file_get_contents($file);

// h) file upload vulnerability
if($_FILES['f']['type'] == "image/png")
  move_uploaded_file($_FILES['f']['tmp_name'],
                     $_FILES['f']['name']);
```

Listing 3: Code examples for eight vulnerability types.

SQL query. This is similar to the way fuzzers like `AFL` insert magic values into the input [66], e. g., −1 or 255, with the goal of triggering integer overflows. For most of our bug oracles, random mutations of the input alone are sufficient (e. g., to add quotation marks), but we found that trigger strings still speed up this process. Although heuristics like these may appear limited, they work well in practice, as demonstrated in our evaluation (see Section 5). In particular, we found no false positives on the test suites, while catching significantly more bugs than existing analysis tools.

**New bug oracles.** In total, we propose custom bug oracles for the eight types of server-side web application vulnerabilities discussed in Section 2.3. In particular, they differ in the fact how *suspicious function behavior* is identified and how the fuzzing input is *tracked to the vulnerable sink*.

**1) SQL Injection.** This bug oracle reports a vulnerability if an input causes a syntax error when processing an SQL query that contains a fuzzer-controlled input. Intuitively, only unsanitized inputs should be able to break the syntax of the SQL query by changing the query such that it eludes the intended constraints (quotation marks, etc.).

**2) Remote Code Execution.** The oracle reports a vulnerability if a fuzzer-controlled input raises a syntax error while compiling dynamic PHP code, e. g., during a call to `eval()`. Similar to the SQL injection oracle, there is a high probability that a syntax error will occur if an attacker-controlled input happens to be interpreted, as most random inputs are invalid PHP code. Listing 3b provides a brief example for the `eval()` function. Additionally, ATROPOS injects valid PHP code that reports back a vulnerability when executed, as an attacker should not be able to run custom code within the context of the web application.

**3) Remote Command Execution.** Identifying a remote command execution, such as the one shown in Listing 3c, is more difficult, as we do not have an explicit error message. Instead, this oracle monitors for attempts to execute non-existent binaries, which is the case if the binary name is under the attacker's control. Additionally, the fuzzer tries to inject a command that attempts to execute a custom binary we placed in the VM that automatically triggers this oracle.

**4) Local and Remote File Inclusion.** A file inclusion vulnerability is reported if a call to file-related functions such as `include()` or `require()` results in an error indicating that the file does not exist, while the file path contains input controlled by the fuzzer (examples are shown in Listing 3d). In some cases, applications check whether the file exists before including it. To identify file inclusion vulnerabilities, even if they are guarded by such a check for existence, ATROPOS occasionally inserts a path to a file that reports back a file inclusion vulnerability when included.

**5) PHP Object Injection.** An object injection vulnerability is reported if attacker-controlled input ends up in a deserialization call (`unserialize()`, cf. Listing 3e). Since serialized data is a structured input, parsing errors can be used to detect suspicious behavior, similar to the SQL injection oracle.

**6) Server-Side Request Forgery.** If a resource request can be made pointing into private address ranges (e. g., `http://192.168.0.1`), while also containing fuzzer-controlled input, this oracle reports an SSRF vulnerability (e. g., by controlling the host of a call to `file_get_contents()`, see Listing 3f).

**7) Arbitrary File Read and Write.** As it is difficult to determine whether some observed file operation is malicious, we conservatively constrain this bug oracle to PHP files. This oracle is triggered if the web application tries to read, write, delete, or rename a PHP file while also containing fuzzer-controlled input in the filename (cf. Listing 3g). Additionally, ATROPOS tries to supply a canary PHP file that triggers the oracle and reports a bug according to the file operation applied to it.

**8) File Upload.** This category is also highly context-dependent: Uploading certain files might be allowed for one application but constitute a vulnerability for another. We defensively consider only uploading PHP files (i. e., files ending with `.php`) a security issue, as this is the least ambiguous violation. If uploading a PHP file via the respective hooked function succeeds (e. g., `move_uploaded_file()` in Listing 3h), we consider this bug as triggered.

In summary, these custom bug oracles are sensitive to specific server-side web application vulnerabilities. Combining these bug oracles with our fuzzer's ability to infer both keys

and values used by the web application, as well as its snapshot-based design, we can efficiently fuzz web applications.

## 4 Implementation

We implement our design in a prototype called ATROPOS in about 3,700 lines of C, Python, and Nim code. ATROPOS is split into two components: (1) the frontend, which generates inputs and decides which seed in the corpus to fuzz next, and (2) the backend, which runs the web application inside a virtual machine. Both components exchange information via shared memory and hypercalls. Moreover, the implementation of ATROPOS revolves around the primary fuzzing execution loop, which needs to (1) generate and mutate inputs, (2) receive feedback on which code regions of the web application were executed, (3) report back any vulnerabilities, and (4) restore the environment to its original state.

**General Setup.** Before the fuzzing run can start, the web application and all components must be installed. We use *Docker* containers to prepare the environment to simplify this process. This allows manual preparations such as logging into the target web application as a user, so that the fuzzer is provided with an initial set of capabilities.

**Generating Inputs.** Unlike binary targets, PHP web applications expect the input to be in the form of key-value pairs. The main functionality of the ATROPOS frontend is implemented as a custom mutator for AFL++, where the key and value inputs can be mutated individually, using our advanced feedback methods explained in Section 3.2. Before each input is executed, it is converted into FastCGI parameters and passed to the agent inside the VM on the backend.

As stated earlier, web applications rarely access these inputs directly. Instead, the PHP interpreter converts these parameters into an easily accessible associative array consisting of key-value pairs. There are four major types of external input sources that we can fuzz: (1) the `$_GET` superglobal contains all inputs passed via the URL, (2) `$_POST` mainly contains inputs sent via an HTML form, (3) `$_COOKIE` allows access to cookies, and (4) `$_SERVER` provides a variety of information, e. g., hostname, user-agent, etc.

**Executing Inputs.** As explained in the design (see Section 3), full-system virtualization is crucial for our approach. While there are multiple virtualization solutions available, we chose to build ATROPOS on top of the NYX framework [47] because it provides fast snapshot restores [48, 49] (about 6,000 to 10,000 reloads per second), which even allows it to be used in the context of native application fuzzing.

In our prototype implementation, we use *Ubuntu 22.04* as the guest operating system. An application called *agent* runs inside the virtual machine and can create or restore snapshots via specific hypercalls. It also governs the processing of input passed to the VM from the ATROPOS frontend and forwards that input to the web application. The agent is written in *Nim* and communicates with the PHP interpreter via FastCGI. Essentially, the agent mimics the communication between a web browser and a web server and, in the role of the web server, communicates with the web application. In addition to the actual web application, we need to run an SQL server in the background to allow the SQL queries to succeed. At the moment, we support both MySQL and SQLite databases, but adding new database support is a straightforward one-time engineering effort. Once the web application finishes the execution of one input, the agent requests to restore the full system to its initial state via the snapshot mechanism.

**PHP Interpreter and Code Coverage.** Our bug oracles require certain hooks to be installed (for a full list, refer to Table 5 in the appendix). We modify the PHP interpreter to execute our corresponding code when specific PHP functions are called, e. g., `move_uploaded_file()` for file upload vulnerabilities. We instrumented PHP 7.4, as 71.5% of all PHP websites use PHP version 7 [62]. It also showed the best compatibility and performance with the code coverage module `pcov` [61], which we use to retrieve coverage feedback. We patched `pcov` to provide feedback via an AFL-compatible bitmap that is shared with the fuzzer outside the VM.

**Runtime Feedback.** Besides the coverage feedback retrieved via `pcov`, ATROPOS implements our advanced feedback that allows us to infer keys and values. This feedback is mainly implemented in the PHP interpreter and the `pcov` module, such that they return information on failed comparisons, missing keys, and regular expression executions. The advanced feedback is primarily communicated via a NYX hypercall, which transfers results from inside the VM to the frontend via a shared memory file. This file can be read by the custom AFL++ mutator from the outside. Since advanced feedback is only recorded once per seed file, the overhead of the hypercall is negligible.

**Performance Improvements.** Before creating the initial snapshot, we use PHP's *Opcache* [44] to compile all PHP files once to avoid re-compilation on each execution. Additionally, we patch the PHP interpreter to prevent functions such as `sleep()` from slowing down the execution.

## 5 Evaluation

We evaluate our prototype implementation of ATROPOS in three experiments. First, we check if ATROPOS can detect more bugs in various benchmarks than state-of-the-art tools and how it compares to them in terms of false positives. Second, we analyze the code coverage of our fuzzer compared to existing work. Last, we study whether ATROPOS manages to find new vulnerabilities in real-world PHP web applications.

### 5.1 Setup

We use five machines with Intel Xeon Gold 6230 @ 2.10GHz processors (40 physical cores) and 192GB RAM, backed by SSD storage, if not declared otherwise.

**Test Suites.** To compare the bug-finding capabilities of ATROPOS, its false positive rate, and its coverage against state-of-the-art tools, we select a ground truth of three popular test suites specifically designed to be vulnerable, covering a wide range of bug types:

- Damn Vulnerable Web Application (DVWA) [63]: We use 16 of the provided bugs as ground truth.
- Xtreme Vulnerable Web Application (XVWA) [57]: An alternative to DVWA that provides nine relevant bugs.
- buggy web application (bWAPP) [32]: It offers 27 vulnerabilities relevant for our evaluation. We use a custom version adapted to run on newer PHP versions [33].

Crucially, these projects provide a ground truth, which we can use to calculate accurate true positive and false positive rates. In total, the three test suites contain 177,000 lines of code and 52 server-side vulnerabilities that are relevant for our evaluation, covering all eight of our supported vulnerability types (see Table 4 in the appendix for details). For the evaluation, we do not consider vulnerability classes that are outside of the scope of our work, e. g., client-side Cross-Site Scripting (XSS) vulnerabilities.

**Experimental Setup.** To better assess the capabilities of ATROPOS, we consider two scenarios: Running ATROPOS on a single core, which we use as a baseline, and running ATROPOS on 40 physical CPU cores, which is a more realistic scenario in practice [9, 18, 28, 50]. This distinction allows us to measure how ATROPOS scales given additional resources.

For our experiment, we follow the guidelines established by Klees et al. [25] for evaluating fuzzers. Since fuzzing is an inherently random process, we run our fuzzer ten times for 24 hours for the single-core mode (as suggested). For the 40-core mode, we complete only three runs, as a single run already requires 960 CPU hours. ATROPOS always starts with an empty seed. For comparison with web scanners and the static analyzers, which need to be run only once, we use the median of the number of bugs (rounded down to the nearest number). We calculate the *true positive rate* (i. e., the proportion of reported actual bugs within all bugs) and *precision* (i. e., the proportion of actual bugs within reported bugs). For a fair comparison, we kept the parameters identical for all targets. In particular, we did not adjust the toolset or configuration flags to perform better on specific targets. We explain the concrete configurations on our GitHub page available at https://github.com/cispa-syssec/atropos-legacy.

## 5.2 Experiment 1: Finding Bugs

In the first experiment, we assess the capability of identifying bugs in the three test suites and measure the number of false positives. As a baseline for static analyzers, we select four that (1) work on modern PHP versions, (2) support the detection of (most) vulnerabilities relevant to our work, and (3) have been actively maintained over the last four years. The selected tools are:

- SONARQUBE [52]: A commercial product where the PHP module was previously known as RIPS [12, 13]. We use the https://SonarCloud.io interface.
- PROGPILOT [14]: A static analysis tool specifically designed to find security vulnerabilities.
- PSALM [59]: A static analysis tool developed by Vimeo, which uses taint analysis to find security vulnerabilities.
- PHPCS-SECURITY-AUDIT [55]: A set of security vulnerability rules for the PHP_CODESNIFFER tool, which checks if the provided PHP project adheres to a previously defined coding standard.

In addition, we also evaluate against the popular web vulnerability scanners ZED ATTACK PROXY (ZAP) [68] and WAPITI [56][1]. In contrast to ATROPOS, these tools do not use coverage-guided fuzzing; instead, they rely on web crawling and scanning the output for hints of security vulnerabilities. In terms of fuzzers, we compare against WITCHER [58], which uses feedback-driven fuzzing but is limited to SQL injections and remote code execution bugs, WFUZZ, a simple fuzzer that replaces a keyword by the value of a given payload, and CEFUZZ, which combines static analysis and fuzzing to check for remote code/command execution bugs. We provide all tools with valid sessions (or login credentials) and a list of all accessible PHP files.

In the best-case scenario, an automated bug-finding tool would discover all bugs while reporting no false positives, i. e., it should have both a high *true positive rate* (also called *recall*) and a high *precision*. In practice, the purpose of automated bug-finding tools is to reduce a security analyst's manual workload. False positives increase the amount of work without providing any benefit. In addition, many false positives may lull developers into a false sense of security, causing them to ignore critical findings or postpone their investigation.

**Results.** Table 1 and Figure 2 summarize the results of this experiment. We observe that ATROPOS finds more bugs in total compared to the tested static analysis tools (overall, ATROPOS finds 75% of all possible bugs in single-core mode and 94% of all bugs when run with 40 cores in parallel). Furthermore, only ATROPOS, SONARQUBE and WITCHER report no false positives, but at the same time, ATROPOS also finds more bugs than the other two combined. Moreover, our approach finds 23% more bugs than the best static analyzer we tested against (in terms of bugs discovered), namely PHPCS-SECURITY-AUDIT. The three web scanners ZAP, WAPITI and WFUZZ lag clearly behind and find only a few bugs in total. Their poor performance is an inherent limitation: Web scanners rely on the application's output to find key-value pairs and explore the application. Vulnerability detection is limited to what the application exposes in its output or in

[1] The free version of BURP SUITE [45], a popular web scanner, does not feature automated scanning functionality. Instead we select the open-source WAPITI.

Table 1: Evaluation of ATROPOS against state-of-the-art analysis tools. True positives (*TP*) is the number of actual vulnerabilities found. False positives (*FP*) is the number of falsely reported vulnerabilities. *Precision* is the percentage of the vulnerability reports that are actual vulnerabilities. The true positive rate (*TPR*) is the percentage of actual vulnerabilities that were found (in total, 52 vulnerabilities exist). Except for false positives (*FP*), larger values are better. Bold values highlight the best results.

| | DVWA | | XVWA | | bWAPP | | Precision | TPR |
|---|---|---|---|---|---|---|---|---|
| | TP | FP | TP | FP | TP | FP | | |
| **ATROPOS (40 cores)** | **15** | 0 | **7** | 0 | **27** | 0 | **100% (49 / 49)** | **94% (49 / 52)** |
| ATROPOS (1 core) | 14 | 0 | 6 | 0 | 19 | 0 | **100% (39 / 39)** | 75% (39 / 52) |
| SONARQUBE | 5 | 0 | 4 | 0 | 20 | 0 | **100% (29 / 29)** | 56% (29 / 52) |
| PROGPILOT | 12 | 2 | 4 | 0 | 18 | 5 | 83% (34 / 41) | 65% (34 / 52) |
| PSALM | 7 | 1 | 6 | 0 | 22 | 5 | 85% (35 / 41) | 67% (35 / 52) |
| PHPCS-SECURITY-AUDIT | 13 | 28 | 5 | 2 | 19 | 29 | 39% (37 / 96) | 71% (37 / 52)[1] |
| WITCHER | 0 | 0 | 3 | 0 | 0 | 0 | **100% (3 / 3)** | 6% (3 / 52)[2] |
| ZAP | 2 | 4 | 2 | 1 | 3 | 6 | 39% (7 / 18) | 13% (7 / 52) |
| WAPITI | 0 | 17 | 4 | 15 | 0 | 2 | 11% (4 / 38) | 8% (4 / 52) |
| WFUZZ | 0 | – | 0 | – | 0 | – | – | 0% (0 / 52) |
| CEFUZZ | 3 | – | – | – | 5 | – | – | 15% (8 / 52)[3] |

[1] PHPCS-SECURITY-AUDIT does not support PHP object injection and file inclusion bugs. If excluding the six bugs of these types, its adjusted TPR is 80% (37 / 46).
[2] WITCHER only supports SQL and command injection bugs. The adjusted TPR is 8% (3 / 38)
[3] CEFUZZ only supports remote command/code execution bugs. We compare against the results they *report* for DVWA and bWAPP. They do not report on false positive numbers. Excluding the 46 bugs it does not support, CEFUZZ's adjusted TPR is 100% (8 / 8).

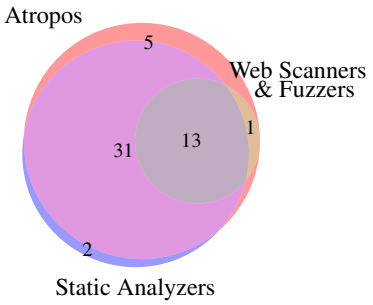


Figure 2: Venn diagram of bugs found in Experiment 1. We combine the results of *all* web scanners and fuzzers as well as *all* static analyzers. Still, ATROPOS found five bugs which not a single one of the static analyzers or web scanners could find. All static analyzers combined found two upload vulnerabilities ATROPOS missed (we discuss them in Section 5.2).

measurable behavioral changes. ATROPOS' advanced feedback mechanism and introspection capabilities allow for a more effective and efficient program exploration and vulnerability detection. Verifying our results for ZAP on DVWA, we find ZAP explores files containing the bugs, yet it does not set a key required to reach the vulnerable code section. Additionally, ZAP seems to be unable to change the difficulty level automatically (low, medium, high, impossible) and is thus limited to vulnerabilities of a single level. Even when manually setting a security level as pointed out in ZAP's FAQ, it only finds two bugs.

WFUZZ is unable to find any of the vulnerabilities because its crawling and bug finding capabilities are severely limited unless it is manually directed to a specific target URL with the correct key-value pairs setting GET, POST, and cookie parameters. Letting it fuzz on its own does not uncover any bugs. WITCHER is limited to SQL injections and remote code execution vulnerabilities; for these two, it finds three vulnerabilities in XVWA but fails to find any in the other two test suites. We believe this is not an inherent limitation of WITCHER but rather due to the limited nature of its academic prototype. However, WITCHER further confirms the notion that fuzzers are less likely to report false positives. Compared to the results reported for CEFUZZ [69], which finds all eight remote command/code execution bugs in both DVWA and bWAPP, we find that ATROPOS achieves the same result (but we also cover additional classes of bugs). As CEFUZZ does not report false positives, we cannot calculate the precision.

**Missed bugs.** Noteworthy, ATROPOS' median run with 40 cores only missed three bugs, two of which are upload-related vulnerabilities (in DVWA and XVWA) and the third is a remote command execution in XVWA. The latter could potentially be discovered given more time or better scheduling (manual inspection showed that ATROPOS did not fully cover the respective file). One of the missed upload vulnerabilities is likely due to the fact that HTTP uploads require a special `Content-Type` header, along with five additional key-value pairs to trigger an upload; in these cases, ATROPOS might have lacked a good seed to discover all the necessary keys to trigger the upload. This could be improved with a better feedback mechanism. The other upload-related vulnerability requires the input to pass a rarely seen check involving `getimagesize()`, which the application uses to detect if the upload looks like an image. We verified that ATROPOS passed the check and triggered this vulnerability by providing it with a seed input involving a short `GIF89` header. This shows the importance of a good seed corpus, which is orthogonal to our work. We used empty seeds for evaluation purposes.

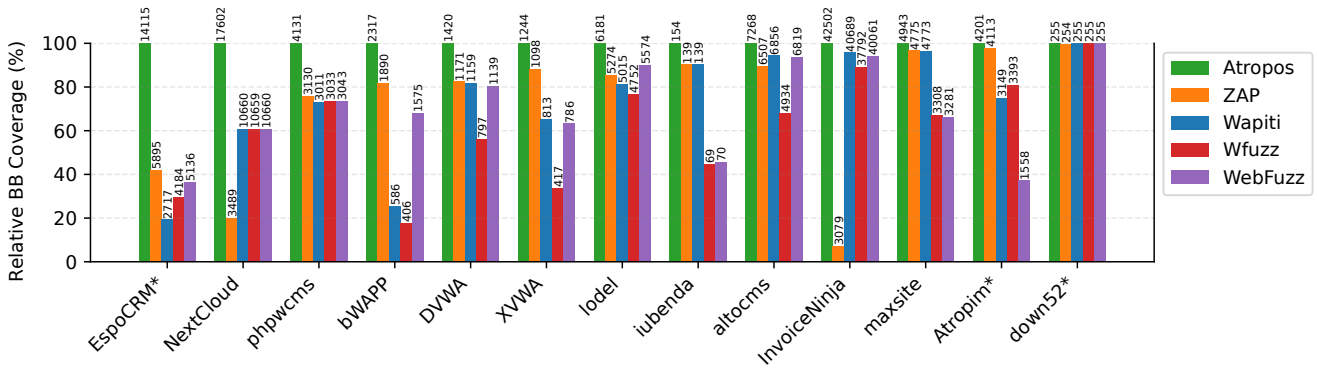


Figure 3: Basic block coverage relative to the best performing tool per target after 24h (some tools may terminate earlier). Single-page applications are marked with an asterisk (*). Numbers on top of the bars are the median basic block coverage.

**Second-order vulnerability.** Interestingly, one of the vulnerabilities in DVWA is a second-order vulnerability, which requires an attacker to first execute one PHP file and set a value stored inside a session, and then execute another PHP file where this value is used without sanitization in an SQL query. ATROPOS was able to trigger this vulnerability successfully. Of the static analysis and web scanning tools, only PHPCS-SECURITY-AUDIT detected this vulnerability; however, it also falsely considered many SQL queries in general as vulnerable. While PHPCS-SECURITY-AUDIT found all seven SQL injections in DVWA, it also reported an additional 27 false positives for this vulnerability class. A likely explanation is that ATROPOS does not need to perform taint analysis to track input across multiple execution steps, making this vulnerability class hard for the static-analysis tools to detect.

In summary, ATROPOS not only succeeds in finding significantly more bugs than state-of-the-art static analysis tools, but also has no false positives on all test suites.

## 5.3 Experiment 2: Code Coverage Evaluation

We evaluate against the coverage-guided fuzzer WEBFUZZ [46] and the black box web application fuzzer WFUZZ [31] as a baseline. We further measure coverage for ZAP and WAPITI to compare against web crawlers. As targets, we select the three intentionally vulnerable web applications as well as ten real-world web applications, including three single-page applications (SPAs)[2].

**Setup.** To make the code coverage results comparable between the tools, we use WEBFUZZ's instrumentation for all tools. The main reason is that WEBFUZZ modifies the source code of the application under test for instrumentation. Therefore, the line numbers are not comparable to ATROPOS' coverage data. We modify WEBFUZZ's instrumentation to dump code coverage information (filename and line number) into a file for later processing. This process allows for an tool-agnostic instrumentation, which enables a fair comparison. WFUZZ, a black box web application fuzzer, replaces the `FUZZ` keyword with payloads from a dictionary, requiring further action. We let WFUZZ run in two stages: (1) visit all possible PHP files once, then (2) fuzz all possible key-value pairs for `$_POST` for all possible PHP files using the provided dictionary consisting of common words. For all tools but ATROPOS, we manually remove the files responsible for logging out or resetting the database to avoid issues with maintaining state. ATROPOS does not require any such manual preprocessing. We keep the initial seed minimal, as per the recommendations of Klees et al. [25]. All tools have access to a list of accessible PHP files for the target and the cookies of a logged-in user. The web scanners ZAP and Wapiti are configured with the respective user login credentials. For ZAP, we select the Crawljax crawler for the three single-page applications (*espoCRM*, *AtroPIM* and *down_52_pojie* hereafter referred to as *down52*), as these are heavily dependent on JavaScript, and we use the default crawler for the remaining web applications. Unlike continuously running fuzzers, web scanners terminate as soon as they have nothing left to process, causing runs to end prematurely before the 24h timeout is reached.

We run all tools ten times for 24h on a single core and use the Mann-Whitney U test for a statistical significance analysis. We derive the average improvement by calculating the geomean over all relative improvements per web application.

**Results.** As visible in Figure 3, ATROPOS achieves the highest code coverage in 12 of 13 web applications; for one application, it is identical to the second-best tool.

On average, ATROPOS reaches 63% more basic blocks than ZAP, 46% more than WAPITI, 80% more than WFUZZ, and 50% more than WEBFUZZ. The differences in coverage are statistically significant at a significance level of $p < 0.05$ according to the Mann-Whitney-U test, except in the case of DOWN52, where nearly all tools and runs saturated the code coverage.

This experiment demonstrates that trying all possible in-

[2]For the SPAs, we picked the three PHP-based SPAs with most stars from https://github.com/topics/single-page-application.

Table 2: Static analysis tools and their performance on the real-world web application vulnerabilities. We indicate whether the vulnerabilities found by Atropos were within these reports (✓) or not (✗). *Oom* denotes that the tool ran out of memory during analysis, *to* indicates that it did not terminate within 24 hours, and "–" that the tool does not support the bug type.

| Web Application | ATROPOS | SONARQUBE | PROGPILOT | PSALM | PHPCS-SECURITY-AUDIT | Witcher | wfuzz | ZAP | Wapiti |
|---|---|---|---|---|---|---|---|---|---|
| AltoCMS | ✓ | ✗ | ✗ | ✗ | ✗ | – | ✗ | ✗ | *to* |
| nextCloud | ✓ | ✓ | ✗ | ✓ | ✗ | – | ✗ | ✗ | ✗ |
| Invoice Ninja | ✓ | ✓ | ✗ | ✓ | ✗ | – | ✗ | ✗ | ✗ |
| Iubenda | ✓ | ✗ | ✗ | ✓ | ✓ | – | ✗ | ✗ | ✗ |
| lodel | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| MaxSite | ✓ | ✓ | *oom* | ✓ | ✗ | – | ✗ | ✗ | ✗ |
| phpwcms | ✓ | ✓ | ✗ | ✓ | ✓ | – | ✗ | ✗ | ✗ |

puts is severely limited, as shown with WFUZZ, while simple web crawling mechanisms are more helpful as seen for ZAP, WAPITI and WEBFUZZ. The more critical factor is the ability to solve conditions typical for web applications (e. g., string comparisons) and discover keys and values not directly embedded in the HTML output. The design of ATROPOS enables us to do both, resulting in a significantly larger coverage.

**Execution Throughput.** Beyond code coverage, we have measured the number of executions performed by WEBFUZZ and ATROPOS on a sample basis for bWAPP and calculated that ATROPOS executed 2.3 times as many inputs as WEBFUZZ in the same time frame, despite the significant slowdown introduced by the additional WEBFUZZ instrumentation. Furthermore, WEBFUZZ is configured to use eight concurrent worker threads on a single core (while ATROPOS uses just one) and makes no attempt at restoring the environment. This demonstrates how avoiding HTTP requests (and thus the web server) in the fuzzing pipeline yields significant performance gains that allow us to perform other heavy-lifting tasks, such as restoring snapshots after each execution.

## 5.4 Experiment 3: Finding Real-world Bugs

We also investigate the capability of ATROPOS to identify software vulnerabilities in real-world code. We select projects by filtering for web applications that run on PHP 7.4, that were updated at least once in the last four years, and that have either acquired more than 100 stars on GitHub or are otherwise considered popular (e. g., they report a high number of users or are an established product for an extended period of time), for example in niche categories with a limited audience but widespread adoption among this audience. Table 6 in the appendix lists the tested applications.

We automated the testing process and fuzzed a total of 56 web applications. Overall, we found seven security-critical vulnerabilities and disclosed them to the developers in a coordinated way. We note that one of the projects has issued a CVE. These bugs affect four CMS platforms, one invoicing software, one WordPress plugin, and one common extension of a file-hosting software (see Table 6 in the appendix for details). One of the vulnerabilities is in a popular PHP module with more than 900,000 installs. We looked at a sample of products that import this module and found at least one additional vulnerable popular project, an invoicing solution with 200,000 active users. To verify whether the static analysis and web scanning tools find the same bugs or ATROPOS has an advantage, we run the tools on all targets and check if they find the bug reported by ATROPOS. We find that ATROPOS is the only tool to uncover all bugs, with the static analysis tools finding only about 40% of them on average. ZAP, WAPITI, and WFUZZ find none of the bugs. Witcher by design could potentially only find the bug in lodel, but also fails to do so. The full details are reported in Table 2.

**Case Study**. We discuss a remote code execution vulnerability with complex input handling that all four static analysis tools from the previous experiment failed to detect. ATROPOS has identified this vulnerability while fuzzing a CMS designed for scientific publishing. The web application provides a template system where the designer can place macros to be replaced later, as shown in Listing 4 in the appendix.

The template system does not only allow data to be placed at these specific points, but automatically identifies and executes PHP code by scanning for a PHP opening tag. While this approach to templating enables dynamic web design, the implementation also allows arbitrary macros to be set by user input. Critically, only `$_GET` variables are sanitized by stripping HTML and PHP tags, while `$_POST` data is inherited as is. This bug allows an unauthenticated user to overwrite the `CSSDATA` macro by setting the input via `$_POST` and thus execute it as PHP code. Importantly, no static analysis tool detected this vulnerability, most likely due to the complexity involved in tracking the input to the sink using their taint analysis. PHPCS-SECURITY-AUDIT notes that the `eval()` function should be avoided in general, but does not indicate a vulnerability.

This example demonstrates the interplay between our feedback mechanism and the ability to find vulnerabilities triggered by hard-to-track inputs through different layers for taint analysis. First, there is no reference to `CSSDATA` in any PHP file, only in the template macro, which most static string extractions are likely to miss since it is a custom format. In contrast, our feedback mechanism extracts these keys, as they are processed at runtime. Second, there are multiple steps

Table 3: Results of the ablation study for bWAPP, 24h runtime, three runs. Shown numbers are the median results.

| | Bugs | #BB Coverage |
|---|---|---|
| ATROPOS$_b$ | 0 | 1,096 |
| ATROPOS$_{bk}$ | 0 | 1,109 |
| ATROPOS$_{bkv}$ | 22 | 1,650 |
| ATROPOS$_{bkvc}$ | 23 | 1,714 |
| ATROPOS$_{bkvcr}$ | **24** | **1,742** |

from the POST data to the final destination in the template, none of which had to be computed by taint analysis in our tool. Instead, if a user-supplied input triggered a Remote Code Execution (RCE) vulnerability, it was detected by our bug oracle at runtime.

**False Positives.** During our fuzzing campaigns, ATROPOS reported three false positives. In two of the cases, ATROPOS reported a vulnerability where the behavior was explicitly desired, allowing administrators to issue arbitrary SQL queries (PHPMYADMIN) and upload arbitrary files (E107). In the third case, certain inputs caused the web application to trigger invalid SQL queries that resulted in a syntax error. While this is a bug, it was not related to the sanitization of attacker-controlled input and was not security-relevant.

## 5.5 Experiment 4: Ablation Study

We investigate the impact that individual components of ATROPOS contribute to its overall success. More specifically, we devise a baseline version of ATROPOS with all advanced feedback mechanisms deactivated (ATROPOS$_b$) and then add one feature after another. This way, we can measure the impact of the key extraction (ATROPOS$_{bk}$), value identification (ATROPOS$_{bkv}$), crawling optimization (ATROPOS$_{bkvc}$), and our regular expression handling (ATROPOS$_{bkvcr}$). We further study the overhead of our string comparisons, as hooking potentially costs a lot of performance, and disable this method in a variant (ATROPOS$_{no_str}$). This ablation study allows us to analyze the effectiveness of individual components and their impact on ATROPOS' performance.

We select bWAPP for this case study, as it comes with a ground truth and contains the most vulnerabilities of the test suites. We run all variants three times for 24h and measure the number of bugs discovered and the median code coverage achieved. We run this experiment on five Intel Xeon Gold 5320 CPU @ 2.20 GHz with 52 cores and 252GB of RAM.

**Results.** We observe that the number of errors found generally increases with the number of activated feedback mechanisms (see Table 3). ATROPOS$_b$ and ATROPOS$_{bk}$ both find zero bugs, while ATROPOS$_{bkv}$ reports 22 bugs, ATROPOS$_{bkvc}$ finds 23, and ATROPOS$_{bkvcr}$ locates 24 bugs. The largest impact is observable for the baseline ATROPOS$_b$, where all advanced feedback mechanisms were removed. Here, we see a drop from 24 found bugs in ATROPOS$_{bkvcr}$ to zero. Deactivating the key-value extraction also lead to a significant reduction in terms of coverage, as randomly generated strings are unlikely to explore deeper parts of the application and, hence, cannot trigger bugs hidden there. In fact, the median code coverage difference between ATROPOS$_b$ and ATROPOS$_{bkvcr}$ is +59%. While the relative difference between ATROPOS$_{bkv}$, ATROPOS$_{bkvc}$ and ATROPOS$_{bkvcr}$ are in the single-digit area percentage-wise, the effect may vary depending on the web application, i. e., one making heavy use of regular expressions will be impacted more by the removal of that advanced feedback mechanism. At the same time, the number of executions of the median run after 24h drops from 19,032,817 (ATROPOS$_b$) to 16,721,061 (ATROPOS), indicating the performance overhead of our instrumentation is 12% in terms of total executions. We believe the higher code coverage and bug finding capability justifies this performance decrease.

Running ATROPOS on bWAPP for one hour, we find that the difference to ATROPOS$_{no_str}$ is less than 1% in terms of total executions: ATROPOS$_{no_str}$ achieves roughly 720,000 executions and ATROPOS only about 715,000. This is because we use the advanced feedback only once for new queue entries and not for every execution.

# 6 Discussion and Limitations

Some of ATROPOS' aspects warrant further discussion.

**Resource Usage.** A significant difference between static analysis and dynamic approaches—especially fuzzing—are the resources required for the latter. While tools such as PSALM complete their analysis of evaluated test suites in less than a minute, ATROPOS requires several hours to produce comparable results. Although this may limit security analysts who need fast results, we argue that, in most scenarios, running an automated process, even with 40 CPU cores for 24 hours, is still significantly cheaper than hiring a security analyst to evaluate the reported results manually. This aspect is especially true for false positives, where a human analyst must spend significant time reviewing and rejecting the reported vulnerabilities. In contrast, ATROPOS provides the security analyst with a concrete input (e. g., as a *curl* command) that triggers the vulnerability, saving time that would otherwise be spent manually searching for inputs that reach a specific vulnerable line of code. Additionally, our evaluation shows that ATROPOS outperforms competing tools while running on a single core (see Table 1).

**Other Vulnerability Classes.** Currently, ATROPOS supports eight classes of server-side vulnerabilities. There are certain limitations to the oracles themselves, e. g., our upload oracle conservatively only detects whether arbitrary PHP files can be uploaded, but other extensions could also be considered dangerous. Furthermore, there are other vulnerability classes like *session fixation* [38] or *XML external entity* [53] that ATROPOS currently cannot detect. However, these vulnerabilities are less common, and we focused on covering the

most common vulnerability types. The engineering effort to add support for new bug oracles varies depending on the oracle type and its complexity. Most of our bug oracles require only small changes to the interpreter, limiting the engineering effort required to implement them. In addition, we ignored client-side vulnerabilities such as XSS, as these would require a client-side component. Recent work covers such bugs [46].

Beyond these bug classes, applications may contain logical faults specific to their functionality, e. g., a user with low privileges should not be able to read profile information from administrator accounts. ATROPOS cannot detect these violations because they are specific to a particular application.

**Improper Sanitization.** Our bug oracles are based on the insight that inputs may end up essentially unchanged as arguments for functions. Hence we can efficiently detect whether an input ends up in a sink without needing expensive taint analysis. Although this approach empirically seems to work well, improper sanitization attempts may change the input to an unrecognizable degree. We argue that in most cases, there is only one way to sanitize user input for a particular use case, e. g., shell arguments must be escaped with `escapeshellarg()` before they are executed. None of the sanitization solutions in PHP we examined changed the alphanumeric magic trigger string to an undetectable degree. Note that DVWA exhibited some non-standard custom sanitization attempts, but ATROPOS was still able to track the input to the sink. Theoretically, there could be cases where PHP developers perform their custom sanitization that renders our methods ineffective. We have not been able to observe this behavior ourselves.

**Execution Speed.** Since we need to run the web application with (informed) random inputs, the application must execute as fast as possible. While web applications like DVWA can achieve 200-400 exec/s on a single CPU core (which scales to 4,000 exec/s distributed over 40 cores), we have found that heavyweight applications are particularly slow, e. g., WORDPRESS took about 800 ms to process a single request (even without ATROPOS). Complex web applications like this would require many CPU cores to achieve a desirable number of executions per second. While the same problem applies to binary fuzzing (e. g., fuzzing complex applications like web browsers), we believe there is room for improvement by using incremental snapshots to skip the initialization part of the web application [49] or focusing only on fuzzing specific functions, similar to LIBFUZZER [29].

**False Negatives and False Positives.** Checking whether the attacker-controlled input is contained in the sink, which is the second precondition all our oracles use for bug detection, may lead to false positives or false negatives. False positives happen when, for example, attacker-controlled input *is* sanitized, yet the sanitized string still causes a warning or error to be generated. In this case, our oracles would observe both suspicious function behavior (generated error) and attacker-controllable input in the sink, even though the input may have been sanitized properly. Empirically, we found only a single occurrence of such a false positive across all 59 tested targets.

False negatives may occur due to our conservative reporting: We only report arbitrary read/write bugs when PHP files are involved, as web applications may write other files during normal processing; however, this means, we may not report such bugs when only specific, non-PHP files can be written/read by an attacker. Similarly, we only report a file upload vulnerability if a PHP file can be uploaded. This means, we may miss bugs limited to other file types.

**Web Scanners.** Compared to web scanners based on crawling, such as ZAP [68], WAPITI [56], or BURP SUITE [45], using fuzzing has a fundamental advantage: Instead of having to crawl the web application's output to find further web pages, our approach can directly test all PHP files via FastCGI. Other than web scanners, it is not driven by exploring new pages but by exploring new code coverage, which is a more fine-granular distinction as a single web page may contain multiple code paths. Our evaluation shows that fuzzing is more effective in uncovering bugs in web applications than web scanners. The advantage of Atropos is that it can explore different functions of a web application, which existing approaches may fail at. Figure 3 also shows that Atropos has a higher overall code coverage. However, Atropos may suffer from analyzing code that is deeply embedded and triggered only by several state-dependent page redirections because theoretically, the analysis time could be exponential in the worst case. For example, if a web application contains a bug that is only triggered if a user adds hundreds of different items to a shopping cart before proceeding to checkout, Atropos may fail to uncover the bug. Similarly, a bug might not be found if it is hidden after a complex setup wizard of an application, which connects many files that must be traversed in the correct order. In these cases, web crawlers may be more efficient at traversing the page redirections, as they are designed to efficiently follow page connections. In the future, Atropos can be improved to use HTML crawling to dump relevant information and just use it for the next fuzzing input.

## 7 Related Work

In the past decade, there has been much research on securing web applications. Previous research mainly focused on static analysis methods [6, 12, 13, 15, 23, 51, 60, 64], while several dynamic methods were proposed [1, 2, 5, 17, 22, 46, 58, 69].

The latter category is more closely related to our approach: WEBFUZZ [46] uses grey-box fuzzing methods to instrument and control its mutations but focuses only on XSS and reflected XSS vulnerabilities. Moreover, our evaluation shows that WEBFUZZ does not reach deeper parts of code because its mutation strategy lacks critical runtime information that ATROPOS has access to due to our advanced feedback mechanism. Another work is CEFUZZ [69], which first performs static analysis and then uses fuzzing methods to find remote command/code execution vulnerabilities. Both approaches

do not use snapshots to preserve the state of the web application, suffer from the overhead of running a web server, and focus on only one or two bug classes. Closest to our work is WITCHER, a concurrent fuzzer that, just as ATROPOS, combines coverage feedback with custom bug oracles. While WITCHER supports web applications in multiple languages, it only supports two bug types, SQL injections and remote code execution vulnerabilities. ATROPOS, on the other hand, focuses on PHP but supports eight vulnerability types and proposes advanced feedback mechanisms, extracting useful information directly from the PHP interpreter. The difference of focus translated also to the respective architectures: WITCHER achieves generality by using a web server as a proxy to the web application. Instead of deploying advanced feedback to infer keys or values, it ships an HTTP mutator, allowing it to make informed mutations. In contrast, ATROPOS focuses exclusively on PHP, allowing us to deploy our advanced feedback mechanism to the interpreter and placing our bug oracles within the interpreter. ATROPOS further uses snapshots and directly communicates with the web application to reduce the overhead.

Concerning file upload vulnerabilities, several works focus exclusively on this class of bugs, e. g., UCHECKER [21], FUSE [27], or UFUZZER [22]. While UCHECKER uses symbolic execution and SMT solvers, FUSE sends actual upload requests to the web server. UFUZZER uses static analysis to find the vulnerability and then applies fuzzing to confirm the results. This combination reduces the false positive rate to zero, further confirming that the actual execution of the input via fuzzing can be used to reduce false positive results.

Other work, including BackREST [17] and RESTler [5], focus on testing the APIs exposed by a web application, making them an orthogonal approach to Atropos. RESTler analyzes the API specification and derives sequences of requests to test them, whereas ATROPOS uses feedback to infer interesting inputs for the web applications. Similar to ATROPOS, closed-source BackREST uses a feedback-driven fuzzer but additionally relies on taint tracking, which ATROPOS avoids for performance reasons. BackREST uses a state-aware crawler to explore JavaScript-heavy Node.JS applications and test a maximum number of endpoints. ATROPOS does not focus on API testing and has no crawler; it could be interesting future work to combine ATROPOS with API inference and state-aware crawling.

## 8 Conclusion

In this work, we have shown how binary fuzzing principles can be extended and adapted to be used in the context of web applications to effectively and efficiently detect server-side vulnerabilities. Our work provides a practical alternative to static analysis for detecting vulnerabilities in web applications.


## Acknowledgements

This work was funded by the European Research Council (ERC) under the consolidator grant RS$^3$ (101045669) and the German Federal Ministry of Education and Research under the grant KMU-Fuzz (16KIS1898). This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2092 CASA – 390781972.


## References


[1] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained Automated Workflow-based Exploit Generation. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[2] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *USENIX Security Symposium*, 2018.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: Stateful REST API fuzzing. In *International Conference on Software Engineering (ICSE)*, 2019.

[6] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *European Symposium on Security and Privacy (EuroS&P)*, 2017.

[7] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F Donaldson. Combining Static Analysis Error Traces with Dynamic Symbolic Execution (Experience Paper). In *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[8] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*, 2019.

[10] Colm O'Connor. Library to generate random strings from regular expressions. https://github.com/crdoconnor/xeger. Accessed: October 9, 2023.

[11] CWE Content Team. CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion') (4.8). https://cwe.mitre.org/data/definitions/98.html. Accessed: October 9, 2023.

[12] Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[13] Johannes Dahse and Thorsten Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2014.

[14] Design Security. A Static Analysis Tool for Security. https://github.com/designsecurity/progpilot. Accessed: October 9, 2023.

[15] Yong Fang, Shengjun Han, Cheng Huang, and Runpu Wu. TAP: A Static Analysis Model for PHP Vulnerabilities based on Token and Deep Learning Technology. *PloS one*, 2019.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[17] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BackREST. In *European Conference on Object-Oriented Programming (ECOOP)*, 2022.

[18] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[19] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Technical Report: Effective Fuzzing of Web Applications for Server-side Vulnerabilities. , 2023.

[20] Security-oriented Fuzzer with Powerful Analysis Options. https://github.com/google/honggfuzz. Accessed: October 9, 2023.

[21] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. UChecker: Automatically Detecting PHP-based Unrestricted File Upload Vulnerabilities. In *Conference on Dependable Systems and Networks (DSN)*, 2019.

[22] Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. UFuzzer: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[23] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[24] Hong Jin Kang, Khai Loong Aw, and David Lo. Detecting False Alarms from Automatic Static Analysis Tools: How Far Are We? In *International Conference on Software Engineering (ICSE)*, 2022.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[26] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/. Accessed: October 9, 2023.

[27] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. FUSE: Finding File Upload Bugs via Penetration Testing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[28] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2018.

[29] LibFuzzer. https://www.llvm.org/docs/LibFuzzer.html. Accessed: October 9, 2023.

[30] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[31] Xavi Mendez. Wfuzz: The Web fuzzer. https://www.wfuzz.io. Accessed: October 9, 2023.

[32] Malik Mesellem. bWAPP, a buggy web application. http://www.itsecgames.com/. Accessed: October 9, 2023.

[33] Larisa Moroz. bWAPP latest modified for PHP7. https://github.com/lmoroz/bWAPP. Accessed: October 9, 2023.

[34] OWASP Foundation. Code Injection. https://owasp.org/www-community/attacks/Code_Injection. Accessed: October 9, 2023.

[35] OWASP Foundation. Command Injection. https://owasp.org/www-community/attacks/Command_Injection. Accessed: October 9, 2023.

[36] OWASP Foundation. OWASP Top Ten 2017. A8:2017-Insecure Deserialization. https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization. Accessed: October 9, 2023.

[37] OWASP Foundation. Server Side Request Forgery. https://owasp.org/www-community/attacks/Server_Side_Request_Forgery. Accessed: October 9, 2023.

[38] OWASP Foundation. Session Fixation. https://owasp.org/www-community/attacks/Session_fixation. Accessed: October 9, 2023.

[39] OWASP Foundation. SQL Injection. https://owasp.org/www-community/attacks/SQL_Injection. Accessed: October 9, 2023.

[40] OWASP Foundation. Unrestricted File Upload. https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload. Accessed: October 9, 2023.

[41] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. Fugio: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *USENIX Security Symposium*, 2022.

[42] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[43] PHP. FastCGI Process Manager (FPM). https://www.php.net/manual/en/install.fpm.php. Accessed: October 9, 2023.

[44] PHP. PHP: OPcache - Manual. https://www.php.net/manual/en/book.opcache.php. Accessed: October 9, 2023.

[45] PortSwigger Ltd. Burp Suite. https://portswigger.net/burp, 2003.

[46] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webFuzz: Grey-Box Fuzzing for Web Applications. In *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[47] Sergej Schumilo and Cornelius Aschermann. Nyx Framework. https://github.com/nyx-fuzz, 2021. Accessed: October 9, 2023.

[48] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[49] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys)*, 2022.

[50] Kostya Serebryany. OSS-Fuzz-Google's Continuous Fuzzing Service for Open Source Software, 2017.

[51] Sooel Son and Vitaly Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2011.

[52] SonarSource. Code Quality and Code Security | SonarQube. https://www.sonarqube.org/. Accessed: October 9, 2023.

[53] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML Parser Vulnerabilities. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[54] SQLite Team. How SQLite Is Tested. https://www.sqlite.org/testing.html, 2022.

[55] Squiz Labs. PHP_CodeSniffer tokenizes PHP files and detects violations of a defined set of coding standards. https://github.com/squizlabs/PHP_CodeSniffer. Accessed: October 9, 2023.

[56] Nicolas Surribas. Wapiti: The Web-application Vulnerability Scanner. https://wapiti-scanner.github.io/, 2006.

[57] Sanoop Thomas. XVWA is a badly coded web application written in PHP/MySQL that helps security enthusiasts to learn application security. https://github.com/s4n7h0/xvwa. Accessed: October 9, 2023.

[58] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a Fault to your Witcher: Applying Grey-box Coverage-guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[59] vimeo. Psalm - A Static Analysis Tool for PHP. https://psalm.dev/. Accessed: October 9, 2023.

[60] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[61] Joe Watkins. PCOV: A self contained CodeCoverage compatible driver for PHP. https://github.com/krakjoe/pcov. Accessed: October 9, 2023.

[62] Web Technology Surveys. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php. Accessed: October 9, 2023.

[63] Robin Wood. Damn Vulnerable Web Application (DVWA). https://github.com/digininja/DVWA. Accessed: October 9, 2023.

[64] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.

[65] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[66] Michał Zalewski. AFL/config.h. https://github.com/google/AFL/blob/61037103ae3722c8060ff7082994836a794f978e/config.h#L229. Accessed: October 9, 2023.

[67] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: October 9, 2023.

[68] ZAP Dev Team. Zed Attack Proxy (ZAP). https://www.zaproxy.org/, 2012.

[69] Jiazhen Zhao, Yuliang Lu, Kailong Zhu, Zehan Chen, and Hui Huang. Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability. *Electronics*, 11(5):758, 2022.

# A Additional Information

We report versions of the test suites (Table 4), the functions we hook (Table 5), and details on real-world apps (Table 6).

Table 4: Additional information on the used test suites.

| Test Suite | Commit | Information |
|---|---|---|
| DVWA | `#423ac71` | 7 x SQL injection,<br>3 x Remote Command/Code Execution,<br>3 x File Inclusion,<br>3 x File Upload |
| XVWA | `#fb30fa5` | 3 x Remote Code/Command Execution,<br>1 x PHP Object Injection,<br>1 x SSRF,<br>2 x SQL Injection,<br>1 x File Inclusion,<br>1 x File Upload |
| bWAPP | `#f3f423c` | 18 x SQL Injection,<br>5 x Remote Code/Command Execution,<br>2 x File Upload,<br>1 x Arbitrary Read,<br>1 x File Inclusion |

Table 5: Functions hooked by each bug oracle.

| Bug Oracle | Hooked Functions |
|---|---|
| SQL Injection | We hook any query that goes through MySQL (`mysqli_query()`) or produces an error in SQLite (`_pdo_sqlite_error()`). |
| Remote Code Execution | `eval()`, `call_user_func()` and `call_user_func_array()`. |
| Remote Command Execution | We hook PHP's internal `php_exec_ex()` function (which multiple PHP functions end up calling, like `system()` and `exec()`) as well as `proc_open()` and `shell_exec()`. |
| PHP Object Injection | `unserialize()` |
| Local and Remote File Inclusion | We hook PHP's internal error reporting functions to catch the filepath and the reports indicating that the included file does not exist (`php_verror()` and `php_error_cb()`). |
| Server-side Request Forgery | `file_get_contents()`, `socket_connect()`, `file_put_contents()`, `file()`, `readfile()`, `php_if_open()`, `fsockopen()` and `stream_socket_client()`. |
| Arbitrary File Read and Write | `file_get_contents()`, `file_put_contents()`, `fwrite()`, `readfile()`, `rename` and `copy()`. |
| File Upload | `move_uploaded_file()`. |

Table 6: Overview of all web applications we have fuzzed with Atropos and the vulnerabilities we have discovered. Some details have been anonymized and omitted due to responsible disclosure procedures.

| Vulnerability | Web Apps | Web Technologies | Notes |
|---|---|---|---|
| PHP Object Injection | AltoCMS | Multi-page CMS, MySQL, MVC | Session cookie gets fed into unserialize without protection. |
| | MaxSite | Multi-page website-builder, MySQL | While one session cookie was protected with HMAC, another one was not. |
| | phpwcms | Multi-page CMS, MySQL | Information is transferred via serialize and unserialize instead of JSON. |
| Server-Side Request-Forgery | InvoiceNinja | Laravel, Flutter, React | 200k+ installs. Same vulnerability as "nextcloud" below. |
| | Iubenda | WordPress Plugin | Popular GDPR compliance WordPress plugin with 100k+ active installs, allows request to local network HTTP services. |
| | Nextcloud | JavaScript-heavy, Vue | CVE-2022-31132. Mail extension by Nextcloud, recommended after setup, contains an SSRF in a third-party module (csstidy, 800k+ downloads), which also affects many other products, including Invoice Ninja above. |
| Remote Code Execution | lodel | Multi-Page, MySQL, Templating | Template system executes user input as PHP code. |
| SQL Injection | lodel | | A parameter is fed into an SQL query unsanitized. |
| No vulnerability detected | Ampache, AsgardCMS, b2evolution, Bigtree, Bolt, Carbon-forum, Chamilo, Cockpit, ConcreteCMS, CoreShop, CouchCMS, Croogo, CubeCart, DaybydayCRM, DokuWiki, dolibarr, e107, ForkCMS, GetSimpleCMS, GLPI, grav, HotCRP, kanboard, Kirby, LibreNMS, LimeSurvey, MantisBT, Matomo, MediaWiki, Microweber, MODX, MyBB, OpenEMR, PageKit, phpBB, phpMyAdmin, pico, Piwigo, PrestaShop, qdPM, Serendipity, SimplePie, slimCMS, SPIP, SuiteCRM, SymphonyCMS, Thelia, TypiCMS, WordPress, Zen Cart | | |

```
<form action="<?php echo $PHP_SELF; ?>">
<IF CONDITION="[#CSSDATA]">
  <style type="text/css">[#CSSDATA]</style>
</IF>
```

```
foreach($_GET as $k=>$v)
    macro_vars[$k] = strip_tags($v);
foreach($_POST as $k=>&$v)
    macro_vars[$k] =& $v;
```

Listing 4: Sample macro file (top), simplified for readability, and vulnerable PHP file (bottom).