Predator: Directed Web Application Fuzzing for Efficient Vulnerability Validation

Chenlin Wang
The Chinese University of Hong Kong
clwang23@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

Changhua Luo
The Chinese University of Hong Kong
Wuhan University
chluo@cse.cuhk.edu.hk

Penghui Li
Columbia University
pl2689@columbia.edu

***Abstract*—Web application vulnerabilities continue to pose a significant challenge. Static analysis is currently the mainstream approach to this issue, while dynamic analysis is not as widely used in comparison. However, both techniques have their limitations. While current static analysis tools are plagued by high false-positive rates, necessitating fine-grained analysis and substantial expertise, it is also the case that dynamic analysis tools are underdeveloped. Current fuzzing-based tools are often limited by inefficiency in exploring deeper code locations. Moreover, state-of-the-art grey-box fuzzers often struggle to capture effective parameters from user interfaces, thereby failing to explore the input space efficiently.**

In this paper, we propose PREDATOR, a directed fuzzing framework equipped with selective dynamic instrumentation for effective and efficient web application vulnerability detection and validation. We use static analysis techniques and dynamic analysis techniques to complement each other. Our lightweight static analysis provides relevant URLs and parameters of the directed fuzzing targets and thus facilitates dynamic validation of static analysis reports. Additionally, we propose a runtime distance supplementation mechanism and tailored mutation strategies to address the dynamic features of interpreted languages like PHP. The evaluation shows PREDATOR effectively triggers more vulnerabilities and outperforms state-of-the-art grey-box fuzzers by up to 43.8 times in terms of time to exposure. Moreover, PREDATOR detects 26 previously unknown vulnerabilities in real-world applications, further demonstrating its effectiveness. At the time of writing, 7 of the 26 vulnerabilities have been confirmed and patched by the corresponding vendors.

1. Introduction

Web applications are extensively deployed in Internet services, powering a variety of functionalities, from online shopping to social networking. These applications, irrespective of the underlying technology, are susceptible to a wide range of security threats. These threats can result in severe consequences, including data breaches, financial losses, and reputational damage [1]. Common security threats across web applications include SQL injection [2], command injection [3], and cross-site scripting (XSS) [4]. PHP is one of the most popular server-side programming languages for developing web applications, powering over 76% of websites [5]. As an interpreted language, PHP offers developers flexibility and ease of use. However, this flexibility can often result in security vulnerabilities, making PHP applications particularly prone to these threats.

To mitigate these threats, security analysts have developed various techniques to detect vulnerabilities in web applications. For instance, static analysis methods such as TChecker [6] and RIPS [7] can scalably screen the source code to identify suspicious security issues using predefined heuristics. Unfortunately, they often suffer from potential high false-positive rates, a situation that is counterproductive [8–11]. Additionally, even state-of-the-art static analysis tools may not be able to fully model the dynamic features of PHP, which can lead to imprecise results [6].

Dynamic analysis methods, such as black-box scanning [12–14] and grey-box fuzzing [15–19], aim to identify vulnerabilities by executing the application and observing its behavior. Witcher [16] leverages AFL [20] to detect SQL injection and command injection vulnerabilities for web applications in multiple languages. By converting semantic errors into segmentation fault signals detectable by AFL through customized bug oracles, it enhances the effectiveness of detecting injection vulnerabilities. Concurrent to our work, Atropos [18] employs a snapshot-based, feedback-driven fuzzing method tailored for PHP-based web applications. It introduces a feedback mechanism to automatically infer the key-value structure at runtime. Dynamic analysis methods typically have lower false-positive rates compared to static analysis methods as they can observe the actual behavior of the application, rather than relying on heuristics.

Despite the advantages, dynamic methods also exhibit significant limitations, primarily due to the intricate architecture and state-dependent nature of modern web applications [21, 22]. Modern web applications often involve complex user interactions and maintain rich stateful behaviors

across multiple sessions and interactions. Although coverage-guided fuzzers are designed to navigate through these states, they inadvertently expend energy on non-critical areas instead of focusing on truly buggy states. As a result, certain errors that only manifest under specific states or conditions may remain undetected. Furthermore, with the integration of various third-party modules and libraries, ensuring thorough coverage becomes increasingly difficult. Each additional layer of complexity not only makes it harder to predict application behavior but also increases the resources and time required for effective testing. This underscores the need for developers to complement these methods with more sophisticated testing strategies, focusing on high-risk code locations instead of exhaustively exploring all possible states.

We aim to harness the strengths and mitigate the weaknesses of these methods. Employing directed fuzzing to validate static analysis reports offers significant potential for enhancing the detection of vulnerabilities in web applications. By concentrating resources on identified high-risk targets generated by static analysis, we can reduce the number of states requiring exploration in dynamic testing. Directed fuzzing can, in turn, lower the costs of manual audits introduced by the high false-positive rates of static analysis. Unfortunately, applying widely-used directed fuzzing techniques to web applications is non-trivial, primarily because of the characteristics inherent to the interpreted languages used in these applications, such as PHP. For example, weak typing and high levels of dynamism impact the precision of static analysis, which is crucial for preparation work like distance calculation.

Although directed fuzzing technique is well-developed for native programs [23–29], it is not widely available for web applications. To the best of our knowledge, Cefuzz [30] is the only directed fuzzer for exclusively detecting remote code execution (RCE) vulnerabilities. However, there are several limitations to Cefuzz. 1) Cefuzz uses a *coarse-grained feedback mechanism*, which fails to accurately reflect the distance between the seed and the target. It prioritizes seeds based on the number of reachable basic blocks (*i.e.*, blocks from which a path to the target exists on the control-flow graph (CFG)) they pass through, which does not accurately reflect the potential of the seeds. When there are multiple paths of different lengths that can eventually reach the target code, Cefuzz tends to give priority to the seed that takes the longest path. 2) Cefuzz performs *static instrumentation*, which increases the script length and risks altering the behavior of the application. It inserts echo statements before and after statements using the concatenation (*i.e.*, `new_line = "echo('x');\n".old_line."echo('y');\n";`). In such cases, if the branch statements do not correctly use curly braces to encapsulate the statements, it can lead to unpredictable consequences. 3) Cefuzz is *closed-source*, which limits its further development and usability.

In this paper, we propose PREDATOR, an efficient directed fuzzing tool for web applications. By selectively instrumenting the web application at runtime, PREDATOR stores inter-procedural block distances without altering the source code. The distance feedback prioritizes in test cases that are closer to predefined high-risk code locations, thereby avoiding inefficient random state exploration. Moreover, PREDATOR updates the block distances at runtime to correct the imprecise static analysis results and enable exploring more promising paths. PREDATOR employs novel tailored mutation strategies for web applications, which are guided by distance and parameters. It probabilistically adopts different mutation strategies based on the current input distance to enhance effectiveness in both exploration and exploitation stages. By employing parameter-to-condition analysis, PREDATOR concentrates mutations on a group of parameters involved in condition expressions, targeting deeper areas that current fuzzers struggle to reach.

To evaluate the effectiveness and efficiency of PREDATOR, we conducted a comprehensive assessment on both synthetic and real-world web applications, covering a wide range of vulnerabilities. Compared to state-of-the-art fuzzers, PREDATOR detected more vulnerabilities and triggered them up to 43.8 times faster. We demonstrated that each component of PREDATOR contributed to the overall performance. We showcased the effectiveness of PREDATOR in validating static analysis reports and reducing the manual auditing efforts. When applying PREDATOR to detect new vulnerabilities, we identified 26 previously unknown vulnerabilities in real-world web applications, 7 of which have been acknowledged and patched.

In summary, the contributions of this paper are as follows:

- We propose PREDATOR, an efficient directed fuzzing tool for web applications to detect SQL injection, command injection, and XSS vulnerabilities.
- We introduce novel techniques to enable and enhance directed fuzzing for web applications, including selective dynamic instrumentation and tailored mutation strategies guided by parameters and distance.
- We evaluate PREDATOR on 3 test suites and 11 real-world web applications. The results show that PREDATOR detects more vulnerabilities and triggers them more efficiently than state-of-the-art tools.
- We find 26 previously unknown vulnerabilities in real-world web applications with PREDATOR. At the time of writing, 7 of them have been patched, 6 new CVE IDs (`CVE-2024-404[47-52]`) have been assigned.

To foster the future research in this area, we will release the source code at https://github.com/cuhk-seclab/Predator.

2. Background and Motivation

In this section, we provide high-level overview of web application fuzzing (§2.1) and directed fuzzing for web applications (§2.2), followed by the research motivation (§2.3).

2.1. Web Application Fuzzing

Fuzzing has been extensively applied to detect vulnerabilities in a wide range of software systems, including web applications. They can generally be categorized based on the information they rely on, falling into two main groups:

black-box fuzzing (scanning) and grey-box fuzzing. Black-box scanning operates under the assumption of no prior internal information about the target web application. For instance, Black Widow [12] and Enemy of the State [13] exemplify black-box approaches by modeling navigation and constructing the state graph of web applications without prior internal knowledge. On the other hand, grey-box fuzzing assumes the availability of such information, such as code coverage feedback. Atropos [18] applies snapshot-based techniques to fuzz web applications and infers parameter keys and values at runtime. WebFuzz [15] instruments the target web applications for code coverage feedback and Witcher [16] enhances the runtime.

Witcher tests web applications using PHP common gateway interface (CGI) binary. The fuzzing process orchestrated by Witcher unfolds in two distinct phases. Initially, it functions as a black-box crawler that discovers pages and parameters. In the second phase, Witcher utilizes the discovered request information as the foundational seeds for the fuzzer. Following this, Witcher generates random payloads to populate *GET*, *POST*, and *COOKIE*. To enhance the effectiveness of the fuzzing process, Witcher incorporates an HTTP parameter mutator and HTTP dictionary mutator. The former facilitates the cross-pollination of interesting parameter names and values, while the latter introduces new key-value pairs to the test cases. A dynamic coverage feedback mechanism based on the PHP opcodes is also implemented to guide the fuzzing process.

2.2. Directed Fuzzing for Web Applications

Directed fuzzing specially tests target code of interests. Its effectiveness has been demonstrated primarily in the context of native applications, as exemplified by the success of tools like AFLGo [23]. However, the application of directed fuzzing to web applications is relatively limited. Cefuzz [30] is a directed fuzzer for detecting RCE vulnerabilities in web applications. It first conducts static taint analysis to identify potential sinks. Subsequently, it instruments the source code to trace the execution paths. It prioritizes seeds by measuring the number of reachable basic blocks they pass through without relying on any distance metrics. Unfortunately, its static instrumentation method not only introduces additional code but also fails to correctly perform seed selection when the application is complex. It is ineffective when static analysis results are imprecise due to the dynamic features of interpreted languages like PHP. Moreover, it only detects RCE vulnerabilities, and its source code is not available.

2.3. Motivation

Limitations of Existing Methods. Both static and dynamic analysis techniques have their limitations. Static analysis generates reports with many false positives [6, 10, 11, 31]. This issue mostly stems from the fact that static analysis evaluates code without executing it, leading to outcomes based on theoretical inferences rather than actual runtime behavior. As a result, the tool might label code as vulnerable when certain conditions for a vulnerability to be triggered may never materialize at runtime. Furthermore, dynamic languages such as PHP introduce complexities in their features and execution behaviors, which static analysis tools often struggle to precisely model [31]. These issues necessitate extensive manual review and validation, which becomes a bottleneck in the efficiency of static analysis tools. Dynamic analysis tools for web applications primarily utilize coverage-guided strategies. They implement random exploration techniques, aiming to traverse through various execution paths in search of potential security vulnerabilities. However, this approach faces limitations due to the intrinsic uneven distribution of vulnerabilities within an application, with most vulnerabilities often residing in specific, critical functions. This approach makes random exploration inefficient, as it does not focus on code segments with higher likelihood of containing vulnerabilities. Moreover, deep vulnerabilities require navigating through complex execution paths, which existing coverage-guided fuzzing might not effectively address.

Note that while vulnerability information can often be sourced from multiple avenues, such as static analysis tools and databases of known CVEs, a practical proof of concept (PoC) to exploit these vulnerabilities frequently remain unavailable. Therefore, it is crucial to develop a tool capable of both identifying and effectively triggering these known vulnerabilities to validate their impact and scope.

A Promising Approach to Bridge the Gap. Utilizing directed fuzzing for automated static analysis report validation shows great promise in web application vulnerability detection. By leveraging directed fuzzing, we can mitigate the extensive manual review and validation workload, primarily induced by the high false-positive rates of static analysis. This is particularly beneficial for dynamic programming languages like PHP, where the intricacies and dynamic features pose modeling challenges for static analysis.

3. Challenges

Directed fuzzing achieves remarkable success in native programs. However, applying it to web applications faces unique challenges. We list the primary ones below.

Challenge 1: Providing Entry URLs and Request Parameters in a Targeted Manner. Fuzzing web applications requires identifying both entry URLs and request parameters. In web applications, *request parameters* are defined as key-value pairs submitted by clients. Each parameter consists of a key (a unique identifier) and a value (the corresponding data). For targeted vulnerability detection via directed fuzzing, the potentially vulnerable code locations are only accessible through specific *entry scripts*. These entry scripts are primarily associated with the front-end components of the applications. They are designed to provide users with access to the application's functionalities. The URLs of these entry scripts are referred to as the *entry URLs* for the targets.

Existing tools usually rely on crawlers [12, 13, 16, 17, 32] and manual efforts [15, 33, 34] to identify URLs. While

effective in coverage-guided fuzzing, crawlers randomly extract URLs, potentially missing critical entry URLs that lead to vulnerable code locations and cluttering the dataset with irrelevant URLs. Manually identifying entry URLs is inefficient and does not align with our research goal.

Triggering specific vulnerabilities necessitates manipulating both parameter keys and values [35]. Fuzzers often struggle when lacking the necessary parameters to target vulnerabilities. Crawlers extract request parameters from the user interface but lack insight into the application's internal logic. Such superficial information fails to assist directed fuzzers in triggering deeper vulnerabilities.

Challenge 2: Instrumenting Web Applications and Gathering Distance Feedback. Instrumenting web applications for directed fuzzing is non-trivial. Existing coverage-guided tools primarily employ two methods: static instrumentation by modifying source code [15, 30, 34], and dynamic instrumentation during runtime [16]. Current static instrumentation methods cannot insert necessary hooks without altering the source code, which risks enlarging the codebase and even changing the application's behavior unintentionally [15]. Moreover, dynamic languages like PHP may require adjustments to block distances at runtime due to potential inaccuracies in static analysis results. Given that the distance information is embedded within the source code, any required adjustments necessitate re-instrumenting the entire application, which is excessively time-consuming. Dynamic instrumentation involves instrumenting bytecode during program execution. Witcher [16] leverages the line number and opcode of the current and prior bytecode instructions to update the code coverage. On the one hand, it may be expensive to instrument every bytecode instruction. On the other hand, directed fuzzing requires computing and maintaining the distance of each basic block, which bytecode-level instrumentation cannot fully address.

In addition to instrumenting the web application code, another common approach involves instrumenting the PHP interpreter, which is written in C, and testing with a fuzzer for native programs [20]. By analyzing the coverage of the interpreter's code, it may implicitly reflect the coverage of the target web application. However, this method is impractical for directed fuzzing purposes. First, it results in a significant amount of noise. Even if we run a simple PHP script containing a few lines of code, the interpreter would execute a large number of instructions. Second, instrumenting the interpreter collects execution traces for the interpreter but leaves the fuzzer unaware of the actual execution flow of the web application's code. When performing directed fuzzing, the fuzzer requires accurate distance feedback from the target application, not the interpreter.

Challenge 3: Addressing Dynamic Natures of Interpreted Languages. Directed fuzzing typically requires preparatory work based on static analysis. For instance, tools utilizing a distance metric need to initially calculate and assign distances to basic blocks based on the control flow of the target application [23, 26]. During dynamic testing, they collect distance feedback and conduct power scheduling, prioritizing

```php
<?php
class ClassA {
    public static function foo($action, $query) {
        $attempt = 0;
        while (!validate($query) && $attempt < 3) {
            $query = sanitize($query);
            $attempt++;
        }
        if ($action === 'lookup')
            benign_func($query);
        else if ($action === 'edit')
            vuln_func($query);
        ...
    }
}

class ClassB {
    public function foo($action, $query) {
        if ($action === 'edit')
            if ($_SERVER['STATUS'] === 'LOGGED_IN')
                vuln_func($query);
        ...
    }
}

$action = $_REQUEST['a'] ?? 'defaultAction';
$query = $_REQUEST['q'] ?? 'defaultQuery';

ClassA::foo($action, $query);
...
$cName = 'ClassB';
$mName = 'foo';
(new $cName())->$mName($action, $query);
```

Listing 1: An example server-side PHP script contains both static and dynamic method calls.

in seeds that are closer to the target. However, static analysis tools often struggle to effectively model dynamic features of interpreted languages like PHP. Inaccurate modeling of dynamic features by static analysis tools can lead to imprecise or erroneous distance calculations.

We provide an example in Listing 1 based on common practices in real-world PHP applications to better illustrate this challenge. There are two main concerns: 1) Some static analysis tools designed for PHP use method names for matching and constructing call edges [6, 10]. In scenarios where the target application contains multiple classes with methods sharing the same name, *e.g.*, `foo()` in Listing 1, these tools fail to construct an accurate call graph (CG). In this context, some paths can hardly be accurately identified. However, these overlooked paths can be reachable to the target location. In the example, we can only identify the execution path `$ClassA::foo()` using these static analysis tools, but remain unaware of the second path `$ClassB->foo()`, even if the second path can trigger the vulnerability more easily. 2) The code dynamically invokes methods based on variable values (`$cName` and `$mName`), making it challenging for static analysis tools to accurately infer all possible execution paths. This feature is also known as variable function calls, which are common in PHP applications. For example, current static analysis tools cannot derive the call edge from line 33 to line 18. When attempting to calculate distances, the results can be imprecise or even erroneous.

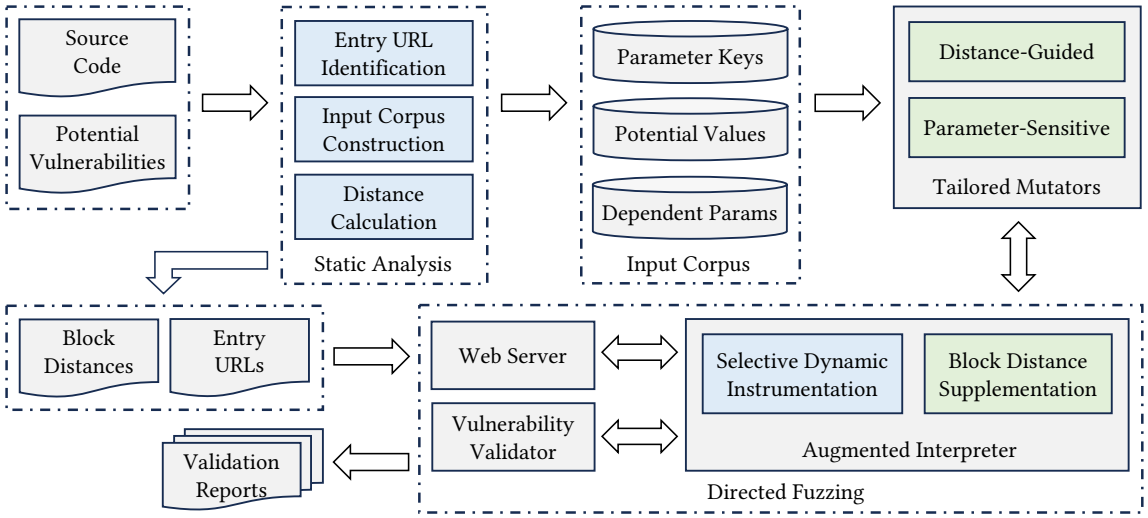

Figure 1: Overall architecture of PREDATOR. Seven additional components are introduced. The blue components are essential for enabling directed fuzzing, while the green components enhance its performance.

4. PREDATOR

In this section, we present PREDATOR, a directed fuzzing tool specifically designed for detecting vulnerabilities in web applications and addressing the aforementioned challenges.

4.1. Architecture Overview

The overall architecture of PREDATOR is illustrated in Figure 1. PREDATOR primarily consists of two stages: the static analysis stage and the directed fuzzing stage. During the static analysis stage, PREDATOR takes the source code of the web applications and existing static analysis reports that provide fuzzing targets as inputs. To enable directed fuzzing, it analyzes the potential vulnerable targets reported, identifies the entry URLs, constructs the input corpus, and performs distance calculations. During the directed fuzzing stage, PREDATOR utilizes selective dynamic instrumentation to provide basic block-level distance feedback for choosing promising seed inputs. It introduces two novel mechanisms to augment the directed fuzzing process for web applications. First, it dynamically supplements the block distance information, thereby enhancing the effectiveness of distance feedback. Second, the mutation strategies based on distance and parameters improve the efficiency of the exploration and exploitation phases.

4.2. Static Analysis for Directed Web Fuzzing

We need to achieve three main tasks using static analysis techniques. First, we analyze control dependencies to extract entry URLs for the specified targets and compute block-level distances. Second, we assess data dependencies to construct the input corpus. This step involves identifying how data flows through the application, which helps determine which parameters are influential in reaching and testing different code paths. By analyzing data dependencies, we can create a targeted input corpus that effectively stimulates diverse application behaviors during fuzzing. Third, to gather distance feedback at runtime and leverage it for power scheduling and seed selection, we need to develop a suitable instrumentation mechanism.

4.2.1. Entry URL Identification. To narrow the search scope, we statically identify the entry scripts and their corresponding entry URLs essential for reaching the target code locations. We perform a backward analysis, starting from the target locations and tracing back to the entry scripts.

PHP's inclusion mechanism alters the execution flow by dynamically injecting additional scripts into the runtime environment. Code in a script might be executed by other scripts that include it through statements like `require()` and `include()`. For a given target, an intuitive approach is to consider all scripts that ultimately include the target's script as entry scripts. However, this approach is too coarse-grained and considered impractical, as a script can be recursively included by other scripts. It could result in a large number of potential entry scripts, although many of them do not lead to the target code location.

We propose to use call chain and script inclusion chain analysis to determine which scripts receive user requests and direct to the target code locations. This process leverages the principle that if a target is in a function body, tracing the call chain can pinpoint the entry functions that initiate the execution. Essentially, a call chain provides a roadmap of function calls, detailing each step from the entry functions to the target. The script inclusion chain analysis then identifies only the scripts that include the known entry scripts.

Call Chain Analysis. We trace the call chain backward from the target to determine which scripts originally initiate the control flow. We build a call graph by analyzing the

Algorithm 1: Input corpus construction.

```
Input  : The target t
Output : A dictionary of request parameters DR, A
         dictionary of dependent parameters DP
Init   : DR = {}, DP = {}, BS = [], RP = [],
         V = []
BS ← identify_BS(t)
foreach bs in BS do
    RP ← identify_RP(bs)
    DP ← DP ∪ {bs, RP}
    foreach rp in RP do
        V ← extract_values(rp)
        DR ← DR ∪ {rp, V}
    end
end
```

inter-procedural control flow of the application to assist in identifying the entry scripts. Each function along the path possesses zero, one, or multiple call edges linking it to its call sites. We meticulously track these call edges to pinpoint the entry functions and record the scripts in which they are invoked. These scripts are then marked as entry scripts. If a target is not within a function body, we consider the scripts containing the target as entry scripts.

Script Inclusion Chain Analysis. We assess the script inclusion chain to identify which scripts include the entry scripts, consequently designating those as entry scripts as well. Then, we convert the file system paths of the entry scripts into entry URLs. They are then fed into the fuzzer to indicate the scripts responsible for processing test cases.

4.2.2. Targeted Input Corpus Construction. An targeted input corpus fosters the generation and mutation of test cases to reach the target code locations effectively. We conduct an inter-procedural backward data-flow analysis to identify critical request parameters that may influence the execution paths (*i.e.*, control flow) of the target application. The input corpus comprises request parameter keys along with potential values that can be assigned to them, enabling the fuzzer to generate test cases effectively. Furthermore, we mark the parameters that affect the condition expression of a branch statement as its *dependent parameters*. These dependent parameters guide the mutation mechanism, which will be detailed in §4.3.2.

The simplified workflow is shown in Algorithm 1. Specifically, we first identify all branch statements (*BS*) along the control flow paths that extend from the entry scripts to the specified target. For each branch statement in *BS*, we extract all request parameters into *RP* and add the ones that influence the condition expressions to the dictionary *DP* by tracing the data-flow path backward. We then analyze the AST nodes along the path to extract all possible values of the request parameters and store them in *V*, specifically extracting all relevant right-hand values from assignments, conditional statements, *etc.* Finally, we append all these possible values together with the request parameters into the dictionary *DR*. Take the code in Listing 1 as an example. The server-side script chooses which function to call based on the value of `$action`. We identify that `$a` is the dependent parameter influencing the value of `$action` in the branch statements on lines 9, 11, and 19. We then extract all potential values of `$a`, which include `lookup`, `edit`, and `defaultAction`.

4.2.3. Selective Dynamic Instrumentation. Inspired by SelectFuzz [26], we propose to instrument only path-divergent basic block, which is the last intersection block of a reachable path and an unreachable one to the target. To achieve this, PREDATOR needs to perform basic block-level instrumentation. As discussed in §3, employing traditional instrumentation techniques is impractical. Instead, PREDATOR augments the PHP interpreter to dynamically instrument only a small group of specific opcodes, as shown in Table 1. This process is transparent to the target application and does not require any modification to the source code. It contains three categories of opcodes, *i.e.*, conditional and unconditional jumps, function calls, and include or eval statements. These specific opcodes represent the starting point of a basic block. By instrumenting these opcodes, PREDATOR can provide precise and efficient monitoring of the execution flow.

Jumps. PREDATOR detects the control-flow transfer caused by either a conditional or unconditional jump. At this point, the execution stream is at a branch statement, which is the starting position of a basic block. PREDATOR checks the file name and starting line number of this basic block to determine if it is a path-divergent basic block. It updates the input distance of the current test case if the block distance is shorter than the current input distance.

Function Calls. When executing a function call opcode, PREDATOR detects the control flow transfer. The process of handling distance information is similar to that used for jump opcodes. Additionally, PREDATOR records the filename and line number of the call sites to dynamically update imprecise block distances, which will be detailed in §4.3.1.

Include or Eval Statements. Executing include or eval statements can significantly alter the control flow of a

TABLE 1: Selected opcodes for instrumentation.

Opcode	Description
`ZEND_JMP`	Unconditional jump
`ZEND_JMPZ`	Jump if zero
`ZEND_JMPNZ`	Jump if not zero
`ZEND_JMPZNZ`	Jump if zero, else to another address
`ZEND_JMPZ_EX`	Jump if zero with extra check
`ZEND_JMPNZ_EX`	Jump if not zero with extra check
`ZEND_DO_FCALL`	Execute function call
`ZEND_DO_ICALL`	Execute indirect call
`ZEND_DO_UCALL`	Execute unresolved call
`ZEND_DO_FCALL_BY_NAME`	Execute function call by name
`ZEND_CALL_TRAMPOLINE`	Call trampoline
`ZEND_FAST_CALL`	Fast call used for user-defined functions
`ZEND_INCLUDE_OR_EVAL`	Execute include or eval statements

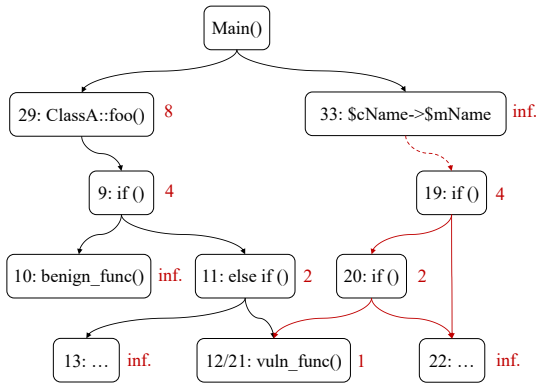

Figure 2: The iCFG of the example script, with red numbers representing the block distance to the target `vuln_func()`, and the red dashed directed line indicating paths that cannot be recognized by static analysis tools.

PHP script, as it introduces new code paths and potential function calls that are not present or known in static analysis. Additionally, the included or evaluated code can contain any number of function calls, conditional statements, or even additional include or eval statements. PREDATOR treats include as a jump and eval as a function call.

If we set line 12 in Listing 1 as the target, PREDATOR will instrument lines 9, 11, and 12, as they contain the selected opcodes and are path-divergent blocks. The simplified bytecode of ClassA can be found in Listing 2 in the appendix.

When calculating block distances for PHP applications, the distance can be inaccurate without executing the scripts. Figure 2 illustrates this issue. In this example, `vuln_func()` is the target, and the red numbers indicate the block distance to the target, which is calculated using SelectFuzz's block distance calculation algorithm. This algorithm works on the inter-procedural control-flow graph (iCFG), however, there are paths within PHP applications that static analysis tools cannot identify. We observed that the path represented by the red dashed directed line in the figure, from line 33 to line 19, is undetectable by static analysis tools. This leads to an inability to determine the distance between the basic block at line 33 and the target, resulting in the fuzzer considering it an unreachable block. When an input causes the application to execute line 33, the fuzzer does not add this input to the queue because it does not perceive the input as reaching a new or closer basic block. Then, the likelihood of executing the paths represented by all the red arrowed lines will decrease due to the wrong action, thereby severely affecting the effectiveness and efficiency of executing target code and triggering vulnerabilities. We will detail the solution to this issue in §4.3.1.

4.3. Tailored Directed Fuzzing Techniques

We develop new directed web fuzzing techniques to tailor towards the specific dynamic features of web applications as they render the fuzzing process inefficient. These techniques encompass block distance supplementation at runtime and tailored mutation strategies. Block distance supplementation assists in updating imprecise block distances during the fuzzing process, while tailored mutation strategies are designed to improve the mutation effectiveness.

4.3.1. Block Distance Supplementation. Inaccurate static analysis results caused by variable function calls can be supplemented during dynamic testing. PREDATOR can dynamically identify paths that were not recognized during the static analysis phase and update the block distances accordingly. The simplified workflow is shown in Algorithm 2.

PREDATOR places a sentinel at the beginning of each function. When executing a variable function, PREDATOR checks whether the path from the call site to the callee already exists. If not, PREDATOR records this path and calculates the block distance for the call site. Note that it is impractical to recalculate all the distances every time a new call edge is identified. Thus, we make a trade-off between the time overhead incurred by updating distances and the accuracy during the process of distance update. PREDATOR sets the block distance of the call site as a constant multiple of the block distance of the callee, where the constant is empirically set to 2. It continuously adopts this approach during the fuzzing process. Following this, the distance information dynamically updated allows the fuzzer to more accurately identify the distance between the currently executed basic block and the target, thereby more effectively guiding the fuzzing process. In §4.2.3, we present a simple example where the performance of the fuzzer is impacted due to a path not being identified, leading to some promising test cases not being added to the seed queue. By using this supplementation method at runtime, the path represented by the red dashed directed line in Figure 2 is identified, and the block distance of line 33 is updated with 8.

4.3.2. Tailored Input Mutation Strategies. Since existing mutation strategies designed for native programs do not guarantee effectiveness when testing web applications, we present new ones specifically tailored for web applications, namely dependent parameter-sensitive mutation and distance-guided probabilistic mutation.

Algorithm 2: Block distance supplementation.

```
1  Input  : A list of functions F, Distance dictionary D
2  Output : Updated distance dictionary D'
3  Init   : Insert a sentinel at the start of each function in F
4  D' ← D
5  foreach func in F do
6      if sentinel_detected(callSite, func) then
7          if !path_exists(callSite, func) then
8              record_path(callSite, func)
9              D' ← D' ∪ {func : Const × D'[func]}
10         end
11     end
12 end
```

Dependent Parameter-Sensitive Mutation. PREDATOR utilizes the dependent parameters identified in §4.2.2 to guide the fuzzer in targeted mutations, thereby quickly limiting the search space of the exploration stage. Angora [24] introduced a similar technique by performing byte-level taint analysis on user inputs and then mutating these bytes. For web applications, we need to manipulate parameter keys and values by conducting lightweight parameter-to-condition analysis. At directed fuzzing stage, PREDATOR first logs the closest basic block to the target during the web application's execution. It then mutates the dependent parameters of this basic block to approach the target via the block. Specifically, for any branch statement, PREDATOR identify several dependent parameters during static data-flow analysis. For the closest branch statement after a round of fuzzing, PREDATOR will utilize the dependent parameters as candidate parameter keys for the next round to mutate. By this means, PREDATOR can guide the fuzzer to fulfill the specific conditions and reach the target more effectively.

Distance-Guided Probabilistic Mutation. We propose a distance-guided probabilistic mutation method that combines the advantages of both dictionary-based methods [16] and byte stream-based mutation methods [20]. Dictionary-based mutation methods can help the exploration stage by changing the decision of a branch statement. However, for the exploitation stage, these methods may be less effective. Specifically, when utilizing error-based bug oracles, triggering a vulnerability may require a specific attack string, such as a single quote for SQL injection.

The key idea of our approach is to mainly adopt dictionary-based mutation methods when the current input distance is relatively large, and byte stream-based mutation methods when the distance gets smaller. PREDATOR uses Formula 1 to calculate the probability to use the byte stream-based mutation methods, where D_{cur} is the input distance in the current round, D_{max} and D_{min} are the maximum and minimum input distances in the previous rounds, respectively. The probability is higher when the current distance is smaller, and lower when the current distance is larger. We set the maximum probability to 90% based on empirical observation, meaning that the byte stream-based mutation methods will be applied with a 90% probability when the current distance is 1. PREDATOR leverages the maximum and minimum distances to modulate the rate of change in the probability function, instead of directly employing a linear function. Specifically, it mostly chooses byte stream-based mutation methods when the current distance is sufficiently far from the maximum distance and sufficiently close to the minimum distance.

$$P_{\text{b}}(D_{\text{cur}}, D_{\text{max}}, D_{\text{min}}) = \begin{cases} 0 & \text{if } D_{\text{cur}} < 1 \\ 0.9 \cdot e^{-\left(\frac{D_{\text{min}}}{D_{\text{max}}}\right)(D_{\text{cur}}-1)^2} & \text{if } D_{\text{cur}} \geq 1 \end{cases} \quad (1)$$

The three variables, D_{cur}, D_{max}, and D_{min}, can exhibit only three possible states in practice. Initially, all are set to -1, indicating that no basic blocks have been executed. At runtime, all three values are greater than or equal to 1 when at least one path-divergent basic block has been executed, and the current run also encounters such blocks. If D_{cur} is -1 while D_{max} and D_{min} are no less than 1, it indicates that basic blocks with distance information have been executed in previous runs, but no such blocks were encountered in the current execution. Hence, this function remains valid across all scenarios.

5. Implementation

We implement a prototype of PREDATOR comprising around 2,300 lines of code, which includes 1,500 lines of C code for directed fuzzing, and 800 lines of Python code for lightweight static analysis. The main components of PREDATOR is built atop Witcher [16], a grey-box coverage-guided fuzzer for web applications. We could implement PREDATOR based on Atropos, but its code is not available at the time of writing. The static analysis part is realized atop PHPJoern [10] and TChecker [6]. Next, we discuss several important implementation details.

Static Analysis. To identify potential targets for dynamic validation, we utilize TChecker [6]. It is a context-sensitive inter-procedural static analysis tool to detect taint-style vulnerabilities in PHP applications. We use the nodes and edges generated by PHPJoern [10] to construct the abstract syntax tree (AST) and corresponding graphs to perform lightweight control- and data-flow analysis.

Selective Dynamic Instrumentation. We modify the PHP interpreter to enable directed fuzzing. Inspired by Witcher [16], we utilize the `VM_TRACE()` macro, which is introduced in the PHP interpreter from version 7.3. This macro enables detailed tracing of the execution of opcodes within the PHP virtual machine. We modify the macro to report the distance of the current basic block when control-flow transferring opcodes are executed. When executing a control-flow transferring opcode, the PHP interpreter searches if the line number of the opcode and the file name of the current script exist in the recorded block distance information. If the current test case reaches a closer block, the interpreter updates its input distance by modifying the shared memory. After the execution of the test case, the fuzzer reads the distance and saves the test case if the distance is the shortest so far. We store the block distance information in the file system, which is to permanently record the distances when block distance supplementation happens at runtime. When testing the same target, we load from the file system to avoid re-updating the block distance information.

XSS Detection. We further implement a simple reflected XSS detector [14, 36] working with PREDATOR, as Witcher [16] is able to identify only SQL injection and command injection vulnerabilities. PREDATOR injects a crafted payload into the target web application and checks whether the payload is reflected in the response. Technically, we could employ a JavaScript engine to execute the response for more accurate detection, similar to what Black Widow [12] does. However, doing so would significantly impair the efficiency of the dynamic validation phase, especially for triggering other types of vulnerabilities. We do not emphasize this component as our contribution and we only aim to

demonstrate the potential to detect taint-style vulnerabilities using PREDATOR.

6. Evaluation

In this section, we answer the following research questions to evaluate PREDATOR:

- **RQ1.** How does PREDATOR compare with existing tools in terms of effectiveness and efficiency in reproducing known vulnerabilities?
- **RQ2.** How does each component of PREDATOR contribute to its performance?
- **RQ3.** How capable is PREDATOR in validating static analysis reports?
- **RQ4.** How well does PREDATOR perform in discovering new vulnerabilities in real-world applications?

All experiments are conducted in Docker running on a 64-bit Debian server equipped with 4 $\times$ Intel Xeon Platinum 8160 processors.

6.1. Known Vulnerability Reproduction (RQ1)

Evaluation Dataset. We select applications evaluated in prior studies [6, 12, 15, 16, 30] as our dataset for evaluation. These encompass 3 synthetic test suites and 11 real-world applications, comprising around 46K files and 2M LoC in total. They collectively contain a total of 51 SQL injection (SQLi), 8 command injection (CMDi), and 26 reflected cross-site scripting (XSS) vulnerabilities. We manually collect known vulnerabilities from the CVE database [37] in a best effort manner, yet we acknowledge the possibility of omissions. We exclude applications that only run on PHP 5, which has long been obsolete and is no longer supported by the PHP community. For these applications, we believe that the benefit of detecting their vulnerabilities is not significant.

To the best of our knowledge, no existing fuzzer encompasses a vulnerability scope identical to that of PREDATOR. Therefore, we compare PREDATOR with various state-of-the-art fuzzers for different types of vulnerabilities, including Witcher [16] and Atropos [18] for SQLi and CMDi vulnerabilities, Cefuzz [30] for CMDi vulnerabilities, and WebFuzz [15] for XSS vulnerabilities. To better compare the performance of PREDATOR with Witcher, we add the XSS detector to Witcher and denote it as Witcher+. The source code of Atropos—a concurrent work—is not available at the time of writing. Cefuzz is closed-source and we are not able to run it for a direct comparison. We thus borrow the evaluation results presented in the papers [18, 30] of Atropos and Cefuzz for a comparison. We manually check the testing results generated by each tool to confirm if they are true positives.

6.1.1. Effectiveness. We first evaluate the effectiveness of PREDATOR in reproducing known vulnerabilities and compare it with other fuzzers. For each application, we search and select the reported code locations of all known vulnerabilities as targets for PREDATOR. We run the crawler of Witcher to collect URLs of the target vulnerabilities for Witcher and Witcher+. We adopt the settings recommended in the paper [16], providing the crawler with valid credentials and run it for four hours. For WebFuzz, we manually browse the applications to collect URLs and ensure it can successfully access all pages and forms related to the target vulnerabilities. We then exclude irrelevant URLs that do not lead to the target vulnerable locations and assign an equal time budget for each of the remaining URLs. This measure allows all tools to focus on testing target vulnerabilities instead of exploring unrelated execution paths, thereby ensuring a fair comparison. If we consider a scenario where no irrelevant URLs are excluded and an equal time budget is allocated for each URL, several challenges would arise. A prolonged budget may result in excessive time spent on irrelevant URLs, while a budget that is too short may prevent thorough testing of vulnerability-related URLs. If we set the same total time budget for each tool for analyzing one application, the actual testing duration on vulnerability-related URLs may vary significantly across different tools. In both scenarios, these discrepancies would negatively impact the fairness of the comparison.

Results. Table 2 summarizes the results on synthetic test suites and Table 3 presents the results on real-world applications. Combining two datasets, PREDATOR effectively identified 32 SQL injection vulnerabilities, 5 command injection vulnerabilities, and 12 XSS vulnerabilities across all 17 applications. Witcher successfully triggered 16 SQLi vulnerabilities and 1 CMDi vulnerability, respectively. According to the original papers, Atropos successfully triggered 21 SQLi vulnerabilities and 6 CMDi vulnerabilities in the synthetic test suites [18]; Cefuzz triggered all 5 CMDi vulnerabilities in the synthetic test suites [30]. WebFuzz successfully identified 9 XSS vulnerabilities, whereas Witcher+ only detected 2. On real-world applications, PREDATOR detected all vulnerabilities that other tools could detect, and additionally discovered 9 vulnerabilities that other tools failed to detect.

Comparison with Witcher. The performance improvements of PREDATOR primarily result from the entry URL identification and targeted input corpus construction techniques. For instance, we observed that Witcher struggled to collect effective entry URLs for some applications, notably bWAPP, OpenEMR, and rConfig. We conducted additional crawling for several applications using Witcher's crawler. However, after the crawler stopped or reached the time budget, we observed that the results never included an effective entry URL for certain vulnerabilities. This is consistent with the findings in the paper [16].

Comparison with Atropos. The performances of PREDATOR and Atropos regarding bWAPP and XVWA show marginal differences. Discrepancy is primarily observed when evaluating DVWA. This is due to the inherent limitation of Witcher on which PREDATOR depends—it cannot carry hidden random CSRF tokens, rendering PREDATOR ineffective in testing DVWA. In contrast, Atropos operates on a snapshot-

TABLE 2: The evaluation results on synthetic test suites. We list the total number of vulnerabilities by type, in the format of *SQLi + CMDi + XSS*. As the source code is not available at the time of writing, we borrow the detection results from the papers of Atropos and Cefuzz. "-" denotes that no detection results are available.

Application			SQLi			CMDi				XSS		
# ID	Name	Vulnerability	PREDATOR	Witcher	Atropos	PREDATOR	Witcher	Atropos	Cefuzz	PREDATOR	WebFuzz	Witcher+
1	bWAPP	17 + 2 + 12	11	0	13	2	0	2	2	7	6	0
2	DVWA	6 + 3 + 3	0	0	6	0	0	3	3	0	0	0
3	XVWA	2 + 1 + 1	2	2	2	1	1	1	-	1	1	1
Total		25 + 6 + 16	13	2	21	3	1	6	5	8	7	1

TABLE 3: The evaluation results on real-world applications. We list the total number of vulnerabilities by type, in the format of *SQLi + CMDi + XSS*. †We remove the hidden tokens in WeBid for a comparison between PREDATOR and Witcher.

Application				SQLi		CMDi		XSS		
# ID	Name	Version	Vulnerability	PREDATOR	Witcher	PREDATOR	Witcher	PREDATOR	WebFuzz	Witcher+
4	Login Mgmt.	2.1	1 + 0 + 0	1	1	0	0	0	0	0
5	Hosp. Mgmt.	4.0	9 + 0 + 1	9	8	0	0	1	0	0
6	Doctor Appt.	1.0	1 + 0 + 0	1	1	0	0	0	0	0
7	Piwigo	13.6.0	2 + 0 + 0	0	0	0	0	0	0	0
8	rConfig	3.9.2	1 + 2 + 0	1	0	2	0	0	0	0
9	OpenEMR	5.0.1.7	5 + 0 + 5	3	1	0	0	2	1	0
10	WeBid†	1.2.2	1 + 0 + 2	1	0	0	0	1	1	1
11	Joomla	3.7.0	1 + 0 + 2	0	0	0	0	0	0	0
12	WebChess	0.9	2 + 0 + 0	2	2	0	0	0	0	0
13	WordPress	6.0	2 + 0 + 0	0	0	0	0	0	0	0
14	ECShop	4.1.5	1 + 0 + 0	1	1	0	0	0	0	0
Total			26 + 2 + 10	19	14	2	0	4	2	1

based approach, making the random tokens deterministic during fuzzing [18]. We cite Atropos' detection results in the best-case scenario as the ground truths used in the evaluation of Atropos and PREDATOR are not identical. When evaluating PREDATOR on reproducing *known* vulnerabilities, we did not count any true-positive vulnerabilities outside the ground truth, whereas Atropos did. The current testing results for Atropos are limited to three synthetic test suites, which hinders our ability to perform a comprehensive comparison on real-world applications.

Comparison with WebFuzz. The performance of PREDATOR is slightly better than that of WebFuzz. We noticed that WebFuzz occasionally got stuck on certain pages, preventing further progress. For instance, when testing Hospital Management System, despite providing valid login credentials, WebFuzz consistently attempted to visit inaccessible pages. This likely represents a design or implementation flaw, leading it to waste considerable time exploring irrelevant paths. When testing other applications, its performance was satisfactory yet remained marginally inferior to that of PREDATOR.

False Negatives. We analyze the reasons behind the false negatives. In bWAPP, 6 SQLi vulnerabilities were not detected, 3 of which were due to the lack of oracle for sqlite. This could be easily addressed by detecting the corresponding error feedback messages. However, we refrained from improving it to maintain fairness in comparison with Witcher. This is one of the reasons why PREDATOR underperformed Atropos in detecting SQLi vulnerabilities in bWAPP. The remaining 3 vulnerabilities could not be triggered by our current prototype due to the presence of CAPTCHAs and the requirement of specific User Agent (UA) header values. For a fair comparison with Atropos, we did not remove hidden CSRF tokens in DVWA. As Witcher and PREDATOR cannot handle hidden tokens, they failed to trigger any vulnerabilities in DVWA. The vulnerabilities undetected in OpenEMR could be identified by concurrently fuzzing with multiple cores or increasing the time budget. Moreover, we discovered that PREDATOR was unable to detect any vulnerabilities within WordPress, Piwigo and Joomla due to the failure of entry URL identification. For example, they dynamically includes server-side files based on the values of query strings, which are impractical to identify using static analysis methods. During the XSS vulnerability detection, we noted that the absence of injected payloads in the source code of some pages prevented the matching of the attack string. As a result, PREDATOR failed to detect the vulnerabilities. This is an implementation limitation of the matching-based XSS detection method.

False Positives. All 6 false positives were reported during the XSS vulnerability detection. This is possibly due to the high false-positive rate of the string matching-based XSS

detection method. This risk of false positives arises whenever an application outputs user input on a page. Addressing this issue could involve utilizing a JavaScript engine to render the responses.

6.1.2. Efficiency. We further compare PREDATOR with Witcher on the efficiency. The comparison was exclusively made with Witcher because PREDATOR is implemented atop Witcher, and it significantly differs from the other tools. Additionally, Cefuzz neither has available source code nor an evaluation of time to exposure in its paper, thus we are unable to compare PREDATOR with it. While we acknowledge that comparing PREDATOR, a directed fuzzer, to Witcher may seem unfair, it is important to note that our intention is not to criticize Witcher. Rather, our goal is to emphasize the advantages of directed fuzzing for web applications. We select vulnerabilities in real-world applications that could be successfully triggered and allocate a 24-hour time budget for each one and run the experiment ten times. Moreover, to enhance fairness as much as possible, we provide both tools with the same input corpus and entry URLs, and set the initial seeds to be empty.

Results. The average time to exposure (μTTE) is shown in Table 4. We used the Mann-Whitney U test to assess the statistical significance of our experimental results. In 22 out of 25 cases, the results are statistically significant, with p-values less than 0.05. In the other 3 cases, both tools either failed to detect the vulnerability, resulting in a 24-hour *TTE*, or they triggered the vulnerabilities within similar timeframes in multiple trials, yielding higher p-values.

PREDATOR outperformed Witcher in 21 out of 25 cases (highlighted in bold in the table), with p-values less than 0.05. PREDATOR demonstrated shorter μTTE and successfully triggered more vulnerabilities in ten attempts. Specifically, both PREDATOR and Witcher triggered the vulnerability `CVE-2020-29283` in all ten attempts, with PREDATOR achieving a μTTE that was 43.8 times shorter. We observed that Witcher's μTTE is marginally lower when detecting `CVE-2020-15713`. This can be attributed to the simplicity of the target vulnerability. Both tools triggered it in seconds, but the throughput of PREDATOR was relatively lower. Note that in this experiment we provided Witcher with the entry URLs and input corpus obtained by PREDATOR. Consequently, Witcher triggered some vulnerabilities that could not be identified when using its crawler in Table 3. However, the input corpus, derived from analyzing the target script, yielded a greater number of potential parameters than those obtained using a crawler, thereby expanding the search space. Witcher was unable to efficiently identify which parameters could be used to reach the target locations. In contrast, due to PREDATOR's directed fuzzing mechanism, it can quickly identify promising parameter keys and values.

6.1.3. Scalability. We evaluate the scalability of PREDATOR by measuring the static analysis time and the throughput of directed fuzzing. The static analysis time consists of two parts: the time spent on the distance calculation and the time spent on identifying entry URLs and constructing the input corpus. We compare PREDATOR with Witcher on their throughputs to examine the impact of selective dynamic instrumentation. We run each tool for one hour on every entry URL of each application and take the average as the throughput, measured in requests per second (req/s). The dataset contains 10 web applications, excluding those where neither PREDATOR nor Witcher identified any vulnerabilities.

Results. The results are shown in Table 5. The distance calculation time is negligible for most applications, except for OpenEMR, which has a large codebase. The cost for identifying entry URLs and constructing the input corpus is also low, except for OpenEMR and ECShop. The time spent on static analysis is typically less than 1 hour, demonstrating that it is both fast and scalable. We analyze the relative difference in fuzzing throughput between PREDATOR and Witcher. PREDATOR's throughput relative to Witcher ranges

TABLE 4: The results of average time to exposure (μTTE) comparison. Each application is denoted by an ID, with the specific correspondence available in Table 3. *T.O.* indicates that in all ten attempts, the vulnerability is not triggered within the time budget. In other cases, a timeout results in a time to exposure of 24 hours. *Runs* denotes the number of successfully triggered attempts. We mark items where PREDATOR shows superior performance and the p-value is below 0.05 in bold.

# ID	# CVE-ID	PREDATOR		Witcher		*p*
		Runs	μTTE	Runs	μTTE	
4	2020-25952	10/10	**0.15 h**	0/10	T.O.	0.000
5	2020-22168	10/10	**0.14 h**	3/10	17.53 h	0.000
	2020-22169	10/10	**0.32 h**	2/10	19.45 h	0.001
	2020-22165	10/10	**0.71 h**	1/10	19.38 h	0.001
	2020-22174	9/10	**4.97 h**	0/10	T.O.	0.000
	2020-22173	10/10	**0.24 h**	3/10	17.68 h	0.001
	2020-22172	10/10	**0.08 h**	7/10	7.36 h	0.045
	2020-22166	9/10	**2.69 h**	5/10	16.94 h	0.001
	2020-22171	9/10	**2.75 h**	4/10	18.70 h	0.008
	2020-22164	10/10	**0.04 h**	8/10	2.69 h	0.000
	2021-39411	10/10	**0.02 h**	0/10	T.O.	0.000
6	2020-29283	10/10	**0.05 h**	10/10	2.19 h	0.000
8	2019-16662	9/10	2.43 h	8/10	9.61 h	0.073
	2019-16663	7/10	**7.28 h**	2/10	20.23 h	0.009
	2020-15713	10/10	0.05 h	10/10	0.01 h	0.000
9	2019-14529	8/10	**7.67 h**	8/10	18.20 h	0.037
	2019-16404	9/10	**10.52 h**	1/10	21.72 h	0.003
	2018-17179	9/10	5.03 h	9/10	5.62 h	0.257
	2023-2949	10/10	**0.04 h**	0/10	T.O.	0.000
	2022-4502	10/10	**0.05 h**	0/10	T.O.	0.000
10	2018-1000867	10/10	6.33 h	10/10	6.82 h	0.372
	2018-1000868	10/10	**0.02 h**	1/10	21.81 h	0.000
12	2023-22959	10/10	**0.23 h**	10/10	0.88 h	0.001
	2019-20896	10/10	**0.34 h**	10/10	0.95 h	0.031
14	2023-5293	10/10	**0.01 h**	10/10	0.03 h	0.005

TABLE 5: The results of static analysis time and fuzzing throughput.

Application	Static Analysis (mins)			Throughput (req/s)		
	Distance	URL & Input	Total	Witcher	PREDATOR	Difference
bWAPP	0.31	0.09	0.40	140.93	101.64	↓ 27.88%
XVWA	0.21	0.02	0.23	106.29	132.92	↑ 25.05%
Login Mgmt.	0.01	0.01	0.02	90.14	97.50	↑ 8.17%
Hosp. Mgmt.	0.12	0.01	0.13	73.64	83.58	↑ 13.50%
Doctor Appt.	0.01	0.01	0.02	100.37	95.03	↓ 5.32%
rConfig	0.78	0.10	0.88	58.68	37.45	↓ 36.18%
OpenEMR	33.71	15.64	49.35	5.04	4.59	↓ 8.93%
WeBid	2.83	0.27	3.10	115.27	146.41	↑ 27.01%
WebChess	0.18	0.07	0.25	64.97	60.17	↓ 7.39%
ECShop	1.73	10.52	12.25	28.58	41.45	↑ 45.03%
Average	3.99	2.67	6.66	78.39	80.07	↑ 2.14%

from 63.82% to 145.03%, where PREDATOR performs better in half cases. Overall, PREDATOR achieves an average throughput of 80.07 req/s, while Witcher achieves 78.39 req/s, differing by 2.14%. This indicates that the additional overhead introduced by our selective dynamic instrumentation is low in practice.

6.2. Ablation Study (RQ2)

We assess the role each component of PREDATOR plays in its overall performance. We start with a baseline version of PREDATOR, disabling all proposed techniques (PREDATOR_b), and then sequentially add each technique to evaluate its impact. We denote each configuration as PREDATOR_e, $\text{PREDATOR}_\text{ed}$, $\text{PREDATOR}_\text{eds}$, $\text{PREDATOR}_\text{edsp}$, and PREDATOR, representing the gradual inclusion of the static analysis method for entry URL identification and targeted input corpus construction, the selective dynamic instrumentation, the block distance supplementation, the distance-guided probabilistic mutation, and the dependent parameter-sensitive mutation, respectively. We achieved different configurations by modifying and recompiling the relevant components of PREDATOR. We selected the SQLi and CMDi vulnerabilities successfully triggered in bWAPP as the dataset. This is primarily for two reasons. Firstly, bWAPP contains the most known vulnerabilities among all applications. Secondly, there is only one entry URL for each vulnerability, which simplifies the experimental setup. We set the time budget to 1 hour, which is sufficient according to our observations in the vulnerability reproduction experiments, and conduct 10 runs for each configuration.

Results. The results are shown in Figure 3. Different colored bars represent the μTTE required by the corresponding configuration. A timeout results in a time to exposure of 1 hour. We investigated the performance across 6 different configurations. The baseline prototype PREDATOR_b used Witcher's crawler for extracting entry URLs and parameters. However, it consistently failed to extract any valid entry URL, thus was unable to trigger any vulnerabilities. With the implementation of entry URL identification and targeted input corpus construction, PREDATOR_e successfully detected 8 out of the 13 vulnerabilities. In the remaining 5 cases it failed to trigger vulnerabilities in all 10 runs. $\text{PREDATOR}_\text{ed}$ further improved the performance due to the addition of selective dynamic instrumentation, notably in cases 1, 8, and 10. For two cases 7 and 11, we observed an increase in μTTE after enabling block distance supplementation in $\text{PREDATOR}_\text{eds}$, attributable to its runtime overhead. The supplementation incurs extra overhead as it involves tracking call sites with unknown callee functions and determining whether the callees can reach the target. $\text{PREDATOR}_\text{edsp}$ managed to reduce the time cost in most cases, as it mainly adopts dictionary-based mutation methods in exploration stage and byte stream-based mutation methods in exploitation stage. Upon enabling the dependent parameter-sensitive mutation, PREDATOR improves significantly in cases 6, 11, and 13.

6.3. Validating Static Analysis Reports (RQ3)

We further assess the capability of PREDATOR in validating static analysis reports. We use the set of vulnerabilities reported by TChecker [6] as the ground truth, and then perform automatic validation using PREDATOR. We set the locations reported by TChecker as the targets. Some applications evaluated by TChecker are excluded, and we provide detailed explanations in Table 8 in the appendix.

Table 6 summarizes the number of vulnerabilities reported by TChecker, validated by PREDATOR, and not validated by PREDATOR. PREDATOR successfully validated 65 out of 84 vulnerabilities (77.38%) reported by TChecker.

False Negatives. We analyze the reasons behind the false negatives. In Codiad, 5 targets are non-functional because they rely on the official Marketplace, which has been offline long ago. If we exclude the 5 targets in Codiad, the ratio of validated vulnerabilities increases to 82.28%. WebChess contains multiple vulnerabilities within the same function. When the first vulnerability was encountered, a segmentation fault signal was sent to AFL. AFL then believed that the application just crashed, and did not test subsequent code to validate the 3 remaining vulnerabilities. In WeBid, one target requires a valid database name, which is difficult for the fuzzer to generate randomly. In the case of Joomla, PREDATOR could not extract the corresponding entry URLs. Since PREDATOR detects only reflected XSS vulnerabilities

TABLE 6: The results of static analysis report validation. †The stored XSS vulnerability in Collabtive is not supported by PREDATOR.

Application	Version	TChecker	Validated	Failed
Codiad	2.8.4	33	27	6
WebChess	0.9	27	22	5
WeBid	1.2.2	18	15	3
Joomla	3.7.0	3	0	3
CPG	1.6.12	1	1	0
PHPLiteAdmin	1.9.8.2	1	0	1
Collabtive	3.1	1†	0	1
Total		84	65	19

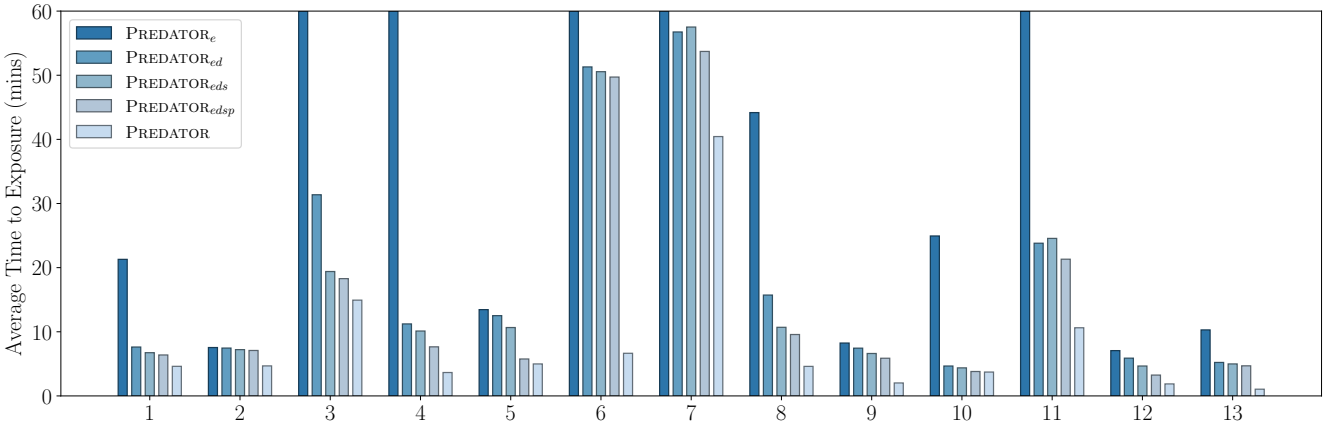

Figure 3: Results of the ablation study. As PREDATOR$_b$ fails to extract any valid entry URLs using Witcher's crawler, we exclude it from this figure.

currently, the stored XSS vulnerability in Collabtive was not triggered. The other 6 vulnerabilities that PREDATOR failed to validate were not triggered within the time limit.

6.4. Discovering New Vulnerabilities (RQ4)

In this section, we showcase the capability of PREDATOR to detect new vulnerabilities in real-world web applications. We include 22 popular web applications from previous work [6, 12, 15, 16, 38] and select the latest versions that are compatible with PHP 7. We first analyze the applications using TChecker, subsequently providing the potential vulnerable targets to PREDATOR for automated dynamic validation. We set the time budget to 24 hours for each application. The dataset, static analysis reports, and results after dynamic validation by PREDATOR are summarized in Table 7.

PREDATOR identified a total of 26 new vulnerabilities, including 11 XSS and 15 SQLi vulnerabilities. PREDATOR detects new vulnerabilities by validating the reports of static analysis tools. Specifically, as a directed fuzzer, PREDATOR sets the target locations based on the static analysis results. If the target locations do contain vulnerabilities, PREDATOR can trigger them automatically, thereby reducing the manual effort needed to validate the static analysis reports. PREDATOR found no new vulnerabilities in applications such as WordPress, phpBB, and Joomla, as the potentially vulnerable locations identified by TChecker did not contain any vulnerabilities. This finding is consistent with the results presented in the paper [6]. We also used Witcher to analyze the applications with the same time budget, but it only detected 7 out of 26 vulnerabilities. We have responsibly reported the vulnerabilities to the respective vendors. At the time of writing, 7 XSS vulnerabilities have been acknowledged and patched, 6 new CVE IDs (`CVE-2024-404[47-52]`) have been assigned.

Our detection approach offers two key advantages. First, we observed only 2 false positives for XSS during the validation process, indicating that vulnerabilities confirmed by PREDATOR generally do not require additional manual effort. This allows limited human resources to be focused on auditing unconfirmed reports. Second, developers can promptly patch vulnerabilities confirmed by PREDATOR, saving time and reducing the potential damage of the vulnerabilities compared to patching after manual auditing, which may take weeks or even months.

TABLE 7: The results of automated validation for new vulnerabilities.

Application	Version	# LoC	SQLi	CMDi	XSS
HAXCMS	8.0.2	238,070	-	-	1
PHPVibe	11.0.46	233,884	-	-	6
Shopping System	#a5f3b4e	7,073	15	-	4
No vulnerability detected	OpenCart, ForkCMS, phpBB, SuiteCRM, osCommerce, phpMyAdmin, Zen Cart, Inventory System, WordPress, Piwigo, PHP Fusion, Rental Manager, Joomla, iCMS, Silverstripe, OpenEMR, ECShop, rConfig, Xenforo				

We discovered that TChecker encountered errors when analyzing certain applications, such as ForkCMS and SuiteCRM. This limitation pertains to the static analysis tool used, rather than to PREDATOR. Moreover, it is important to note that the current XSS detector inherently has the potential for false positives, which is a limitation due to implementation issues. Developing more advanced detectors could overcome this limitation.

7. Discussion

PREDATOR achieved good results in the evaluation, yet there remains room for improvement. In this section, we discuss some limitations and possible future work.

Performance of Static Analysis Tools. The effectiveness of PREDATOR highly depends on the performance of the static analysis tools. For reproducing known vulnerabilities, these tools need to identify potential execution paths as accurately as possible. This assists in analyzing the paths

that are likely to lead to the target locations and calculating the block distances. The current tools are not perfect in this regard, especially when the target application is complex or contains new language syntax that the tools cannot handle [6, 10]. Additionally, some PHP frameworks, such as Laravel, may dynamically register HTTP URLs and their handling functions to define URL entries, causing PREDATOR to potentially miss these entry URLs.

The capability of PREDATOR to detect new vulnerabilities relies heavily on the results of static analysis. PREDATOR can dynamically validate the potential targets to reduce the false positives, but the false negatives are still a problem. Unfortunately, many vulnerable locations are not identified by the current static analysis tools [9]. We observed that TChecker failed to analyze some applications, possibly due to the complexity of the applications or its inability to model new PHP syntax. This is a common problem for static analysis tools, which we leave for future work.

Directed Web Fuzzing. As a directed fuzzer, PREDATOR guides the fuzzing process towards the target locations. It requires preparatory work to identify the potential targets. This is a language-specific task, thus we need to use a specific static analysis tool for one language. As PHP is the most popular server-side language for web development [5], we chose PHP as the target language in our prototype. The distance feedback mechanism is currently implemented in the PHP interpreter, which could be extended to other interpreted languages.

One distinguishing aspect of web fuzzing, compared to other fuzzing domains, is the necessity to analyze the URLs associated with the application's different functionalities and the structured parameters included in these URLs. PREDATOR extracts URLs and builds input corpus from source code through lightweight static analysis, differing from the commonly employed crawling method. However, the static analysis method cannot handle dynamically generated URLs. One possible approach is to integrate static analysis with a crawler. The results from the crawler can facilitate the exploitation of easily accessible vulnerabilities, while static analysis provides more detailed insights into exploring deeper execution paths.

Bug Oracles. PREDATOR leverages Witcher's customized bug oracles to detect SQL injection and command injection vulnerabilities. A reflected XSS detector is implemented to demonstrate its potential to detect more types of vulnerabilities. However, there are limitations of the current bug oracles. For example, the current SQL injection oracle only supports MySQL and PostgreSQL, limiting its applicability across diverse database environments. The XSS detector introduces false positives. Additionally, PREDATOR currently does not support a broader spectrum of vulnerabilities, such as server-side request forgery (SSRF). This issue could be addressed by extending the current bug oracles or implementing PREDATOR upon other fuzzers capable of detecting more types of vulnerabilities.

8. Related Work

Securing Web Applications. In recent years, many approaches have been proposed to secure web applications. Static analysis methods [6, 7, 10, 11, 39–43] scrutinize the source code of web applications to identify taint-style or second-order vulnerabilities. Recent work typically based on code property graph [6, 10] or abstract syntax tree [39, 40] to model the source code. However, these tools often suffer from potential high false positive rates.

Dynamic approaches [12–19, 30, 32, 33, 44, 45] are more relevant to our work. Witcher [16] is a grey-box fuzzer for web applications and the closest to our work. It uses code coverage feedback to guide the fuzzing process and customized bug oracles to detect SQL injection and command injection vulnerabilities. It collects initial seed inputs and targets using a crawler to support up to 6 languages. PREDATOR extends Witcher by introducing an static analysis stage to identify potential targets and a distance feedback mechanism to guide the directed fuzzing process. It only focuses on PHP applications. The reasons are twofold: 1) PHP is the most popular server-side languages for web development [5], and 2) PHP applications are more vulnerable to taint-style vulnerabilities [6]. However, the techniques used in PREDATOR can be easily extended to other languages. WebFuzz [15] uses static instrumentation to collect code coverage feedback and employs a crossover method to generate new test cases. It significantly enlarges the source code of the target application. PREDATOR does not require any modification to the target application. BackREST [17] is a closed-source feedback-driven fuzzer designed for testing RESTful APIs. It conducts dynamic taint analysis during runtime to support the fuzzing process. PREDATOR only conducts static analysis before the fuzzing process to avoid introducing high runtime overhead. At runtime, PREDATOR only collects distance feedback and leverages the dependent parameters to improve the mutation effectiveness. As a concurrent work, Atropos [18] employs snapshot-based techniques and runtime inference to detect eight types of vulnerabilities. Another recent work, Phuzz [19], is a modular fuzzing framework for detecting 5 types of server-side and 2 types of client-side vulnerabilities. PREDATOR, based on Witcher, only detects three kinds of vulnerabilities. While developing more bug oracles is not our primary research objective, it is possible to detect additional types by migrating the approach of PREDATOR to any other fuzzing tool.

Directed Fuzzing. There are sufficient directed fuzzing tools for native programs [23–29] and kernels [46–48], yet few for web applications [30]. Cefuzz [30] claims to be a directed fuzzer for web applications. Yet its static instrumentation method introduces additional code and cannot accurately evaluate the seeds when the application is complex. Furthermore, Cefuzz is closed-source, and the evaluation is limited to a small number of applications. It cannot address the dynamic features of interpreted languages like PHP, *e.g.*, the variable function calls. PREDATOR bridges the gap of the lack of efficient directed fuzzing tools for web applications without altering the source code of the

target application. Additionally, PREDATOR introduces novel methods to enhance directed fuzzing for web applications, which encompass block distance supplementation, mutation strategies guided by dependent parameters and distance.

9. Conclusion

In this paper, we present PREDATOR, an efficient directed fuzzer for web applications. To solve the challenges of applying directed fuzzing to web applications, PREDATOR introduces several novel techniques to enable and augment the directed fuzzing process, especially the block distance supplementation and the mutation strategies tailored for web applications. In the evaluation, PREDATOR shows effectiveness and efficiency in detecting vulnerabilities in both synthetic and real-world web applications. Moreover, it finds 26 previously unknown vulnerabilities in real-world applications and 7 of them are acknowledged and patched with 6 new CVE IDs assigned. We believe that PREDATOR can foster research in the field of web application security and contribute to the development of more sophisticated directed fuzzers for web vulnerability detection.

10. Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable suggestions and comments. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14209323).

References

[1] O. Foundation, "Top 10 web application security risks," 2021, https://owasp.org/www-project-top-ten.

[2] ——, "Sql injection," 2024, https://owasp.org/www-community/attacks/SQL_Injection.

[3] ——, "Command injection," 2024, https://owasp.org/www-community/attacks/Command_Injection.

[4] ——, "Cross site scripting," 2024, https://owasp.org/www-community/attacks/xss.

[5] W3Techs, "Usage statistics of php for websites," Apr. 2024, https://w3techs.com/technologies/details/pl-php.

[6] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, USA, Nov. 2022.

[7] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2014.

[8] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA, May 2022.

[9] P. Nunes, I. Medeiros, J. M. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Transactions on Reliability*, 2018.

[10] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*, Paris, France, Apr. 2017.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, USA, May 2006.

[12] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2021.

[13] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, USA, Aug. 2012.

[14] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: Evolutionary fuzzing for black-box xss detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014.

[15] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos, "webfuzz: Grey-box fuzzing for web applications," in *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*, Virtual event, Oct. 2021.

[16] E. Trickel, F. Pagani, C. Zhu, L. Dresel, G. Vigna, C. Kruegel, R. Wang, T. Bao, Y. Shoshitaishvili, and A. Doupé, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2023.

[17] F. Gauthier, B. Hassanshahi, B. Selwyn-Smith, T. N. Mai, M. Schlüter, and M. Williams, "Experience: Model-based, feedback-driven, greybox web fuzzing with backrest," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, 2022.

[18] E. Güler, S. Schumilo, M. Schloegel, N. Bars, P. Görz, X. Xu, C. Kaygusuz, and T. Holz, "Atropos: Effective fuzzing of web applications for server-side vulnerabilities," in *Proceedings of the 33th USENIX Security Symposium (Security)*, Philadelphia, PA, USA, Aug. 2024.

[19] S. Neef, L. Kleissner, and J.-P. Seifert, "What all the phuzz is about: A coverage-guided fuzzer for finding vulnerabilities in php web applications," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, New York, NY, USA, Apr. 2024.

[20] M. Zalewski, "American fuzzy lop," 2021, https://github.com/google/AFL.

[21] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, 2010.

[22] C. E. Silva and J. C. Campos, "Combining static and dynamic analysis for the reverse engineering of web applications," in *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, 2013.

[23] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, USA, Oct.–Nov. 2017.

[24] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2018.

[25] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[26] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2023.

[27] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *Proceedings*

of the 43th IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, May 2022.

[28] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviation basic blocks," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, USA, May 2022.

[29] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[30] J. Zhao, Y. Lu, K. Zhu, Z. Chen, and H. Huang, "Cefuzz: An directed fuzzing framework for php rce vulnerability," *Electronics*, 2022.

[31] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, 2015.

[32] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "Navex: Precise and scalable exploit generation for dynamic web applications," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[33] "Wfuzz," 2020, https://github.com/xmendez/wfuzz.

[34] Nikic, "Php-fuzzer," Aug. 2023, https://github.com/nikic/PHP-Fuzzer.

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

[36] Z. Jingyu, H. Hongchao, H. Shumin, and L. Huanruo, "A xss attack detection method based on subsequence matching algorithm," in *2021 IEEE International Conference on Artificial Intelligence and Industrial Design (AIID)*, 2021.

[37] M. Corporation, "Cve database," 2024, https://cve.mitre.org/.

[38] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *Proceedings of the 15th USENIX Security Symposium (Security)*, Vancouver, Canada, Jul. 2006.

[39] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of the Web Conference (WWW)*, Ljubljana, Slovenia, Apr. 2021.

[40] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *Proceedings of the 2019 International Conference on Dependable Systems and Networks (DSN)*, Portland, OR, USA, Jun. 2019.

[41] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, USA, Oct. 2015.

[42] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, USA, Aug. 2014.

[43] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages." in *Proceedings of the 15th USENIX Security Symposium (Security)*, Vancouver, Canada, Jul. 2006.

[44] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[45] P. Li and M. Zhang, "Fuzzcache: Optimizing web application fuzzing through software-based data cache," in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, USA, Oct. 2024.

[46] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, Nov. 2023.

[47] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "Actor: Action-guided kernel fuzzing," in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, USA, Aug. 2023.

[48] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *Proceedings of the 43th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2022.

[49] "bwapp," 2024, https://github.com/lmoroz/bWAPP.

[50] "Dvwa," 2024, https://github.com/digininja/DVWA.

[51] "Xvwa," 2020, https://github.com/s4n7h0/xvwa.

[52] "User login management system," 2020, https://phpgurukul.com/user-registration-login-and-user-management-system-with-admin-panel/.

[53] "Hospital management system," 2019, https://phpgurukul.com/hospital-management-system-in-php/.

[54] "Doctor appointment booking system," 2020, https://projectworlds.in/free-projects/php-projects/online-doctor-appointment-booking-system-php-and-mysql/.

[55] "Piwigo," 2024, https://github.com/Piwigo/Piwigo.

[56] "rconfig," 2023, https://github.com/rconfig/rconfig-v3.

[57] "openemr," 2018, https://www.open-emr.org/wiki/index.php/OpenEMR_5.0.2_Linux_Installation.

[58] "Webid," 2017, https://github.com/renlok/WeBid.

[59] "Joomla," 2017, https://downloads.joomla.org/cms/joomla3/3-7-0.

[60] "Webchess," 2019, https://github.com/halojoy/PHP7-Webchess.

[61] "Wordpress," 2022, https://wordpress.org/download/releases/#branch-60.

[62] "Ecshop," 2021, https://www.ecshopok.com/article-789.html.

[63] "Haxcms," 2024, https://github.com/elmsln/HAXcms.

[64] "Phpvibe," 2024, https://github.com/PHPVibe/PHPVibe.

[65] "Online shopping system," 2022, https://github.com/PuneethReddyHC/online-shopping-system-advanced.

[66] "Collabtive," 2017, https://github.com/philippK-de/Collabtive.

[67] "Codiad," 2020, https://github.com/Codiad/Codiad.

[68] "Coppermine photo gallery," 2021, https://github.com/coppermine-gallery/cpg1.6.x.

[69] "Phpliteadmin," 2019, https://github.com/phpLiteAdmin/pla.

Appendix A.
The Simplified Bytecode of ClassA

```
1  line     op                      return    operands
2  -----------------------------------------------------
3     5     JMP                               ->10
4     6     INIT_FCALL_BY_NAME                'sanitize'
5           SEND_VAR_EX                       !1
6           DO_FCALL                $4
7           ASSIGN                            !1, $4
8     7     POST_INC                ~6        !2
9           FREE                              ~6
10    5     INIT_FCALL_BY_NAME                'validate'
11          SEND_VAR_EX                       !1
12          DO_FCALL                $7
13          BOOL_NOT                ~8        $7
14          JMPZ_EX                 ~8        ~8, ->17
15          IS_SMALLER              ~9        !2, 3
16          BOOL                    ~8        ~9
17          JMPNZ                             ~8, ->4
18    9     IS_IDENTICAL            ~10       !0, 'lookup'
19          JMPZ                              ~10, ->24
20    10    INIT_FCALL_BY_NAME                'benign_func'
21          SEND_VAR_EX                       !1
22          DO_FCALL
23          JMP                               ->30
24    11    IS_IDENTICAL            ~12       !0, 'edit'
25          JMPZ                              ~12, ->30
26    12    INIT_FCALL_BY_NAME                'vuln_func'
27          SEND_VAR_EX                       !1
28          DO_FCALL
29    13    ...                     ...       ...
30    14    RETURN                            null
```

Listing 2: The simplified bytecode of ClassA in Listing 1. *line* refers to the line number in the source code that corresponds to the given bytecode, and the line number on the far left denotes the line number of the bytecode itself.

Appendix B.
The Excluded Applications in RQ3

TABLE 8: The excluded applications and reasons.

Reason	Application	Version
No TP reported by TChecker	MediaWiki	1.36.2
	WordPress	5.4.8
	Joomla	3.10.3
	phpBB	3.3.3
Incompatible with PHP 7	osCommerce2	2.3.4.1
	Zen-Cart	1.5.5
	Zen-Cart	1.3.8
	Monstra	3.0.4
Unknown application version	Ecommerce-CodeIgniter	-
	stock-management	-

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper describes a system called PREDATOR, which is a directed fuzzing framework on PHP, to detect vulnerabilities using a combination of static and dynamic analysis. Building on prior work in fuzzing web applications, this paper proposes uses the output of static analysis, using prior work TChecker, then using this to drive a fuzzer, using prior work Witcher, to confirm the vulnerability. While there have been directed fuzzers in many contexts, this is the first directed web application fuzzer.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) PREDATOR can detect 26 previously unknown vulnerabilities in real-world applications and so far seven have been confirmed and patched by their vendors.
2) This paper expands the field of directed fuzzing (which has been shown to be difficult in other domains) to web application fuzzing.
3) This paper will open-source the system, thus allowing future researchers to build on this work.