Holistic Concolic Execution for Dynamic Web Applications via Symbolic Interpreter Analysis

Penghui Li
Zhongguancun Laboratory
lipenghui315@gmail.com

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

Mingxue Zhang*
Zhejiang University
mxzhang97@zju.edu.cn

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

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

Abstract*—Symbolic execution for dynamic web applications is challenging due to their multilingual nature. Prior solutions often fall short in limited syntax support and excessive engineering costs. We propose a novel approach called *symbolic interpreter analysis (SIA)* for web applications written in interpreted languages. SIA tackles the limitations by leveraging the comprehensive syntax support of language interpreters and incorporating established engineering from existing symbolic execution engines. Since web application logic is handled by the interpreter, SIA leverages an off-the-shelf symbolic execution engine to analyze the corresponding interpreter code to symbolically comprehend the behavior of the web application. Indeed, SIA entails solving several technical challenges in web application symbolic execution such as web application exploration, database interactions, *etc.

We have implemented our approach in SYMPHP, a concolic execution engine for PHP-based web applications. Our extensive evaluation shows that SYMPHP could effectively explore web application code with comprehensive PHP syntax support and high code coverage. It achieved high code coverage and successfully identified 77.23% of known vulnerabilities in our dataset, significantly outperforming prior approaches. The hybrid fuzzing framework built atop SYMPHP significantly boosted fuzzing and detected ten new vulnerabilities.

1. Introduction

Web applications serve as the foundation of the Internet and have powered many important services, such as banking, e-commerce, and social networks. Web applications are typically executed by language interpreters, processing request data from external users. They thus particularly suffer from taint-style vulnerabilities [1–3]. Exploitation of such vulnerabilities can lead to dire consequences, such as sensitive data leakage, privilege escalation, and even server compromises [1, 2]. In the real world, 64% of industry businesses were reported to have experienced web-based attacks such as malicious code execution [4].

Symbolic execution has emerged as a promising technique for detecting security vulnerabilities across various domains [5–9]. It symbolically reasons the behaviors of programs and consults constraint solvers to check the compliance of program properties. In the past, notable symbolic execution engines have successfully identified thousands of critical vulnerabilities in real-world software such as Linux kernel and Chrome browser [6, 7, 9–12].

However, symbolic execution for web applications presents unique challenges compared to other types of programs. One fundamental problem arises from the *multilingual nature* of web applications [13, 14]. Web applications often comprise multiple components implemented in different programming languages. For example, though PHP-based web applications are written in the high-level language PHP, the basic functionalities of PHP are implemented within the PHP interpreter in the low-level language C [13, 14]. This multilingual aspect poses a significant obstacle to symbolic execution, as it needs to comprehensively understand and reason about all the components in different languages. Developing a holistic understanding of the interplay between these components is a non-trivial task.

Existing works bridge the language gap by converting both languages into a common representation and abstracting/modeling the code behaviors of the languages. The conventional approach involves initially extracting constraint formulas from the high-level web application code's various representations. These formulas consist of variables, basic operators, and built-in functions in the high-level language. Subsequently, these extracted formulas are integrated with the corresponding syntax abstraction of the operators and built-in functions implemented in the low-level language, all within the same representation. For example, Navex [15] employs Three-Address-Code (TAC) as the formula representation. It extracts TAC formulas from the application's code property graph and seamlessly integrates them with manually crafted syntax abstraction of the low-level language.

The existing approach is incomplete and inaccurate, and

Mingxue Zhang is with the State Key Laboratory of Blockchain and Data Security, and ZJU-Hangzhou Global Scientific and Technological Innovation Center.

demands excessive engineering efforts. First, it often uses a representation of the high-level web programming language, *e.g.*, code property graph, for symbolic execution. Given the syntax complexity of high-level language, the solution is difficult to implement and often incomplete. Moreover, the abstraction of the low-level language into the high-level representation further leads to incomplete and inaccurate reasoning. The incompletion and inaccuracy could cause both false positives and false negatives in vulnerability detection (more details in §2.3). Second, the abstraction process can be challenging and time-consuming, especially given the complexity of these components. It demands *significant engineering efforts* and thus is *unscalable*.

We observe that the intrinsic problems do not exist in concrete execution when processing concrete inputs. The reason is that the language interpreter can invoke the corresponding implementation to operate on concrete inputs without additional engineering efforts. The language interpreter provides a *complete* description of the language of the application. Every syntax has its corresponding implementation in the interpreter code. Therefore, a natural thought is to symbolically analyze the corresponding interpreter code to achieve the symbolic reasoning of the language syntax. The analysis, if automated, can also avoid excessive engineering efforts.

In our work, we propose a new approach called *symbolic interpreter analysis (SIA)* for symbolic reasoning of web applications implemented in interpreted languages. Since every piece of web application code is handled by the corresponding interpreter code, we symbolically analyze the interpreter code to indirectly analyze the application. The interpreter itself is often implemented in low-level compiled languages like C/C++. We can leverage an off-the-shelf symbolic execution engine (*e.g.*, S2E [7]) to automate the symbolic analysis of the interpreter and comprehend the behavior of the application code. By specifying the web input as symbolic, the engine can symbolically execute web application instructions on the symbolic data.

Despite SIA's advantages, we need to address several challenges. First, the underlying symbolic execution engine, originally designed to analyze the interpreter, lacks awareness of the execution flows of the web application. This can lead to a shallow exploration as multiple code locations in the web application may trigger the same code of the interpreter. To overcome this, we instrument the interpreter to record the locations of the currently executed web application code in the application state. Second, to improve the exploration of web applications, we propose a novel *state scheduling algorithm*. A symbolic execution engine maintains a set of program states for further analysis. This algorithm clusters the states into groups based on the recorded locations and ensures different locations can be well explored. Third, web applications frequently interact with database systems, which are outside the scope of language interpreters. To accurately model these interactions, we concretize the symbolic variables used in database operations via constraint solving. It allows us to issue concrete queries to the database systems and retrieve the corresponding concrete results.

SIA is a holistic symbolic execution solution for dynamic web applications because it addresses the multilingual problem. SIA is a generic approach for interpreted web programming languages. Since PHP is the most popular server-side programming language with 76.8% of market share [16], we implement our design of SIA in a system called SymPHP for PHP-based web applications. By maintaining both the concrete inputs and symbolic states, SymPHP results in a concolic execution engine. SymPHP follows specific paths directed by the concrete inputs and mitigates the path explosion issue in conventional symbolic execution.

We extensively evaluated SymPHP on a comprehensive set of web applications. Compared to two state-of-the-art solutions (XSym [14], and AnimateDead [17]), SymPHP could fully support PHP syntaxes without raising errors during the experiments and unit testing. SymPHP also achieved a reasonably high code coverage of 51.57% on average across the applications. Experiments on a ground-truth vulnerability dataset further demonstrate that SymPHP could detect 77.23% of the vulnerabilities, which significantly outperformed the two prior solutions. We further integrated SymPHP with a web fuzzer Witcher [18], and the resulting hybrid framework could improve the code coverage of fuzzing by up to 85.71%. The framework has detected ten new vulnerabilities in the latest versions of real-world web applications. To date, two of them have been confirmed.

In this paper, we make the following contributions.

- We proposed SIA, the first solution that tackles the multilingual challenge and enables complete symbolic exploration of web applications.
- We developed a holistic concolic execution engine SymPHP for PHP-based web applications.
- We demonstrated the benefits of SymPHP in hybrid fuzzing and discovered ten new vulnerabilities.
- We plan to open-source SymPHP at https://github.com/secureweb/symphp.

2. Background and Motivation

In this section, we first provide the background knowledge of web applications and symbolic execution. We then show the limitations of existing approaches to motivate our research.

2.1. Web Applications

Web applications are deployed on web servers and provide services to client users. Since web applications are viable on the majority of desktop and mobile platforms where web browsers are installed, they have become desired targets for various vulnerability exploitation and attacks [1]. Injection vulnerabilities are among the most severe types of flaws on the web where attackers can inject malicious payloads into web applications. For example, a SQL injection vulnerability would enable external attackers to steal or destroy critical application data. A command injection vulnerability would allow shell command execution in a remote victim server. A

```php
1  <?php
2  if(isset($_GET['name'])) {
3      $user = mysqli_real_escape_string($_GET['name']);
4      $q1 = "SELECT * from users WHERE name = '$user'";
5      $result = mysqli_query($user_conn, $q1);
6
7      if(mysqli_num_rows($result) == 1) {
8          // if returning only one row
9          $row = mysqli_fetch_assoc($result);
10
11         if(strtolower($row["plan"]) == "premium" ) {
12             // check the language if in a premium plan
13             $q2 = "SELECT * from languages WHERE language = " .
                ↪  $_GET['lang'];
14             mysqli_query($lang_conn, $q2); // SQL injection
15         }
16     }
17 }
```

Listing 1: A SQL injection vulnerability.

cross-site scripting (XSS) vulnerability could leak sensitive user data on the client.

For high flexibility, developers often implement web applications using dynamic interpreted programming languages such as PHP, JavaScript, and Python. Among them, PHP is the most prevalent one, powering 76.8% of websites today [16]. The most popular content management system framework WordPress [19] is written in PHP. The execution of web applications often concerns multiple components implemented in different programming languages and are *multilingual*. Listing 1 presents a PHP code snippet. Each PHP statement corresponds to certain C code implementations in the PHP interpreter. Its execution involves both PHP for application code and C for interpreter code. The example shows a typical SQL injection vulnerability. At line 14, the program queries the database system with the query `$q2`, which embeds unsanitized external data `$_GET['lang']`.

2.2. Symbolic Execution

Symbolic execution is extensively used for analyzing programs written in static compiled languages and binary programs. According to Poeplau and Francillon [11], symbolic execution is generally constructed from three components: 1) symbolic expression generation, 2) symbolic backend, and 3) state scheduling. Symbolic execution treats program inputs as symbolic. It simulates program execution on the symbolic inputs and represents the program variables as symbolic expressions, *i.e.*, functions of symbolic inputs. The symbolic backend reasons about symbolic expressions using constraint solvers. Symbolic execution starts with an initial program state and forks new states in branch instructions. In each state, it maintains the program counter (PC), which is the address (location) of the currently executed instruction of the analyzed program. The state scheduling component aims to orchestrate the different execution states and prioritize them according to some strategies [7, 11].

Symbolic execution can be static and directly analyze program [14]. Static symbolic execution offers the advantage of not requiring actual program execution. It can also be combined with concrete execution into *concolic execution*—the focus of this work. In concolic execution, an engine

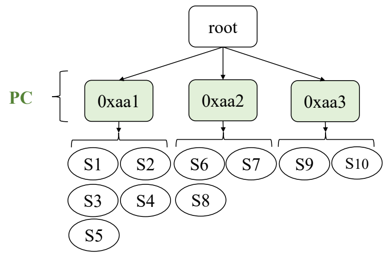

Figure 1: A state clustering tree of S2E.

maintains both concrete values and symbolic expressions for program variables. The analysis typically follows the path directed by the execution of a concrete input. To explore paths that deviate from the current concrete path, the engine checks the feasibility of the branch target opposite to the concrete direction [20]. If feasible, it generates a corresponding concrete input to drive further analysis. Concolic execution can be integrated with fuzzing for hybrid analysis.

S2E. S2E [7] is a widely-used concolic execution engine. It by default employs a state scheduling algorithm (*a.k.a.*, state searcher [21]) based on PC values. S2E groups the states by their PC values. For example, Figure 1 shows the state clustering tree with ten states, where states are stored in the leaf nodes. Multiple states can share the same PC value because they can execute the same instruction via different paths. S2E first randomly selects a group. Each group thus has the same probability to be selected. After that, it randomly selects a state in the group. As shown in Figure 1, a state (*e.g.*, `S1`) in the group of `0xaa1` has the probability of $\frac{1}{3} \times \frac{1}{5}$ to be chosen for further analysis, whereas a state (*e.g.*, `S9`) in the group of `0xaa3` has $\frac{1}{3} \times \frac{1}{2}$. Compared to randomly selecting a state among all states (each state has the same probability of $\frac{1}{10}$), this avoids the potential bias towards the groups with more states. Therefore, different locations (PC values) could be uniformly exercised with equal likelihood.

2.3. Motivation

The multilingual nature of web applications makes their symbolic execution extremely challenging [13, 14]. A symbolic execution engine for web applications has to come across the language boundaries to seamlessly and holistically reason all the components in different languages. The differences in languages among components are vast. For example, PHP is a dynamic interpreted language, while C is a static compiled language. Designing holistic symbolic reasoning that covers all components/languages is difficult.

However, holistic reasoning is necessary for symbolically analyzing web application programs. Lack of reasoning of any component would lead to inaccurate and incorrect analysis. Failure to understand the interpreter code responsible for handling the application logic makes the entire symbolic execution unusable. Even incomplete reasoning of a component would cause dire consequences. According to our analysis of the source code [22] and descriptions in the papers [23, 24], the incomplete support of sanitization functions, *e.g.*, `mysqli_real_escape_string()` in Listing 1, has caused incorrect symbolic data propagation and false positives in vulnerability detection.

Previous approaches solve the multilingual problem of web applications by converting both languages into a common representation. They often extract web application formulas from the high-level language code's representations like code property graph [15, 25, 26], control-flow graph [14, 17, 27], and PHP Opcode [23, 24, 28] and then link to the syntax abstraction of the low-level language. However, as mentioned in §1, these approaches present two intrinsic issues.

❶ **Language Syntaxes.** Web programming languages like PHP are complex with massive language syntaxes. It is difficult to implement the symbolic execution *completely and correctly* at existing high-level representations. For instance, PHP defines hundreds of operations and more than one thousand distinct internal functions [13]. The syntaxes are often more complex compared to the ones in low-level languages like C/C++. Moreover, the languages are dynamic. For example, program variables are used without type declarations. Their types can be dynamically changed through the execution, and some are only determinable at runtime. The dynamic type system makes it hard to infer variable types in symbolic execution, which, however, are necessary for constraint solving.

❷ **Engineering Efforts.** Prior symbolic execution demands a significant amount of engineering work and is unscalable [11]. Navex and XSym were implemented with 5K and 11K lines of code (LoC), respectively. A recent tool AnimateDead was developed using more than 13 person-months as described in their paper [17]. For example, to handle the internal functions, each function ought to be specially modeled to facilitate the analysis in the symbolic backend [14, 15, 23, 24]. Apollo [23, 24], Navex [15], AnimateDead [17], and SYNTHDB [28] all have engineered function models based on the developers' understanding of the function behaviors. The manual approach is unscalable and erroneous. Moreover, web programming languages have been evolving with frequent feature updates. Prior approaches entail additional efforts in each version update.

3. Insight

We propose a new approach of symbolic interpreter analysis (SIA). SIA transforms the language interpreter that executes a web application into a symbolic execution engine, which in turn smartly tackles the multilingual issue. In this section, we illustrate how SIA is derived.

3.1. Revisiting Concrete Execution

We observe that the two intrinsic problems (§2.3) do not exist in concrete execution. A web application often processes concrete external inputs (*e.g.*, GET requests) from users. The application code is dynamically executed by the corresponding language runtime, which is usually an interpreter (*e.g.*, the PHP interpreter). The language interpreter can directly handle the language syntaxes and application code with its corresponding interpreter code. Listing 2 shows the implementation of the `add` operation in the PHP interpreter. It can be regarded as the precise and official definition of the operation. Different language syntaxes are all similarly supported in the corresponding interpreter code.

```c
int add_function_fast(zval *result, zval *op1, zval *op2) {
    zend_uchar type_pair = TYPE_PAIR(Z_TYPE_P(op1),
    ↪  Z_TYPE_P(op2)); // get the types of operands

    if (type_pair == TYPE_PAIR(IS_DOUBLE, IS_DOUBLE)) {
        ZVAL_DOUBLE(result, Z_DVAL_P(op1) + Z_DVAL_P(op2));
        return SUCCESS;
    }
    else if (type_pair == TYPE_PAIR(IS_DOUBLE, IS_LONG)) {
        ZVAL_DOUBLE(result, Z_DVAL_P(op1) +
        ↪  ((double)Z_LVAL_P(op2)));
        return SUCCESS;
    }
    ...
}
```

Listing 2: Implementation of `add` operation in PHP.

3.2. Symbolic Execution of Interpreter Code

By revisiting the concrete execution, we observe that symbolic execution of the interpreter code can simultaneously tackle both the intrinsic problems mentioned in §2.3. The language interpreter provides a complete description of the language of the application. Every syntax has its corresponding implementation in the interpreter code. Therefore, by symbolically analyzing the corresponding interpreter code, a solution can achieve the symbolic reasoning of language syntaxes, thus resolving limitation ❶. For example, a symbolic execution engine can symbolically analyze the interpreter code shown in Listing 2 to fully comprehend its behavior. It would carefully track the symbolic states over the function arguments (`op1` and `op2`) and manage the memory object of `result`. Since interpreters are often implemented in static compiled languages, *e.g.*, C/C++, we thus can directly leverage existing mature execution engines [7, 10, 11] to analyze the interpreter code. By automating the process on different syntaxes, the solution can alleviate excessive engineering efforts, addressing limitation ❷.

One might leverage the symbolic analysis of the interpreter code as a means of syntax reasoning and then integrate it with symbolic analysis on code representations, *i.e.*, integrating two separate symbolic execution components. However, this is difficult because the symbolic execution of the interpreter code and the symbolic execution of the web code representations run in two completely different settings. For example, KLEE for the interpreter code is atop LLVM intermediate representation (IR), whereas Navex for the application code is atop the PHP code property graph. Bridging the gap between two analyses is at least as difficult as bridging the language gap between application code and interpreter code.

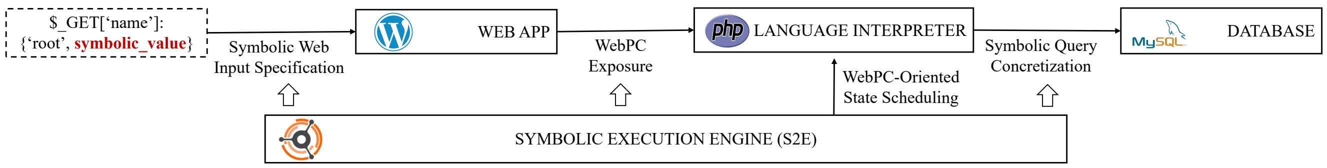

Figure 2: Concolic execution of a web application through SIA. The dotted box denotes the concrete/symbolic input of the analysis.

3.3. SIA: Symbolic Interpreter Analysis

We propose a new approach of *symbolic interpreter analysis (SIA)* for web application symbolic execution. The key idea is to simulate the execution process of a web application and symbolically analyze the corresponding interpreter code. A web application comprises a sequence of application code instructions. The functionality of each code instruction is correspondingly achieved in the interpreter code. Since they are functionality-wise equivalent, symbolically analyzing the corresponding sequence of the interpreter code can realize the symbolic execution of the web application. In our approach, SIA directly applies the target web application to the interpreter so that SIA can perceive the interpreter code sequence. SIA then leverages an off-the-shelf symbolic execution engine to symbolically execute the interpreter code, thus avoiding excessive engineering efforts.

SIA addresses the multilingual problem and is holistic. To the best of our knowledge, SIA is the first solution that enables complete symbolic reasoning of web applications. The workflow of SIA is depicted in Figure 2. Initially, web inputs such as request data are stored in the memory of the language interpreter before the interpreter executes the web application code. SIA leverages the underlying symbolic execution engine to specify the web inputs as symbolic in the memory of the interpreter. In every web application instruction, the underlying engine symbolically analyzes the corresponding interpreter code, which performs the necessary computation for the application code. It tracks the computation and propagation of symbolic data by its inherent symbolic analysis capability of interpreter code. Following the execution of the web application, this ultimately achieves the symbolic execution of the target web application.

We design SIA into a concolic execution method driven by concrete inputs. SIA maintains symbolic states for concrete web inputs and specifies them as symbolic on demand. Concolic execution can potentially mitigate the path explosion problem by effectively exploring a selective set of interesting program paths. It shows great promise and can assist other program analysis methods. For example, it can validate static analysis results by generating proof-of-concept inputs. It can also solve hard-to-solve path constraints to help fuzzers explore difficult paths.

3.3.1. SIA in an Example. Due to the complexity of language implementations, we illustrate our solution in a simple case of `$x = $sym + 1`. We assume `$sym` is a symbolic variable manipulable by external users, and it has an initial concrete value `0` in our concolic execution. Before the

```
1 void zend_assign_to_variable_reference(zval *variable_ptr, zval
  ↪  *value_ptr) {
2     zend_reference *ref;
3     ...
4     ref = Z_REF_P(value_ptr);
5     GC_ADDREF(ref);
6     ...
7     ZVAL_REF(variable_ptr, ref);
8 }
```

Listing 3: Implementation of `assign` operation in PHP.

analysis, we first specify web inputs as the initial symbolic data. In the implementation of Listing 2, the PHP interpreter performs the value addition based on the operand types. For example, line 9 converts `op2` to `double` type and adds it with `op1`. The result is saved to `result` in the function. At the PHP code level, an intermediate result of `$sym + 1` can be computed according to the type of `$sym`. Since `$sym` is a symbolic variable, the intermediate result is also regarded as symbolic. The type judgment in the implementation resolves the dynamic type systems of the application because the underlying engine can symbolically reason the types of operands.

Symbolic data propagates mainly at assignment operations. Listing 3 shows the simplified implementation of `assign` operation from right-hand-side (RHS) value to left-hand-side (LHS) variable. It first obtains the pointer reference to RHS value (`value_ptr`) at line 4. The reference is then assigned to the LHS variable (`variable_ptr`) at line 7. As a result, the LHS variable points to the same memory region of the RHS value. In the assignment operation of `$x = $sym + 1`, it has the RHS value of `$sym + 1` and the LHS variable of `$x`. The RHS value has been previously computed into the intermediate result. Therefore, symbolically executing the assignment operation makes `$x` point to the symbolic memory region of the intermediate result. `$x` then becomes symbolic.

When a symbolic variable is used in branch instructions, the underlying engine will consult constraint solvers to check the feasibility of branch targets.

3.3.2. Challenges and Solutions. Despite SIA's advantages, there remain three technical challenges.

Unawareness of Execution Flows of Web Applications. Multiple code locations in a web application can trigger the same portion of code of the interpreter, *e.g.*, the same operation. Though the language interpreter executes a web application, the underlying symbolic execution engine is only aware of the interpreter code but not the application code. By monitoring the interpreter alone, it cannot perceive the execution state of the web application. Specifically, the underlying symbolic execution engine maintains only the states of the interpreter. The maintained states would have the same PC value of the interpreter even though they execute different code locations of the web application—*PC value collision*. For example, the two equality comparison operations at line 7 and line 11 in Listing 1 correspond to the same code implementations in the PHP interpreter because they are comparing values in the same types. The underlying engine cannot distinguish between the two lines.

Solution: We solve this challenge by instrumenting the language interpreter. In particular, we define WebPC—the locations (*e.g.*, line number and operation types) of the executed web application code. We maintain the WebPC in the interpreter and expose it to the underlying symbolic execution engine. This is also shown as *WebPC exposure* in Figure 2. Therefore, the underlying engine can monitor the execution states of the web application.

Inefficient State Scheduling. The original state scheduling strategy of the underlying engine is designed for analyzing the interpreter. It would aim to expand the code coverage of the language interpreter regardless of the execution progress of the web application. It cannot extensively explore the web application. In particular, the PC value collision problem makes the state scheduling strategy incapable of properly arranging the states for advancing the analysis of the web application.

Solution: We incorporate the application state into the interpreter state for application state exploration. We first include WebPC in the maintained states and employ a *WebPC-oriented state scheduling algorithm* to solve this challenge. In particular, our new state scheduling algorithm refines the clustering of states (§2.2) into a hierarchical clustering tree, where WebPC is used to group the states. It then randomly selects a group so that groups with different WebPC values can be uniformly explored.

Database Interactions. Web applications commonly interact with external environments, especially databases for the application data. The data retrieved from databases can affect the execution flows [28]. The databases are managed by orthogonal database management systems (*e.g.*, MySQL [29]) that are not included in part of the language interpreters. The interpreters only provide extensions in the means of internal functions of the languages, *e.g.*, `mysqli_query()`, to interact with them. Symbolic interpreter analysis cannot symbolically reason their behaviors by analyzing the application code and interpreter code. Most prior symbolic execution tools over-approximate the retrieved data as arbitrary symbolic data [14, 15] or directly discard database operations [23, 24]. This would overlook the necessary data constraints such as data types specified in the database schema, and lead to non-negligible false results.

Solution: SIA follows the common practice in concolic execution to interact with database systems, especially when symbolic data is involved in database operations. In particular, SIA includes a module of *symbolic query concretization*, which invokes constraint solvers to generate concrete values for the symbolic data used in the queries. These resulting concrete queries are then passed to actual database operations and valid concrete results are retrieved.

4. SymPHP

In this section, we describe how we prototype SIA in a system named SymPHP for PHP-based web applications. SIA can be similarly implemented to other interpreted languages such as JavaScript and Python.

4.1. Underlying Symbolic Execution Engine

SIA employs an off-the-shelf symbolic execution engine to analyze the language interpreter. Since the PHP interpreter is purely implemented in C, engines capable of C language can meet our requirements. After our preliminary experiments, we decide to adopt S2E [7] for our implementation of SymPHP. We have tried other options including KLEE [10], QSYM [30], SymCC [11], and SymSan [20]. However, our attempts were unsuccessful. We present our experience in §A. We believe once these engines are robust enough to analyze the PHP interpreter, they can be used to facilitate our realization of SIA.

4.2. Handling Symbolic Data

4.2.1. Symbolic Web Input Specification. An initial step of concolic execution is to specify symbolic data. Super global variables are often regarded as the source of external inputs in symbolic execution [2, 14]. PHP interpreter maintains all variables and values of the web application code in its memory, including super global variables. SymPHP thus specifies the memory regions of super global variables as symbolic in the interpreter. SymPHP directly invokes the `s2e_make_symbolic()` API function for the specification after the request data is copied to the super global variables, thereby enabling the concolic execution driven by the concrete request data [21]. It is worth noting that our solution is based on the PHP interpreter. It can be applied to the target web applications without modifications to the applications.

Some symbolic execution engines might define their own custom API functions for flexibly specifying symbolic data, *e.g.*, `make_symbolic()`. We take a different option for two reasons. First, using such API functions requires modifications of the target web applications, *e.g.*, calling the corresponding API function in the application code. This is often undesired especially when super global variables are extensively used across the application modules, and some are inlined. Second, directly marking the corresponding super global variables as symbolic when they are initialized in the interpreter can ensure the completeness of the specification.

4.2.2. Tracking Symbolic Data. SymPHP relies on the underlying symbolic execution engine to track the symbolic data. We have illustrated this process in §3.3.1. Specifically, S2E can track the usage of the specified symbolic memory regions in different PHP operations by symbolically analyzing the corresponding interpreter code. It automatically propagates the symbolic data in value assignment statements. Such a design naturally avoids significant engineering efforts.

4.2.3. Concretizing Queries for Database Operations. Web applications frequently interact with database systems. Symbolic data can propagate to the database operations, constituting database queries. For example, the `$_GET['name']` is used as a part of the query `$q1` at line 4 of Listing 1. The query becomes symbolic if the external input `$_GET['name']` is set symbolic. To solve this issue, one can extend the scope of symbolic execution to the database management system. However, this is computationally expensive as the engine has to analyze much more code. Existing approaches often approximate database operations and treat the retrieved data (*e.g.*, line 5) as arbitrary data [15, 23]. This is imprecise and can introduce false positives.

We propose concertizing the queries for database operations when symbolic data is involved as part of the queries. Specifically, we first solve path constraints to obtain concrete values for symbolic data, which compose in part concrete queries. The concrete queries are then directly passed to the actual database operations to retrieve concrete database data. This is precise as always valid data is retrieved. For example, the symbolic data in queries (`$q1` and `$q2`) are concretized when they are passed to `mysqli_query()` function to query database systems. The PHP interpreter uses several internal functions as the binding layer to interact with database systems. We thus extend the related internal functions in the PHP interpreter to achieve symbolic query concretization.

4.3. Exploration Strategy

SymPHP symbolically analyzes web applications on the symbolic inputs. In this part, we first describe how we expose the execution information of the web application to S2E (§4.3.1). We then present the design of our new state scheduling algorithm (§4.3.2).

4.3.1. WebPC Exposure. The maintained states of S2E include PC values of the target program—PHP interpreter in our case. We propose incorporating the application state into the interpreter state for effective application state exploration. We define *WebPC* as the location of the web application code being executed presently. We include WebPC values in the states maintained by S2E.

PHP programs are executed in the form of a sequence of byte-code instructions because the PHP interpreter first transforms PHP source code files into byte code. The byte code is dynamically generated on the fly when a PHP script is requested. It is thus non-trivial to directly locate a dynamically generated byte-code instruction. Therefore, we correlate the line numbers of the PHP programs with the byte-code instructions and alternatively record the line number in WebPC for each instruction.

One single line of PHP code can generate multiple byte-code instructions, each corresponding to one basic operation.

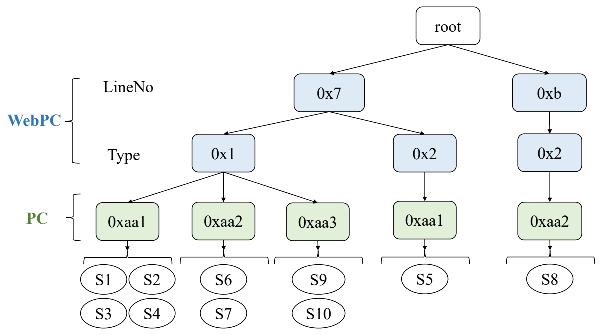

Figure 3: A state clustering tree based on WebPC.

Multiple byte-code instructions thus can share the same line number. For example, the internal function call of `strtolower()` and equality comparison at line 11 of Listing 1 correspond to two distinct byte-code instructions; however, they have the same line number of 11 in the PHP program. This leads to a collision problem. To mitigate this issue, we additionally include the type of byte-code instruction in WebPC. The type specifies the class of instructions. Therefore, most byte-code instructions can be uniquely located, resulting in more accurate WebPC in our design.

$$WebPC(ins) = LineNo_{web}(ins) << 8 \mid Type(ins) \quad (1)$$

We formalize WebPC in Formula 1 as a function of the currently executed byte-code instruction ($WebPC(ins)$). It is a 32-bit integer and constitutes two parts: 1) $LineNo(ins)$, the corresponding line number in PHP code described in the first 24 bits, and 2) $Type(ins)$, its instruction type described in the last 8 bits. It is not necessary to include the file names of PHP scripts in WebPC because we take a similar approach like Witcher [18] to separately test each PHP script (more details in §5.4). However, adding the file name can be easily achieved, for example, by concatenating the hash value of the file name to the current form of WebPC. We leave this extension to interested readers.

We instrument the PHP interpreter to expose WebPC of currently executed PHP byte-code instructions to S2E. The Zend virtual machine of the PHP interpreter provides a trace module. We enhance it to capture the required information of WebPC. We then use the `s2e_invoke_plugin()` API function to notify S2E of the WebPC value so that S2E can include it in the maintained states. As a result, each state in S2E contains both WebPC and PC.

4.3.2. WebPC-Oriented State Scheduling. We develop a new state scheduling algorithm to select states based on the exposed WebPC. It aims to expand the coverage of the web application code. Similar to the state clustering tree of S2E (§2.2), our algorithm first constructs a state clustering tree based on the values of WebPC and PC. Specifically, it first categorizes the states by their WebPC values (LineNo and Type, respectively). States with the same values are grouped together. Within each group, it further clusters the states by their PC values. This results in a hierarchical state clustering

Algorithm 1: State scheduling algorithm.

```
    input  : states
    output : nextState
1   root ← treeConstructor(states)
2   nextState ← selectOne(root)
3   return nextState
4
5   function selectOne(root):
6       // randomly select one child of root
7       numChildern ← size(root.children)
8       if numChildern > 0 then
9           randomIndex ← randomInteger(0,
              numChildern) // get a random integer
10          randomChild ← root.children[randomIndex]
11          return selectOne(randomChild)
12      else
13          return root
14      end
15  end
```

tree, where the states are stored in the leaf nodes. Take the ten states in Figure 1 as an example, Figure 3 shows the state clustering tree under the new clustering method. Based on the maintained WebPC values, the states are accordingly adjusted in the leaf nodes.

Our state scheduling algorithm is performed at the top of the state clustering tree. The key insight is to properly select states so that groups with distinct WebPC values get uniformly explored. Meanwhile, the PC values also supplement the state selection. As shown in Algorithm 1, the `treeConstructor()` function at line 1 constructs the state clustering tree and returns its root node. The algorithm then selects a state for further analysis by the `selectOne()` function (line 2). Specifically, the function randomly selects a child of the root node if it has at least one child (line 10). This is done by generating a random integer (`randomIndex`) as the index to select the child. Such a strategy ensures that all children of a root node have the same probability to be selected. The process iterates on the selected child until it reaches the leaf node, where the corresponding state is returned (line 11 and line 13). Since WebPC values incorporate the location of the web application code, our algorithm ensures that specific code locations are not excessively executed, effectively mitigating path explosion in loops and recursions.

We illustrate our algorithm with the example in Figure 3. Initially, the root node has two children: `LineNo:0x7` and `LineNo:0xb`. They are given the same chance (*i.e.*, $\frac{1}{2}$) since they refer to distinct locations in the PHP application. Suppose `LineNo:0x7` is selected, its two children `Type:0x1` and `Type:0x2` have the same selection probability, *i.e.*, $\frac{1}{2} \times \frac{1}{2}$. Ultimately, the selection probabilities of `S1` and `S9` are $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4}$ and $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{2}$, respectively. They are lower than those in the original S2E algorithm because the two states both execute the same locations (line 7) in the PHP program. On the other hand, `S8` has a selection probability of $\frac{1}{2}$ because it is the only state that executes line 11 (0xb). As shown, states with different locations in the PHP program can be well distinguished and prioritized.

TABLE 1: Breakdowns of implementation details.

Component	**Modification (LoC)**
S2E [7]	1,500 (C++)
PHP Interpreter	300 (C)
Coordinator	300 (Bash and Python)

4.4. Implementation

We handle the web request using the Common Gateway Interface (CGI) of PHP. Normally, a web server is configured with a PHP module. It receives from clients HTTP requests for PHP scripts and delivers the task to the PHP module. The module executes the PHP code to dynamically generate the HTML or other content. Such a workflow glues the web server and the PHP interpreter, making it hard to symbolically analyze only the PHP interpreter through SIA. Fortunately, PHP CGI allows directly invoking the web application through environment variables and standard inputs from the command line. We thus translate HTTP requests into CGI requests and then separately leverage the CGI mode of PHP interpreter to handle the requests. Witcher also takes a similar strategy [18].

For vulnerability detection, *we emphasize that the key contribution of this work is to support symbolic exploration of web applications via SIA, rather than techniques to detect specific types of vulnerabilities.* We thus leverage the fault escalation technique proposed in Witcher to identify SQL injection and command injection vulnerabilities. Besides, we also equip SYMPHP with a basic detector for reflected server-side XSS vulnerabilities [31].[1] The detector injects a specially crafted payload and checks if the payload appears in the response. We leave supporting other types of vulnerabilities as the future work.

We implement SYMPHP for PHP 7 and PHP 8. The two versions together account for 80.9% of PHP usage on the web [16]. We summarize our efforts for prototyping SYMPHP in Table 1. Our implementation takes only 2.1K LoC in total, with 1.5K LoC for the core S2E component. The modifications are much smaller than prior solutions including XSym [14] and Navex [15]. We develop our state scheduling algorithm as a plugin of S2E. Our modifications are not tailored to PHP and can be ported to other interpreters. Our experience shows that prototyping SIA to a different version of PHP would take only *several hundred LoC using weeks of effort.*

5. Evaluation

In this section, we first describe our experimental setup (§5.1). We evaluate the comprehensiveness of PHP syntax support (§5.2) and web application code exploration (§5.3). We then evaluate the vulnerability detection capability (§5.4)

1. Without further clarification, the term XSS in this context specifically refers to reflected server-side XSS.

and investigate the internals of SYMPHP (§5.5). We also validate the benefits of SYMPHP in hybrid fuzzing (§5.6).

5.1. Experimental Setup

Dataset. We aim to construct a comprehensive and unbiased dataset for the evaluation. Our selection criteria are to include the popular and complex web applications that have been well evaluated by recent PHP program analysis works [2, 18, 28, 32]. As a result, we include 17 web applications in our evaluation, shown in Table 2. Some web applications are excluded from our dataset, even though they are evaluated in the above-mentioned tools. There are two reasons. First, some web applications (*e.g.*, SchoolMate) can only run on PHP 5, which has become obsolete since 2004 and therefore is not supported in our current implementation of SYMPHP. Second, due to unresolved dependency issues, we failed to install some web applications in our environments. For instance, one of the authors spent over 20 hours trying to install Codiad [33], but still did not succeed. It is worth noting that we use old versions of the applications as in prior works [15, 28, 32] because the previously known vulnerabilities in them allow us to better evaluate the vulnerability detection capability. Nevertheless, SYMPHP can be applied to the latest versions to detect previously unknown vulnerabilities, and we demonstrate that in §5.6.

The rest of the web applications included in our dataset are popular ones. For example, web applications such as MediaWiki [34], WordPress [35], PHPBB [36], Drupal [37], and OpenEMR [38] have millions of deployments on the Internet. The applications are complex. In total, the web applications contain around 51K files and 9M LoC. The only exception is Witcher-test, which is a benchmark constructed by the authors of Witcher [18] with only 246 LoC. It aims to test a tool's capability of generating specially crafted testing inputs. It well suits the assessment of concolic execution engines like SYMPHP. Therefore, we also add it to our dataset.

We manually installed the web applications and initialized the databases in the default settings using a considerable amount of time. For most web applications, we created a user account and set up its credentials. We can use them to achieve automated authentication in further testing. The containers for the experiments are running Ubuntu 22.04 with a shared memory of 1GB. They are set up on a server with a 36-core Intel Xeon CPU and 96GB RAM.

Comparison Targets. We include the state-of-the-art PHP symbolic execution engine, XSym [14], in the evaluation as a comparison target. Other symbolic execution engines are excluded from the comparison because their code is not publicly available [23, 24, 28]. Although Navex is open-sourced, its code was said to be incomplete [22, 39, 40]. One author spent more than one week setting up Navex but still failed to fix it. We thus exclude the quantitative comparison with Navex in this part. It is worth noting that XSym is a static symbolic execution engine rather than a concolic one. It directly analyzes the application's code representations to check vulnerabilities. We acknowledge that the comparison is not perfectly fair. However, it could still allow us to systematically compare concolic execution and conventional symbolic execution.

AnimateDead [41] is a concolic execution engine for web application debloating. It removes code not exercised/covered during concolic execution. We would like to also include it in the comparison. However, we did not completely succeed in the tool setup. We instead directly reuse the data presented in the paper [17] for a general comparison.

Experiment Procedures. As a concolic execution, SYMPHP requires concrete inputs to drive its exploration. Similar to SymCC [11] and SymSan [20], we employ a fuzzer to generate such concrete inputs. There are a few web application fuzzers that can facilitate our needs, including Black Widow [42], Witcher [18], and WebFuzz [43]. We choose Witcher in our current experiments because it is considered the state-of-the-art grey-box fuzzer for web applications as shown in its evaluation results.

During fuzzing, we provide Witcher with the entry URL of the application, login URL, and user credentials. Witcher operates in two steps by design: initially identifying the entry URLs and subsequently conducting fuzzing on each URL. Following the instructions of Witcher, we limit Witcher's crawler to run for four hours per application. In the second step, Witcher separately tests each URL for a time budget. In our testing, we set the time budget to ten minutes. As a result, the total testing time would vary across web applications depending on the number of URLs identified by the crawler. Normally, the number of URLs is positively related to the complexity of web applications [18].

In the second step, we collect the corresponding concrete inputs on the server side. We provide the collected concrete inputs to SYMPHP to perform concolic execution. Each concrete input includes the URL, request method (*e.g.*, POST or GET), request data, and query string. SymCC and SymSan configure a time budget for the analysis of a concrete input. We set a time budget of ten minutes for each URL and uniformly distribute the budget to the concrete inputs associated with the URL. This is because modern web applications have thousands of URLs, and each URL is further associated with a number of concrete inputs. The total number of concrete inputs would be substantially huge. Setting a time budget per URL is a necessary adjustment to ensure the experiments can finish within a reasonable amount of time. In our experiments, we apply the same time budget used by SYMPHP per application to XSym.

5.2. Syntax Support

During the experiments, we check if SYMPHP can fully support PHP syntaxes as we expect. We thus monitor the analysis of SYMPHP and check the logs for any syntax error. The results show no such syntax errors occurred during the entire experiments. XSym rather reported multiple (more than 20) issues with incomprehensive syntax support regarding arrays, dynamic function calls, *etc.* Though being a static and

TABLE 2: Evaluation results on vulnerable web applications. SymPHP$_{ns}$ is a variant of SymPHP with the WebPC-oriented state scheduling disabled. TO means the tool reaches the time limit before completing the analysis. ERR means the analysis could not finish because of unexpected errors. $*$ denotes a previously unknown vulnerability that remains in the latest version. † denotes the average value across applications.

Application				Code Coverage (%)			Vulnerability Detection (#)				
ID	Name	# Files	# LoC	Witcher	SymPHP	SymPHP$_{ns}$	Known	Witcher	XSym	SymPHP	SymPHP$_{ns}$
1	MediaWiki 1.2.7	3,264	184,683	25.19	34.22	28.84	7	2	4	6	2
2	WordPress 4.7	591	26,801	25.34	43.97	27.52	10	3	TO	10	4
3	WordPress 5.1	706	336,985	15.91	32.98	29.01	5	2	TO	4	3
4	OSCommerce 2	533	54,561	33.28	51.10	42.19	6	2	3	6	2
5	PHPBB 3.1.10	1,401	330,269	21.4	43.98	26.78	3	0	TO	3	1
6	Witcher-Test	8	246	100	100	100	7	7	7	7	7
7	Drupal 8.0	8,111	152,656	19.54	40.82	22.48	5	1	5	4	2
8	Drupal 9.0	11,078	252,249	23.72	49.09	26.92	4	0	1	1	1
9	OpenEMR 5.0.2	9,004	225,221	42.12	53.90	44.92	13	3	TO	10	3
10	OpenEMR 6.0.2	8,565	6,794,691	21.87	35.24	27.67	5	3	TO	4	3
11	Opencart 2.3.0.2	2,179	291,403	42.51	67.43	50.53	3	2	TO	3	2
12	Opencart 3.0.3.8	2,601	93,220	50.48	59.12	53.38	2	1	0	2	1
13	Webid 1.2.2	322	55,738	13.23	27.98	15.93	4	0	2	2	0
14	Silverstripe 3.5.3	777	200,808	21.47	39.23	27.28	5	3	ERR	3	3
15	WebChess 7	29	5,017	49.05	89.90	61.26	4	3	2+1*	4+1*	3
16	HSM 19	817	9,181	58.32	75.18	66.17	4	1	2	2	1
17	phpMyAdmin 4.7	1920	130,389	22.49	43.09	33.91	14	4	8	7	6
Total	-	51,906	9,144,118	35.21†	51.57†	40.07†	101	37	34+1*	78+1*	44

dynamic symbolic execution, respectively, the experiments can still demonstrate the significant benefits of SIA. Although Azad *et al.* claimed that AnimateDead [17] could also support all syntaxes of PHP 7, it heavily relies on their dedicated engineering. Adding support for new features in PHP 8 will still take a considerable amount of time.

We further assess the syntax support using unit testing. We apply SymPHP and XSym to the PHP test suite equipped in the official PHP interpreter repository [44]. Each test case in the suite contains a simple PHP program exercising specific PHP syntax. We first randomly set a variable in the PHP program symbolic and use SymPHP to find a satisfying concrete value.[2] We then check if the tools 1) do not raise syntax errors and 2) do produce expected results. For the latter, we apply the concrete value again to the test suite to verify its correctness. Regarding the results, out of 152 test cases, SymPHP passed all of them, while XSym succeeded for 86. This further confirms the comprehensive syntax support of SymPHP.

5.3. Code Coverage

To understand how comprehensive SymPHP can symbolically explore target web applications, we measure the code coverage SymPHP can achieve. We replay the concrete inputs generated by SymPHP during symbolic analysis and use Xdebug [45] to measure the total exercised code lines.

2. We excluded simple tests that do not use any variables.

However, we cannot measure the coverage achieved by XSym because XSym only generates concrete inputs for those suspicious locations it labels. Replaying these inputs does not reflect its actual code coverage.

SymPHP is highly effective in exploring web application code. We show in Table 2 the proportion of exercised code of both Witcher and SymPHP. For this, we used PHP-Parser [27] to parse the application's source code to compute full code coverage at code line level. The results suggest that SymPHP could significantly improve code coverage atop the concrete inputs of Witcher. Specifically, SymPHP and Witcher achieved an average code coverage of 51.57% and 35.21%, respectively. This means that after thorough fuzzing, SymPHP could still improve the code coverage by 46.46%.

Besides syntax support and engineering effort advantages, the code coverage of SymPHP is comparably better than other dynamic concolic execution works [17, 28]. However, among the applications, WordPress and Webid have relatively lower coverage. We manually inspected them and identified two possible reasons. First, we used the default installation configuration to run the experiments. Some code modules are not reachable in our configurations [46]. For instance, we chose English as the language setting for all applications. Code related to other languages could not be covered in this setting. To cover such code, special configurations are required. Second, some code is only reachable after multi-step navigation due to the dependencies among web pages. Our current concolic execution tests a URL at a time and cannot simulate the page navigation. Other advanced techniques such as the navigation models [15, 42] can help mitigate this issue.

Comparison with AnimateDead. Two out of the four applications evaluated by Azad *et al.* are also included in our dataset [17]. The proportion of remaining code in their paper after debloating denotes the ultimate code coverage. AnimateDead covered 54% and 31% of code for WordPress and phpMyAdmin, respectively. We find that AnimateDead selectively employs forced execution for complex path conditions—when it explores both branches of a conditional statement. The authors discussed that this would overestimate the code coverage by 4% to 15%. SymPHP achieved similar results on the two applications compared to AnimateDead, which was developed using over 13 person-months.

5.4. Vulnerability Detection

Besides code coverage, we assess the capability of SymPHP in detecting vulnerabilities. In the application dataset, we include known vulnerabilities (*i.e.*, command injection, SQL injection, and reflected server-side XSS) to facilitate false negative assessment. We investigated the CVE website and/or the applications' official vulnerability report channels, *e.g.*, GitHub issues. Specifically, we searched keywords specifying the vulnerability type and application version and manually identified the vulnerabilities.[3] Within the 17 web applications, we identify 101 in-scope known vulnerabilities. We acknowledge that our manual vulnerability collection may not be exhaustive. However, the search is done in a *best-effort manner*. An alternative method to constructing an unbiased vulnerability dataset is to find all the vulnerabilities in the popular web frameworks from the CVE database in recent years. However, we abandon it because the vulnerabilities can concern a large number of application versions, resulting in unaffordable installation efforts.

SymPHP is highly effective in detecting known injection vulnerabilities in our dataset. As shown in Table 2, SymPHP successfully detected 78 (77.23%) out of 101 vulnerabilities. Besides, SymPHP even identified a previously unknown vulnerability in WebChess that remains in the latest version. We have responsibly reported the vulnerability to the developers.

False Negatives. As a dynamic analysis approach, SymPHP has false negatives. We systematically analyzed the 23 missed vulnerabilities and found 11 of them were because the crawler could not successfully find the corresponding URLs. Therefore, the relevant code was not exercised by SymPHP at all, leaving the vulnerabilities unexposed. Improving the crawler or extending the crawling time can potentially mitigate this issue. For the remaining 12 cases, the URLs were successfully extracted by the crawler but SymPHP could not generate a concrete input to trigger the vulnerabilities. Given the complex and random nature of exploration in concolic execution, we could not confidently conclude a cause for them. Nevertheless, the false-negative rate is still considered low, as compared to other dynamic analysis tools [18, 18, 28].

False Positives. We observed 12 false positive cases in SymPHP's results. All of them are about XSS vulnerabilities. This is basically because of the simple static XSS detector. Our XSS detector statically checks if certain special payloads exist in the response through literal textual matching. It does not launch a browser to dynamically load the response (including HTML and JavaScript). As a result, it still labels vulnerabilities when the payloads appear in error information or the responses with `application/json` content-type header. Nevertheless, we do not emphasize the contribution of XSS detection, and the issue can be resolved with more advanced detection techniques [42, 43].

5.4.1. Comparison.

We present the comparison results with the state-of-the-art solutions.

Witcher. We find that the concrete inputs generated by Witcher could directly trigger 37 (36.63%) vulnerabilities. All of them were identified by SymPHP since SymPHP is driven by the concrete inputs. SymPHP could additionally find 41 more vulnerabilities, which stands for a significant improvement of 110.81% over Witcher.

XSym. SymPHP significantly outperforms XSym in vulnerability detection. For six applications, XSym reached the time limit (shown as `TO`) and could not finish the analysis. It also aborted its analysis on one application because of unexpected errors (shown as `ERR`). In the remaining ten applications, XSym was able to find 34 (33.66%) known vulnerabilities. This suggests that SymPHP achieves a much higher vulnerability detection efficacy and analysis applicability. Among the 34 detected vulnerabilities, SymPHP could identify 30 of them. The reason why four vulnerabilities were not identified by SymPHP is that the corresponding URLs were not uncovered by the crawler. Therefore, SymPHP did not test the scripts, leaving the vulnerability unexposed. Conventional symbolic execution engines like XSym do not require concrete inputs to drive the analysis and could identify them via whole-application analysis. In our later investigation, we manually feed the URLs to SymPHP. SymPHP could also trigger the vulnerabilities within ten minutes. Therefore, we believe this is not a limitation of SymPHP—once the URL is extracted, SymPHP could identify it. XSym also detected the same new vulnerability in WebChess.

We now investigate why XSym failed to analyze the seven applications. The analysis of XSym is in two stages. The first stage performs path forking and collects the corresponding constraints for potentially vulnerable locations. In the second stage, constraint solvers are invoked to check the feasibility of the collected constraints. For the six `TO` cases, XSym did not even advance to the second stage. The cause is probably because of the path explosion in its simple path-forking strategy. We also inspected the `ERR` case. We noticed a high usage of memory, which could be a potential reason why XSym aborted the analysis.

XSym has many false positives and negatives. XSym missed all 46 vulnerabilities in the abovementioned seven

3. Some vulnerabilities existing in the optional functionalities, *e.g.*, WordPress plugins and themes, are not considered because they are not included in the main codebase of the application by default.

applications and 21 vulnerabilities on the remaining ten applications. It also reported around 40 false positives. Though interesting, it is hard to attribute every false case to specific causes. Our analysis reveals that XSym's incomplete support of PHP features stands for an essential cause. Though XSym aims to automatically analyze PHP internal functions, it only supports PHP 3, which has become obsolete since 2000. Program paths with newer internal functions mostly cannot be correctly analyzed. We have illustrated this in the motivation example (§2.3). We additionally provide the analysis of a false negative case in §B.

AnimateDead. Although not designed for vulnerability detection, the vulnerabilities (*i.e.*, vulnerable code locations) preserved by AnimateDead can be used to (overly) approximate its capability in vulnerability detection. As stated in the paper [17], AnimateDead could reach 65% and 35% of vulnerability locations in WordPress and phpMyAdmin, respectively. SYMPHP achieved similar (if not better) detection results on the two applications. Note that the estimation of AnimateDead's results is likely the upper bound because 1) reaching code location does not necessarily mean triggering the vulnerability, and 2) AnimateDead relaxes the path constraints in its forced execution.

5.5. Understanding the Performance

We further evaluate the efficacy of the state scheduling algorithm to help understand the performance of SYMPHP. We also measure the system initialization overhead and the performance difference in the concrete mode of SYMPHP and concrete execution.

Efficacy of WebPC-Oriented State Scheduling. We design SYMPHP$_{ns}$, a variant of SYMPHP with the WebPC-oriented state scheduling disabled. It uses the default PC-based state scheduling algorithm of S2E. We evaluate SYMPHP$_{ns}$ on the same dataset and present results of vulnerability detection and code coverage in Table 2. Since SYMPHP$_{ns}$ also uses the concrete inputs generated by Witcher, it triggers all code and vulnerabilities triggered by Witcher.

By comparing the SYMPHP's and SYMPHP$_{ns}$'s improvements above Witcher, we conclude that our state scheduling algorithm could greatly improve the exploration of web applications. Specifically, we observe that SYMPHP$_{ns}$ has a similar performance to Witcher, demonstrating its negligible improvements over Witcher. For example, SYMPHP$_{ns}$ could only identify one additional vulnerability in Drupal 9.0, and cover 4.83% more code compared to Witcher. As mentioned earlier, the full-fledged SYMPHP could significantly improve Witcher in both vulnerability discovery and code coverage.

Initialization Overhead. SYMPHP has an initialization phase, during which SYMPHP starts S2E's virtual machine to prepare the environment for the symbolic execution. We measure the duration used for the initialization. Our results find that SYMPHP takes around six to nine seconds to start up. Note that the initialization overhead is independent of target web applications. It normally needs to be done only once for the analysis of a web application. We thus believe the initialization overhead is negligible in continuous web testing.

TABLE 3: Average RTT (millisecond) in concrete execution and concrete mode of SYMPHP.

ID	Concrete Exe.	Concrete Mode	Factor
1	258	1,729	6.70
2	247	1,076	4.35
3	198	1,391	7.14
4	419	3,163	7.55
5	381	1,738	4.56
6	48	181	3.77
7	261	2,817	10.79
8	378	2528	6.69
9	398	2,903	7.29
10	165	1,792	10.86
11	232	1,401	6.04
12	274	965	3.52
13	391	2,571	6.58
14	249	2,310	9.28
15	102	477	4.68
16	518	1943	3.76
17	498	2492	5.00
Average	295	1852	6.39

Overhead in Concrete Mode. When no symbolic input is specified, SYMPHP runs in a concrete mode without the symbolic backend invoked. We compare the processing time in the concrete mode of SYMPHP to pure concrete execution. To do this, we replay the collected concrete inputs on the web applications deployed on a server. We measure the round-trip time (RTT) from when a client sends a request to when it receives the first response from the server. The client and server are hosted on the same machine to eliminate the network latency. Therefore, RTT can accurately reflect the real processing time. We run the experiments in two settings: 1) concrete execution, and 2) concrete mode of SYMPHP.

The average RTT per application is shown in Table 3. In concrete execution, most of the applications can process a request in several hundred milliseconds. The concrete mode of SYMPHP, which runs on S2E's virtual machine, requires around two thousand milliseconds on average. We calculate the ratio between the two as a factor value. It shows that the concrete mode of SYMPHP brings an average overhead of 6.39×. The overhead is mainly due to checks for memory accesses according to the descriptions of Chipounov *et al.* [7]. Besides, access to database systems is also a reason.

5.6. Application of SYMPHP: Hybrid Fuzzing

Though the goal of this work is to generally advance concolic execution, we showcase one of its typical applications,

TABLE 4: New vulnerabilities discovered in hybrid fuzzing.

Application	# Cmd Inj.	# SQL Inj.	# XSS
WebChess 7	1	1	2
OpenEMR 7.0.1	1	-	-
Symfony 7.0	1	-	-
Opencart 4.0	-	1	-
Silverstrip 5.0	-	-	1
HSM 20	-	-	2

hybrid fuzzing. We integrate SYMPHP with the state-of-the-art fuzzer Witcher into a hybrid fuzzing framework. We aim to check if and how well SYMPHP could supplement fuzzing. XSym is a static symbolic execution engine, and it cannot be directly used for hybrid fuzzing. Therefore, it is excluded from this evaluation. We run Witcher in the master-slave mode with two fuzzer instances. The two instances can actively synchronize their favored inputs during testing. We then start another instance of SYMPHP, which utilizes the inputs generated by the fuzzer instances. The symbolic execution instance also provides interesting inputs (GET, POST, and Cookie data) to the fuzzer instances in return. This is a basic way for implementing hybrid fuzzing [11, 20]. Other advanced techniques [47] can potentially improve our naive hybrid fuzzing framework. However, we think they are out of the scope of this research. Note that this is a different setting to §5.4 as the instances of fuzzer and concolic execution are running simultaneously and cooperatively. We also run another version with only the fuzzer instances as the baseline.

Since we have already evaluated SYMPHP on old versions of web applications and thoroughly analyzed its efficacy, we mainly use the hybrid fuzzing framework to find new vulnerabilities in the *latest versions*. We apply the framework to a set of latest version web applications, most of which correspond to the ones listed in Table 2. We found ten previously unknown vulnerabilities, including three command injections, two SQL injections (including the one mentioned in §5.4), and five XSS. Details are shown in Table 4. It is worth highlighting that we even identified one critical vulnerability in the famous Symfony application, which has been thoroughly tested by its developers. None of the new vulnerabilities are discovered in the baseline with only fuzzing. This demonstrates the optimistic effects of SYMPHP in identifying real-world vulnerabilities. We have responsibly disclosed all identified vulnerabilities to relevant vendors and closely worked with the developers to confirm and fix them. To date, the developers have confirmed two new cases.

We further depict the code coverage over time in Figure 4. Since we separately test each URL, we accumulate the covered code of all URLs at a time and show the sum within the time budget of ten minutes. Due to the space limit, we only present the results for four web applications where SYMPHP has identified new vulnerabilities. The results show that the hybrid fuzzing framework could improve the code

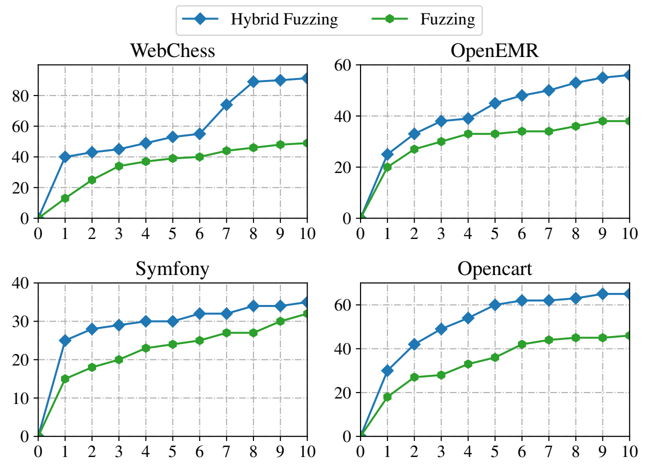

Figure 4: Code coverage (%) over time (hour).

coverage significantly by up to 85.71%, compared with the baseline. This additionally proves the practical benefits of SYMPHP to supplement dynamic fuzzing.

6. Discussion

State Changes. Web applications are stateful. Earlier executions have impacts on database states and future exploration. In the two-step design of Witcher, the second step can potentially operate on invalid or obsolete states. To mitigate this state change problem, we currently restore target web applications to the initial clean states for each entry URL. To better solve it, one can take snapshots on the states [48] and restore them when necessary. We leave it as a future work.

State Scheduling Algorithm. SYMPHP employs a WebPC-based state scheduling algorithm to address the inefficient state scheduling issue. We have demonstrated that it could allow SYMPHP to explore diverse locations and enhance overall code coverage. However, the algorithm is based on uniform randomness; each child node of the root node has an equal probability of being selected. Numerous advanced algorithms (*e.g.*, weighted randomness, breadth-first-search, depth-first-search) from the literature [7, 10, 49, 50] can be applied to improve the current algorithm. For example, one could customize SYMPHP to prioritize critical code locations with higher weights for vulnerability validation. Nevertheless, we believe the current strategy has paved a good foundation for such enhancement or customization. We thus leave this as a future work.

Runtime Overhead. The runtime overhead of SYMPHP can be improved, as we present in §5.5. This can be mitigated by using more efficient underlying symbolic execution engines. Recent advances of compilation-based symbolic execution approaches are desired directions to boost the performance of SYMPHP. Though we failed to apply them as we present in §A, we do not see any fundamental difficulty that prevents us from using them once these engineering issues are fixed. We are eager to work in this direction in the future.

Detecting Other Vulnerabilities. We can extend SymPHP to find other types of vulnerabilities by equiping other vulnerabilities checkers. For example, FUSE [3] employs an upload validator to detect file upload bugs. Black Widow [42] checks the reflection of a given random token to identify stored cross-site scripting vulnerabilities. FUGIO [51] checks the execution trace of the injection objects to identify PHP object injection vulnerabilities. SymPHP can be directly extended with these techniques. Moreover, to identify second-order vulnerabilities [18, 52, 53], it might require to set other results, *e.g.*, database data, as symbolic. This could be potentially achieved by hooking the relevant database functions and marking the returned data as symbolic.

Analyzing Other Web Applications. SIA can be ported to other interpreted programming languages like Python and JavaScript by design. In our current implementation, we developed the core part of SymPHP in the underlying engine, which takes as input the web application code and the exposed WebPC. To extend SIA to other interpreted languages, WebPC exposure is needed. Interpreters typically contain a main interpretation loop that switches over the instruction types and invokes specific handlers. Therefore, WebPC exposure can be realized by enhancing the interpretation loop with modest engineering. Besides, SIA is by no means limited to only database-backed PHP applications. Our preliminary investigation shows that SymPHP can help test standalone PHP libraries and applications. For example, we have attempted to integrate SymPHP with the PHP library fuzzer, PHP-Fuzzer [54].

7. Related Work

Symbolic Execution. Symbolic execution has been widely used for web security [14, 15, 17, 23, 24]. For example, SYNTHDB [28] employs concolic execution to help generate a comprehensive database for dynamic testing. MalMax [55] leverages concolic execution to scan server-side malware. As we introduced in §2.3, these symbolic execution engines have non-trivial limitations such as the demand of excessive engineering. SymPHP is a completely different concolic execution engine that is based on the idea of SIA. Bucur *et al.* propose Chef to prototype symbolic execution engines for Python and Lua [56]. However, Chef does not consider the domain-specific issues of web applications such as database interactions, web data processing, *etc.*, which we have tackled in this work.

Static Analysis for Web Applications. Static approaches analyze the source code (or its representations) to identify vulnerabilities. Especially, most of them apply taint analysis to detect taint-style vulnerabilities such as SQL injection and server-side cross-site scripting [1, 2, 26, 32, 57, 58]. For example, TChecker [2] and PHP Joern [26] transform the source code of PHP-based web applications to code property graphs and perform query-based analysis on the graph. UChecker [58] and LChecker [32] are based on abstract syntax trees generated by PHP-Parser [27]. SymPHP differs from these works as a concolic execution approach. Nevertheless, static analysis can assist the exploration of symbolic execution. For example, we can use static analysis to pre-screen the web applications and leverage symbolic execution engines like SymPHP to selectively explore potentially vulnerable paths.

Dynamic Analysis for Web Applications. Dynamic approaches generate concrete inputs to test web applications [18, 42, 59]. Since web applications are dynamic and stateful, many approaches model the states of web applications, aiming to improve the coverage during dynamic testing. Enemy of the State [59] infers the server-side states by comparing the differences in response on the client side. Navex [15] instead monitors on the server side by tracking session creation and database queries. Besides, jÄk [60] and Black Widow [42] also consider client-side events such as form submissions. By modeling the states, dynamic approaches can achieve better code coverage. We demonstrated that SymPHP could supplement dynamic fuzzing in §5.6. SymPHP would also benefit from state models.

8. Conclusion

We have introduced SIA, a new and holistic concolic execution approach for web applications. By leveraging an off-the-shelf symbolic execution engine to analyze the language interpreter, SIA addresses the intrinsic limitations of prior solutions and achieves effective symbolic execution. Our prototype implementation for PHP-based web applications has demonstrated the merits of SIA in terms of language syntax support, code coverage, vulnerability detection, and hybrid fuzzing. We believe that SIA would open up new possibilities for web application symbolic execution and significantly enhance web security.

Acknowledgments

The authors would like to thank the anonymous reviewers and the shepherd for their valuable suggestions. The authors also thank the S2E team for open-sourcing such a remarkable tool. The work described in this paper was supported in part by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14209323).

References

[1] 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, Feb. 2014.

[2] 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.

[3] T. Lee, S. Wi, S. Lee, and S. Son, “Fuse: Finding file upload bugs via penetration testing.” in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2020.

[4] “How often do cyber attacks occur?” Jun. 2020, https://aag-it.com/how-often-do-cyber-attacks-occur/.

[5] Q. Wu, Y. He, S. McCamant, and K. Lu, “Precisely characterizing security impact in a flood of patches via symbolic rule comparison,” in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[6] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, “Precise and scalable detection of double-fetch bugs in os kernels,” in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[7] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[8] J. Song, C. Cadar, and P. Pietzuch, “Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications,” *IEEE Transactions on Software Engineering*, 2014.

[9] F. Brown, D. Stefan, and D. Engler, “Sys: a static/symbolic tool for finding good bugs in good (browser) code,” in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[10] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[11] S. Poeplau and A. Francillon, “Symbolic execution with symcc: Don’t interpret, compile!” in *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[12] C. Liu, Y. Chen, and L. Lu, “Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel,” in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2021.

[13] “The php interpreter,” Dec. 2021, https://github.com/php/php-src.

[14] P. Li, W. Meng, K. Lu, and C. Luo, “On the feasibility of automated built-in function modeling for php symbolic execution,” in *Proceedings of the Web Conference (WWW)*, Ljubljana, Slovenia, Apr. 2021.

[15] 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, Aug. 2018.

[16] W3Techs, “Usage statistics of PHP for websites,” Nov. 2023, https://w3techs.com/technologies/details/pl-php.

[17] B. A. Azad, R. Jahanshahi, C. Tsoukaladelis, M. Egele, and N. Nikiforakis, “AnimateDead: Debloating web applications using concolic execution,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, USA, Aug. 2023.

[18] 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 (Oakland)*, San Francisco, CA, USA, May 2023.

[19] W3Techs, “Usage statistics and market share of WordPress,” Nov. 2023, https://w3techs.com/technologies/details/cm-wordpress.

[20] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, “Symsan: Time and space efficient concolic execution via dynamic data-flow analysis,” in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2022.

[21] “Instrumenting program source code for s2e,” Nov. 2023, https://s2e.systems/docs/Tutorials/BasicLinuxSymbex/SourceCode.html/.

[22] “Navex,” Nov. 2023, https://github.com/aalhuz/navex.

[23] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in dynamic web applications,” in *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, Jul. 2008.

[24] ——, “Finding bugs in web applications using dynamic test generation and explicit-state model checking,” *IEEE Transactions on Software Engineering*, 2010.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, USA, May 2014.

[26] 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.

[27] Nikic, “A PHP parser written in PHP,” Nov. 2023, https://github.com/nikic/PHP-Parser.

[28] A. Chen, J. Lee, B. Chaulagain, Y. Kwon, and K. H. Lee, “Synthdb: Synthesizing database via program analysis for security testing of web applications,” in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, Feb. 2023.

[29] “Mysql,” Nov. 2023, https://www.mysql.com/.

[30] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM: A practical concolic execution engine tailored for hybrid fuzzing,” in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[31] “Reflected xss,” Nov. 2023, https://owasp.org/www-community/attacks/xss/#reflected-xss-attacks.

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

[33] “Codiad,” Nov. 2023, https://github.com/Codiad/Codiad.

[34] “Mediawiki,” Nov. 2023, https://www.mediawiki.org/wiki/MediaWiki.

[35] “Wordpress,” Nov. 2023, https://wordpress.com/.

[36] “Phpbb,” Nov. 2023, https://www.phpbb.com/.

[37] “Drupal,” Nov. 2023, https://drupal.org/.

[38] “Openemr,” Nov. 2023, https://www.open-emr.org/.

[39] “Docker container for navex,” Nov. 2023, https://github.com/aalhuz/navex/issues/3.

[40] “Code is not complete,” Nov. 2023, https://github.com/aalhuz/navex/issues/6.

[41] “Animatedead,” Nov. 2023, https://github.com/silverfoxy/distributed_animate_dead.

[42] 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 (Oakland)*, San Francisco, CA, USA, May 2021.

[43] 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.

[44] “Php-src,” Nov. 2023, https://github.com/php/php-src/tree/master/tests.

[45] “Xdebug,” Nov. 2023, https://xdebug.org/.

[46] B. A. Azad, P. Laperdrix, and N. Nikiforakis, “Less is more: Quantifying the security benefits of debloating web applications.” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, USA, Aug. 2019.

[47] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, “Evaluating and improving hybrid fuzzing,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 2023.

[48] 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 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, USA, Aug. 2024.

[49] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Indianapolis, IN, USA, Oct. 2013.

[50] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *18th International Symposium (SAS)*, Venice, Italy, Sep. 2011.

[51] S. Park, D. Kim, S. Jana, and S. Son, "Fugio: Automatic exploit generation for php object injection vulnerabilities," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, USA, Aug. 2022.

[52] 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.

[53] 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.

[54] "Php fuzzer," Nov. 2023, https://github.com/nikic/PHP-Fuzzer.

[55] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M.-R. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[56] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, USA, Mar. 2014.

[57] 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 (Oakland)*, Oakland, CA, USA, May 2006.

[58] 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.

[59] 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.

[60] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Kyoto, Japan, Nov. 2015.

[61] "Clang: a c language family frontend for llvm," Nov. 2023, https://clang.llvm.org/.

[62] "Wllvm," Nov. 2023, https://github.com/travitch/whole-program-llvm.

[63] "Klee issue 678," Nov. 2023, https://github.com/klee/klee/issues/678.

[64] "Support latest kernel and compilers," Nov. 2023, https://github.com/sslab-gatech/qsym/issues/4.

[65] "Cve-2020-36243 detail," Nov. 2023, https://nvd.nist.gov/vuln/detail/CVE-2020-36243.

Appendix A. Experience in Attempting Other Engines

We have tried other options including KLEE [10], QSYM [30], SymCC [11], and SymSan [20]. Though they were reported to be effective, we failed to successfully apply them to any recent versions of the PHP interpreters. In particular, KLEE [10] requires first compiling the target program into LLVM bitcode. We leveraged clang compiler [61] and WLLVM [62] to produce the LLVM bitcode. However, the bitcode could not be successfully analyzed by KLEE due to intrinsic instructions such as saturating arithmetic intrinsic instructions. We followed the instructions of KLEE and attempted to manually add wrapper functions for some intrinsic instructions [63]. We ultimately had to abandon it due to the large number of unsupported intrinsic instructions. QSYM [30] also did not work in our analysis due to the unresolved dependency of a newer version of PIN [64]. Its code repository on GitHub has been archived and is no longer maintained since March 2023.

SymCC [11] and SymSan [20] are two recent compilation-based concolic execution engines for C/C++ programs. However, the two engines are not robust according to our experience. Specifically, we first used their compilation component (*i.e.*, enhanced clang compiler) to compile the PHP interpreter. Unfortunately, their compilation component raised several unexpected exceptions. Our attempts to fix the errors were unsuccessful after weeks of engineering.

Appendix B. A False Negative of XSym

Listing 4 shows a missed vulnerability of XSym. The application issues a system command to delete existing records in a database table. The command injection vulnerability occurs when the improperly sanitized user data flows to the system command (line 21). The application only uses `add_escape_custom()` function to prevent SQL injection at line 10. However, it does not fully protect the application against command injection. XSym could not identify the command injection because it does not correctly handle the `mysqli_real_escape_string()` function that performs sanitization.

```
1  <?php
2  function add_escape_custom($s) {
3      //prepare for safe mysql insertion
4      $s = mysqli_real_escape_string($GLOBALS['dbh'], $s);
5      return $s;
6  }
7
8  foreach ($_POST['form_sel_layouts'] as $layoutid) {
9      if (IS_WINDOWS) {
10         $cmd .= " echo DELETE FROM layout_options WHERE form_id
       ↪   = '" . add_escape_custom($layoutid) . "'; >> " .
       ↪   escapeshellarg($EXPORT_FILE) . " & ";
11     }
12     else {
13         $cmd .= "echo \"DELETE FROM layout_options WHERE
       ↪   form_id = '" . add_escape_custom($layoutid) . "';\"
       ↪   >> " . escapeshellarg($EXPORT_FILE) . ";";
14     }
15 }
16
17 exec($cmd);
```

Listing 4: CVE-2020-36243 [65] in OpenEMR 5.0.2.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper presents a novel method to utilize mature symbolic execution engines for low-level languages to handle higher-level languages like PHP, without the need for manual modeling of functions. With their new technique called Symbolic Interpreter Analysis (SIA), the authors enhance the way how web applications based on interpreted languages are tested. Since web app code is handled by the interpreter, they symbolically analyze the interpreter code to indirectly analyze the application. This new approach is implemented in their tool SymPHP, a concolic execution engine for PHP-based web applications. They compare their tool against other tools as well as use it together with fuzzing to find new vulnerabilities.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

C.3. Reasons for Acceptance

1) The work presents a different approach to an already established field by using the analysis of language interpreter as an execution engine for symbolic analysis of web applications, which resolves several issues.
2) The method and tool reduce the required manual efforts, making it easier for both industry and academia to use and build on. Multiple future tracks can be built on this as they plan to make it publicly available.
3) They showed that their tool can be used to discover new vulnerabilities in PHP Web applications.

C.4. Noteworthy Concerns

The comparison was only done with tools with different technological backgrounds. Also, Witcher was used to generate the input links for SymPHP, which naturally results in SymPHP outperforming it.