On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution

Penghui Li
Chinese University of Hong Kong
phli@cse.cuhk.edu.hk

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

Kangjie Lu
University of Minnesota
kjlu@umn.edu

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

ABSTRACT

Modeling language-specific built-in functions is essential for symbolic execution. Since built-in functions tend to be complicated and are typically implemented in low-level languages, a common strategy is to manually translate them into SMT solver specifications for constraint solving. Such translation requires an excessive amount of human efforts and deep understandings of the function behaviors. Incorrect translation can invalidate the results and bring security problems, *e.g.,* a true positive case is missed. This problem aggravates in PHP applications because the built-in functions are written in C, but the rest functions are in PHP, *i.e.,* the cross-language nature.

In this paper, we explore the feasibility of automating the process of modeling PHP built-in functions. We synthesize C programs by transforming the constraint solving task in PHP symbolic execution into a C-compliant format and integrating them with C implementations of the built-in functions. We apply symbolic execution against the synthesized C program to find a feasible path, which gives a solution that can be applied to the original PHP constraints. This way, we automate the modeling of built-in functions in PHP applications. We thoroughly compare our automated method with the state-of-the-art manual modeling tool. The evaluation results demonstrate that our automated method is more accurate with a higher coverage, and can exploit a similar number of vulnerabilities. Our empirical analysis also shows that the manual and automated methods have different strengths, which complement each other in certain scenarios. Therefore, the best practice is to combine both of them to optimize the accuracy, correctness, and coverage for symbolic execution.

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

PHP; Constraint solving; Symbolic execution

ACM Reference Format:

Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2021. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *Proceedings of The Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 13 pages. https://doi.org/xxx

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN xxx.
https://doi.org/xxx

1 INTRODUCTION

Web applications have been one of the primary channels connecting service providers and hundreds of millions of end-users. Because of their importance, these applications and their valuable users have been the primary targets of cyber attacks. A study [35] has reported that 64% of companies were said to have experienced web-based attacks. A recent report in 2018 [37] also shows that among 43 popular web applications, each web application on average contained 33 vulnerabilities, and the number of those critical vulnerabilities per web application grew by three times compared to the year of 2017.

Due to the popularity and critical uses of web applications, the detection of their vulnerabilities has been an active and important research topic in the past decades. In particular, symbolic execution uses multiple symbolized inputs to test certain program properties, and is proven to be effective in detecting vulnerabilities and testing their exploitability with the advances in constraint-satisfiability theory [12]. It has been applied to multiple program analysis tasks in the OS kernel, distributed systems, *etc.* In web applications, many works are using symbolic execution to detect SQL injection (SQLi), cross-site scripting (XSS), remote code execution (RCE) vulnerabilities [2].

A general challenge in symbolic execution is handling language-specific built-in functions. Such built-in functions are commonly used to provide basic operations like string processing and number operations. Therefore, a correct understanding of the function semantics and overall program logic requires the analysis of built-in functions. However, to generate concrete solutions to determine the reachability and exploitability further needs to precisely model their behaviors for constraints solving. As a common strategy, prior works model a small number of built-in functions into SMT-LIB specifications [36] for constraint solving and ignore the other ones [2]. First, such a modeling process is expensive, as it typically requires an excessive amount of human effort and domain knowledge of the function behaviors. Second, manual models can be error-prone, and lead to false results (both false positives and false negatives) of their client applications. For example, incorrect modeling can bring soundness problems that a true positive case can be classified as a negative [8, 42].

Unlike some languages (*e.g.,* C) whose built-in (library) functions are written using the same language, PHP, as a dynamic language, however, implements its built-in function in a static language—C. Such a cross-language nature poses many challenges for precisely modeling these built-in functions in symbolic execution. For example, the language feature inconsistency (*e.g.,* operators and type systems). Some operators in one language might not exist in the other, making it hard to understand the behaviors of built-in functions with such operators.

In this paper, we aim to explore the feasibility of automatically modeling built-in function in PHP symbolic execution. We also compare it with manual modeling and shed light on future modeling of built-in functions. Automated modeling of built-in function faces several challenges. First, to the best of our knowledge, there exists no such an automated tool for modeling PHP built-in function yet. The cross-language nature renders the modeling very hard. Second, it is hard to achieve a high *coverage*. There are a large number of built-in functions in a programming language, with different function definitions. An automated method shall be capable of many of these built-in functions. Third, it is difficult to achieve an acceptable *correctness*. A built-in function can be designed to behave for several tasks. Under different arguments, the results can be different. The consequences of inaccurate modeling can invalidate the results of their applications.

We target PHP, the most popular server-side programming language for web applications [38]. To make use of the C implementations of PHP built-in functions, we propose a cross-language program synthesis method to automate the built-in function modeling. We convert the constraint solving task in PHP symbolic execution into a C program and integrate the C implementations of built-in functions. We then employ a C symbolic execution to solve. We map the PHP syntax (*e.g.,* PHP built-in functions) in the task into their original implementations, and overcome the challenge of language feature inconsistency. Because the synthesized C program retains the semantics of the original constraint solving task, the solutions can be applied back to the PHP symbolic execution and the PHP program. This way, we achieve the goal of automatically modeling PHP built-in functions.

We implemented our methodology into XSym. We successfully applied it to automatically model 287 (87.90%) PHP built-in functions. We demonstrate that the models can accurately represent the internal semantics of the built-in functions and achieve similar performance as the ones modeled by humans. Furthermore, with the help of XSym, we can exploit 141 vulnerabilities in the testing dataset. We also pass the constraints collected in PHP symbolic execution to a state-of-the-art manual modeling tool, which exploits 133 vulnerabilities. XSym also exploits 13 vulnerabilities that cannot be exploited by the manual modeling tool. Our manual verification shows that the manual modeling tool produces wrong results for 27 cases, while XSym has only two. This supports that manual modeling can be error-prone and can cause soundness bugs.

We further thoroughly characterize our automated modeling with the manual modeling for PHP built-in functions. Compared with manual methods that are *modeled on demand* and *specialized*, our automated method achieves a higher *coverage* and *correctness*. It can be easily applied to most of in-scope built-in functions and achieve high performance. To maintain an acceptable level of complexity, manual methods, and our automated method complement each other. Besides, we find that our automated method can even be used as a means for verifying the correctness of manual methods.

In summary, we make the following contributions in this work.

- **A new cross-language model.** We propose a cross-language program synthesis method for PHP applications. We explore the feasibility of automated modeling of the PHP built-in functions for PHP symbolic execution. To this end, we propose multiple new techniques such as type inference and cross-language syntax mapping.
- **Extensive evaluation.** We demonstrate that our automated method can achieve a high function coverage and correctness. It is also practical in exploiting vulnerabilities—We can also exploit 141 vulnerabilities in 26 real-world applications, with fewer false-positive reports and false-negative reports.
- **Thorough characterization.** We summarize the characteristics of manual and automated models. We provide insightful suggestions to guide future modeling of built-in functions.

2 BACKGROUND

2.1 PHP

PHP has been the most popular server-side programming language, used by 79% of the websites today [38]. It is important to detect vulnerabilities in PHP applications. However, vulnerability detection for PHP applications can be challenging for some characteristics of PHP. In particular, PHP is a dynamically- and weakly-typed language, *i.e.,* developers do not need to declare a variable before use and can change its type implicitly. More importantly, to improve performance, PHP code often calls built-in functions implemented in C, which can affect the results of bug detection.

2.1.1 PHP Symbolic Execution. Symbolic execution is a widely used program analysis technique that helps determine the inputs to guide control and data flows. It runs a program with symbolized inputs to check whether certain program properties can be satisfied or violated [5], and has been used in PHP program analysis for various tasks, such as bug detection and exploit generation.

PHP programs are interpreted by a framework—the Zend engine [41]—into Zend bytecode at runtime. The Facebook HHVM [30] is a virtual machine that could convert PHP code into its HipHop bytecode through a just-in-time compiler and had stopped supporting PHP since version 4 [23]. To the best of our knowledge, there is no existing tool that can directly perform symbolic execution above the PHP Zend bytecode or HHVM bytecode. Existing symbolic execution frameworks operating on LLVM IR [9–11] cannot be used for PHP, either, because there exists no tool that can fully compile PHP into LLVM IR. Therefore, previous works design their own customized intermediate languages for their PHP symbolic execution. For example, Torpedo [29] builds its own control flow graphs (CFGs) above the abstract syntax trees (ASTs) generated from PHP-Parser [28]; Navex [2] uses PHP code property graphs (CPGs) in PHP Joern [4] that are initially designed for C programs [43].

During PHP symbolic execution, path constraints, which stand for the conditions of input values that lead to specific locations, are collected to help determine the feasibility of particular execution paths or bugs. By solving path constraints with satisfiability modulo theory (SMT) solvers, a decision can be given to know if a path can be taken or a bug can be triggered. However, SMT solvers cannot directly interpret and understand a class of functions—the uninterpreted functions (*e.g.,* PHP built-in functions). These functions commonly appear in constraint formulas. The semantics of solver-uninterpreted functions, directly stop a solver from producing a correct solution for any constraint formula that contains them.

2.1.2 PHP Built-in Functions. Similar to other programming languages, PHP has a large number of internal (built-in) functions.

The usage of PHP built-in functions. To understand if PHP built-in functions are widely used, we conducted a preliminary study to count the number of built-in functions appearing in the source code. We downloaded the top 1000 highly-rated (by stars) projects on GitHub and traversed the ASTs for built-in function invocation statements. We were able to parse only 987 projects with PHP-Parser [28], with around 401K PHP files and 56M lines of code. We found that there were a total of 1.7M built-in functions among the 987 projects; on average, there were around 1710 built-in functions per project, and one built-in function every 33 lines of code. The results show that built-in functions are very common in practice.

The need of modeling PHP built-in functions. Most of the PHP built-in functions are solver-uninterpreted, and thus not supported directly by the solver. This poses a serious problem for symbolically executing PHP programs and generating tests/exploits. As explained in §2.1.1, the commonly used solver-uninterpreted PHP built-in functions stop us from exploring more paths and finding bugs. Previous works on modeling the solver-uninterpreted functions can be divided into two classes: 1) manually modeling functions based on their definitions and descriptions [2]; 2) concretely executing a function with selective inputs or runtime values [22]. The first method requires to check and understand specific functions, and then *"translates"* the functions to a solver-understandable definition. This requires an excessive amount of human effort and domain knowledge. Because of that, this method can only be applied to a limited number of solver-uninterpreted functions. The second method can create an imprecise model of built-in functions by collecting multiple input-output pairs. However, it does not scale as it can cover only very few concrete inputs. Therefore, an automated and accurate approach to modeling the PHP built-in functions is necessary.

2.2 Satisfiability Modulo Theories

Satisfiability is the basic and ubiquitous problem of determining if a formula expressing a constraint has a model or a solution [18]. Many problems can be described in terms of satisfiability, including puzzles, program verification, exploit generation, *etc.* It is also a key component in symbolic execution.

2.2.1 SMT Solvers. SMT solvers check the satisfiability of first-order logic formulas from various theories [18, 40, 42] such as booleans, bit-vectors, strings *etc.*, and are under continuous development. For instance, Z3 [17], a state-of-the-art SMT solver, has more than 5K stars on GitHub.

The SMT-LIB language is the current standard input language for SMT solvers [36]. Formulas in SMT-LIB format are now accepted by the majority of current SMT solvers [7]. SMT-LIB supports basic arithmetic and logic operations. Similar to other programming languages, the SMT-LIB language also defines several statements to perform operations. For example, `declare-fun` and `declare-variable` can be used to declare functions and variables; `assert` statement can specify constraints. Note that the operations are specified in the prefix notation and nested operations in layers are also allowed. For instance, `(assert (< var (* var 2)))` specifies the constraint of `(var > (var * 2))`. Besides, multiple assertions can be viewed as the conjunction of the constraints in individual `assert` statements.

The SMT solvers can output three decisions for input formulas: 1) SAT if satisfiable, 2) UNSAT if unsatisfiable under any circumstances, and 3) UNKNOWN if they are not able to decide its satisfiability. For the formulas receiving a satisfiable decision, models (solutions) that satisfy the constraints with concrete value for each defined variable can be provided using `get-model` statement.

2.2.2 SMT Built-in Functions. To facilitate specifying constraints into SMT-LIB compatible formulas, SMT solvers provide several necessary built-in functions. However, the number of such SMT built-in functions is limited and their functionalities are restricted. For example, Z3 only provides around 40 built-in functions, some advanced functionalities (*e.g.,* string splitting) are not directly supported with its built-in functions. The analysts thus may have to implement such advanced features on their own. Prior works make use of SMT built-in functions to model PHP built-in functions for constraints solving [1, 2].

3 UNDERSTANDING MANUAL MODELING

To understand how manual models are constructed, we study the models of Navex, a state-of-the-art tool that equips with manual built-in function models [2]. We try to summarize and generalize some common practices in them. These characteristics also apply to many other manual modeling approaches to certain extent.

Ignoring for compatibility. For compatibility reasons, Navex chooses to ignore those unsupported built-in functions. This relaxes the entire constraints to certain extent. This is a common trade-off design practice that enables the tool to study more cases, but, consequently, can bring side-effects such as wrong satisfiable decisions and wrong solutions. For instance, if the function `f()` in constraint formula `f($x) == 1` is unsupported, the ignoring of `f()` regards the term `f($x)` as unconstrained, resulting in `$x` as unconstrained as well. The SMT solver will give sample solutions for the unconstrained variable `$x`, *e.g.,* 0.

Functionality simplification. A built-in function is designed to perform certain tasks. However, supporting the entire behavior of a built-in function in SMT-LIB specifications is non-trivial. Manual modeling might choose to perform *functionality simplification* to cover only a part of the entire functionality of a built-in function. Such a design might potentially cause wrong results, but can also bring the benefits of reducing the complexity of constraints as only part of the function is considered. As an example we study, the PHP built-in function `rtrim(str, [optional] character_mask)` by definition strips white space (or other characters in `character_mask`) from the right of a string. In Navex, its model can be used to query the argument given a return value in certain situations. However, the reverse operation, *i.e.,* querying the return value when certain arguments are given, is not modeled. In detail, given a reverse constraint as `$ret=rtrim("testcase\t\n")`, to query the `$ret`, line 1-3 first declare three variables (`str`, `ret`, `var1`) as `Strings`; the assertion

```
1 (declare-variable str String)
2 (declare-variable ret String)
3 (declare-variable var1 String)
4 (assert (= str "testcase\t\n"))
5 (assert (= var1 str))
6 (assert (= ret var1))
```

Listing 1: SMT-LIB specifications of constraint `$ret=rtrim("testcase\t\n")` by Navex.

in line 4 forces `str` to be equal to the input string `"testcase\t\n"`. Thus, the SMT solver does not actually strip the white space, and gives a wrong solution of `$ret="testcase\t\n"`.

4 PROBLEM STATEMENT

4.1 Research Problem and Research Goals

Symbolic execution requires to understand the behaviors of built-in functions for constraint solving. As introduced earlier, built-in functions are common in PHP applications but are hard to model. The current program-analysis tools normally ignore such built-in functions or support only a small number of built-in functions through manual modeling. Manual methods usually take an excessive amount of human efforts and require deep understandings of the function behaviors to accurately model. They can also be error-prone.

In this work, we first aim to explore the possibility of automating the modeling of built-in functions for symbolic execution. In particular, we focus on the PHP programming language, which is the most popular server-side language. Second, we hope to systematically evaluate the automated models with the state-of-the-art tool, in terms of the function coverage, accuracy, and applications. Third, we want to summarize the lessons we learn in our exploration of an automated method and provide some insightful suggestions to shed light on future works in modeling built-in functions.

4.2 Research Challenges

We face several challenges to automatically model PHP built-in functions for symbolic execution. First, modeling built-in functions requires to understand the behaviors of them. To automate such process, we need to find a way to understand the behaviors of built-in functions without human efforts, which is technically difficult. To the best of our knowledge, this problem has not been well studied yet, as there currently exists no such a tool. Besides, due to the large number of built-in functions, it is hard to achieve a high *coverage*. Prior manual works thus choose to only model those most frequently used ones and ignore the rest. Furthermore, it is also hard to achieve a high *correctness*. Each function may contain diverse functionalities, under different input arguments, different features can be thus enabled or disabled. To achieve a high accuracy means to thoroughly support the entire functionality of the function.

5 METHODOLOGY

5.1 Overview

We explore the feasibility of automated built-in function (written in C) modeling for PHP symbolic execution with a high *coverage* and *correctness*. As the implementations of PHP built-in functions are

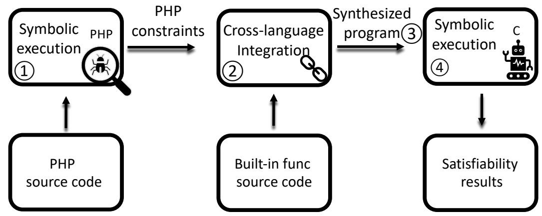

Figure 1: The workflow of overall methodology.

available in the PHP interpreter, we can perform symbolic execution on them to understand their behaviors and automate the modeling. However, PHP and C, are two inherently different languages with different language features. It is hard to seamlessly integrate two languages and the two symbolic execution engines for them. Due to the complexity and dynamic nature of PHP, translating the whole PHP language system to another static language is hard or even infeasible [23]. Thus we cannot simply convert the whole PHP application into a C program and employ C symbolic execution. The low-level C implementations of all built-in functions can hardly be converted into a high-level language, PHP, either.

Therefore, we propose to convert only the results of PHP symbolic execution (*e.g.,* constraints) into a C program that equally describes the task in the C symbolic execution. Unlike the whole language system translation, the constraint solving task can be transformed because it contains only a subset of the whole PHP dynamic language features. We then integrate the C program with the implementations of PHP built-in functions. We call the integrated result *synthesized program*. Afterwards, we leverage C symbolic execution on the synthesized C program for constraint solving. Because the synthesized C program retains the original constraint solving task, the solutions generated by the C symbolic execution thus can equally be applied to the original PHP constraints. Relying on the C symbolic execution, we can achieve the automated built-in function modeling.

For the example in Listing 2, to exploit the XSS vulnerability in line 8, instead of the whole 10 lines of the PHP program, we only need to convert the task of finding a possible solution for the control flow constraints, *e.g.,* `strtolower('PHPBB_' . $_GET['uname']) == 'PHPBB_root' && $_GET['passwd'] == 'mypassword'` and the data flow constraints *e.g.,* `$_GET['ann'] == "alert('XXS')"` collected in PHP symbolic execution. As an example, the synthesized C program is illustrated in Listing 3. Apart from the variable declarations in line 5-7, line 8-9 describe the control flow and data flow in the PHP constraints, where the PHP operators are replaced with the corresponding functions. The C symbolic execution can be directed to the C implementations to analyze and model these PHP built-in functions. We also synthesize an assertion in line 10 so that when the C symbolic execution attempts to find the assertion error, the conditions in line 8-9 are satisfiable, and a set of value assignments to the variables (*e.g.,* `_GET_uname`) can be provided. The solutions are also applicable to the original PHP constraints and PHP applications.

Challenges. There are several technical challenges. First, the language inconsistency between PHP and C makes the constraint solving task conversion difficult. PHP is a weakly-typed programming language that can initialize and use variables with assignments, while C, a statically-typed language, requires variable declarations

```
1  <?php
2  $user = 'PHPBB_' . $_GET['uname'];
3  $password = $_GET['passwd'];
4  $announcement = $_GET['ann'];
5
6  if(strtolower($user) == 'PHPBB_root') {
7      if($password == 'mypassword') {
8          echo $announcement; // XSS
9      }
10 }
```

Listing 2: An XSS vulnerability for demonstration.

```
// include built-in function
#include "php-built-in.h"

int syn_pro() {
    php_string _GET_uname; // to symbolize
    php_string _GET_passwd; //to symbolize
    php_string _GET_ann; //to symbolize
    if(php_is_equal(strtolower(php_concat("PHPBB_",
     _GET_uname)), "PHPBB_root") && php_is_equal(
     _GET_passwd, "mypasswd")) { //control flow
        if(php_is_equal(_GET_ann, "aleart('XSS')")) {
     // data flow
            assert(0); // synthesized error
        }
    }
}
```

Listing 3: The synthesized program for code in Listing 2.

before use. Such type information is thus missing in the PHP code and the constraints. Besides, the constraint solving task contains many PHP operators (*e.g.,* the concatenation in line 8 of Listing 2) that are not defined in C. Second, PHP built-in functions are embedded inside the PHP interpreter and interact with other modules through complicated PHP internal APIs. Identifying such APIs and isolating built-in functions out are challenging. Third, synthesizing a C program to guarantee the PHP constraint solving task is accurately preserved is hard.

We overcome these challenges with a cross-language program synthesis method. The workflow of the overall methodology is presented in Figure 1. To the best of our knowledge, there does not exist a well maintained open-source symbolic execution framework for PHP applications. Also as stated in §2.1.1, many prior works design their own symbolic execution on their custom intermediate formats. Thus we first design a PHP symbolic execution in §5.2. We propose a type inference algorithm to infer the variable types, and a *light-weight syntax mapping* to handle other PHP language features (*e.g.,* operators and built-in functions) in §5.3. We seamlessly convert the constraint solving task and synthesize C programs in §5.4. Last, we construct symbolic inputs and leverage a C symbolic execution engine for the synthesized C program in §5.5.

5.2 PHP Symbolic Execution

As stated earlier, there is no open-source framework or standard intermediate format for PHP symbolic execution. We thus take a similar approach as common practices [2, 29] do to perform symbolic execution over our custom CFGs constructed from ASTs. Our symbolic execution follows the common practices [2, 29] to collect relevant constraints for constraint solving and further verification. We create symbolic variables for the inputs, and walk through the CFGs for constraint collections. Next, we briefly describe the key techniques in our symbolic execution.

5.2.1 Memory Space Management. Symbolic execution simulates the real execution with symbolic inputs. In our design, we treat all values that are not concrete or cannot be directly interpreted as symbolic variables. We design dedicated data structures to manage the symbolic variables and the operations above them (*e.g.,* logical operations). The whole memory space of both symbolic variables and concrete values are managed in a global array, `Memory`, which maintains the pairs of keys and corresponding values. The keys are viewed as addresses of variables. We organize a dictionary, `Keydict`, to store the mappings from PHP variables to keys in `Memory`. For assignments that initialize new variables, we first create a mapping of the variable name (left side value) and an unused key in `Keydict`, and then store the variable value (right side value) to the corresponding location in `Memory` pointed by the key. Variable value fetches are conducted in a similar but reverse direction. However, we cannot support the cases when the variable name cannot be interpreted given the information provided in `Memory`, *e.g.,* `$$x = 1;` where `$x` is a symbolic variable already, thus `$$x` cannot be determined statically. It is a generic challenge of PHP analysis [15], thus we currently do not handle it.

5.2.2 Constraint Collection. The CFGs we construct contain mainly two types of nodes: conditional nodes (*e.g.,* `if` statements), and non-conditional nodes (*e.g.,* assignment statements). Conditional nodes usually include the boolean conditions as the prerequisites to execute the statements in the following branches (*e.g.,* the statements in the body of an `if` branch). We collect the conditions in the conditional statements, and interpret them as parts of the path constraints by applying the symbolic values stored in `Memory` to all variables in them. In non-conditional nodes, we update the `Keydict` and `Memory` accordingly. The control flow constraints in the conditional statements determine the reachability of a particular location. Data flow constraints, which usually integrate the values of critical variables in `Memory` for the sinks with some attack payloads, can be also collected when required. Once reaching interesting code locations (*e.g.,* sinks), relevant path constraints are outputted. For example, to study the exploitability of an XSS vulnerability in a simple echo statement `echo $x;`, the value of `$x` in `Memory` is collected for crafting attack input to launch the attacks [2, 29].

The output constraints describe the paths to specific program locations. The PHP user-defined functions have been analyzed and expanded before integrating into the constraints, therefore, the constraints we collect in PHP symbolic execution can be represented in Equation 1.

$$\begin{aligned} Term \quad t &:= c \mid v \mid f(t) \\ Formula \quad F &:= true \mid false \mid t_1 \; op \; t_2 \mid F_1 \; op \; F_2 \end{aligned} \tag{1}$$

The simplest formula is a term, which can be either a constant value (c), a symbolic variable (v), or a PHP built-in function ($f(t)$). A formula can be further extended by performing a logical or arithmetic operation with another formula to generate a new formula. The formula system belongs to the standard first-order theory of equality with uninterpreted functions [33], which is applicable for the state-of-the-art solvers *e.g.,* Z3 [17].

5.2.3 Path Forking. Conditional nodes always lead to different branches and paths. Thus we need to perform path forking. To collect constraints in different paths, we always break into the branches without considering the satisfiability of conditions at that moment. For path forking, we simply create a duplicate memory space (`Memory`) for the path to be executed next. The memory space is then garbage-collected after finishing exploring that path. The loops statements are only unrolled once and treated as `if` statements, which removes many paths that shall appear in dynamic symbolic execution tools. Such a path forking strategy is simple; some advanced methods can be applied to optimize the forking process [6].

Summary. Our symbolic execution is generic and can be applied for several tasks. In the example of Listing 2, it forks at the two `if` statements and explores four paths in total. It reasons about the sources of the values in the conditions, and these variables appear in the conditions are replaced with the values in `Memory`. Therefore, the control flow constrains (`strtolower('PHPBB_' . $_GET['uname']) == 'PHPBB_root' && $_GET['passwd'] == 'mypassword'`) are collected. For the data flow, the critical variable `$announcement` in the `echo` statement is combined with additional attack payloads to constitute the final data flow constraints *e.g.,* (`$_GET['ann'] == "alert('XXS')"`).

5.3 Cross-Language Integration

The PHP symbolic execution outputs the PHP constraints that are collected for certain analysis tasks. The constraints inherent some PHP language features that are not present in C. In particular, there are two issues we need to tackle: *1) Lack of type information;* and *2) Syntax inconsistency.* First, PHP is a dynamically typed programming language, where variables are initialized by assignment statements. However, C, a typical static programming language, requires all the variables and functions to be clearly declared before use. To synthesize a C program, we have to infer the variable types and declare them explicitly in the C code. Second, PHP and C are two completely different languages that define different operators and built-in functions. The operators in one language might not appear in another language. For example, PHP has the `identical` operator (*i.e.,* ===) that compares both the types and values of operands, which C does not naturally support.

We propose a type inference algorithm to fill the lack of type information, and a light-weight syntax mapping to overcome the syntax inconsistency. With these techniques, we then synthesize a C program that equally represents the PHP constraints.

5.3.1 Type Inference. We perform type inference in the constraints to accurately determine variable types. Our algorithm is based on the fact that, although PHP variables can change their types through the execution, in a specific path, the variable types are determined by the execution context. The constraints we collect from PHP symbolic execution just describe such a context that limits the types of variables. Our type inference algorithm starts by collecting an initial set of types based on the operators and function signatures. Then, it employs an iterative algorithm to infer types of remaining variables.

Initial type inference based on operations. The overall constraints represent the execution context of the variables, thus limit the legitimate types for variables. Locally, the operator behaves as the context for its operands. Thus the types of one operand can often be inferred based on the other operand or the corresponding operators. Besides, the function signatures describe the types of arguments and return values of their call sites. Based on the observation, we first decide operand types based on operators and the arguments and return value types of call sites based on the definitions. Certain operand types are only applicable to specific operators. For example, the addition operator (+) requires the operands to be numbers. Therefore, we identify its operands as of numeric type in the constraints. Similarly, the result of a concatenation operation (`.`) needs to be a string in PHP. Any variables that we cannot obtain their types from the first step are not restricted within the local operator context.

Second, after the first step, we consider the comparative operators such as ==. We perform an overestimation that the operands are of the same type if allowed. This is sensible because: 1) these operands are free of the local context, and adding additional type information for variables with undetermined types (in the first step) does not invalidate the correctness of the syntax; and 2) a comparison usually targets variables of the same type in most of the uses. Therefore, we target a list of comparative operators and identify the types of their operands accordingly.

We apply the two-step procedure to each operation. We put the operand variables whose types are already inferred in the first step into a list, `L`. For those operators satisfied in the second step, we put the operand variables into a corresponding individual *type set*, which we will join through the following steps.

Iterative type propagation. We perform an iterative type inference by propagating the variables with known types in `L` to the remaining unknown variables in the type sets. Since the variables in one type set have the same type, we can infer the types of all other variables in the set if the type of one variable is already known. Therefore, for each variable with inferred type in `L`, we pop it from the list and propagate its type to other variables in the type sets that contain this variable. We also add the new variables which we just identify the types into the list `L`. We repeat this process till the list `L` is empty. In case there are any variables whose types cannot be inferred, we set a default type of *string* as it is the most commonly used type in PHP programs, and continue the propagation process.

Using the control flow constraints in Listing 2 as an illustration example, in the constraints (`strtolower('PHPBB_' . $_GET['uname']) == 'PHPBB_root' && $_GET['passwd'] == 'mypassword'`), because of the concatenation operator, `$_GET['uname']` is inferred as string type. Also from the function definition of `strtolower()`, its return value is inferred in string type as well. Because of the usage of equality operator (==) in `$_GET['passwd'] == 'mypassword'`, we obtain that `$_GET['passwd']` and string `'mypassword'` need to take the same type in the constraints, so we put them in a type set [`$_GET['passwd']`, `'mypassword'`]. Accordingly, the type of `$_GET['passwd']` is inferred as string type finally.

5.3.2 Syntax Mapping. We perform a light-weight syntax mapping to map the PHP operators in PHP constraints into their C implementations in the PHP interpreter. We consider the PHP operators, PHP type systems, and PHP built-in functions that appear in our constraint formula system Equation 1.

PHP defines over 100 operators, including many advanced operators for facilitating server-side scripting. For example, a three-way operator spaceship (*i.e.,* `<=>`) in PHP can perform greater than, less than, and equal comparisons between two operands. However, only less than 40 operators are defined in C. Thus we cannot simply map an operator in PHP to the one in C or vice versa. To address the first inconsistency, we alternatively choose to map all PHP operators into their original C implementations.

The PHP interpreter provides macro definitions for each specific operator and implements corresponding operator handlers. We add wrappers to allow calling these functions from external C programs. For example, the equality operator (*i.e.,* `==`) is defined with a macro `IS_EQUAL`. Thus we define a wrapper function `php_is_equal(arg1, arg2)` that takes two arguments. Similar rule also applies to concatenation operation (`php_concat(arg1, arg2)`), and all other PHP operators. Because there are explicit macros and signatures for these functions, we can make it fully automated and scale to all PHP operators.

Besides the operators, we also do a similar type definition mapping, *i.e.,* one PHP type can be directed to its original definition. We investigate the code parser of PHP interpreter and study how the initialized variables are represented in their C source code. We find that there is a general prototype data structure, `pval`, that is the overall carrier for most types of variable values. The different specifications of the fields in the `pval` can carry different types of variable values. For example, by specifying the `type` field to `IS_STRING`, we can use `strvalue` and `len` fields for strings. We thus wrap them into C language structures and allow directly declaring variable explicitly with these types, *e.g.,* we define a type wrapper `php_string` over `pval` to allow declaring a PHP string type variable in C.

To include PHP built-in function into the analysis scope, we need to clean the implementations of built-in functions from complicated inner APIs inside the PHP interpreter. Some functions use explicit ways to pass the arguments in their C implementations, *e.g.,* `struct pval* is_int(struct pval)`, which can be easily handled. However, some functions do not accept arguments directly. Instead, they are provided with only the address of a hash table, which stores the real PHP arguments. For example, the PHP built-in function `strtolower()` has the function signature of `strtolower(INTERNAL_FUNCTION_PARAMETERS)`. The macro of `INTERNAL_FUNCTION_PARAMETERS` takes a pointer of hash tables to pass argument values. The special argument-fetching design requires complex computation for obtaining and parsing the arguments in the hash table in built-in functions. To tackle this, we use another approach by allocating memory on the heap or the stack and passing the address as the hash table address for them. This is feasible because there are internal type-conversion functions in the PHP interpreter that can be leveraged to transform the data in memory into the anticipated argument types. Therefore, we apply such type-conversion functions to the allocated memory to convert the type to the anticipated type. With this, we can analyze the stand-alone behaviors of these PHP built-in functions.

5.4 Synthesizing C Programs

We synthesize a C program that equally represents the semantics of the PHP constraints and directly executes the C implementations of those PHP built-in functions. There are mainly three steps to synthesize a C program for our purpose.

First, we need to declare variable types before use. Based on the type inference, we obtain the exact types of PHP variables in the PHP constraints. Since we already map the PHP type systems into their implementations and add wrappers for them, we can explicitly declare the necessary variables. For the synthesized C program (`syn_proc()`) in Listing 3, lines 5-7 declare three variables as `php_string`. Note that an array element in the superglobal `$_GET` of PHP is transformed as a simple variable, *e.g.,* `$_GET['uname']` turns to be `_GET_uname`. Second, we replace all the PHP operators with their wrappers above their C implementations by putting the operands as the arguments of the wrapper functions, *e.g.,* `php_concat()` and `php_is_equal()`. Last, we construct the overall logic and finalize the C program synthesis, *i.e.,* we include the C implementations of PHP built-in functions (line 2), represent the control flow and data flow constraints into the conditions of `if` statements (line 8-9), and synthesize an error that can be triggered when the conditions are met (line 10).

The type inference and type system mapping guarantee the correctness of basic C syntax. With the operator and built-in function mapping, the `if` statements in the synthesized C program can retain their functionalities as in PHP constraints. Once the synthesized error is triggered, the conditions in the `if` statements are definitely satisfied. In other words, the synthesized C program can equally represent the corresponding PHP constraints.

5.5 Symbolic Execution on Synthesized Programs

We perform symbolic execution on the synthesized C program. In this work, we use KLEE [10], a state-of-the-art and popular dynamic symbolic execution engine for LLVM IR. We first compile the synthesized C program together with the C implementations of PHP built-in functions into LLVM IR. We use the primitives of KLEE (*e.g.,* `klee_make_symbolic()`) to declare variables as symbolic inputs, *e.g.,* `_GET_uname`. After that, we can symbolically execute the synthesized C program and invoke PHP built-in functions from their C implementations. The symbolic execution on the synthesized C program can determine the satisfiability of PHP constraints by searching a path to reach the error we synthesize in the code. Taking advantages of the searching heuristic inside the symbolic execution, we turn the PHP constraint solving problem into a path searching problem. Thus we can automate the process of built-in function modeling for PHP symbolic execution.

Similar to directly using SMT-LIB on constraints with manual models, the C symbolic execution is capable of giving a solution to the synthesized C program if it can find a satisfiable path to reach the error; an unsatisfiable decision will be given if the condition can never be satisfied; otherwise, the symbolic execution will keep running until it reaches the timeout. Because the synthesized C program are equally transformed from PHP constraints, the solutions can be naturally applied to the original PHP constraints and PHP programs.

6 IMPLEMENTATION

We implemented our PHP symbolic execution engine on top of the PHP-Parser [28] with about 7K LoC in PHP. We use the PHP-Parser to parse PHP source code into ASTs, and then construct control-flow graphs. To synthesize the C program, we automated the wrapper constructions with 2K LoC in Python, and modified the PHP interpreter (v3.0.18) with 1.2K LoC in C. We modified KLEE [10] for analyzing the synthesized program with about 500 LoC in C++.

We integrate the synthesized program with the C implementations in PHP interpreter v3.0.18. We did not select the latest version of the PHP interpreter because KLEE is not able to well support the intrinsic functions in recent versions of PHP interpreters. In particular, the PHP interpreter had been re-engineered significantly. The compiled LLVM IR code of the latest versions includes a lot of architecture-dependent intrinsic functions that KLEE does not support. We do notice that the newer versions have introduced some but not many new functions. However, we observe that the basic definitions of most PHP operators, types, and built-in functions remain the same across PHP version updates. Therefore, we believe targeting a relatively older version of PHP is reasonable. Our methodology shall work for the newer versions of PHP as long as KLEE includes support for those intrinsic functions. Nevertheless, as we will demonstrate next, working on this version of PHP already allows us to achieve good performance.

7 EVALUATION

In this section, we evaluate XSym in the three aspects: 1) *coverage*, 2) *correctness*, and *application*. First, one major goal of XSym is to automatically model PHP built-in functions. We measure how many built-in functions can be supported with our approach. Second, the correctness of the built-in function models is a key factor for ensuring the effectiveness of applications using them. We investigate how accurate our models are. Third, we study if XSym can help develop better symbolic execution applications, *e.g.,* exploit generation.

We first apply XSym to model PHP built-in functions, and evaluate the function coverage in §7.2 and the correctness in §7.3. Next, in §7.4, we demonstrate the efficacy of XSym in exploiting real-world applications. Last, we characterize manual and automated methods in §7.5.

7.1 Experimental Setup

We specified XSym to use Z3 SMT solver, and configured a 10-GB maximum memory usage and a 5-hour timeout. We conduct all the experiments on a server running Debian Stretch (Linux Kernel 4.9.0) with 96 GB RAM, and four 2.1 GHz Intel Xeon E5-2695 CPUs.

We systematically compare XSym with the state-of-the-art PHP symbolic execution tool—Navex [2]. Though Navex had been open-sourced, unfortunately, the source code (for bug detection and constraint collection) is incomplete and no longer maintained. Our attempts failed to reach the authors. For a fair comparison, we could only use its constraint solving component—which is independent and includes their function models—for solving the same PHP constraints collected by XSym. We evaluate the tools on the dataset used in [2]. It includes 1) popular and complex PHP applications such as Joomla, HotCRP, and WordPress, and 2) the same applications tested by other state-of-the-art tools in exploit generation (*e.g.,* Chainsaw [1]) and vulnerability analysis (*e.g.,* RIPS [15]).

7.2 Coverage

7.2.1 *In-scope Built-in Functions.*

XSym requires the source code of a program, and cannot model a function if not all its code is available. A common scenario is that a built-in function relies on some external modules of which the code is unavailable. For example, function `imap_check()` checks information from a mailbox, and relies on the external mail service. The version of PHP we use includes a total of 923 built-in functions, of which 603 rely on external modules. So we finally select 320 built-in functions for evaluation. We emphasize that, as shown in [16], this set has covered more than 90% of the most popular functions.

7.2.2 *Coverage for Built-in Functions.*

XSym has a high coverage. The evaluation results show XSym is able to automatically model 287 (89.70%) functions. In contrast, Navex modeled only 35 (10.93%) built-in functions in their released source code. XSym fails to support the rest functions for the following reasons: 1) implementation issues and 2) implicit dependency issues. First, our current implementation of XSym has some limitations inherited from KLEE. For example, KLEE does not support float point numbers and assembly code that are used in some built-in functions, such as the ones for mathematical calculations (*e.g.,* `sin()`). Second, some functions have implicit dependencies with other functions, and require others to be called first (not internally called). For instance, function `get_magic_quotes_gpc()` gets the current configuration setting of `magic_quotes_gpc` in global variables which must be provided by an earlier function call. Since these prerequisite functions are not internally called, XSym currently cannot identify such implicit dependency. 4 cases not supported by XSym fall in this category.

7.3 Correctness

7.3.1 *Evaluation Method.*

We evaluate the correctness of each individual built-in function. We separately synthesize PHP constraints and C program for each built-in function, and then checking whether XSym can produce correct solutions for that function to evaluate its correctness. In detail, for each built-in function $f(t)$, we put the PHP constraint formula (`f($t) == $ret`) into the condition of an `if` statement, and similarly synthesize an error in the `if` body. To evaluate the PHP constrain formula (`strtolower("TESTCASE") == $sol`), we synthesize the C program in Listing 4. We try to symbolize either the arguments (`$t`) or the return variable (`$ret`), and query KLEE to solve. In the example, the `$sol` is to be symbolized and to be solved. KLEE might generate solutions for the symbolized variables in the constraints. A test would *pass* if the solution is correct; or *fail* if KLEE is not able to give a solution or the solution is incorrect. A constraint formula can have multiple solutions. This is because different arguments can result in the same return value for some functions, *e.g.,* the formula (`strlen($str) == 1`) can have many possible values for `$str`. We separately execute the function concretely with the KLEE provided solutions and compare the concrete return values.

To pave the ground truth of correctness, we leverage the test suite shipped with the PHP interpreter, which include the expected

```c
// include built-in function
#include "php-built-in.h"

int syn_pro() {
    php_string sol; //symbolic return value
    if(php_is_equal(strtolower("TESTCASE"), sol) ) {
            assert(0); // synthesized error
    }
}
```

Listing 4: A synthesized C program for evaluating correctness.

Table 1: Statistics of function accuracy.

Func. Types	# Func	# Tests	# Passed	Proportion
String	47	296	254	85.81%
Arithmetic	21	98	82	83.67%
Others	20	120	63	65.00%

return values for executing built-in functions with the provided concrete arguments. For the functions in category `others`, because the PHP interpreter does not include test cases for many of them, we select the ones with test cases in our evaluation.

7.3.2 Results. The evaluation results are shown in Table 1. Since string-related and arithmetical functions are the most prevalent in symbolic execution, we divided these 287 functions into three classes: string-related functions, arithmetical functions, and the others. We observe that XSym has a reasonably high accuracy. It passed 85.81% of the tests for string-related functions and 83.67% of the tests for arithmetical functions. It also passed 65.00% of tests in the `others` category. The results suggest that our automatic approach can correctly model the behavior of many built-in functions.

XSym did not pass certain tests for the following reason. Symbolic execution cannot cover all paths for complex functions because of path explosion. Therefore, we cannot pass some test cases if the provided inputs traverse the paths not explored in symbolic execution. This is the inherent limitation of symbolic execution; however, this can be mitigated through dynamic state merging [26].

Comparing with the state-of-the-art. We also evaluate the correctness of manual models using the same method described above. Among 25 out of 35 PHP built-in functions that Navex manually supports, XSym outperformed Navex by passing 48 more test cases.

For the rest functions, they are not directly compared because of the nature of functions and the different versions of PHP the two tools modeled. In detail, four functions produce non-deterministic results. Thus, it is impossible to verify the correctness. Further, six functions were added to PHP since v5.4, which are not included in the version of PHP for which we automatically modeled.

Summary. We have two findings in the evaluation part. First, to certain extent, modeling PHP built-in functions is a process of translating their behavior defined in one language to another language that is understandable by the solver. We find that some functions cannot be easily supported even by experts, because of the language feature inconsistency. Second, our analysis demonstrates the automated modeling of built-in functions can be much more accurate, compared to the manual modeling.

7.4 Vulnerability Detection

To understand how our automated models help with security applications, we apply XSym for the detection of SQL injection and cross-site scripting vulnerabilities, which are the dominating types of severe threats to server-side applications. We first perform a standard static taint analysis to identify the vulnerabilities, then use XSym with Z3 to validate and exploit the vulnerabilities. We also ask Navex to solve the same set of constraints for comparison. The SMT solver may directly output an UNKNOWN decision for a constraint. We also set the output as UNKNOWN if the tool is unable to produce a decision within the time limit. Note that we do not investigate vulnerabilities depending on client-side code and the multiple-step nature of web applications, as they have been thoroughly studied in Navex [2] and are orthogonal to our work.

7.4.1 Overall Results. The evaluation results are shown in Table 2. As presented in the column `Sinks`, the taint analysis marked 172 SQLi and 139 XSS cases in 18 out of 26 applications. The results of the constraint solving for each tool are shown in the columns `Sol` and `Rep` in Table 2. A case is solvable if the SMT solver can give either a SAT or an UNSAT decision within the time limit; otherwise, it is unsolvable. XSym solved 110 out of the 172 SQLi cases, and 95 out of the 139 XSS cases; and Navex solved 120 SQLi cases and 105 XSS cases. In our experiment, Navex triggered some syntax errors that violated SMT-LIB specifications while analyzing 15 cases, and reported them as unsolvable cases.

A case is considered as a positive by a tool if its constraint receives a SAT solution. In summary, XSym identified 81/62 positive SQLi/XSS cases, and Navex reported 84/72 positive SQLi/XSS cases. Navex solved more constraints and reported more positive cases, which result from its overestimation and oversimplification of the constraint formulas (see §3). However, as we demonstrate next, Navex has a quite high number of false positives.

To evaluate the correctness of the constraint solving results, we manually analyzed all the vulnerable cases found in taint analysis and tested the solutions given by each tool. We present the true positive cases in the columns `TP`, and denote the false positive cases in parentheses of columns `Rep` in Table 2. Both tools have false positives (FP) and false negatives (FN): XSym has 1/1 FP SQLi/XSS case, while Navex has 11/12 FP SQLi/XSS cases; XSym has 7/6 FN SQLi/XSS cases, while Navex has 14/8 FN SQLi/XSS cases.

7.4.2 Analysis. To clearly understand the capability of each tool, we depict the distribution of results in Figure 2. We highlight the number of constraints including built-in functions supported by only XSym as the first number in each parenthesis. Overall, 99 (31.83%) out of 311 sinks, and 66 (24.53%) out of 265 solvable constraints include such XSym-only built-in functions. This demonstrates the support for more built-in functions is well needed. We next study different situations in detail.

Solvable and unsolvable cases. The majority of cases (`A` - `D`) are solvable by both XSym and Navex. However, some cases (*e.g.,* eight in `D`, 13 in `J`) contain XSym-only built-in functions. Navex could "solve" such cases, because it ignores those functions that it does not support by treating them as free symbols for compatibility reason.

There exist many cases that are solvable by only one tool (*e.g.,* Navex could not solve cases in `E`, `F`, and `I` but XSym could), and

Table 2: Statistics of vulnerability detection for SQLi and XSS. Sinks, Sol, Rep, and TP denote the number of sinks (taint analysis), solvable cases (including satisfiable and unsatisfiable), tool reported cases, and manually confirmed (true positive) cases, respectively. The subscripts X and N denote the results of XSym and Navex. The numbers in parenthesis mean false positives.

App	Files	LoC	SQLi						XSS					
			Sinks	Sol_X	Sol_N	Rep_X	Rep_N	TP	Sinks	Sol_X	Sol_N	Rep_X	Rep_N	TP
myBloggie (2.1.4)	56	9090	25	12	12	7	7	7	2	2	2	1	1	2
WebChess (0.9)	29	5219	13	6	8	4	7 (4)	5	16	11	12	8	9	8
WordPress (4.7.4)	699	181257	0	0	0	0	0	0	0	0	0	0	0	0
HotCRP (2.1.00)	145	57717	0	0	0	0	0	0	0	0	0	0	0	0
SchoolMate (1.5.4)	63	15375	50	33	36	25	29 (4)	28	11	8	9	6	8 (2)	7
HotCRP (2.6.0)	43	14870	0	0	0	0	0	0	4	4	4	2	3 (1)	2
Zen-Cart (1.5.5)	1010	109896	0	0	0	0	0	0	0	0	0	0	0	0
Geccbblite (0.1)	11	323	4	3	3	3	2	3	0	0	0	0	0	0
OpenConf (6.71)	134	21108	0	0	0	0	0	0	0	0	0	0	0	0
osCommerce (2.3.3)	541	49378	1	1	1	1	0	1	47	28	33	18	21 (2)	21
osCommerce (2.3.4)	684	63631	0	0	0	0	0	0	5	3	3	2	2	2
Drupal (8.3.2)	8626	585094	0	0	0	0	0	0	0	0	0	0	0	0
WeBid (0.5.4)	300	65302	43	38	39	29 (1)	26 (2)	30	13	10	8	4	4	5
Gallery (3.0.9)	510	39218	0	0	0	0	0	0	0	0	0	0	0	0
Scarf Beta	19	978	0	0	0	0	0	0	3	2	2	2	2	2
DNScript	60	1322	2	1	2	1	1	1	1	1	1	1	1	1
Joomla (3.7.0)	2764	302701	0	0	0	0	0	0	0	0	0	0	0	0
FAQForge (1.3.2)	17	1676	17	5	8	3	4	4	7	4	4	3	3	3
LimeSurvey (3.1.1)	3217	965164	0	0	0	0	0	0	0	0	0	0	0	0
Collabtive (3.1)	836	172564	0	0	0	0	0	0	0	0	0	0	0	0
Eve (1.0)	8	905	5	2	3	2	2	2	2	2	2	2	2	2
Elgg (2.3.5)	3201	215870	0	0	0	0	0	0	0	0	0	0	0	0
CPG (1.5.46)	359	305245	3	2	3	2	2	2	11	7	9	5	5 (1)	6
MediaWiki (1.30.0)	3680	537913	0	0	0	0	0	0	1	0	1	0	0	0
PHPBB (3.0.11)	74	29164	4	3	3	3	3 (1)	3	16	13	16	8 (1)	11 (6)	7
PHPBB (3.0.23)	387	158756	5	4	4	1	1	1	0	0	0	0	0	0
Total	27473	3909736	172	110	120	81 (1)	84 (11)	87	139	95	105	62 (1)	72 (12)	68

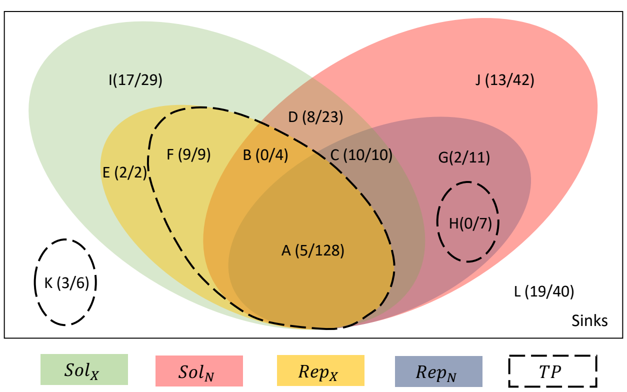

Figure 2: Distribution of vulnerability detection results. The alphabets (A-L) denote different situations. The numbers in parenthesis denote (number of cases including built-in functions supported by only XSym/ total number of cases).

even ones that neither tools can solve (*e.g.,* the six in K). On the one hand, XSym, as a general symbolic execution tool, suffers from path explosion problem as it turns the constraint solving problem into a path search problem. Thus, it cannot generate a solution within the time limit if one constraint is very complex. On the other hand, Navex cannot solve some complex cases as well, because the SMT solvers intrinsically suffer from also the excessively high computation complexity [19].

True positives (TP) and true negatives (TN). 128 cases in A were provided with a SAT solution by both tools. However, two solutions given by Navex were incorrect, and thus the corresponding two vulnerabilities were not really exploitable using the incorrect solutions. This is caused by the *functionality simplification* in Navex. Nine TP cases in F included XSym-only functions and were solvable by only XSym. Seven TP cases in H did not contain XSym-only functions, but were solvable only by Navex. We find that these seven constraints involved complex multiple-layer calls of built-in functions. Navex's functionality simplification could reduce the complexity to some extent. In contrast, XSym aimed to cover all the functionalities, and particularly suffered from the complexity problem.

XSym and Navex reported the same TNs in D, but also different ones in I, and J, respectively. Especially, XSym determined 10 TNs in C, which were wrongly classified as positives by Navex. All these 10 cases included XSym-only functions, which XSym was able to correctly model their functionalities and accordingly generated the correct UNSAT decisions. In contrast, Navex incorrectly relaxed the constraint because of its ignoring for compatibility, and generated incorrect SAT decisions.

False positives (FP). XSym had two FPs in E. We found in our analysis that these two constraints included calls of *uninterpreted* user-defined functions, of which the PHP source code was not available. Accordingly, XSym had to replace them with free symbols for generating solutions, which were wrong. This is a well-known challenge in PHP program analysis [2, 4], but not a limitation of our

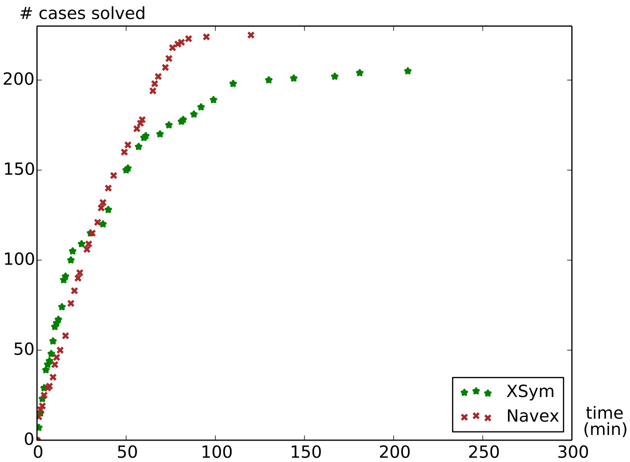

Figure 3: Number of solved cases over time for XSym and Navex

automated modeling approach. Navex had 23 FPs. The reasons for the 12 FPs in A and C have been discussed above. The other 11 FPs in G were also caused by its *ignoring for compatibility*. The results suggest that Navex could have generated more accurate results if more built-in functions were supported.

False negatives (FN). As explained earlier, XSym had six FNs in K and seven FNs in H that it could not solve. Similarly, Navex had 15 FNs in K and F that it could not solve. Particularly, four cases in B that Navex solved were FNs, because of the *functionality simplification* in its models. For example, Navex only modeled a subset of the entire functionalities of certain built-in functions, while the satisfiable functionalities were not included. Therefore, the underlining SMT solver was unable to satisfy the whole constraints. On the contrary, XSym was able to generate the correct exploits because of its correct and complete modeling of these functions.

Summary. XSym and Navex can solve the majority of reported cases, and exploit most true positive cases. Compared with Navex, XSym has a lower false-positive rate and false-negative rate. Navex, because of the *ignoring for compatibility* and *functionality simplifications*, produces wrong decisions or solutions for 27 cases, while XSym only has two.

7.4.3 Performance. As we already specified the maximum memory usage to 10 GB for both tools, here we only measure the time usage in them. Specifically, for all solvable cases, we present the numbers of solved cases over time in Figure 3. From the figure, we observe that, more than half of the cases were solved in the first 100 minutes, and no more cases were solved after 210 minutes. This suggests that the timeout of 5 hours is sufficient to evaluate both tools in our settings. As a comparison, XSym has a relatively longer time requirement than Navex. Again, higher efficiency with Navex is a result of its manual over-approximation, which however sacrifices accuracy.

7.5 Characterizing Manual and Automated Modeling

A manual method is usually *modeled on demand* and *specialized*. It focuses on a relatively small set of built-in functions that are usually required for certain analysis tasks. Due to the unaffordable manual effort and significant complicated logic, manual modeling typically cannot cover all built-in functions or provides accurate results. That said, we found that a manual method can specialize the features to meet their needs like functionality simplification. However, this can bring side-effects, for example, the wrong solutions. It also makes use of the SMT-LIB built-in functions to assist their implementations. By contrast, our automated method is with a high *coverage*, *correctness*, and *completeness*. Our automated method can scale to a large number of built-in functions and manage an acceptable accuracy. Our automated method considers the entire function semantics, and is more complete, however, compared with the manual models using *functionality simplification*, it results in a higher analysis complexity. Therefore, only the shadow (paths) functionalities can be explored and modeled.

We further find that manual and automated methods can complement each other. For those frequently-invoked built-in functions, manual methods can be leveraged to specialize them with best efforts for the analysis tasks. For other relatively less used built-in functions, our automated model can scale to support them with basic functionalities. We suggest to further combine them together.

Our automated modeling can be used as a possible means to verify the correctness of manual methods. As discussed earlier, our manual method has a high accuracy and high true positive reports, and it can solve most of the cases that the manual method can solve. Therefore, it can be a feasible way to help verify the decisions and solutions given by the manual methods.

8 DISCUSSION

Built-in functions dependent on external modules. Our automated method requires all the code to be available. Built-in functions that rely on external modules such as operating systems and database systems can not be directly supported when the code of the external modules are not in our analysis scope. We can also try to solve this problem by including the code of external modules in our analysis scope. For example, S2E [14] adopts the whole-system symbolic execution to cover all involved modules. However, potential challenges in whole-system symbolic execution include that it has to analyze binary code (when the source code is not available) and that it may not scale well. We believe that the approach of XSym is generic, and supporting external modules is an orthogonal topic.

Combining automated and manual method. Our methodology uses a C symbolic execution tool, KLEE, to analyze the synthesized C program. To integrate the manual modeling into our automated modeling, we may have to redirect the function calls to such manual modeled functions to their manual models, and seamlessly join them into their execution context in the C symbolic execution. We leave it as our future work.

9 RELATED WORK

Symbolic execution. Symbolic execution has been widely used for web security. SAFELI [20] performs symbolic execution against the instrumented bytecode of Java-based web applications for SQLi. Kudzu [34], a JavaScript symbolic execution, studies the client-side vulnerabilities. Differently,XSym targets server-side PHP applications, Apollo [3] and Navex [2] use concolic execution for test generation. They either instrument the Zend engine or use xdebug to get runtime information for constraint solving. Compared with XSym, they remain relying on analysts to translate PHP built-in functions into SMT-LIB specifications, which is found to be error-prone.

Program Synthesis. Program synthesis as the task of generating programs from user intent, has been widely used for studying security problems. Singularity [39] transforms the complexity testing problem to optimal program synthesis to identify performance bugs. Aspire [13] synthesizes application specifications from input-output examples to meet user intent and to guarantee the security. Many fuzzing works [24, 27, 31] use the language syntax to synthesize code fragments as test cases. However, XSym applies a program synthesis method to construct automated models for PHP built-in functions.

Modeling of built-in functions. Analyzing built-in functions is also common in static analysis. Pixy [25] configures 29, and RIPS [15] classifies and analyzes over 900 built-in functions in static analysis for taint propagation and sanitization. In symbolic execution and test generation, SMART [21] proposes a summary to describe the behaviors of a function, but it only targets C applications instead of PHP applications that involve cross-language features. DART [32] isolates library functions from the whole constraints and concretely executes these library functions to mitigate the built-in function problem for Java programs; Godefroid records concrete input-output pairs for uninterpreted built-in functions and reuses them in constraint solving [22]. However, they can only cover a very few function situations. Tools like Chainsaw [1] and Navex [2] that manually model built-in functions are shown to be error-prone. In comparison, XSym stands for using symbolic execution on C implementations of PHP built-in functions to automatically model their behaviors.

10 CONCLUSION

Modeling PHP built-in function in symbolic execution is important. In this paper, we explored the feasibility of automatically modeling PHP built-in functions for PHP symbolic execution. We proposed a cross-language program synthesis method that transforms relevant constraints collected in PHP symbolic execution into a C-compliant program and integrates with the C implementations of PHP built-in functions. We then leveraged a C symbolic execution to analyze the synthesized program, which achieved the goal of automating built-in function modeling. Our evaluation shows that the automated method is scalable and accurate. With it, we successfully exploited 141 vulnerabilities in 26 real-world web applications. We believe that our study would offer new opportunities for modeling built-in functions.

11 ACKNOWLEDGMENT

We thank ..

REFERENCES

[1] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.
[2] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.
[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA)*. Seattle, WA.
[4] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the 2nd IEEE Symposium on Security and Privacy (Oakland)*. Paris, France.
[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* (2018).
[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* (2018).
[7] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*.
[8] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. 2006. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in SMT. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer.
[9] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running symbolic execution forever. In *Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA)*. Los Angeles, US.
[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. 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.
[11] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security* (2008).
[12] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* (2013).
[13] Kevin Chen, Warren He, Devdatta Akhawe, Vijay D'Silva, Prateek Mittal, and Dawn Song. 2015. ASPIRE: Iterative Specification Synthesis for Security. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*. Kartause Ittingen, Switzerland.
[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. *SIGPLAN Not.* (March 2011).
[15] Johannes Dahse and Thorsten Holz. 2014. 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.
[16] Dams. 2018. Top 100 PHP functions. https://www.exakat.io/top-100-php-functions/.
[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Budapest, Hungary.
[18] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. 2007. A tutorial on satisfiability modulo theories. In *International Conference on Computer Aided Verification*. Berlin, Germany.
[19] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. 2006. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation* (2006).
[20] Xiang Fu and Kai Qian. 2008. SAFELI: SQL injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*.
[21] Patrice Godefroid. 2007. Compositional dynamic test generation. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)*. Nice, France.
[22] Patrice Godefroid. 2011. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA.
[23] HHVM. 2018. Ending PHP Support, and The Future Of Hack. https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html.
[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA.
[25] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2010. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security* (2010).

[26] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* (2012).

[27] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1044–1051.

[28] Nikic. 2020. A PHP parser written in PHP. https://github.com/nikic/PHP-Parser.

[29] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. 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, Colorado.

[30] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, PA.

[31] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[32] Corina S Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*. Toronto, Canada.

[33] G Robinson and Lawrence Wos. 1983. Paramodulation and theorem-proving in first-order theories with equality. In *Automation of Reasoning*. Springer.

[34] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[35] AAG IT Services. 2019. How often do Cyber Attacks occur? https://aag-it.com/how-often-do-cyber-attacks-occur/.

[36] SMT-LIB. 2020. SMT-LIB. http://smtlib.cs.uiowa.edu/.

[37] Positive Technologies. 2019. Web application vulnerabilities: statistics for 2018. https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/.

[38] W3Techs. 2020. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.

[39] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.

[40] Wekipedia. 2020. Satisfiability modulo theories. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories.

[41] Wikipedia. 2020. Zend Engine. https://en.wikipedia.org/wiki/Zend_Engine.

[42] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion.. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK.

[43] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.