Detecting Correctness, Security, and Performance Bugs in Software Systems with Automated Analysis and Testing

LI, Penghui

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

The Chinese University of Hong Kong

July 2023

Thesis Assessment Committee

Professor LYU Rung Tsong Michael (Chair)

Professor MENG Wei (Thesis Supervisor)

Professor LUI Chi Shing John (Committee Member)

Professor ZHANG Chao (External Examiner)

Abstract of thesis entitled:

Detecting Correctness, Security, and Performance Bugs in Software Systems with Automated Analysis and Testing

Submitted by LI, Penghui

for the degree of Doctor of Philosophy

at The Chinese University of Hong Kong in July 2023

Computer systems are becoming increasingly complex and are deployed at massive scales. Bugs routinely occur in production-level computer systems, compromising their reliability and leading to severe consequences. Successful exploitations of software bugs cost over 100 billion US dollars yearly, and the trend has been growing over time. However, manual code auditing to examine software bugs is often unscalable and incomprehensive in the complex digital era. This thesis thus aims to detect software bugs automatically so that software developers can timely fix them.

Modern software systems have three key properties, namely correctness, security, and performance, whereas various types of software bugs often undermine them. In this thesis, we design automated approaches to identifying different categories of software bugs that break these properties, aiming to provide a holistic support for software correctness, security, and performance. We first discover a new category of correctness bugs that produce incorrect results caused by erroneous programming language designs, namely loose comparison bugs. In particular, the dynamic and weakly-typed features of programming languages might be incorrectly used. We develop a novel static code analysis tool, LChecker, to automatically detect loose comparison bugs. LChecker uses taint analysis to identify user-manipulable inputs and incorporates a variable type inference algorithm to locate relevant variables. It solves the scalability challenge in analyzing complex software with a context-sensitive function summary technique. LChecker has found 42 previously unknown bugs in seven million lines of code using an hour.

Second, we design dynamic fuzzing approaches to finding security and performance bugs. We develop a new directed fuzzing system, SDFuzz, to detect security bugs. SDFuzz refines the exploration directions towards security bugs through selective code instrumentation and early execution termination, and quickly exposes security bugs

with significantly less time. Furthermore, we design MdPerfFuzz to expose performance bugs. MdPerfFuzz integrates a grammar-aware input generation technique and a statistical model to precisely label suspicious performance issues. It also incorporates a bug de-duplication technique based on trace similarity. The two fuzzing tools beat prior solutions and have detected four new security bugs and 209 new performance bugs to date.

Third, we advance automated bug detection capability through testing and improving foundational program analysis techniques. In particular, we focus on symbolic execution. We find that the support of internal functions (*e.g.*, common library functions and built-in functions) is the bottleneck of symbolic execution. We thus first test the internal function models with a new differential analysis technique, which compares the behaviors in a symbolic execution function model to those in concrete execution. We use data-flow based static analysis to identify the scope of the models and develop SEDiff. SEDiff has found 46 new bugs in the state-of-the-art symbolic execution engines of C/C++, PHP, and binary programs. Since internal function models need to be constructed manually and are often error-prone, we also seek to automate the support of internal functions. Because the required function semantics of internal functions are usually conveyed in the interpreters of the languages in source code form, we thus perform another layer of symbolic execution on the interpreters to extract the semantics instead of manually constructing models. This solution eliminates the intensive manual efforts and meanwhile ensures correctness. We realize a prototype symbolic execution framework XSym for PHP programs and the solution can be ported to other programs. Experiment results show that XSym outperforms prior manual solutions with better function coverage, accuracy, and exploit generation capability.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Problem Statement

The ubiquitous computing devices and services around us are actuated by software systems such as web applications. Since these software systems are the foundation of computing devices and services, their quality is of great importance. However, modern software systems are complex, with millions of lines of code, opening the door for various bugs, vulnerabilities, exploitations, and attacks. In practice, software bugs routinely occur in production-level software systems and impair their intended functionalities. In 2022, there were over 25 thousand software bugs publicly reported, and the number had been growing yearly [48]. At the same time, software bugs cause vast financial loss, costing the world economy over 100 billion US dollars per year.

Given the complexity of modern software, performing manual code audit to analyze software systems is unscalable and incomprehensive. Automated bug detection approaches to improving software quality are thus desired. Although there were tremendous research efforts in the past to automatically analyze software systems, their techniques mostly could not sufficiently resolve the issues and challenges arising from the rapidly evolving software systems. For instance, new classes of software bugs have overwhelmed the capability of prior bug detection tools. New solutions and system designs are urgently needed to guarantee the functionalities of software systems and computing devices today.

Software systems have three key properties, namely correctness [92, 124, 166, 188],

Table 1.1: Categories of software bugs and thesis contributions.

Category	Consequence	Example	Thesis Contributions	
			Tailored Detection	Generic Approach
Correctness	Incorrect results	Bad type casting	LChecker [105]	SEDiff [106] XSym [107]
Security	Corruptions	Memory-safety bugs	SDFuzz	
Performance	Irresponsiveness	CPU-exhaustion DoS	MdPerfFuzz [104]	

security [12, 64], and performance [86, 113], whereas software bugs often undermine them and break the intended functionalities. The correctness property describes if the software behaves as intended without producing incorrect results; the security property requests additional software attributes such as confidentiality, availability, integrity, *etc.*; the performance property emphasizes the acceptable efficiency such as processing time and service responsiveness. This thesis thus aims to provide *a holistic support* to improve software correctness, security, and performance through detecting and eliminating software bugs. On the one hand, we aim to identify new threats and bugs in today's software systems and design novel techniques to automatically detect them before attackers can exploit. This will allow software developers to timely fix them and reduce the risks. On the other hand, we want to advance foundational program analysis techniques, which will benefit software analysts and the community in the long term.

1.2 Thesis Overview

With the goal of improving software quality through automated bug detection, we first characterize past reported software bugs, as shown in Table 1.1. In particular, there are three typical categories of software bugs, namely correctness bugs, security bugs, and performance bugs, corresponding to the respective properties of software systems. All categories of software bugs can cause severe consequences, such as computing incorrect results, causing system corruptions, and computing resource exhaustion.

This thesis proposes to improve software correctness, security, and performance in three directions: 1) detecting correctness bugs with static analysis, 2) exposing security and performance bugs through fuzzing, and 3) testing and improving foundational program analysis techniques. Since bugs in different categories have distinct features,

the first two directions aim to identify the bugs using tailored approaches based on the domain-specific features. Some generic program analysis techniques have wide application scope for bug detection. The third direction thus advances a representative generic program analysis technique, symbolic execution, through testing and modeling.

In the following sections, we provide an overview of the three directions of our approaches. They incorporate the existing studies in the rich literature and significantly improve them with new observations, insights, and systems.

1.2.1 Detecting Correctness Bugs with Static Analysis

Correctness is the foundation for software systems to behave as expected. Software systems without a correctness guarantee would not be practically used and deployed in the real world. Correctness issues are difficult to detect due to the lack of oracle, *i.e.*, to define and distinguish (in)correct situations. Dynamic analysis solutions such as differential testing [142, 166] often require similar implementation variants of the testing objects, which are not available in the majority of real-world software systems. Therefore, we use static analysis and formalize heuristics to solve this oracle issue. We thereby design LChecker, to detect a representative type of correctness bugs related to bad type casting [99].

1.2.1.1 LChecker: Detecting Loose Comparison Bugs Caused by Bad Type Casting

Type casting (conversion) is a common programming practice but can become *bad*. We find that weakly-typed languages such as PHP support loosely comparing two operands by implicitly converting their types and values. In certain conditions, loose comparisons can cause unexpected results, leading to authentication bypass and other functionality problems. In this thesis, we present the first in-depth study of such loose comparison bugs. We develop LChecker, a system to statically detect PHP loose comparison bugs. It employs a context-sensitive inter-procedural data-flow analysis together with several new techniques. We also enhance the PHP interpreter to help dynamically validate the detected bugs. Our evaluation shows that LChecker can both effectively and efficiently

detect PHP loose comparison bugs with a reasonably low false-positive rate. It also successfully detected all previously known bugs in our evaluation dataset with no false negatives. Using LChecker, we discovered 42 new loose comparison bugs and were assigned nine new CVE IDs.

1.2.2 Exposing Security and Performance Bugs through Fuzzing

Besides correctness bugs, finding security and performance bugs can further protect the users and improve the user experiences. In this part, we apply the effective dynamic approach—fuzz testing—to expose security and performance bugs. We design SDFuzz, a directed fuzzer, to facilitate reproducing and validating vulnerabilities. We also develop MdPerfFuzz [104] to identify performance bugs that consume excessive CPU resources. Both tools have tackled a set of limitations of prior solutions in order to achieve high effectiveness.

1.2.2.1 SDFuzz: Target State Driven Directed Fuzzing for Security Bugs

Fuzzing has shown great promise in finding security bugs such as crashes. Directed fuzzing further optimizes general coverage-guided fuzzing by driving the testing towards highly-valuable target site locations. However, past directed fuzzers still unnecessarily explore program paths that cannot trigger the target vulnerabilities due to their random exploration strategy. Fortunately, we observe that the major application scenarios of directed fuzzing provide detailed vulnerability descriptions, from which highly-valuable program states (*i.e.*, target states) can be derived, *e.g.*, call traces when a vulnerability gets triggered. By driving the testing to expose such target states, directed fuzzers can exclude massive unnecessary exploration.

Inspired by the observation, we present SDFuzz, an effective directed fuzzing system driven by target states. SDFuzz first automatically extracts target states in vulnerability reports and static analysis results. It employs a selective instrumentation technique to reduce the fuzzing scope to the required code for achieving target states. SDFuzz then early terminates the execution of a test case once SDFuzz probes the remaining execution cannot achieve the target states. It further uses a new target state feedback and

refines prior imprecise distance metric into a two-dimensional feedback mechanism to proactively drive the exploration towards the target states. We thoroughly evaluated SDFuzz on known vulnerabilities and compared it to related works. The results show that SDFuzz could improve vulnerability exposure capability with more vulnerability triggered and less time used. Regarding vulnerability exposure time, SDFuzz outperformed the state-of-the-art directed fuzzers, including AFLGo, WindRanger, Beacon, and SieveFuzz, with an average speedup of 4.79×, 4.63×, 1.57×, and 2.58×, respectively. Our ablation study shows that selective instrumentation and execution termination are the dominant contributing factors to the superior performance of SDFuzz. Our application of SDFuzz to automatically validate the static analysis results successfully discovered four new vulnerabilities in well-tested applications. Three of them have been acknowledged by developers.

1.2.2.2 MdPerfFuzz: Grammar-Aware Fuzzing for Performance Bugs

Performance bugs can cause performance degradation and resource exhaustion. Though powerful, fuzzing cannot be directly applied to find performance bugs. This is mainly due to three obstacles, namely input generation, testing oracle, and result duplication. For instance, two inputs might trigger an identical performance bug yet exhibit different program behaviors. In this thesis, we develop MdPerfFuzz, a fuzzing framework with a syntax-tree based mutation strategy to efficiently generate test cases to manifest performance bugs. MdPerfFuzz uses a statistical model to precisely label suspicious performance issues. It also uses an execution trace similarity algorithm to de-duplicate the bug reports.

These techniques are generic and can be utilized for various programs. In our current implementation and evaluation, we target at the compilers of domain-specific languages. In particular, we apply MdPerfFuzz to the compilers of a representative domain-specific language—Markdown. With MdPerfFuzz, we successfully identified 216 new performance bugs in real-world compilers and applications. Our work demonstrates that the performance bugs are a common, severe, yet previously overlooked problem in domain-specific language compilers. We believe MdPerfFuzz can benefit other applications and scenarios in the long term.

1.2.3 Testing and Improving Symbolic Execution

Symbolic execution has become a foundational program analysis technique, powering a large variety of program analysis and bug detection tasks such as automated test generation, exploit synthesis, *etc.* The academia and industry have proposed many symbolic execution engines to facilitate these tasks. However, the effectiveness of existing symbolic execution engines is still questionable. In this part, we not only test but also improve the foundational symbolic execution technique. We identify that the support of internal functions (*e.g.*, PHP built-in functions, C standard library functions) is the performance bottleneck of symbolic execution. We thus first propose SEDiff [106] to test symbolic execution engines and reveal several bugs in the state-of-the-art engines. SEDiff is capable of testing symbolic execution engines of C/C++, PHP, and binary programs. Furthermore, we propose an automated system, XSym [107], to better support internal functions in symbolic execution. We currently realize a prototype implementation for PHP applications but the techniques can naturally be ported to C/C++ and binary programs.

1.2.3.1 SEDiff: Scope-Aware Differential Fuzzing to Test Internal Function Models in Symbolic Execution

Just like concrete execution, performing symbolic execution unavoidably encounters internal functions (*e.g.*, library functions) that provide basic operations such as string processing. Many symbolic execution engines construct internal function models that abstract function behaviors for scalability and compatibility concerns. Due to the high complexity of constructing the models, developers intentionally summarize only partial behaviors of a function, namely modeled functionalities, in the models. The correctness of the internal function models is critical because it would impact all applications of symbolic execution, *e.g.*, bug detection and model checking.

A naive solution to testing the correctness of internal function models is to cross-check whether the behaviors of the models comply with their corresponding original function implementations. However, such a solution would mostly detect overwhelming inconsistencies concerning the unmodeled functionalities, which are out of the

scope of models and are thus considered false reports. We argue that a reasonable testing approach should target only the functionalities that developers intend to model. While being necessary, automatically identifying the modeled functionalities, *i.e.*, the scope, is a significant challenge.

In this thesis, we propose a scope-aware differential testing framework, SEDiff, to tackle this problem. We design a novel algorithm to automatically map the modeled functionalities to the code in the original implementations. SEDiff then applies scope-aware grey-box differential fuzzing to relevant code in the original implementations. It also uses a new scope-aware input generator and a tailored bug checker that efficiently and correctly detect erroneous inconsistencies. We extensively evaluated SEDiff on several popular real-world symbolic execution engines targeting binary, web and kernel. Our manual investigation shows that SEDiff precisely identifies the modeled functionalities and detects 46 new bugs in the internal function models used in the symbolic execution engines.

1.2.3.2 XSym: Automating Internal Function Modeling in Symbolic Execution

Modeling language-specific internal (built-in) functions is essential for symbolic execution. Since internal functions tend to be complex and are typically implemented in low-level languages, a common strategy is to manually translate them into the SMT-LIB language for constraint solving. Such translation requires an excessive amount of human effort and deep understandings of the function behaviors and is often error-prone. This problem aggravates in PHP applications because of their cross-language nature, *i.e.*, the built-in functions are written in C, but the rest code is in PHP.

In this thesis, we explore the feasibility of automating the process of modeling PHP built-in functions for symbolic execution. 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 on the synthesized C program to find a feasible path, which gives a solution that can be applied to the original PHP constraints. In 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 function 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 of symbolic execution.

1.3 Thesis Contributions

In summary, this thesis makes the following contributions.

- **Analysis and formalization of new types of threats.** We identify several new types of software threats, including loose comparison bugs caused by bad type casting, and internal function model bugs.
- **Novel bug detection mechanisms.** We propose novel detection approaches to automatically identifying software bugs spanning different categories.
- **Prototype implementations and open source.** We design and implement prototype systems of our detection approaches and make them publicly available.
- **Findings of real-world bugs.** With the solutions and systems, we have made real-world impacts by identifying over 300 new software bugs in foundational software systems.

□ End of chapter.

Chapter 2

Detecting Correctness Bugs with Static Analysis

Software bugs violating the correctness property cause severe impacts. Since they are typically semantic bugs, automatically detecting them is difficult. Designing a generic oracle or checker supporting all types of correctness bugs is not feasible because there is no general pattern. In this chapter, we make the first step to detect a new yet severe class of correctness bugs—loose comparison bugs—caused by bad type casting. We demonstrate that such bugs are prevalent and our tailored approach could effectively detect them in real-world applications.

2.1 LChecker: Detecting Loose Comparison Bugs Caused by Bad Type Casting

2.1.1 Motivation

Comparison is an essential programming feature. Strongly-typed languages, *e.g.*, Python, perform *strict* comparisons that consider both the values and the types of the comparison operands. In contrast, weakly-typed languages provide the identical and not identical operators (===, !==) for *strict* comparisons, and the equal and not equal operators (==, !=) for *loose* comparisons. Loose comparisons can implicitly convert the operand types, thus allowing for comparing operands in different types.

Loose comparisons are supported in many programming languages, such as PHP, JavaScript, and Perl. In particular, loose comparisons are widely used in PHP programs. At run time, PHP can automatically convert an operand's type and specially interpret its value. This is also known as type juggling [147, 162]. For instance, ("0e12345" == "0e67890") is evaluated as True, because PHP interprets these two string operands as the integer 0 (see §2.1.2.2 for more details), whereas most (if not all) people may intuitively think the result shall be False. Consequently, loose comparisons could sometimes produce unexpected comparison results and break the intended program functionality. We call such loose comparisons that cause unexpected results as *loose comparison bugs*.

Loose comparison bugs can be exploited for malicious purposes and have severe security impacts. Previous reports have shown that attackers can exploit loose comparison bugs to bypass authentication and escalate privilege [128, 176]. The bugs can also be leveraged for executing system commands and overwriting system files [121]. Although prior research has studied some type system bugs, loose comparison bugs and the relevant security issues have not been well studied. TypeDevil [151] and Phantm [95] could detect type inconsistency bugs—a different class of type system bugs—in JavaScript and PHP. Backes *et al.* [16] discussed the possibility of applying their graph traversal based method to *magic hash* bugs, which are one class of loose comparison bugs as we will introduce later (§2.1.2.1). Nevertheless, to the best of our knowledge, there exists no tool that can effectively detect loose comparison bugs.

In this work, we aim to fill the gap by developing approaches to detecting loose comparison bugs in applications developed in PHP—the most popular server-side programming language used by 78.8% of websites [181], and studying their security impacts.

Detecting loose comparison bugs is challenging. First, loose comparison is a language feature and is not a bug unless incorrectly used in the program. It is non-trivial to develop methods to well differentiate incorrectly used loose comparisons from the normal ones. In particular, a specification covering different kinds of misuses is needed. Second, loose comparison bugs belong to general logic bugs as they do not break code properties and semantics. Therefore, it is difficult to find indicators of attacks exploiting them. Moreover, it can be challenging to find such bugs in modern applications, which may have a massive code base with millions of lines of code. Last but not the least, the

dynamic nature of PHP makes the inference of variable types very difficult, which is necessary to determine if a loose comparison can be potentially exploited or not.

To overcome these challenges, we first provide a formal definition of loose comparison bugs, considering both the sources and the types of operands, and excluding the legitimate uses of loose comparisons. We then develop LChecker—a tool specializing in detecting loose comparison bugs. LChecker performs a taint analysis to identify loose comparisons that can be controlled by attackers through malicious inputs. It also employs a type inference algorithm to overcome the challenge of determining the types of variables in dynamic languages such as PHP. LChecker's static analysis is scalable as it employs a context-sensitive function summary in its efficient inter-procedural analysis. The summary can assist in finding nested bugs that only manifest when certain functions are called with special inputs. We also enhance the PHP interpreter to help validate the bugs with minimal human efforts.

We implemented a prototype of LChecker for PHP and will release the source code of our prototype implementation. We thoroughly evaluated LChecker on 26 popular PHP applications, including around 38.7K source files and 7M lines of code. Our evaluation demonstrates that LChecker can *effectively* and *efficiently* detect loose comparison bugs. LChecker successfully identified all eight known bugs, and 42 previously unknown ones with a reasonably low number of false reports, in only 75 minutes. Furthermore, LChecker outperformed the only relevant state-of-the-art analysis tool by detecting 37 more bugs. We reported all new bugs to the relevant vendors and were assigned 9 new CVE IDs.

In summary, this work makes the following contributions:

- We conduct the first systematic study on loose comparison bugs in PHP. We characterize known loose comparison bugs and propose a formal definition of such bugs.
- We design and develop LChecker, a tool to detect loose comparison bugs in PHP.
- With LChecker, we detect and confirm 50 (including 42 previously unknown) loose comparison bugs.

Table 2.1: PHP loose comparison procedures in order of priority.

Operand 1	Operand 2	Computation Procedures
Null/String	String	1) Convert operands to Number if applicable 2) Lexical comparison
Null/Bool	Anything	Convert both operands to Bool
Number	String	Translate String to Number

```php
<?php
/* retrieve userdata from database */
$result = $db->query("SELECT id, passwd FROM members WHERE name = '".$_POST['user']."'");
$userdata = $db->fetch_result($result);

/* password validation */
if ( !$userdata['id'] || md5(stripslashes($_POST['passwd'])) != $userdata['passwd'] ) {
    /* login fails */
} else {
    /* login succeeds */
    $session->update(NULL, $userdata['id']);
}
```

Listing 2.1: A loose comparison bug in CVE-2020-8088.

2.1.2 Problem Statement

2.1.2.1 Real-World Loose Comparison Bugs

Unlike strongly-typed languages, weakly-typed languages employ implicit type conversion to allow comparisons between differently typed operands. PHP follows the procedures listed in Table 2.1 to handle a loose comparison. For example, a string is implicitly converted into a number when compared with a number. Even the operands are of the same type, implicit type conversion can still happen. For example, in ("12" == "10"), the two String operands are actually compared as numbers as in (12 == 10).

However, automatically converting operand types and values in loose comparisons can be problematic. Developers usually use loose comparisons in conditional statements. The comparison strategies can result in strange comparison results that lead to unexpected program behaviors, *e.g.*, an unexpected path is taken. We informally name these unexpected loose comparison problems as *loose comparison bugs*. We will provide a formal definition in §2.1.2.2.

Listing 2.1 shows an example of a loose comparison bug in UseBB (1.0.12) (CVE-2020-8088 [42]) that exploits a *magic hash* bug. The user authentication can be easily

bypassed without correct credentials. In lines 3-4, the user information is first retrieved from the database with a user-provided username ($_POST['user']). Line 7 validates the credential by checking whether 1) the user ID ($userdata [' id ']) is set, and 2) the user-provided password matches the one in the database. The stripslashes () function removes backslashes from a string. The md5() function computes the hash value of a string by generating a 32-character (128-bit) hexadecimal string. The password stored in the database ($userdata [' passwd']) had been hashed previously. If the conditional statement in line 7 is evaluated as False, the program determines that the authentication is successful and takes the else branch (lines 9-11) to update the user session data.

However, an attacker can trick the program into taking the else branch even without providing a correct password because of the loose comparison bug in line 7, if the hashed password string is specially formatted. For instance, if the correct original password is "QLTHNDT", the loose comparisons with the hash values of many other strings, *e.g.*, " PJNPDWY", can be True. This is because the hash values of the two different passwords are the string representations of zero in scientific notation, as shown below:

md5("QLTHNDT")= "0e405967825401955372549139051580"

md5("PJNPDWY")= "0e291529052894702774557631701704"

They both represent numbers like 0×10^n, where the exponent n is a huge integer specified after the letter 'e' in the string. Although the two hash values are in String type, they are automatically converted into the integer 0 in the loose comparison. As a result, the condition in line 7 becomes False and the program takes the else branch as if the provided password is correct, thus granting the attacker the privilege of the victim user.

2.1.2.2 Formal Definition of Loose Comparison Bugs

Loose comparison as a language feature, if inappropriately used, can result in security issues. In detail, we define loose comparison bugs based on the following three conditions.

Cond1: User-controllable loose comparisons. A loose comparison bug should involve data from an untrusted source, *e.g.*, user inputs, so that it can be exploited by an attacker. To form a loose comparison bug, at least one operand of the loose comparison needs to

come from an untrusted source.

Cond2: Exploitable operand types. Only data in certain types can be implicitly converted into another type in loose comparisons, and thus brings about potential security issues. As shown in Table 2.1, Null, Bool, Number or String operands can be converted into a different type (and thus exploited) in loose comparisons. We consider only loose comparisons of which the operands are in these exploitable types as potential bugs.

Cond3: Inconsistent comparison results. Besides the types, the operand values also matter. In the examples we studied in Table 2.1.2.1, the string operands are interpreted as numbers whose conventional string representations (*e.g.*, "0") are different from the literal strings of the operands (*e.g.*, "0e405967 ..."). Consequently, the loose comparison results become unexpected and inconsistent. Loose comparison bugs result from such inconsistent comparison results, which happen when the actual runtime comparison results by comparing the implicitly converted operands are different from the (expected) results by comparing the operand literal values.

2.1.2.3 Research Goals and Research Scope

In this work, we aim to systematically study loose comparison bugs. We aim to develop methods to detect such bugs in PHP applications with a low false-positive rate. We do not, however, aim to detect all loose comparison bugs, *i.e.*, our method is not sound. We aim to also study the main causes and the security impacts of loose comparison bugs in real-world scenarios.

We particularly study three common classes of loose comparison bugs in our work. In each class, an attacker can leverage a type of unexpected values to bypass loose comparison checks. We call the following types of special strings as *magic strings* in this work.

U1: Scientific notation strings. PHP can interpret a scientific notation string as a number. For instance, any string that matches the scientific notation regular expression '0e\d*' is evaluated as an integer zero when being loosely compared with a similar string or a number. We have shown how magic hash bugs use *magic zero strings* to bypass loose comparison checks in Table 2.1.2.1. In addition to magic zeros, scientific notation strings can also be automatically converted into other numbers. In string-to-string

comparisons, implicitly converting scientific notation strings into numeric values is unacceptable in most cases. We believe that, only in rare cases, developers deliberately use such indirect scientific notation string representations for simple numeric comparisons. Thus, scientific notation strings should not be compared loosely in most programs.

U2: Numeric strings. Like scientific notations, other numeric strings can also be interpreted as numbers. As shown in the first row of Table 2.1, numeric strings are converted to numbers for numeric comparisons. This means, if two operands are numeric strings, the loose comparison is actually performed as a numeric comparison. In addition, multiple extra leading zeros can be added to the front of the strings without changing the numeric values. For example, to let ($x == "0.1") be True, $x can be "0.1", and similar strings with many leading zeros, such as "00.1", "000.1", *etc.* Regardless of the number of leading zeros, they are all evaluated as 0.1 in the loose comparison, though they are literally different *strings*.

U3: Numeric-prefix strings. Some string as a whole may not be a valid number. However, if one of its prefixes is a valid numeric string, the prefix is converted into a number for being loosely compared with Number and Bool values [146]. For instance, to let ($x == 0.1) be True, besides 0.1 and "0.1", $x can also be many other strings starting with the prefix "0.1", such as "0.1 xy", "0.1 xyz", *etc.* This is because all these strings are cropped and evaluated as the float number 0.1 in the loose comparison.

The above loose comparison cases are semantically correct. However, they can potentially cause problems because the loose comparison checks can be bypassed in an implicit and unexpected way. Several studies [108, 147, 162] have demonstrated the potential risks caused by such strange/unexpected comparisons.

2.1.2.4 Research Challenges

We face several technical challenges. First, modern applications are very complex and can have millions of lines of code. Loose comparison bugs typically span many functions and are context-sensitive, where they can only be triggered when the operands are within several operand types and are in special values. Performing precise program analysis in huge code bases is naturally challenging.

Second, it is difficult to identify loose comparison bugs with a low false-positive

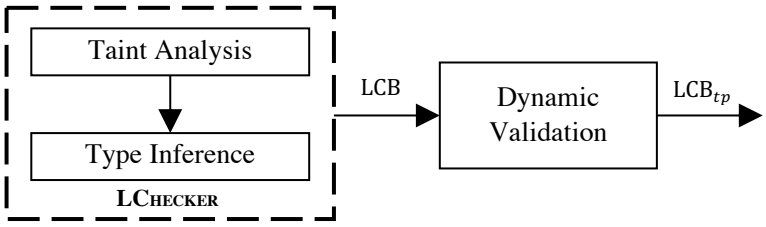


Figure 2.1: The overall methodology of LChecker.

rate. Loose comparison is a language feature and can be exploited only in very special situations. An application may use a massive number of loose comparison operations. Most of the loose comparison operations are not bugs. Not reporting those normal loose comparisons as bugs is challenging.

Last but not the least, the weakly-typed nature of PHP makes the program analysis very challenging. We discuss it in more details in §2.1.3.1 and §2.1.4.

2.1.3 Design of LChecker

We propose an approach to detecting loose comparison bugs in PHP. To the best of our knowledge, it is the first program analysis approach that detects loose comparison bugs in weakly-typed languages. The architecture of our approach is depicted in Figure 2.1. Our approach consists of 1) a static analysis component, LChecker, to identify loose comparison bugs (LCB), and 2) a dynamic analysis to help validate the true positive bugs (LCB$_{tp}$). Specifically, the static analysis part, LChecker, uses taint analysis to identify untrusted user data (Cond1), and tackles the challenge of determining the types of variables in weakly-typed languages with a type inference algorithm (Cond2). LChecker also integrates a context-sensitive function summary to perform an efficient inter-procedural analysis. To reduce the false reports, the dynamic analysis drives an enhanced program execution engine to assist experts to validate true positive bugs (Cond3).

2.1.3.1 Static Analysis

LChecker performs a source-code level static analysis based on the control flow graphs (CFGs) and call graphs (CGs) of a program using PHP-Parser [131]. We develop our own

static analysis because there is no available tool to fully translate PHP into a common intermediate representation for static program analysis due to the dynamic nature of PHP. File inclusions might introduce new code into the analysis scope when constructing the CFGs, thus LChecker uses a fuzzy string match algorithm to identify and include possible files into the current analysis context. This is a common design choice in practice [135, 165]. If the entire inclusion file string is identified literally, LChecker directly includes that file; otherwise, it considers its prefix or suffix and includes all satisfied file candidates. The basic node in the CFGs is formulated in a format consisting of a left operand (*LeftOp*), a right operand (*RightOp*), and an *operator*. The *operands* can represent other nested expression nodes. For instance, in the statement $x = $a + $b, the operator is assignment (=), the *LeftOp* is $x, the *RightOp* is $a + $b, which represents the expression of a plus operation.

Taint Analysis

LChecker performs a standard taint analysis to find loose comparison statements that can be controlled by attackers (Cond1).

LChecker sets as taint sources some untrusted data sources that may expose the PHP program to potential risk. It includes superglobals (*e.g.*, $_GET, $_POST), user file uploads, databases, *etc.*, as taint sources. It also maintains a set of tainted variables for identifying the taint status of variables during the analysis.

LChecker propagates taint from the *RightOp* to the *LeftOp* in assignment-like statements. For example, in the statement $a = $b, if $b is tainted, then $a becomes tainted and thus is put into the tainted variable set. In branch statements, the analysis is first forked and then joined after it. A variable after the branch node is marked as tainted if it gets tainted in any branch. This is reasonable as there actually exists at least one path for attackers to control the variable.

LChecker treats all loose comparison operations as sinks. Generally, if any tainted data reaches one of the operands, the loose comparison operation is tainted. However, we observe that it is not sufficient in practice, because the other untainted operand is usually set with values that do not satisfy loose comparison bugs. Therefore, LChecker reports only the loose comparisons of which both operands are tainted.

Type Inference

LChecker uses a type inference algorithm to check Cond2 and to narrow down the scope of potential bugs.

Different from some prior type inference works [9, 134], we need to consider two new issues in our type analysis. First, in PHP and other dynamically-typed languages (*e.g.*, JavaScript), the values and types of variables are determined only at run time. This is different from some statically-typed languages (*e.g.*, C/C++), whose variables are declared with explicit types before use. Second, a PHP variable is not bound to one type. PHP variables can be overwritten by values in different types through the program execution. Therefore, to perform type inference analysis, LChecker needs to monitor all assignment statements to track variable type changes.

LChecker infers variable type information through data flow analysis. In addition to the taint status, each variable is associated with a type set, which denotes the possible types the variable can be in. We define the following eight basic variable types in our analysis: 1) Null, 2) Bool, 3) Int, 4) Float, 5) String, 6) Object, 7) Array, 8) Mixed. Note that the Numeric type mentioned earlier includes both the Int and the Float types. LChecker then identifies the types of variables and operations with different strategies.

For *scalars and constants*, LChecker literally infer their types, *e.g.*, "1" is in the String type. For *variables*, LChecker directly gets their types from their type sets. This is because, whenever a variable is encountered, it must have been assigned with some value and type before, or have been assigned with other variables. By propagating the types with the data flow, LChecker is able to infer the types of most variables.

Diverse *operators* are frequently used in PHP code. They determine the types of the operation results. For example, in the simple assignment of `$x = 1 . " string "`, the concatenation operator (`.`) concatenates an integer `1` and a string `" string "`. The type of $x would always be String because it is determined by the concatenation operator (`.`). Similarly, the types of results of all other binary operations, unary operations, type casting operations, *etc.* can also be inferred accordingly.

We find that *built-in functions* are usually well documented with explicit specifications. LChecker can thus infer the types of parameters and return value whenever a built-in function is called. However, the parameter types are not always accurate because built-in functions in PHP generally allow users to violate the parameter type

specifications without raising runtime errors. For example, a PHP built-in function, strlen(String):Int, can accept an integer argument (*e.g.*, strlen(1)).

Differently, for *user-defined functions*, LChecker directly analyzes the source code to infer types. LChecker uses a context-sensitive function summary in the inter-procedural analysis to infer return value types. The function summary describes the type relationship between the parameters and the return value. Thus, once the argument types are inferred, the return value type can also be known. The details will be presented in our inter-procedural analysis in §2.1.3.1.

The variable types are transferred from the *RightOp* to the *LeftOp* in assignment-like statements. A conditional statement can lead to multiple program execution paths and a variable can be assigned differently in different branches. Therefore, for each variable, LChecker accumulates all possible types in all branches into its type set after the branches. This may introduce false positives because only one branch can be taken at run time and a variable can only be in one type after that branch. However, it is still acceptable because we observe that most variables are assigned with the same type among different branches. We will further discuss this problem in Table 2.1.5.2.

Context-Sensitive Inter-Procedural Analysis

We need to perform inter-procedural analysis because different call sites can invoke a user-defined function with different argument types and values which can later affect the states of the caller function. LChecker follows the common practices in PHP program analysis to identify the callees in dynamic method calls [16, 65]. Specifically, it searches the method name in the whole program to find a unique match. If several methods have the same method name, it further compares the number of parameters in the method declarations and the number of arguments in the call site.

It is expensive to analyze a function at every call site with explicit arguments, thus LChecker uses a function summary to achieve an efficient inter-procedural analysis for detecting loose comparison bugs. The function summary maintains some necessary information for each function, including function name, class name, *etc.* Whenever a previous unanalyzed user-defined function is called, LChecker constructs the function summary and then applies the function summary to the call site. A function only needs to be analyzed once, then the function summary can be used across the whole analy-

sis among all call sites. This significantly improves the efficiency of inter-procedural analysis. However, there are two main problems we need to tackle for detecting loose comparison bugs.

First, PHP does not require developers to include parameter types in function definitions. The parameter types indeed affect the variable types inside the functions, which ultimately influences our type inference and loose comparison bug detection. Furthermore, different call sites can provide different arguments to invoke the functions. In the example of Listing 2.2, the original parameter of function $\mathrm{f}()$ is directly returned. Line 2 and 3 call $\mathrm{f}()$ with an argument in Array type and String type, respectively. Consequently, $a and $b are expected to be in Array type and String type. Therefore, the function summary should be context-sensitive to model return value types.

Second, loose comparison bugs are context-sensitive. We also need to consider the calling context of a callee in our detection. In Listing 2.2, when constructing the function summary for $\mathrm{f}()$ alone, the loose comparison bug at line 7 is not detected. However, this loose comparison statement can be exploited in some calling contexts. With the inputs like the one in line 3, the loose comparison bug can be exploited because the local variable $x becomes a tainted string. Therefore, the function summary should be context-sensitive so that the bugs in the callee can be detected.

The function summary in LChecker is context-sensitive to support different calling contexts of a function. To achieve this, we add a type placeholder for the parameters ($Param_{idx}$) in the step of type inference, where idx denotes the corresponding index of a parameter. $Param_{idx}$ is propagated in the same way as other types during the type inference. Further, LChecker hooks all return expressions and records the $\mathrm{Returntypes}$ for function return values. In the example of Listing 2.2, $x in line 7 obtains the type $Param_1$ instead of an explicit type (*e.g.*, Array or String). At line 10, LChecker adds $Param_1$ to $\mathrm{Returntypes}$ of the function summary of $\mathrm{f}()$. At the call sites in line 2 and 3, LChecker replaces the type placeholders with explicit argument types to obtain the return value types. Thus $a and $b are inferred as of Array and String types, respectively.

To successfully detect such loose comparison bugs at the call site of $\mathrm{f}()$ at line 3, LChecker records the data sources (*e.g.*, parameters and superglobals) of each variable during the data flow analysis of summary construction. It then includes all loose

```php
<?php
$a = f(array(1, 2)); // pass a constant array
$b = f((string)$_GET['test']); // pass a user string

function f($x) {
    $passwordDB = "0e12345678";
    if($x == $passwordDB) {
        /* login succeeds */
    }
    return $x; // return parameter x
}
```

Listing 2.2: An example of calling a user-defined function in different contexts.

comparisons (sinks) together with the types, values, taint status, data sources of the operands in the function summary. In the example, LChecker includes line 7 ($x == $passwordDB) into its summary. The data source of the *LeftOp* ($x) is the first parameter; its type is $Param_1$; and it is not tainted. The data source of the *RightOp* ($passwordDB) is database; its type is String; and it is tainted. So at the call site of line 3, LChecker applies the taint and type status of the arguments to the loose comparison in the summary. Line 7 thus can be identified as a loose comparison bug. Moreover, to handle the nested user-defined function calls, LChecker recursively adds all callees' loose comparisons (sinks) into the caller's function summary.

Detecting Authentication Related Bugs

LChecker specially handles authentication related bugs due to their commonness and severe security impacts. We observe that the authentication process usually compares a user-provided password with a password stored in the application back end, *e.g.*, database [51]. The passwords are most likely to be hashed before the comparisons to avoid accidental disclosure in a data breach. Therefore, LChecker tags database access and hash computation as additional information during the data flow analysis.

LChecker reports authentication related bugs when an operand of a loose comparison has both the database access and the hash computation tags set. Besides, LChecker disregards some hash functions because they cannot cause inconsistent loose comparison results. For example, the PHP built-in function password_hash() does not generate the magic strings defined in §2.1.2.3.

LChecker hooks the built-in functions that handle database queries and those computing hash values in the analysis. Once these operations are encountered, the corre-

sponding tags are set to True. Besides, since some applications maintain passwords in some back-end files or directly hard-code them in the source code, we propose a keyword matching strategy to assist the detection of authentication related bugs. In detail, we collect a set of frequently-used identifiers for passwords, *e.g.*, $passwd. The variables that use these identifiers are tagged with database access. After that, we follow the similar rules to propagate these tags from their sources to other variables.

2.1.3.2 Dynamic Analysis

The static data flow analysis can detect many potential loose comparison bugs, of which some can be false positives. In addition, our static type inference cannot be 100% accurate because we consider multiple different execution paths at a time. Therefore, we further use a dynamic analysis for validating the loose comparison bugs in a semi-automated manner.

We enhance the PHP interpreter and hook loose comparison operations to validate the loose comparison cases that are statically detected. At run time, we can precisely obtain the types and values of the comparison operands. The enhanced PHP interpreter additionally operates a shadow strict type comparison between the operands of the loose comparisons to check if the two comparisons produce different comparison results. It also further checks whether they fall into the three inconsistent comparison cases (U1-U3), *i.e.*, if an operand is a magic string and is implicitly converted into a different value. Security warnings are generated at run time to notify the true positive bugs to an analyst.

Since our static analysis has already pinpointed a very limited number of potentially vulnerable paths for each identified suspicious case. Thus we can leverage very limited human efforts (see §2.1.5.2) to construct inputs to assist the enhanced program execution engine.

2.1.4 Implementation

We implemented our method with around 3K lines of PHP code and 600 lines of C code. We will release the source code of our prototype implementation. We used PHP-Parser

to parse PHP source code into abstract syntax trees (ASTs) and then construct CFGs and CGs. The taint analysis and type inference were performed by walking through the CFGs and CGs. We also manually identified some password and database related built-in functions for the keyword matching strategy. We implemented the dynamic analysis in the PHP interpreter. We did not modify the generation step of PHP ASTs and opcodes in the PHP Zend virtual machine. Instead, we inserted code in comparison handlers to perform additional strict comparisons and warning generation. We discuss next some important implementation issues we encountered and our solutions.

Arrays. Arrays are usually accessed with keys (*e.g.*, $a[$key]). However, sometimes the concrete values of the keys cannot be inferred statically due to the existence of conditional statements (*e.g.*, if, loops), built-in functions, *etc.* Our data-flow analysis chooses to only model the array items with concrete key values that can be statically inferred. For the other array items, we apply the taint status of the array to them and designate their types to Mixed. We mark an array as tainted if at least one item in it is tainted. This is a commonly used method to model arrays [78].

Loops. To avoid path explosion, we treat loop statements (*e.g.*, for, while and foreach) as if statements and unroll them only once, following the common practice [191]. To achieve this, we remove the back edge pointing from the loop body to the loop header in the CFG construction process. In the case of foreach($array as $key=>$value) loops, a $value and a $key (optional) are created to help iterate over the array items. They are applied to the scope of the loop bodies only. Due to the same challenge of array analysis above, we choose to simplify it by assigning the taint status of the array and a type of Mixed to $value and $key. This allows us to analyze the part of the code inside the loop bodies.

Analysis entries. We implemented LChecker to start the analysis from the main functions in the default application deployment settings. Some user-defined functions are not ever analyzed because they cannot be invoked either directly or indirectly from the main functions. However, these unanalyzed functions themselves might directly interact with untrusted data sources by retrieving values from superglobals, files, *etc.* Thus they can also lead to loose comparison bugs. It is possible that LChecker might miss some loose comparison bugs in the current implementation. Nevertheless, we believe

the implementation choice is reasonable because there is no execution path leading to those bugs.

2.1.5 Evaluation

In this section, we evaluate the effectiveness of LChecker in detecting loose comparison bugs. We apply LChecker to detect loose comparison bugs in several popular PHP applications (§2.1.5.2) and compare it with the related work (§2.1.5.3). We then analyze its performance (§2.1.5.4).

2.1.5.1 Experimental Setup

We select 26 popular PHP applications. They are listed in the first column of Table 2.2. In total, they contain 38.7K PHP source files and 7M LoC and have 49.8K strict comparisons and 49.3K loose comparisons. The evaluation dataset is constructed by selecting 1) applications in the dataset used in a closely related work—Nemesis [51]; 2) popular and large PHP applications such as WordPress, MediaWiki, and HotCRP; and 3) several well-known and highly-rated PHP projects on GitHub. We try to include all applications with known loose comparison bugs in CVE database as the ground truth in our evaluation (the last seven applications in Table 2.2). However, some applications cannot be included because their complete source code is not publicly available (*e.g.*, CVE-2020-10568 [39]) or we are unable to locate the bugs given the limited information in the CVE (*e.g.*, CVE-2019-10231 [38]). We download the source code of each application from its official website or GitHub, and configure it with the default settings in our experiments.

We also compare with PHP Joern, which is the only relevant tool that attempts to detect loose comparison bugs [16]. For a fair comparison, we try to use the same settings as in their paper. We first construct the code property graphs [195] for each application. We then apply the same taint propagation rules mentioned in §2.1.3.1 and traverse the graphs to detect magic hash bugs.

All experiments were conducted on a computer running Debian GNU/Linux 9.12, with a 4-core Intel Xeon CPU and 16GB RAM.

2.1.5.2 Bug Detection

The evaluation results are shown in Table 2.2. We use the superscript L in the column headers to denote the results of LChecker, and subscripts $taint$, $type$, and tp for the results of running only taint analysis, running type inference, and the final true positives. The taint analysis of LChecker checks Cond1 to filter those normal loose comparisons out. Among a total of 49.3K loose comparisons (LC) in the 26 PHP applications, LChecker identified 958 (1.92%) tainted loose comparisons (LCB^L_{taint}) in all applications, while $48{,}363$ (98.08%) loose comparisons did not meet our definition thus were directly excluded. This indicates that our taint analysis can very *effectively* narrow down the scope of buggy loose comparison cases.

After applying the type inference to check Cond2, LChecker removed 773 (80.69%) loose comparisons identified in the taint analysis and reported only 185 cases (LCB^L_{type}) in 17 out of 26 applications. We then analyzed the reported bugs with the enhanced PHP interpreter and confirmed 50 loose comparison bugs (LCB^L_{tp}). 42 bugs are previously unknown. To validate all the 185 reported cases, it took one author, a total of 12 hours with the help of the enhanced PHP interpreter. The manual effort was mainly spent on checking whether Cond3 can hold or not. In most cases, the potentially vulnerable paths were reported already in the taint analysis and the type inference analysis, so the manual analysis was straightforward. We responsibly reported the newly detected bugs to the relevant developers. As of Feburary 17, 2021, 10 bugs, including 9 CVEs[1], have been promptly acknowledged or patched

Effect of Keyword Match Strategy

As mentioned in Listing 2.1.3.1, we collect some frequently-used identifiers to assist the detection of authentication related bugs. We find that 230 out of the 958 cases reported in taint analysis were identified with the help of these identifiers. Further, 44 out of 185 cases reported in type inference and 13 out of 50 real loose comparison bugs were identified with such identifiers. This indicates that the keyword match strategy can help improve the effectiveness of bug detection.

Characterization of Bugs

[1]CVE-2020-23352, CVE-2020-23353, CVE-2020-23355, CVE-2020-23356, CVE-2020-23357, CVE-2020-23358, CVE-2020-23359, CVE-2020-23360, CVE-2020-23361.

Table 2.2: Detection results of loose comparison bugs.

App	Files	LoC	SC	LC	LCB^{L}_{taint}	LCB^{L}_{type}	LCB^{L}_{tp}	Auth^{L}_{tp}	CV^{L}_{tp}	Time^{L}	LCB^{J}_{taint}	LCB^{J}_{tp}	Time^{J}
WordPress (5.4.1)‡	1,474	862,308	5,025	3,439	42	18	0	0	0	10 m	35	0	11 m
MediaWiki (1.3.41)‡	4,289	1,101,308	9,992	3,661	18	0	0	0	0	12 m	21	0	11 m
phpStat (1.5)†	16	2,138	0	112	52	15	4	0	4	1 m	38	0	1 m
Codiad (2.8.4)†	57	10,798	94	241	39	10	6	1	5	2 m	28	1	1 m
Monstra (3.0.4)†	509	422,355	241	500	8	1	0	1	0	3 m	12	1	1 m
PHP-ML (2.0)‡	148	12,895	182	84	19	2	2	0	2	1 m	11	0	2 m
Z-BlogPHP (1.6.0)†	250	46,443	290	1,247	39	6	3	1	2	2 m	11	0	1 m
HotCRP (2.102)‡	224	76,598	4,119	1,907	39	0	0	0	0	2 m	23	0	1 m
FAQforge (1.3.2)†	108	2,984	0	32	17	2	1	1	0	1 m	9	1	1 m
geccbblite (0.1)†	11	327	0	6	4	4	4	0	4	1 m	4	4	1 m
phpMyAdmin (5.0.2)‡	4,289	324,353	4,399	3,542	19	4	0	0	0	2 m	45	0	2 m
PHPiCalendar (2.4)†	83	21,466	64	627	49	5	1	1	0	2 m	33	0	1 m
SCARF†	20	1,687	5	44	19	0	0	0	0	1 m	17	0	1 m
PHPFastNews (0.3)†	22	4,288	18	75	15	2	2	0	2	1 m	11	0	1 m
Drupal (8.7.0)‡	11,644	1,652,690	7,171	3,846	49	0	0	0	0	5 m	63	0	4 m
phpBB (3.3)†	2,964	502,182	6,009	3,536	29	5	0	0	0	6 m	12	0	8 m
MyBB (3.3.0)†	416	174,796	364	6,464	52	10	0	0	0	3 m	21	0	1 m
WeBid (1.2.1)†	416	150,640	141	1,775	63	15	7	0	7	4 m	44	2	3 m
osCommerce (2.3.4)‡	807	93,835	133	2,764	64	14	7	0	7	2 m	72	0	1 m
PHPList (3.5.0)†	7,770	853,603	9,080	2,010	77	24	5	3	2	5 m	63	2	1 m
UseBB (1.0.12)†	80	22,509	66	337	69	12	1	1	0	1 m	59	0	3 m
YOURLS (1.7.3)†	479	46,364	500	263	41	5	1	1	0	2 m	17	1	1 m
Centreon (2.8.26) †	1,793	376,522	824	5,489	15	7	2	1	1	1 m	12	1	1 m
Trovebox (4.0.0-rc5) †	477	82,594	708	673	23	1	1	1	0	1 m	19	0	1 m
PHPLiteAdmin (1.9.6)‡	24	10,110	88	387	46	18	1	1	0	1 m	31	0	1 m
MyBB (1.8.6) †	376	167,722	304	6,291	51	5	1	1	0	3 m	48	0	3 m
Total	38,746	7,023,515	49,817	49,352	958	185	50	14	36	75 m	759	13	64 m

SC, LC, and LCB mean strict comparisons, loose comparisons, and loose comparison bugs, respectively.
The superscripts L and J denote the results of LChecker and PHP Joern.
The subscripts $taint$, $type$, and tp denote the results of taint analysis, type inference, and true positives.
Auth and CV denote authentication bypass vulnerabilities and correctness violation bugs.
‡ denotes relatively more popular applications.
† denotes relatively less popular applications.

Table 2.3: Types of loose comparison bugs.

Type	U1	U2	U3	U1+U2	U1+U3	U2+U3	U1+U2+U3	Total
# Bugs	6	0	0	41	0	0	3	50

We further investigate and characterize these real bugs in this section.

Categorization. We manually investigated the characteristics of the bugs and classified them into two categories: 1) authentication bypass vulnerabilities, allowing attackers to bypass authentication without valid credentials; and 2) correctness violation bugs, breaking the application functionality or correctness and leading to abnormal program behaviors. There are 14 authentication bypass vulnerabilities and 36 correctness violation bugs, respectively. The results are shown in columns Auth^{L}_{tp} and CV^{L}_{tp} in Table 2.2.

We also classify the bugs based on the type of inconsistent loose comparisons they can cause, as shown in Table 2.3. Most buggy programs perform direct plain text (String) loose comparisons, thus can be exploited by both scientific notations strings (U1) and numeric strings (U2). In six bugs, the programs process the operands with hash functions, which can only produce scientific notation strings (U1). Only three bugs involve both same-type (String) and cross-type (String and Numeric) loose comparisons and belong to all the three types of bugs (U1 + U2 + U3).

Popularity of buggy apps. LChecker detected loose comparison bugs in a diverse set of applications. It found bugs in content management system applications, such as Codiad (2.8.4) [36] and Monstra (3.0.4) [126]. It also found bugs in several popular and well-maintained applications. For example, LChecker detected seven bugs in osCommerce (2.3.4) [136], a popular e-commerce software used by over 200K websites; two bugs in PHP-ML (2.0) [148], a popular PHP machine learning library with over 6K stars on GitHub.

We investigate the relationship between loose comparison bugs and the popularity of buggy applications. We classify the applications in our dataset into relatively *more* or *less* popular applications, and mark them in the first column of Table 2.2 with superscripts ‡ and †, respectively. Specifically, apps having over 0.1% market share [180], having over 2K stars or forks on GitHub, or used by over 100K websites are classified as relatively more popular ones. Others are relatively less popular.

As shown in the first column of Table 2.2, 40 bugs were found in 14 out of the 18 relatively less popular applications (with 3.0M LoC). The rest 10 bugs were found in only three out of the eight relatively more popular applications (with 4.0M LoC). We did not detect any bug in those apps with a huge number of lines of code (*e.g.*, WordPress, MediaWiki, and Drupal). This suggests that the size of the application may not be directly related to loose comparison bugs. However, the relatively less popular applications are more likely to have loose comparison bugs, probably because they are less well maintained.

False Negatives

The evaluation result on applications with known bugs demonstrates that LChecker has no false negative in this ground truth dataset. The last seven applications in Table 2.2 have eight previously known loose comparison bugs, including seven CVEs. LChecker identified all eight known bugs. Interestingly, LChecker also detected two new bugs in the old version of PHPList (3.5.0). They remained in the latest version until we reported to the developers.

However, LChecker might still miss potential loose comparison bugs and lead to false negatives. First, LChecker has limitations in call target inference because it builds an incomplete call graph as other works [12, 16]. Therefore, some actual reachable functions might not be analyzed. Second, the loops are unrolled only once, thus many paths are not studied. Some bugs might be exposed only after several iterations. Other imprecise modeling mentioned in §2.1.4 can also result in false negatives.

False Positives

LChecker employs a static analysis and consequently has false positives. As presented in Table 2.2, many statically detected loose comparison bugs were not validated as real bugs. Especially, in 185 reported cases, only 50 were confirmed as real loose comparison bugs, and 135 were false positives. We discuss the main causes of false positives as follows.

Custom sanitizers. Developers can write their custom functions to sanitize untrusted user data. Some tainted loose comparison cases are actually sanitized with these custom sanitizers and are reported as false positives. However, it is difficult to automatically identify all these sanitizers. Around 10% of false positives fall under this category.

Partial dependency. Loose comparison may be used as only part of the entire logical formula in a conditional statement. Therefore, the execution flow does not depend on only loose comparisons. Even though a loose comparison operation alone can be bypassed through U1-U3, the overall condition might still remain the same and lead to the same program execution path. This introduces around 20% of false positives.

Unreachable code. Many statically detected loose comparison bugs are not practically reachable. This is because the global path constraints for reaching them cannot be satisfied. Our manual investigation reveals that around 30% of false positives are caused by this reason. To address it, we propose to use symbolic execution together with constraint solving in our future work.

Uses of safe functions. Many false-positive loose comparison bugs do not produce inconsistent comparison results, because the operands are produced by some *safe* built-in or user-defined function. For example, some password encryption function always generates outputs that do not lead to inconsistent comparison results. Around 20% of false positives fall under this category.

Others. The remaining 20% of false positives are caused by other issues, such as inaccurate modeling of arrays, inaccurate type inference in the branch statements, and dynamic function calls. Although they are orthogonal challenges of static analysis, mechanisms like [61, 79] can be used to improve the analysis accuracy of LChecker.

2.1.5.3 Comparison with Related Work

In Table 2.2, the detection results of PHP Joern are shown in the columns labeled with the superscript J. Since PHP Joern does not support type inference, we list the results of running only taint analysis (LCB^{J}_{taint}) and the true positive cases (LCB^{J}_{tp}). Overall, it reported 759 cases, of which 13 (1.71%) bugs in eight out of 26 applications were confirmed as true bugs. We investigate and demonstrate the detection results of LChecker and PHP Joern in each step in Figure 2.2.

LChecker outperformed PHP Joern with more confirmed bugs and much fewer false reports. LChecker successfully detected all real loose comparison bugs that PHP Joern detected, and 37 more real bugs that PHP Joern failed to detect. In taint analysis,

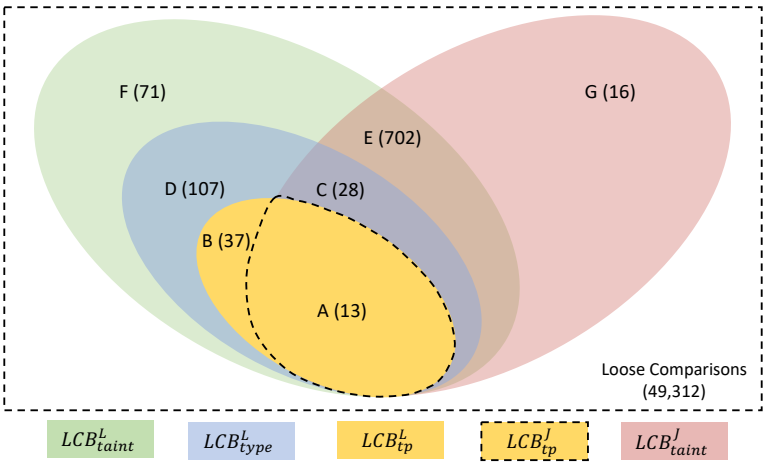


Figure 2.2: Result distribution of LChecker and PHP Joern. Alphabets (A - G) and the numbers in parenthesis denote different situations and the corresponding number of cases in them.

LChecker and PHP Joern reported 985 and 759 loose comparison cases, respectively, among 26 applications. There are 743 cases (A, C, and E) that were found by both tools, and only 13 cases (A) of them were confirmed as real bugs. Since PHP Joern targets at only magic hash problem (U3), 215 more cases (B, D, and F) were reported by LChecker but not by PHP Joern. Consequently, 37 loose comparison bugs (B) were thus identified by only LChecker. With LChecker's type inference, it lowered the number of possible cases to 185—of which 50 (27.03%) were true positives, significantly limiting the cases that need to be semi-manually validated by a human analyst. In contrast, an analyst using PHP Joern would have to examine all 759 potential cases, of which only 13 (1.73%) were true positives.

We notice that 16 cases (G) were reported by PHP Joern but not by LChecker. Our further study finds that they were missed because of LChecker's imprecise modeling of arrays (§2.1.4).

2.1.5.4 Performance

LChecker's static analysis is both efficient and scalable. It statically analyzed 7M LoC in 26 PHP applications within 75 minutes, comparable to PHP Joern which finished the analysis within 64 minutes. The analysis time for each application is shown in column Time^{L} (LChecker) and column Time^{J} (PHP Joern) of Table 2.2. Most applications were analyzed by LChecker within one minute because it employs a context-sensitive inter-

procedural analysis. It spent only 12 minutes on analyzing MediaWiki (1.3.41) [120], which has over one million lines of code. This shows that LChecker is able to efficiently analyze complex modern applications.

2.1.6 Discussion

Validating loose comparison bugs. We enhanced the PHP interpreter to assist humans to validate loose comparison bugs. The difficulty of our semi-manual validation is greatly decreased because some potentially vulnerable paths have been pinpointed by the static analysis. Our evaluation results also demonstrated that the manual efforts spent in validating the bugs were acceptable. However, techniques like symbolic execution [12] and directed fuzzing [22, 185] might be applied to further automate such processes. In the future, we plan to leverage symbolic execution to collect path constraints for reaching the bugs and query constraint solvers for possible solutions that can be used in the dynamic bug validation process.

Patching loose comparison bugs. Some loose comparison bugs can be patched locally, *i.e.*, by simply changing the comparison operations [127]. Converting the loose comparisons to strict ones to eliminate implicit type conversion and enforce operand types can avoid some of the bugs. Fixing some other loose comparison bugs may require changes in the overall logic. For example, a program might be designed to allow the comparison operands to be in multiple types for different paths. Thus directly converting its loose comparisons to strict ones might break the intended functionalities in some types or paths. Therefore, this might require developers to patch each case differently.

Portability. Besides equal and unequal operators, implicit operand type conversion can also happen in other loose comparison operators such as the greater than operator (>). These loose comparison operators are also subject to the similar loose comparison bugs. We currently only implement our method to detect buggy loose comparisons using the equal and unequal operators. The proposed definition and approach, however, can be ported to other loose comparison operators without loss of generality. We leave it as our future work.

2.1.7 Related work

Type system bugs. Recent research has covered some type system bugs other than loose comparison bugs. μPHP [13] formally defines type juggling and implicit type conversion in PHP, but it does not target at detecting bugs caused by such language features. Phantm [95] and PHPLint [149] use static flow-sensitive analysis to identify variable type mismatch errors in PHP. Some prior works detect type system bugs in other programming languages. TypeDevil [151] identifies the type inconsistency bugs in JavaScript with a runtime type analysis. Johnson *et al.* detect user/kernel pointer bugs in Linux kernel with a type qualifier inference method [88]. CAVER [99] identifies bad type casting bugs in C/C++ at run time. We study a novel class of bugs in the type system of PHP.

PHP application bug detection. Many related works have tried to detect logic vulnerabilities, including authentication bypass vulnerabilities and access control vulnerabilities. Dahse and Holz precisely model PHP built-in functions and statically detect bugs with taint analysis [49]. Additionally, they detect *second-order vulnerabilities* that are exploited with the second-order attack inputs [50]. Sun *et al.* compare sitemaps of different user roles to find privileged pages and detect access control vulnerabilities via forced browsing [172]. Nemesis [51] leverages dynamic information flow tracking to prevent authentication and access control vulnerabilities with developer specified access control rules. However, it does not investigate loose comparison bugs. Many other works use taint analysis [89, 119, 165, 173] and symbolic execution [7, 14, 165, 173] to find logic vulnerabilities in web applications. However, these works focus on application-layer logic vulnerabilities. Instead, LChecker targets the logic vulnerabilities caused by the language feature misuse in PHP.

PHP Joern [16] is the only work that discussed identifying magic hash bugs. We extend the research scope to loose comparison bugs and develop a type inference algorithm to reduce the false-positive rate.

□ End of chapter.

Chapter 3

Exposing Security and Performance Bugs through Fuzzing

The previous chapter has explored the static bug detection techniques. However, static analysis often makes many conservative assumptions about a program, and tends to report many false positives. Recently, dynamic analysis techniques have shown great promise for detecting new bugs in real-world software. Dynamic analysis can accurately capture the software behaviors at runtime and report only true positives. Fuzz testing (fuzzing) is especially effective in automatically exposing various vulnerabilities by dynamically monitoring abnormal behavior, *e.g.*, crashes and performance slowdowns. In this chapter, we develop dynamic grey-box fuzzing frameworks to automatically detect security and performance bugs.

3.1 SDFuzz: Target State Driven Directed Fuzzing for Security Bugs

3.1.1 Motivation

Directed grey-box fuzzing (DGF[2]) usually drives the testing towards highly-valuable target site locations. Though being widely used in crash reproduction and vulnerabil-

[2]In this thesis, we use DGF to denote directed grey-box fuzzing or directed grey-box fuzzer interchangeably.

ity validation [30, 58, 100, 144], one primary problem of prior directed fuzzers is that they often explore a large number of program code and paths that cannot trigger the crashes. In the exploration stage of DGF, most existing approaches follow AFLGo's solution [144] and use coverage feedback for expanding the exploration of different code areas. Many code areas are irrelevant and unrequired for reaching the target sites [167]. Such code areas are still unnecessarily explored. In the exploitation stage, prior directed fuzzers utilize the distance metrics based on the control-flow graph (CFG) and call graph (CG) [30, 58, 144]. The distance metrics do not consider the path conditions, such as control-flow and data-flow conditions. As a result, executions that cannot trigger the vulnerabilities at the target sites because of unsatisfiable control-flow or data-flow conditions might still gain short distances and be overly favored [83, 100, 206]. Excessive testing efforts are thus wasted [206].

A solution to mitigate the above-mentioned problem is to first identify the required program code/paths for triggering the crashes and fuzz only the required one(s). SieveFuzz analyzes the inter-procedural control-flow graph (ICFG) to identify required functions for reaching the target sites and terminates the executions once they reach unrequired functions [167]. Beacon computes the preconditions to reach the target sites via a backward interval analysis and early terminates the executions that fail to satisfy the preconditions [83]. SelectFuzz [115] statically identifies the control- and data-dependent code to the target sites. However, they over-approximate the set of allowed program code and paths, significantly restricting their performance. This is because their analysis is conservative. They only consider the control-flow reachability but fail to analyze other conditions for triggering the vulnerabilities, *e.g.*, the expected reaching order of the target sites [100]. Hawkeye [30] and CAFL [100] include call traces in their designs of distance metrics, however, they do not use them to exclude unrequired code/paths.

We solve the problem of unnecessary exploration in DGF with a new finding of *target states, which include the expected call traces and reaching order of the target sites for triggering the vulnerabilities.* In particular, we find that the major tasks of DGF, such as crash reproduction and vulnerability validation, all provide detailed descriptions of vulnerabilities, such as the target site locations and the associated vulnerability information, *e.g.*, crash dumps, backtraces [71], and source-sink flows [171]. Target site lo-

cations describe where the vulnerabilities would occur; the vulnerability information additionally explains the interesting program states (namely target states) about how to trigger the vulnerabilities. The goal of directed fuzzing is not only to reach the target site locations but also to dynamically expose the vulnerabilities. Inspired by this, we aim to drive the fuzzing towards not only the target sites but also the interesting target states. *Thus, besides excluding the exploration that cannot reach target sites, we further avoid the unnecessary executions that cannot achieve the target states.* Therefore, we can further narrow down the fuzzing scope to the required code for achieving the target states, which is a subset of the required code for reaching target sites. Similarly, executions that cannot achieve the target states can be early terminated.

We thereby design SDFuzz, a target State based Directed Fuzzer. SDFuzz first automatically extracts the target states (*i.e.*, expected call traces and the reaching order of target sites) from vulnerability reports and static analysis results. It then takes a selective instrumentation technique to reduce fuzzing scope to the code required for achieving the target states. In particular, based on the functions in the target states, SDFuzz analyzes the calling relationship in the inter-procedural control-flow-graph (ICFG) to identify (un)required code. Unlike prior works that directly terminate all executions on unrequired code (identified based on target sites), SDFuzz still preserves all code of the software but instead removes the code coverage feedback from unrequired code (identified based on target states). The unrequired code is hidden from the fuzzer's perspective, SDFuzz thus would not assign excessive energy to overly test it. Besides, since the target states further constrain the paths to target sites, SDFuzz preserves only a subset of the code preserved in prior works [83, 167]. This design choice also gets rid of the acute side effects of false code elimination and is fault tolerant: executions can go through (wrongly identified) unrequired code to target sites without being (wrongly) terminated.

SDFuzz early terminates the execution of a test case once it probes that the remaining execution cannot achieve the target states. One challenge we face is *how to determine if an execution cannot ultimately achieve the target states without completing the execution.* We solve the problem by monitoring runtime program state. We first formalize a program state as the active call stack and the associated function invocation locations;

target states for a vulnerability are a list of ordered program states that are expected to be sequentially achieved. At runtime, SDFuzz maintains the runtime program state, periodically compares it to the next unachieved target state, and probes if there is any deviation. Situations with any deviation that *cannot be recovered in the subsequent execution of a test case* can be terminated immediately. A deviation can be recovered on if there is a program path on the ICFG, through which the deviation function call can possibly be updated to the expected function call specified in the target state.

Besides reducing resource consumption by avoiding unnecessary exploration, SDFuzz further employs a two-dimensional feedback mechanism to *proactively* drive the testing towards target states. In the first dimension, SDFuzz calculates a novel target state feedback by computing a similarity score from the best runtime program state of a fuzzing trial to the target states. This dimension aims to favor test cases with better runtime program states. In the second dimension, SDFuzz uses the widely-adopted distance metrics in DGF. Instead of using an empirically configured constant coefficient (*e.g.*, 10 in AFLGo [144]) as the inter-procedural distance between functions in CG, SDFuzz uses a new weighted inter-procedural distance by approximating the chance for the caller function to invoke the callee function. The two-dimensional feedback is then used for the power scheduling and seed selection.

We thoroughly evaluated SDFuzz on a diverse set of vulnerabilities from the Google Fuzzer Test Suite [77] and AFLGo's Test Suite [145], and demonstrated its high effectiveness. We compared SDFuzz to four related state-of-the-art directed fuzzers (AFLGo [144], WindRanger [58], Beacon [83], and SieveFuzz [167]). The results show that SDFuzz outperformed these works by exposing five, four, six, and three more vulnerabilities, respectively. In terms of vulnerability exposure time, SDFuzz achieved an average speedup of 4.79×, 4.63×, 1.57×, and 2.58× above AFLGo, WindRanger, Beacon, and SieveFuzz, respectively. SDFuzz achieved the best performance for 80% of the evaluated vulnerabilities. Our ablation study confirmed the benefit of each technique in SDFuzz, especially the selective instrumentation and execution termination techniques. For instance, on average, SDFuzz eliminated 42.39% of unrequired functions and early terminated 60.38% of executions. Furthermore, we applied SDFuzz to automatically extract target states from the results of a static analysis tool—SVF [171]—and validate the results. SDFuzz success-

fully identified four new vulnerabilities in three well-tested file processing applications (*e.g.*, libjpeg). Three of them have been promptly acknowledged by the developers.

In summary, this paper makes the following contributions:

- We proposed a new concept of target states and demonstrated their benefits for DGF.
- We designed SDFuzz, an effective directed fuzzing system driven by target states.
- We incorporated DGF with static analysis to fully automatically validate the results.
- With SDFuzz, we discovered four new vulnerabilities.

3.1.2 Preliminary

In this section, we present the background of directed grey-box fuzzing and its representative use examples. We then analyze existing DGF approaches to motivate our solution.

3.1.2.1 Directed Grey-Box Fuzzing

DGF differentiates itself from general coverage-guided grey-box fuzzing in specializing in testing specific code locations. DGF employs an exploration stage driven by the code coverage feedback to expand the covered code locations. Additionally, DGF has an exploitation stage, where past practices mainly employ distance metrics for providing runtime feedback to guide testing direction [58, 100, 144]. For instance, AFLGo defines the distance of a basic block as the harmonic mean of its shortest path lengths to all target sites [144]. In a shortest path, the path length is the sum of the weighted inter-procedural CG distance from the caller function to the target function and the intra-procedural CFG distance from the basic block to the call site's basic block. During a fuzzing trial, a directed fuzzer obtains the execution trace of a test case and computes the average distance of the executed basic blocks as the seed distance. It then uses the distance for seed selection and power scheduling.

```
1  void main() {
2      int x = input();
3      if(x < 10)
4          option1(x); //(1)
5      else
6          option2(x); //(2)
7      clean();
8  }
9
10 void option2(int opcode) {
11     target(opcode);
12 }
```

```
13 void option1(int opcode) {
14     check();
15     target(opcode + 5);
16 }
17
18 void target(int arg) {
19     if (arg <= 20) {
20         assert(arg < 5);
21     }
22     ...
23 }
```

Listing 3.1: A motivating example.

3.1.2.2 Use Examples of DGF

We show some representative use examples of DGF below.

Reproducing crashes. Software developers often accept vulnerability reports from users. Automated crash report systems [76] collect crash reports from end users and send them to the developers. Since the crash inputs usually convey sensitive user data (*e.g.*, confidential cookies for browser crashes), such systems by default do not attach the crashing proof-of-concept (PoC) inputs. They instead often provide the crash dumps, which contain the crash types and locations, and the involved call traces when the crashes occur. DGF is then helpful for the developers to reproduce the crash specified in the crash dumps. Even if PoC inputs are available, developers may still want to use DGF to comprehend the vulnerabilities and patches.

DGF can address the limitations of symbolic execution based crash reproduction tools. For instance, Star [32], JCharming [153], and BugRedux [87] perform symbolic execution on the slices to target sites. However, they usually collect non-linear, incomplete, and complex path constraints and do not handle external environments, rendering PoC generation through constraint solving difficult [201]. DGFs can address such complex issues by efficient input mutation.

Validating vulnerabilities. Static analysis screens the source code of the program and outputs potential vulnerabilities. The analysis results commonly include vulnerability locations and assumed suspicious flows for the vulnerabilities [155, 169, 170, 195]. To reduce the huge number of false positives, developers can leverage DGF to proactively find PoCs on the suspicious flows and validate the results.

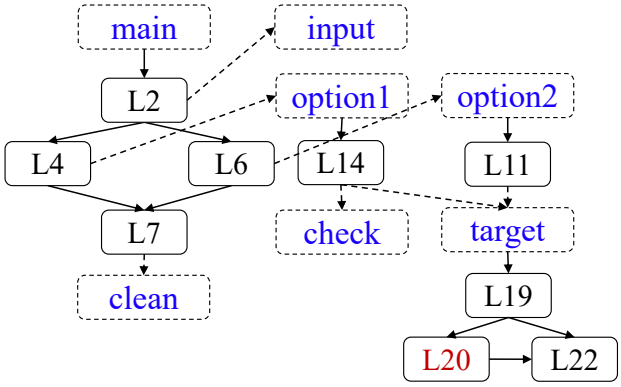


Figure 3.1: The ICFG of code in Listing 3.1.

3.1.2.3 Existing Approaches and Limitations

Directed fuzzers often unnecessarily test code and paths that cannot trigger the vulnerabilities. This occurs in both the exploration and exploitation stages. We illustrate it with a motivating example shown in Listing 3.1. There is an assertion failure at L20, which is often set as the target site in DGF. The execution through L4 in function main() (namely execution ①) can reach the target site and possibly trigger the assertion failure. The execution going through L6 (namely execution ②) can only reach the target site.

Unnecessary exploration stage testing on unrequired code. DGF has the goal of reaching target sites and triggering the vulnerabilities there. Its exploration stage directly adopts the conventional coverage metric to expand the covered code, which would favor the executions/test cases that increase the coverage (*i.e.*, new code discovery). However, a large proportion of code is not required for reaching the target sites and would negatively interfere the exploration direction of DGF. For instance, the function clean() is not helpful for reaching L20.

Unnecessary exploitation stage testing on (un)reachable executions. Even after narrowing down the fuzzing scope to the required code, the exploitation stage still causes unnecessary resource consumption on both executions that can or cannot reach the target sites, *i.e.*, reachable and unreachable executions. As we mentioned in §3.1.2.1, DGFs use distance metrics in the exploitation stage [30, 100, 144]. The distance metrics are simply based on the ICFG without analyzing path conditions. Thus paths with unsatisfiable conditions can be regarded as feasible ones. The corresponding executions

might obtain short distances and get overly favored.

Prior Solutions

A solution to mitigate this problem is to identify the unrequired code/paths for triggering the vulnerabilities. Beacon [83] statically identifies unrequired code based on the reachability to target site locations. It immediately terminates executions through assertions once the executions trigger unrequired code. SieveFuzz [167] further integrates a dynamic analysis to identify unrequired code and then accordingly terminates executions. SelectFuzz [115] identifies the data- and control-dependent code to the target sites. However, they merely leverage the reachability to the target site locations to drive their required code identification. They could not sufficiently exclude other unrequired code.

Beacon also inserts assertion checks to early terminate executions unsatisfying the preconditions for reaching the target sites through a backward interval analysis. However, the current design of Beacon has two foundational problems. First, to terminate the executions early, the backward interval analysis produces complex preconditions, which introduce significant runtime overhead [167]. Second, the preconditions merely embed the control-flow reachability to the target sites without considering additional important information such as call traces and the reaching order of the targets [100]. As a result, Beacon would not terminate reachable executions like execution ②.

Summary. The majority of distance-based DGFs do not leverage any method to exclude unnecessary exploration. They unnecessarily test code and paths in both the exploration and exploitation stages. Prior mitigation approaches [83, 115, 167], however, only consider the reachability to the target sites. Hawkeye [30] and CAFL [100] employ call traces to refine their distance metrics. They do not exclude unnecessary exploration like other distance-based DGFs. In this thesis, we tackle these problems and improve the effectiveness of DGF by further eliminating unnecessary testing.

3.1.3 Target States

DGFs specify the target sites—code locations of interest. The target sites often come from two sources: 1) vulnerability reports and 2) static analysis results. We find that

```
1 0x... in target      file.c:20
2 0x... in option1     file.c:15
3 0x... in main        file.c:4
4 0x... in in ___libc__start__main
```

(a) A crash dump.

TS1=[(main, libc), (option1, L4), (target, L15)]

PS1=[(main, libc), (input, L2)]
PS2=[(main, libc), (option1, L4)]
PS3=[(main, libc), (clean, L7)]

(b) Selected program states.

Figure 3.2: A crash dump and associated program states.

from the two sources, we can additionally identify target states, which include the expected call traces and reaching order of the target sites. The target states are helpful for improving directed fuzzing.

3.1.3.1 Vulnerability Reports

Software developers often accept vulnerability reports from users. Most vulnerability reports in the wild contain associated crash dumps to help developers confirm the vulnerabilities timely. For example, Figure 3.2(a) presents the crash dump (call trace) for the assertion failure in Listing 3.1. It captures a list of function calls that are currently active in the thread and the invocation locations when the crash is triggered. The libc in the crash dump means the library that starts the execution of the main function. Prior DGFs often set the erroneous locations at the top of the crash dump (*i.e.*, crash points) as the fuzzing target sites, *e.g.*, file.c:20 in this example.

However, the crash point is only a part of the crash dump. The crash dump pinpoints a highly-valuable call trace about the reported crash, through which the vulnerability can be easily reproduced. With only the target site locations, directed fuzzers would unnecessarily explore many infeasible program states (*e.g.*, execution ②). Driving a directed fuzzer to invoke the list of functions conforming to the crash dump could enable it to trigger the vulnerability quickly. Furthermore, for multi-target vulnerabilities (*e.g.*, use-after-free), there can be associated call traces for each involved target site, and their expected reaching order can be inferred [100]. *In this thesis, we define the expected call traces and the reaching order of target sites as target states.* No prior directed fuzzer has used the target states to eliminate unnecessary exploration. Prior solutions [83, 167] only use the target site locations and partially exclude exploration that could not reach the target sites.

Crash dumps are available in most of the vulnerability reports, from which target states

can be extracted. We studied four popular file processing applications such as swftophp and lrzip, to which fuzzing has been widely applied [83, 144]. We examined all historical crashes in the applications and summarized an unbiased dataset of 259 crashes. We found that crash dumps existed in 191 (73.75%) cases. This demonstrates that target states can be obtained in most cases. A solution that requires the target states still has immense applicability to the majority of cases.

3.1.3.2 Static Analysis Results

Static analysis is based on heuristics and often outputs detailed information for analysts to comprehend the analysis results. Static analysis tools report the vulnerable flows to the target sites where the conditions of the heuristics are met. For instance, Joern [195]—a taint analysis tool—would report the vulnerable flow from the untrusted external data sources to the target sites. SVF [170, 171] for memory-safety issues would report vulnerable program flows of improper memory usage.

Prior directed fuzzers often set the memory operations in the reported flows as the target sites. This would make DGF explore all possible paths to the target sites. However, there are abundant flows that do not meet the conditions of the heuristics. Testing them causes non-negligible resource waste. To efficiently validate the vulnerability, a fuzzer ought to focus on exhibiting the vulnerable flows instead of all possible ones. Similar to vulnerability reports, we can derive the expected target states from such commonly provided vulnerable flows in static analysis results.

3.1.3.3 Formalization

We define a general representation of target states used in this thesis, as shown in Equation 3.1. We learn from examples (*e.g.*, Figure 3.2(a)) that the crash dumps contain a sequence of ordered function calls and the invocation locations. We thus formalize a target state (TS) as the a stack of function invocations. Each item in the target state is a tuple of the function name and the invocation location ((Func, Loc)). These functions in the stack ought to be reached or called in order. Therefore, Equation 3.1 captures the expected call traces and their reaching order. For example, we can formalize the crash dump in Figure 3.2(a) as the target state (TS1) shown Figure 3.2(b). For multi-target

vulnerabilities, we can usually derive one target state per target site and sort them to obey the required target reaching order (TSs) [100]. Similarly, the target states of static analysis results can be formalized using program states.

$$\begin{aligned} TS &= [(Func_1, Loc_1), (Func_2, Loc_2), ..., (Func_n, Loc_n)] \\ TSs &= [TS_1, TS_2, ..., TS_m] \end{aligned} \tag{3.1}$$

3.1.4 Design of SDFuzz

3.1.4.1 Overview

We advance DGF using target states. In the vast exploration space of a program, a significant proportion cannot trigger the target vulnerabilities. Testing all of it would cause unnecessary resource consumption. Fortunately, target states describe interesting program states where the vulnerabilities would (likely) occur. Our intuition is to drive the fuzzing to exhibit these interesting target states instead of merely reaching the target site locations. We use the target states from vulnerability reports and static analysis results to *exclude unnecessary exploration that cannot achieve the target states*. We further design new techniques to proactively guide the testing. Hawkeye [30] and CAFL [100] also use call traces in their distance design to guide the exploration. They still consider the whole exploration space of a program. SDFuzz starts from a different angle by proactively removing unnecessary testing.

We thereby develop a new directed fuzzing system, SDFuzz, based on the target states. The workflow of SDFuzz is depicted in Figure 3.3. SDFuzz first automatically extracts target states and parses them into specified formats. It then instruments the program to exclude unrequired code and to add distance feedback in the instrumentation phase. In the fuzzing phase, SDFuzz terminates infeasible executions early based on runtime program states and proactively guides the exploration directions. Generally, SDFuzz consists of three key parts:

Selective instrumentation of required code (§3.1.4.3). SDFuzz identifies the required code for achieving the target states and removes other unrequired code from fuzzing. It advances prior solutions [83, 167] in two folds. First, it identifies the required code based on the target states instead of target sites, and thus SDFuzz can precisely

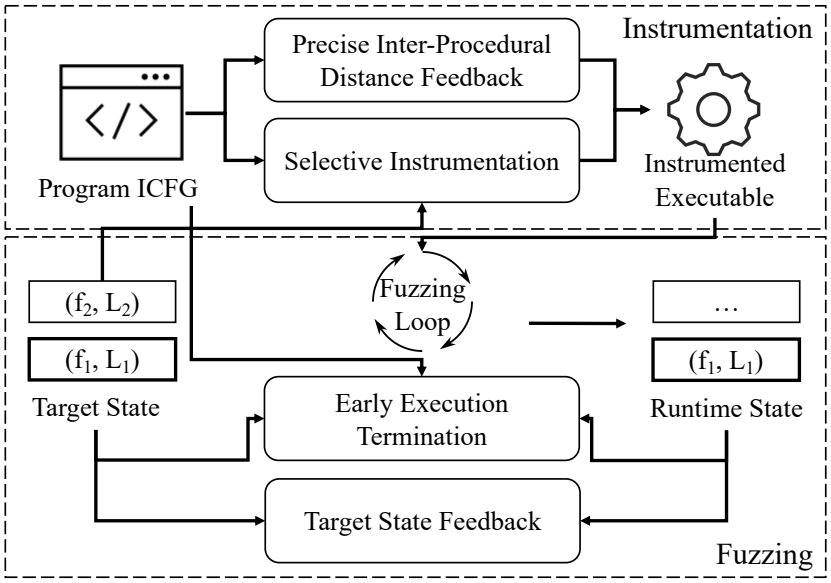


Figure 3.3: The workflow of SDFuzz.

eliminate more unrequired code. Second, SDFuzz avoids the exaggerated side effects of false code elimination through selective coverage instrumentation. It preserves all code of the software but computes no code coverage for unrequired code, which makes our solution fault tolerant.

Early termination of executions (§3.1.4.4). SDFuzz early terminates the executions once it probes that the remaining execution of a test case cannot achieve the target states. SDFuzz monitors the program states and terminates the executions with unrecoverable deviations to the target states. SDFuzz thus can try more test cases in a given amount of time and improve fuzzing throughput (*i.e.*, number of executions per second). SDFuzz also terminates the reachable executions that cannot achieve the target states and dramatically outperforms prior reachability-based execution termination technique [83].

Two-dimensional feedback mechanism (§3.1.4.5). SDFuzz uses a two-dimensional feedback mechanism to proactively guide the testing towards the target states. In the first dimension, SDFuzz measures the similarity between the best runtime state of a test case and the target states, and favors the ones with higher similarity. In the second, SDFuzz adopts the distance metric feedback. SDFuzz solves the limitations of prior distance metrics with a new precisely-weighted inter-procedural distance metric.

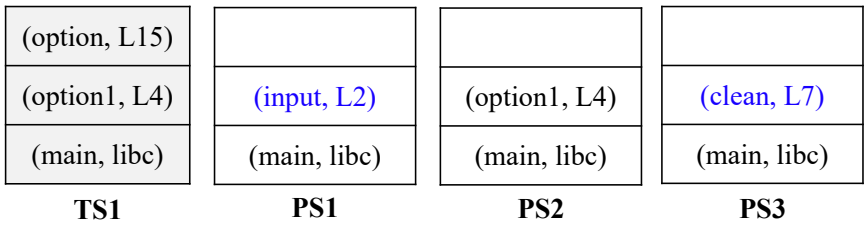


Figure 3.4: Illustration of target state and selected program states. The root deviations are depicted in blue.

3.1.4.2 Extracting Target States

To extract the target states, SDFuzz requires either vulnerability reports or static analysis results. For the latter, SDFuzz employs an existing static analysis tool to analyze the program's source code.

Vulnerability reports. The crash dump consists of the records of the active function calls when a vulnerability is triggered, as shown in Figure 3.2(a). Each record contains: 1) function name (*e.g.*, option1) and 2) invocation location (*e.g.*, file.c:15). Therefore, we first search the vulnerability reports about descriptions containing such information using regular expressions. After extraction, we further parse them to decide if they match the formats defined in Equation 3.1. We also automatically sort the target states based on the vulnerability types and the descriptions. For example, a use-after-free vulnerability often contains multiple target states. We would sort the target states sequentially by the free and the use site.

Static analysis results. SDFuzz also automatically extracts the target states from the static analysis results. Since different static analysis tools take diverse ways to represent their results, naturally, the automated extraction has to be specially designed for each static analysis tool. We currently develop SDFuzz to coordinate with one popular static analysis tool, SVF [170, 171]. To the best of our knowledge, target states can be extracted from other static analysis tools such as CodeQL [68] and Joern [195] with only additional efforts to parse the results. The program's source code is usually required for running such static analysis tools.

Algorithm 1: Required code identification.

```
1  input  : TSs, ICFG
2  output: requiredFuncs
3  initRequiredFuncs ← [ ]
4  requiredFuncs ← [ ]
5  for TS ∈ TSs do
6      for f ∈ TS do
7          initRequiredFuncs.insert(f)
8          funcs ← backwardAnalysis(f, ICFG) // get functions with intra-procedural
             dependencies
9          initRequiredFuncs.insert(funcs)
10 while ! initRequiredFuncs.empty() do
11     f ← initRequiredFuncs.remove()
12     if f ∉ requiredFuncs then
13         requiredFuncs.insert(f)
14         callees ← getCallees(f, ICFG) // get callees of f
15         initRequiredFuncs.insert(callees)
16 return requiredFuncs
```

3.1.4.3 Selective Instrumentation of Required Code

SDFuzz reduces the fuzzing scope by selectively instrumenting only the required code for coverage feedback. SDFuzz first identifies what part of the code is required and then deliberately excludes the other unrequired code from the fuzzing process. Our solution preserves the code required for achieving the target states, which is a subset of the code for reaching the target sites as preserved in SieveFuzz and Beacon. This is because the target states further constrain the paths to reach target sites. It thus can help filter out much more code and improve fuzzing throughput. Our solution selectively instruments the required code for code coverage feedback instead of directly removing it from the source or executable.

We propose a function-level algorithm shown in algorithm 1 to identify the required code. It takes as inputs a set of target states (TSs) and the ICFG of the target program (ICFG). The functions appearing in the target states (namely target state functions) are related to the vulnerabilities and are directly included as the required functions (line 5). Besides, these target state functions might depend on other functions. Our algorithm first performs a backward intra-procedural analysis to identify the functions that a target state function depends on (line 6). A function is included if there is an intra-procedural path between the basic block that has a function call site and the basic block

of a target state function. For instance, function check() is included because function target() at L15 depends on it. Additionally, those newly included functions might invoke other functions to accomplish their functionalities. Therefore, our algorithm analyzes the ICFG (especially the CG) and includes those functions on the CG paths outing from the initially included functions (lines 14-15). In this way, SDFuzz expands the set of functions required for realizing the target states. The callee functions of check() are added to the required code for this.

Instead of directly removing the code from the target executable, SDFuzz employs a instrumentation-based method to exclude unrequired code. We find that DGF requires collecting code coverage feedback, which reveals the existence of code areas. SDFuzz thus selectively instruments only the identified required code for code coverage feedback, which hides the other unrequired functions from the fuzzer and reduces the fuzzing scope. This design is fault-resilient. Even if some code areas are wrongly identified as unrequired, executions can still go through such code areas to further approach the target sites and states. SDFuzz would not assign testing energy to explore the paths that are not instrumented. Thus it gets rid of the critical downside caused by false code elimination in prior solutions [83, 167]. It also reduces the overhead caused by the instrumented coverage tracking code.

3.1.4.4 Early Termination of Executions

SDFuzz uses a new technique to early abort the executions that cannot achieve the target states. If some executions are known to be unsuccessful in achieving the target states, we can terminate them early to save the exploration resources. This can dramatically increase the fuzzing throughput by terminating unnecessary executions early. Unlike prior reachability-based execution termination approaches, SDFuzz also terminates reachable executions that cannot achieve the target states.

To perform the early termination of executions, we have to predict if the execution can ultimately achieve the target states or not. This is extremely difficult because the program state is dynamically updated along the program execution, *e.g.*, via function invocations and exits. Given the high complexity of modern programs, the program state space they can exhibit is incredibly huge.

An Unrecoverable Deviation Based Solution

SDFuzz periodically retrieves the runtime program state when the execution runs to the functions in target states, and checks if to early terminate the execution. We model the runtime program states as the call traces and their invocation locations. As mentioned in §3.1.3.3, target states for a vulnerability are a list of ordered program states, each for a target site; some of them might have been achieved in the earlier execution of a test case. Our algorithm (shown in algorithm 2) thus takes as inputs the current program state (PS) at a point in time, previously achieved target states (achievedTSs), the ordered target states (TSs), and the ICFG. It iterates over the target states to find the first target state that has not been achieved during a fuzzing trial of a test case (lines 3-6). If all target states have been achieved, the algorithm directly returns (lines 7-8). Otherwise, it then checks the deviation function calls, especially the first deviation—*root deviation* through rootDeviation function (line 10). The root deviation denotes where the program state starts to deviate from the unachieved target state. This is done by comparing the runtime program state to the target state (lines 20-26).

If there is any deviation (line 11), our algorithm further measures if the remaining execution can recover the deviations to achieve the target state based on the ICFG (line 12). If an execution has unrecoverable deviations in its program state, it can be immediately terminated. Our algorithm checks the ICFG of the program and probes if there is a program path from the root deviation code location to the expected function call in the target state. Such a path means the deviations might be recovered in the future execution because the execution can return from the root deviation function call and run to the expected one. Accordingly, the execution that might recover the deviation would not be terminated.

We illustrate the workflow of our algorithm with three program states of an execution (namely PS1-PS3) listed in Figure 3.4. PS1 is observed when the execution reaches right after line 2. The program state deviates from the target state (*i.e.*, TS1) in the second item, *i.e.*, (input, L3) *v.s.* (option1, L4). The deviation is possibly recoverable because subsequent execution might return from the function input() and run next to the expected function option1() at L4. From the perspective of ICFG, this can be reflected as the existence of a program path from the deviation location (*e.g.*, L3) to the expected

Algorithm 2: Execution termination and target state similarity.

```
1  input  : PS, TSs, achievedTSs, ICFG
2  output: score, termination, achievedTSs
3  nextTS ← null
4  termination ← false
5  for TS ∈ TSs\achievedTSs do
6      nextTS ← TS // find next target state
7      break
8  if nextTS = null then
9      return 1, false, achievedTSs
10 deviationIdx ← rootDeviation(PS, nextTS)
11 if deviationIdx ≠ nextTS.size then
12     termination ← ! ICFG.path.exists(PS[deviationIdx], nextTS[deviationIdx]) // check
         if recoverable
13     score ← (deviationIdx / nextTS.size + achievedTSs.size) / TS.size
14 else
15     achievedTSs.insert(nextTS) // no deviation
16     score ← achievedTSs.size / TSs.size
17 return score, termination, achievedTSs
18
19 function rootDeviation(PS, TS):
20     index ← 0
21     for index<min(PS.size, TS.size) & PS[index]=TS[index] do
22         index ← index + 1
23     return index
```

one (*e.g.*, L4). Thus the execution would not be terminated at PS1. At PS2, the program state is exactly the prefix of TS1 without additional deviations and does not deviate from TS1. The execution would not get terminated. However, in the case of PS3, it deviates at (clean, L7) against the (option1, L4) in TS1, and there is no path from L7 to L4. The execution would get terminated. Beacon's solution [83], on the other hand, is not able to terminate the execution in the middle.

3.1.4.5 Two-Dimensional Feedback

Target State Feedback

SDFuzz also compares the runtime program states to the target states and computes a similarity score to proactively guide the exploration. The feedback favors test cases with more similar program states to the target states. Some recent works such as CAFL [100], LOLLY [110], and Hawkeye [30] also use the runtime program behaviors. However, the

mechanism in SDFuzz is tailored to target states and considers calling contexts.

The workflow is also shown in algorithm 2. After finding the first unachieved target state (nextTS) at lines 3-6, SDFuzz uses the index of the root deviation to compute the similarity score. If the current program state does not fully match the first unachieved target state (nextTS), SDFuzz first measures how well the current program state fits it by computing the ratio of matched $\mathrm{deviationIdx}$ over its size (line 13). Our algorithm also considers previously achieved target states and sums a score of the ratio and the size of $\mathrm{achievedTSs}$. The score is further normalized using the number of target states and returned. If the current program state matches nextTS, our algorithm directly returns the proportion of achieved target states (line 16). Since the algorithm might be invoked multiple times for the execution of a test case, we assign the best score as the result of the test case.

Distance Feedback

SDFuzz also uses a distance metric to guide the fuzzing process. Prior distance metrics are imprecise because they consider every edge in CG equally. They empirically configure a constant weight (*e.g.*, 10 in AFLGo-based directed fuzzers) to approximate the chance for reaching the target functions [30, 144]. Therefore, executions exhibiting long call chains, even with high chances to reach the target functions, are possibly assigned with large distance values and get deprioritized. Distinct functions ought to be evaluated differently.

SDFuzz mitigates the imprecision with precise edge weights when computing inter-procedural distances. The edge weight is expected to reflect the chance for the caller function to invoke the callee function. SDFuzz computes the edge weight based on the call-site weights. We define the *call-site weight* for a caller function to invoke a callee function as the intra-procedural distance from the start of the caller function to the call site of the callee. Since there might be multiple call sites to the same callee function, the inter-procedural *edge weight* is the shortest call-site weight ($weight(f_i, f_j)$) between the caller function $f_i()$ and the callee function $f_j()$. This is also shown in Equation 3.2, where $d_{f_i}()$ computes the intra-procedural distance in function f_i. For the function $\mathrm{option1}()$ in Listing 3.1, since the function start and the call site of the function $\mathrm{check}()$

reside in the same basic block, their edge weight is 0 instead of 10 as in AFLGo.

$$weight\left(f_i, f_j\right) = min\left(d_{f_i}\left(BB_{f_i-start},\ BB_{f_j-call-site}\right)\right) \tag{3.2}$$

The edge weights between callers and callees form a weighted CG. This allows SDFuzz to compute precise CG distance between two arbitrary functions. We formalize the method to compute inter-procedural distance in Equation 3.3. If there is at least one path from function f_s to function f_e in the CG, their distance is calculated as the sum of edge weights in the shortest path. Otherwise, if there is no path from function f_s to function f_e, the distance is considered as not available or infinite. We explain how we construct CG in §3.1.5.

$$interDistance\left(f_s, f_e\right) = min\left(\sum_{(f_i,f_j)\in path} weight\left(f_i, f_j\right)\right) \tag{3.3}$$

Seed Selection and Power Scheduling

SDFuzz incorporates the two dimensions of feedback to guide the seed selection and power scheduling. To drive the fuzzing towards the target states quickly, SDFuzz sorts the seeds in the corpus sequentially by two attributes—target state feedback and seed distance. Generally, SDFuzz prefers seeds with better target state feedback and shorter distances. It uses target state feedback as the primary sorting attribute and distance as the secondary. The reason is that the target state feedback capturing the runtime context is more precise, and could better help approach target states. SDFuzz also refines the power scheduling algorithm of AFLGo to assign energy to the seeds according to the two-dimensional feedback.

3.1.5 Implementation

We implement a prototype of SDFuzz to fuzz C/C++ programs. SDFuzz first uses static analysis to build a program representation to facilitate the required code identification and execution termination. SDFuzz employs a compile-time analysis to instrument the target program, which inserts the necessary code for tracking coverage, maintaining

program states, computing seed distances, *etc.* Besides, SDFuzz also uses a runtime library to retrieve runtime program states and perform selective execution termination. The main fuzzing component was implemented atop AFLGo [144].

Static analysis. SDFuzz statically analyzes the program to identify the required functions (§3.1.4.3). SDFuzz leverages Andersen's points-to analysis to identify the call targets for indirect calls [158]. Our implementation currently reuses the associated pipeline of SVF [170]. This results in the ICFG (particularly CG) for our analysis. This part is implemented as LLVM passes. We also discuss the potential issues of static analysis in §3.1.7.

Instrumentation. We maintain the list of interesting functions obtained from §3.1.4.3 and selectively instrument them for code coverage feedback. Other functions not in the list are still preserved but are not instrumented for code coverage feedback. As a result, they can still be executed. The instrumentation component is realized as an LLVM pass. Besides, SDFuzz also inserts code to maintain the runtime program call stack in the shared memory.

Fuzzing. SDFuzz employs a runtime library to perform execution termination and target state feedback. In the library, SDFuzz maintains a call stack and function invocation locations. At runtime, SDFuzz periodically invokes the library to check the runtime program state and compare that to the target states. The runtime library thus can early terminate some executions. Besides, SDFuzz also scores the runtime program states to provide target state feedback. We modify the $\mathrm{update\_bitmap\_score()}$ and $\mathrm{cull\_queue()}$ functions in AFL to achieve our seed selection strategies.

3.1.6 Evaluation

We extensively evaluate SDFuzz to answer the following questions.

- What is the capability of SDFuzz in exposing vulnerabilities?
- How effective can SDFuzz reduce unnecessary exploration?
- How do the techniques in SDFuzz contribute to its performance?
- How effective is SDFuzz in discovering new vulnerabilities?

Table 3.1: Vulnerability exposure results of SDFuzz.

Case #	Program	Location	AFLGo			WindRanger			Beacon			SieveFuzz			SDFuzz
			Time	Factor	p-val	Time	Factor	p-val	Time	Factor	p-val	Time	Factor	p-val	Time
1	libming	decompile.c:349	216	2.45	0.003	195	2.22	0.002	147	1.67	0.001	199	2.26	0.007	88
2	libming	decompile.c:398	268	1.71	0.008	348	2.22	0.003	194	1.24	0.05	282	1.80	0.03	157
3	LMS	service.c:227	5	1.67	0.009	8	2.67	0.006	3	1.00	0.001	3	1.00	0.001	3
4	mjs	mjs.c:13732	272	1.36	0.132	204	1.02	0.012	128	0.64	0.003	228	1.14	0.023	200
5	mjs	mjs.c:4908	8	2.67	0.007	5	1.67	0.004	5	1.67	0.006	3	1.00	0.001	3
6	tcpdump	print-ppp.c:729	608	4.68	0.004	708	5.45	0.003	CE	-	-	512	3.94	0.003	130
7	lrzip	stream.c:1747	372	18.60	0.005	251	12.55	0.003	38	1.90	0.001	176	8.80	0.003	20
8	lrzip	stream.c:1756	329	7.48	0.002	224	5.09	0.001	158	3.59	0.003	137	3.11	0.009	44
9	objdump	objdump.c:10875	785	5.38	0.002	752	5.15	0.008	235	1.61	0.003	327	2.24	0.003	146
10	objdump	dwarf2.c:3176	TO	-	-	618	7.92	0.001	CE	-	-	154	1.97	0.019	78
11	libssh	messages.c:1001	TO	-	-	TO	-	-	TO	-	-	TO	-	-	1,112
12	libxml2	valid.c:952	151	2.44	0.009	42	0.68	0.004	52	0.84	0.003	70	1.13	0.001	62
13	libxml2	messages.c:1001	217	1.43	0.003	209	1.38	0.002	78	0.51	0.003	192	1.26	0.018	152
14	libxml2	parser.c:10666	134	3.35	0.012	211	5.28	0.007	TO	-	-	78	1.95	0.009	40
15	libarchive	format_warc.c:537	TO	-	-	TO	-	-	TO	-	-	TO	-	-	1,039
16	Little-CMS	cmsintrp.c:642	382	2.98	0.003	565	4.41	0.003	229	1.79	0.001	258	2.02	0.004	128
17	boringssl	asn1_lib.c:459	511	4.26	0.006	368	3.07	0.004	263	2.19	0.003	346	2.88	0.006	120
18	c-ares	ares_create_query.c:196	3	3.00	0.019	3	3.00	0.122	1	1.00	0.151	1	1.00	0.132	1
19	guetzli	output_image.cc:398	42	10.50	0.030	51	12.75	0.003	17	4.25	0.004	25	6.25	0.012	4
20	harfbuzz	hb-buffer.cc:419	TO	-	-	TO	-	-	TO	-	-	1,350	2.13	0.001	633
21	json	fuzzer-parse_json.cpp:50	8	4.00	0.013	19	9.50	0.003	3	1.50	0.001	5	2.50	0.003	2
22	woff	buffer.h:86	519	1.46	0.019	638	1.79	0.003	389	1.09	0.001	443	1.24	0.003	356
23	vorbis	codebook.c:479	TO	-	-	TO	-	-	198	0.78	0.001	TO	-	-	254
24	re2	nfa.cc:532	1,121	12.18	0.005	654	7.11	0.003	157	1.71	0.001	465	5.05	0.005	92
25	pcre	pcre2_match.c:5968	55	4.23	0.001	30	2.31	0.001	8	0.62	0.005	27	2.08	0.005	13

Factor is the ratio of time used by a tool compared to that of SDFuzz.
CE denotes compilation error.
TO denotes that a tool reaches the time limit (timeout) before triggering a vulnerability.
The best result of a case is underlined.

3.1.6.1 Dataset

We use real-world vulnerabilities from Google Fuzzer Test Suite [77] and AFLGo's Test Suite [145] to evaluate SDFuzz's capability of triggering (known) vulnerabilities. In total, we are able to include 25 unique vulnerabilities in our dataset, and we list them in Table 3.1. These vulnerabilities have been used for the evaluation of prior works [30, 144]. The included vulnerabilities span a comprehensive set of vulnerability types such as buffer overflow, heap overflow, *etc.*, and can well evaluate the capability of SDFuzz. Other vulnerabilities are excluded from our evaluation mainly because we cannot compile them in our environments. All the experiments are conducted on a server running Ubuntu 18.04 with two 18-core Intel Xeon Gold 6140 CPUs and 256GB RAM.

3.1.6.2 Performance of SDFuzz

Target states and seeds. We prepare the target states and the seed inputs for the experiments. We first find the origin of the vulnerability report and extract the target states. SDFuzz successfully extracted the target states for all the cases, demonstrating its advantage of full automation. We then use SDFuzz to test the vulnerabilities for five runs, each with a 24-hour time limit. For the vulnerabilities in Google's Fuzzer Test Suite, we use the seed inputs (if available) provided in the repository; we use empty seed inputs for other cases.

Required code identification. Our selective instrumentation technique can significantly reduce the fuzzing scope to the required code. We first analyze the proportion of the required code that SDFuzz identified for the 25 evaluated vulnerabilities. In particular, SDFuzz eliminated 42.39% of unrequired functions on average and narrowed down the fuzzing scope to the other 57.61% of required functions. For several cases (*e.g.*, #24 in re2), SDFuzz could even eliminate over 80% of unrequired functions and trigger the vulnerabilities.

Vulnerability exposure. We measured the time used for exposing the known vulnerabilities and presented the evaluation results in Table 3.1. SDFuzz could reproduce all 25 vulnerabilities within the time limit of 24 hours (1,440 minutes). SDFuzz could trigger the majority (20 out of 25) of the vulnerabilities within four hours. This demonstrates

the high effectiveness of SDFuzz in exposing known vulnerabilities.

3.1.6.3 Comparison with Other Approaches

We compared SDFuzz to existing directed fuzzers on the same dataset. We thoroughly investigated the literature and included the state-of-the-art open-sourced directed fuzzers as the comparison targets: AFLGo [144], WindRanger [58], and SieveFuzz [167]. Beacon [83] is publicly available in the form of binary [199]. We additionally used the provided binary and included it in the comparison. We could not add some other related works (*e.g.*, Hawkeye [30], CAFL [100]) mainly because they are not open-sourced. We ran these fuzzers also for five 24-hour runs and reported the average vulnerability exposure time. Since these works do not use target states, we acknowledge that our comparison with them can hardly be perfectly fair. Nevertheless, we tried our best to conduct a fair comparison by strictly following the instructions provided by the compared tools, and using the same hardware environments and initial seeds.

Vulnerability detection. The comparison results are shown in Table 3.1. *SDFuzz generally outperformed other directed fuzzers with more vulnerabilities exposed.* Specifically, AFLGo, WindRanger, Beacon, and SieveFuzz exposed 20, 21, 19, and 22 vulnerabilities, respectively. The numbers of the exposed vulnerabilities are fewer than SDFuzz's. We failed to compile two cases (#1 and #10) using Beacon's compilation tool (shown as CE). Beacon did not trigger additional four cases.

We characterize the time each tool used for triggering the vulnerabilities. *SDFuzz used a shorter time than the compared directed fuzzers in most of the exposed cases.* We compute a factor value as the ratio of the time used by a tool to that of SDFuzz in each case. The factor value also describes the performance speedup between tools. The factor value is not available (shown as - in Table 3.1) if a fuzzer does not trigger the vulnerability. We further compute the average speedup as the arithmetic mean of the factor values for those exposed vulnerabilities (excluding CE and TO cases). In general, SDFuzz achieved an average speedup of 4.79×, 4.63×, 1.57×, and 2.58× above AFLGo, WindRanger, Beacon, and SieveFuzz, respectively. SDFuzz also outperformed AFLGo, WindRanger, Beacon, and SieveFuzz by up to 18.60×, 12.55×, 4.25×, and 8.80×, respectively. The best result per vulnerability is underlined in Table 3.1. We observe that

SDFuzz performed the best in most of the cases (20 out of the 25), demonstrating the effectiveness of our new techniques.

We further employed a Mann-Whitney U test [118] on vulnerability exposure time to measure the statistical significance of our experiment results. We find that our results are significant in the majority of the cases under the significance level of 0.05 (*i.e.*, most cases in Table 3.1 have a p-value less than 0.05). Therefore, given the high diversity of the vulnerabilities in our dataset, we confidently conclude that SDFuzz could trigger vulnerabilities more quickly.

We observe that several vulnerabilities were significantly hard for prior directed fuzzers to trigger, which SDFuzz successfully triggered. For instance, cases #11 and #15 were not exposed by all other four evaluated directed fuzzers; cases #20 and #23 were not triggered by three of the other evaluated directed fuzzers. We investigated the source code and the historical input queue and identified the reason why SDFuzz could more easily trigger them. These four cases generally have a significantly large number of paths on the CG from the entry function to the target function(s). A large proportion of them is not feasible dynamically due to the unsatisfiable path constraints. For example, in case #11, a buffer overflow vulnerability, other directed fuzzers could not reproduce it within the time limit. They favored wrong paths and tested this vulnerability in a wrong direction, where the vulnerability-triggering conditions were hardly achieved. SDFuzz, however, chose a feasible path that was more likely to approach the target state. The analysis suggests that the target states provided additional guidance, which directed SDFuzz to trigger the vulnerabilities, whereas the other fuzzers easily got stuck in wrong directions.

Code elimination. SieveFuzz [167] uses a code elimination technique based on target site locations and is open-sourced. We also investigated how well SieveFuzz eliminated code. It removed around 29% of unrequired code on average, which is 46.17% less than what SDFuzz eliminated. This demonstrates the benefits of the target state information for code elimination. We were not able to check such statistics for Beacon [83] since the authors only released its binary, and we could not enhance it to obtain such internal statistics.

Path pruning and fuzzing throughput. The effectiveness of path pruning can be

reflected in the fuzzing throughput, *i.e.*, number of executions per second. We find that *SDFuzz achieved a much higher fuzzing throughput.* Since different programs normally have distinct processing time, we thereby calculate a throughput factor value as the ratio of the throughput of a tool to that of AFLGo on each case. We then compute the arithmetic mean on all cases as the average throughput. On average, SDFuzz, WindRanger, Beacon, and SieveFuzz had the throughput factor value of 34.31, 0.87, 4.94, and 17.32, respectively. This shows that fuzzers employing execution termination techniques have higher throughput compared to the ones with only distance metrics. For instance, SDFuzz, Beacon, and SieveFuzz had higher throughput than the other two. Besides, SDFuzz, driven by target states, achieved the highest fuzzing throughput in the evaluated vulnerabilities. This could be explained as its execution termination to also abort some reachable executions. As for WindRanger, it underperformed AFLGo mainly because of its taint analysis overhead.

Vulnerability-triggering paths. *We found that other fuzzers mostly triggered the vulnerabilities through the identical paths that SDFuzz derived from the target states.* Specifically, we replayed the crashing input that a fuzzer generated to expose a vulnerability and analyzed the triggered program path. We then correlated such paths to the ones in the target states. SDFuzz triggered the vulnerabilities through these paths. On the other hand, AFLGo, WindRanger, Beacon, and SieveFuzz ultimately used the paths in the target states for 15, 12, 11, and 16 cases, respectively. Such an observation has two implications. First, SDFuzz could directly drive the exploration towards such paths without sparing too much effort on other paths. This is the root reason why SDFuzz could have superior performance compared to other directed fuzzers. Second, by driving towards the target states, though SDFuzz could potentially overlook some other paths, this would not significantly undermine the performance of SDFuzz. Therefore, we believe SDFuzz could significantly benefit state-of-the-art crash reproduction.

3.1.6.4 Component-Wise Analysis

We conduct an ablation study on the same dataset to understand how each technique in SDFuzz contributes to the performance. We design four variants for the component-wise evaluation. Since SDFuzz is built atop AFLGo, each variant enables one key tech-

Table 3.2: Speedup achieved by SDFuzz above AFLGo.

Case #	SDFuzz$_{si}$	SDFuzz$_{et}$	SDFuzz$_{sf}$	SDFuzz$_{df}$	SDFuzz
1	1.16	1.95	1.27	1.08	2.45
2	1.00	2.11	2.11	1.34	1.71
3	1.25	1.67	1.00	1.00	1.67
4	1.30	1.17	1.17	1.08	1.36
5	2.00	2.00	0.80	1.60	2.67
6	1.24	3.22	1.12	1.16	4.68
7	2.10	9.79	3.19	2.49	18.60
8	2.19	3.58	1.39	1.36	7.48
9	2.02	2.42	1.54	1.32	5.38
10	✓	✓	×	×	✓
11	×	×	×	×	✓
12	1.96	1.94	1.26	1.16	2.44
13	1.09	1.15	1.09	1.14	1.43
14	1.72	1.97	1.12	1.54	3.35
15	×	×	×	×	✓
16	1.28	1.49	1.07	1.06	2.98
17	1.38	4.02	1.94	1.13	4.26
18	1.50	1.50	1.50	1.50	3.00
19	1.08	1.16	0.78	1.14	10.50
20	×	✓	×	×	✓
21	1.60	1.33	1.12	1.04	4.00
22	1.06	1.30	1.60	1.00	1.46
23	×	✓	×	×	✓
24	6.96	5.66	2.38	2.12	12.18
25	1.28	1.90	1.47	1.08	4.23
Avg.	1.71	2.57	1.45	1.26	4.79

✓ and × denote the variant (or SDFuzz) triggers or not the vulnerability, respectively, when AFLGo does not trigger it.

nique over AFLGo. In particular, SDFuzz$_{si}$, SDFuzz$_{et}$, SDFuzz$_{sf}$, and SDFuzz$_{df}$ enables only selective instrumentation, execution termination, target state feedback, and distance feedback, respectively.

We use the same experimental setup in §3.1.6.2 to run the four variants. We measure the average time used to trigger the vulnerabilities for successful runs. We show the speedup of each variant above the baseline tool, AFLGo, in Table 3.2. We cannot compute the speedup for five cases when AFLGo does not trigger the vulnerability. We thus use ✓ and × to denote if the variant or SDFuzz triggers or not the vulnerability, respectively. Generally speaking, the results validate the effectiveness of the four key techniques. We next analyze the results in detail.

Selective instrumentation. SDFuzz$_{si}$ and SDFuzz share the same required code

identification step. Therefore, SDFuzz$_{si}$ could eliminate the same proportion of code as SDFuzz. We have shown the results in §3.1.6.2 that SDFuzz could eliminate 42.39% of unrequired functions on average, demonstrating its high effectiveness.

From Table 3.2, we find SDFuzz$_{si}$ improved AFLGo by exposing one additional vulnerability and triggered 21 out of the 25 vulnerabilities in total. SDFuzz$_{si}$ improved the performance of AFLGo, with an average speedup of 1.71× on the 20 vulnerabilities AFLGo exposed. We also found SieveFuzz [167] had better performance than SDFuzz$_{si}$ on the vulnerability exposure time even though SDFuzz$_{si}$ eliminated more unrequired code. It achieved an average speedup of 2.05× above AFLGo. The reason is two-fold. First, SDFuzz$_{si}$ sacrifices a bit the performance for fault tolerance by its instrumentation-based code elimination. Second, SieveFuzz also employs other techniques such as diversity heuristics SDFuzz$_{si}$ currently does not support. Integrating such techniques would definitely improve our solution.

Early termination of executions. The execution termination technique in SDFuzz$_{et}$ is highly effective. For 24-hour experiments on the 25 vulnerabilities, SDFuzz$_{et}$ early terminated 60.38% of the executions on average. This also confirms that executions that cannot achieve the target states are widespread in real-world programs. We could not measure the proportion of terminated executions in Beacon because it is close-sourced. Besides, SDFuzz$_{et}$ also improved the fuzzing throughput by 26.71× on average, compared to AFLGo. In terms of vulnerability exposure, SDFuzz$_{et}$ triggered four more vulnerabilities than AFLGo, exposing a total of 23 vulnerabilities. SDFuzz$_{et}$ achieved an average speedup of 2.57× above AFLGo on the 20 vulnerabilities AFLGo triggered.

Feedback mechanism. The feedback mechanism guides the testing to trigger the vulnerabilities. SDFuzz$_{sf}$ triggered one more vulnerability compared to AFLGo. It also accelerated the vulnerability exposure by 1.45×. The rationale lies in that SDFuzz$_{sf}$ appropriately spares the testing efforts to desired paths specified in target states. However, we notice that SDFuzz$_{sf}$ performed slightly worse than AFLGo in cases #5 and #19. The reason mainly came from the runtime overhead caused by the program state maintenance. Since the two cases are relatively simple to trigger, the overhead turned to take a significant proportion in the total used time.

Similarly, the precisely-weighted distance metric feedback in SDFuzz$_{df}$ also improved

Table 3.3: Vulnerability discovery results of SDFuzz.

Program	Statically Reported	Validated
libjpeg	46	2
tinyexr	22	1
pugixml	59	1
ffmpeg	32	0
Total	159	4

fuzzing effectiveness. Compared to AFLGo, SDFuzz$_{df}$ triggered one more vulnerability and achieved an average speedup of 1.26×. However, we notice that the performance improvement of SDFuzz$_{df}$ is less significant than that of SDFuzz$_{sf}$. Our manual analysis found that the new distance metric would not significantly optimize the exploration direction to eliminate unnecessary exploration. The target state feedback, on the other hand, could offer such benefits.

3.1.6.5 New Vulnerability Discovery

We further evaluate the efficacy of SDFuzz in discovering new vulnerabilities. In particular, we explore the feasibility of applying SDFuzz to automatically validate the analysis results of SVF [171]. We apply SVF to a set of well-tested applications that handle or process different types of files, such as Binutils, libjpeg [175], tinyexr [174], pugixml [203], and ffmpeg [60]. Some of the applications have been well-tested by the related tools [58, 144].

We employed the saber checker [171] of SVF to identify memory leakage and double-free vulnerabilities. In total, SVF reported 159 suspicious cases, and we leveraged SDFuzz to validate them. SVF provided a program flow for each suspicious case, which SDFuzz converted to a target state for directed fuzzing. Given the large number of cases, we ran SDFuzz for 12 hours for each case, resulting in a total CPU time of around $2{,}000$ hours. To date, SDFuzz successfully identified four new vulnerabilities. We responsibly disclosed the new vulnerabilities to the vendors. Three of the vulnerabilities have been acknowledged. The detailed vulnerabilities can be found in Table 3.3. Given this, we believe that SDFuzz can be applied in practice as a fully automated solution for vulnerability validation.

To understand the efficacy of target states, we further replayed the crashing inputs

generated by SDFuzz on the four vulnerabilities and analyzed the triggered program paths. Our investigation revealed that the four vulnerabilities were triggered through the exact paths reported by SVF (*i.e.*, paths derived from the target states). This confirmed that the target states could help validate the static analysis results.

3.1.7 Discussion

In this section, we discuss the limitations of SDFuzz and future work.

Requirement of target states. SDFuzz requires target states from two sources—vulnerability reports and static analysis results. This requirement can be satisfied in practice as most of the vulnerability reports provide such target states (§3.1.3). Even if vulnerability reports are not available, SDFuzz can still leverage existing static bug analysis techniques to identify the target states. Compared to exploring all paths, the paths in static analysis results often convey the suspicious flows that better deserve testing. We have already demonstrated the feasibility in §3.1.6.5.

We admit that some application scenarios such as patch testing might not provide available target states. Our solution can extract the target states from the vulnerability where the patch applies by analyzing the vulnerability report. It can also analyze the patch change logs to identify the target reaching order [100]. However, it currently cannot directly derive complete target states from only the patch change logs, *e.g.*, find the problematic program paths in the patches. To mitigate this problem, one possible direction is to employ advanced program analysis techniques such as symbolic execution [27, 35] to synthesize feasible target states for a patch. We leave this as future work.

States other than target states. A downside of our approach is that it might potentially overlook some valuable paths that are not included in the target states, only in crash reproduction. In fact, as shown in our evaluation, SDFuzz could trigger the vulnerabilities in a significantly shorter time. This suggests that overlooking other states would not undermine the performance of SDFuzz.

We believe driving to target states is a reasonable trade-off for two reasons. First, infeasible paths for triggering the vulnerabilities dominate the program paths [206]. The

paths stated in the target states are preferred working ones. Our solution would mostly exclude unnecessary exploration and guide the fuzzer towards these guaranteed working directions. Second, the paths in target states are likely to be the best or simplest ones to trigger the vulnerability as they correspond to the first cases when a vulnerability gets triggered or reported in real world. Other fuzzers ultimately triggered the vulnerabilities also through the paths in the target states. This confirms the rationale of using target states.

In the application scenario of validating static analysis results, the task of applying directed fuzzing is to test the reported suspicious flows. Therefore, our strategy of focusing on the target states (suspicious flows) is reasonable for achieving this goal.

Incomplete call graph. We notice that static analysis is not always precise because some call targets cannot be statically inferred. We admit that this might result in an incomplete call graph. SDFuzz might incorrectly assess the state recovery capability because of wrong path existence judgment. Some executions might be wrongly terminated. This is also a common limitation of all execution termination based DGFs. To mitigate the issue, we can incorporate dynamic tracing to monitor function invocations, and accordingly add additional call edges to the call graph. We will leave it as future work.

3.1.8 Related Work

Eliminating unnecessary explorations can improve fuzzing effectiveness. Beacon [83], SieveFuzz [167], and SelectFuzz [115] are three typical examples considering the reachability to the target sites. As we discussed in §3.1.2.3, SDFuzz can further exclude unnecessary reachable executions that cannot achieve the target states.

The distance metric is also an important factor in directed fuzzing. AFLGo [144] and SemFuzz [201] were the first lines of research about directed fuzzing. They initially proposed the concept of distance metrics to drive the coverage-guided fuzzing towards a direction. Hawkeye [30] further used adjacent-function distance, which considered the number of times a function is called. ParmeSan [137] directed the testing towards locations with more sanitization checks. WindRanger [58] pointed out that only deviation

basic blocks were necessary for distance computation. MC2 [157] approached a new randomized search algorithm to optimize the fuzzing exploration. SDFuzz differentiates itself from these tools by considering the target states for triggering vulnerabilities. Additionally, LOLLY [110] analyzed the execution trace of a test case after its execution and used that as feedback. Its analysis was purely offline, *i.e.*, it did not capture the runtime program behaviors like the calling contexts. It also introduced heavy overhead in recording the complete execution trace in every fuzzing trial. SDFuzz instead selectively monitors the runtime program states with minimal overhead. CAFL [100] measured the order dependency among targets—a part of the target states. SDFuzz considers a comprehensive set of the target states for the feedback.

Unlike SDFuzz, some related works improve DGF from other angles or apply DGF to other scenarios. FuzzGuard [206] used deep learning to predict unreachable test cases and filtered them out from the testing. As for binary programs, UAFuzz [130] considered the target reaching order and detected UAF vulnerabilities; 1dVul [141] analyzed binary patches to identify one-day vulnerabilities.

3.2 MdPerfFuzz: Grammar-Aware Fuzzing for Performance Bugs

3.2.1 Motivation

Performance bugs have become an emerging attack vector for launching denial-of-service (DoS) attacks [40, 41, 43]. Such bugs could cause excessive resource consumption and negatively affect user experiences. By specially crafting inputs to exploit a performance bug on a server, attackers can exhaust the server's computing resources (*e.g.*, memory and CPU) and significantly impair the application's availability for legitimate users. Some performance bugs require only low-bandwidth traffic to exploit and can be leveraged to easily overwhelm a target system [44, 56, 122]. Detecting performance bugs could help mitigate or even prevent such DoS attacks to safeguard the normal operation of many popular and critical services on the Internet.

We seek to detect unknown performance bugs in real-world programs. We focus on CPU-exhaustion performance bugs, which are the dominant type of performance bugs. In particular, we consider leveraging fuzz testing—the go-to approach—to detect performance bugs. Fuzzing is free from some limitations of static analysis techniques *e.g.*, high false positives [86, 132, 133]. It has been extensively applied to detect thousands of vulnerabilities in various real-world software [75, 183, 202].

In this thesis, we investigate the performance issues in domain specific language compilers, which are widely used in real world. Language compilers are widely used for different domains. They serve as the foundation for a lots of applications and functionalities. The performance issues in language compilers could degrade their performance and waste computational resources. In particular, we focus on Markdown, which is an easy-to-use domain-specific markup language. Because of the flexibility Markdown offers, Markdown compilers are commonly included in many application scenarios such as code hosting software, content management systems (CMSs), online Markdown editors, *etc.* For example, two leading code hosting providers—GitHub and GitLab—support Markdown document compilation and rendering at both the server end [67, 70] and the client end [66, 69]; popular CMSs like WordPress [189] and Drupal [57] also provide

support for Markdown content rendering in the posts with their server-side Markdown compilers; among the Alexa top 1 million websites [10] are many online Markdown editors such as StackEdit [156].

Existing fuzzers for performance bugs like SlowFuzz [143], PerfFuzz [103] and SAFFRON [98] are not designed for domain-specific languages such as Markdown hence cannot efficiently generate inputs to thoroughly exercise the compilers. To efficiently fuzz Markdown compilers, a grammar-aware approach would be needed. Therefore, we study the CommonMark [37] specification and model the Markdown grammar. We then extend existing fuzzers with a syntax-tree based mutation strategy [184] specifically for Markdown. Such a mutation strategy could preserve useful Markdown syntaxes during the input mutation process, and help efficiently generate high-quality inputs to fuzz Markdown compilers.

To detect CPU-exhaustion performance bugs in Markdown compilers, we monitor the program execution under the generated inputs. We employ a statistical model using Chebyshev inequality to label abnormal cases as performance bugs. Like in prior works [103, 143], our approach can potentially lead to duplicate bug reports because different yet highly similar inputs could actually trigger the same performance bug. It is time-consuming and impractical to manually de-duplicate them. Yet it is non-trivial to automate this process in the scenario of performance bugs. Existing bug de-duplicating methods in fuzzing use coverage profile and call stack snapshots, which are not applicable to performance bugs. For example, it is hard to obtain an accurate and deterministic call stack snapshot that represents the run-time program state when a performance bug is triggered. Following prior trace analysis works [125, 177, 178, 192], we propose an execution trace similarity algorithm to de-duplicate the reports. Specifically, we represent the execution trace per report into a vector, compute the cosine similarity [53] between vector pairs, and classify highly similar vectors (bug reports) as the same bug.

We integrate the above-mentioned techniques into MdPerfFuzz, a fuzzer specialized in detecting performance bugs in Markdown compilers implemented in C/C++. With MdPerfFuzz, we successfully detected 7 new performance bugs in 2 standalone Markdown compilers. It also outperformed the state-of-the-art works with *more* detected unknown performance bugs, *higher* performance slowdown, and *higher* code coverage.

To detect performance bugs in Markdown compilers and plugins that are not written in C/C++, we further summarized the exploits generated by MdPerfFuzz into 45 attack patterns. By applying these patterns we found 209 new performance bugs, which could potentially affect millions of websites and their users. Our evaluation results demonstrate that such performance bugs were widespread and they existed in the latest stable versions of almost all Markdown compilers we tested. We further showed that such bugs can be easily exploited using only low-rate traffic to launch DoS attacks against a server, demonstrating the severity of this emerging threat.

We also revealed that vulnerable Markdown compilers implemented in different programming languages could be exploited by the same inputs because different compiler developers implemented the compilers in exactly the same buggy ways. We are in the process of responsibly disclosing our findings to the affected parties. At the time of writing, 25 bugs have been acknowledged and one new CVE ID has been assigned. To facilitate future research, we release MdPerfFuzz as an open-source software. Researchers can easily build on MdPerfFuzz their tools to study performance issues in the compilers of other domain-specific languages such as Latex and Wikitext.

In summary, we make the following contributions.

- We developed MdPerfFuzz to generate high-quality test inputs to detect performance bugs in Markdown compilers.
- We proposed an execution trace similarity algorithm to effectively de-duplicate performance bugs.
- We detected 175 new performance bugs in 17 Markdown compilers and 41 new performance bugs in 4 popular real-world applications.

3.2.2 Design of MdPerfFuzz

We try to detect unknown performance bugs in real-world Markdown compilers. We focus on CPU resource exhaustion performance bugs in this work because they are the dominant type of performance bugs.

To avoid the high false-positive rates in static analyses [132, 133], we propose to use dynamic fuzz testing to detect and exploit performance bugs. To do so, we face two technical challenges. First, generating Markdown documents to test Markdown

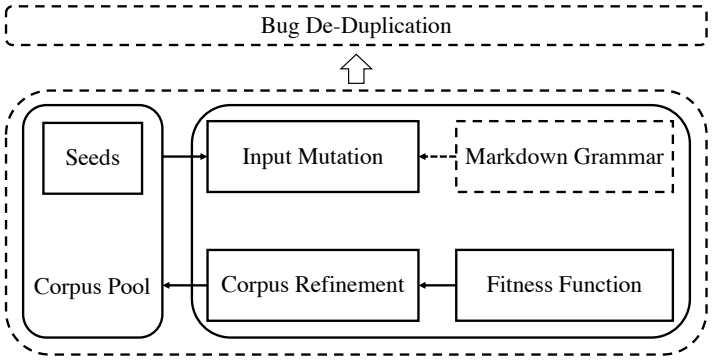


Figure 3.5: The architecture of MdPerfFuzz.

compilers and exploit the performance bugs (if any) is naturally difficult because of the huge document search space. Prior fuzzers [103, 143] are not very efficient in generating the specially formatted inputs to trigger the performance bugs in Markdown compilers (which we will discuss in §3.2.3.3). Second, since many distinct inputs can trigger the same performance bug, it is naturally challenging to accurately de-duplicate the bug reports. Prior performance bug fuzzers [21, 143] do not try to de-duplicate performance bugs. Other fuzzers for detecting *memory corruptions* de-duplicate bugs using the unique memory footprints (*e.g.*, coverage profiles and call stacks [93]) when the bugs are triggered, whereas one *performance bug* can potentially exhibit different memory footprints.

We overcome these challenges with MdPerfFuzz. The overall methodology is depicted in Figure 3.5. MdPerfFuzz follows the general fuzzing workflow and is built on top of AFL [202]. Inside the main fuzzing loop, we particularly design a grammar-aware syntax-tree based mutation strategy to efficiently generate high-quality inputs (§3.2.2.1). We first model Markdown grammar from the CommonMark specification to parse the test cases into abstract syntax trees (ASTs). The mutation strategy then mutates the ASTs while preserving the Markdown syntaxes to well exercise the diverse syntax components of Markdown compilers. To guide the fuzzer to detect CPU-exhaustion performance bugs, we use a fitness function to measure if an input should be favored or not (§3.2.2.2). The fitness function considers both code coverage and resource usage. To report only unique bugs, we compute the cosine similarity between each pair of the vector representations of the execution traces in bug reports and group highly similar reports as duplicate ones (§3.2.2.3). We then present the implementation details (§3.2.2.4).

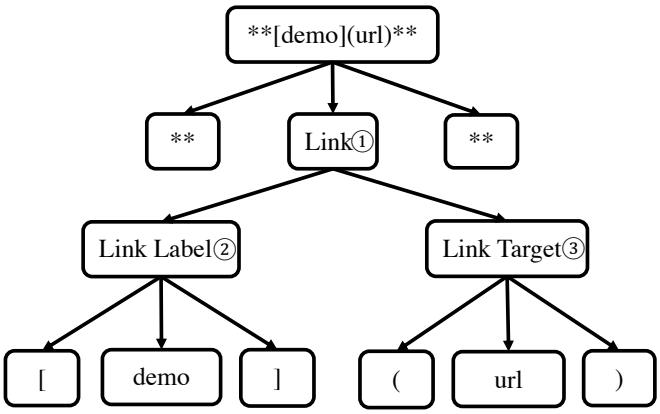


Figure 3.6: AST of the statement '**[demo](url) **' .

3.2.2.1 Syntax-Tree Based Mutation Strategy

Above the default mutation strategies of AFL (*e.g.*, bit flipping), MdPerfFuzz introduces a syntax-tree based mutation strategy to better preserve the Markdown syntaxes. Due to the complexity of Markdown language, no prior work has attempted to formalize its grammar, which is non-trivial. We spent a considerable amount of efforts on analyzing the CommonMark specification and modelling the language grammar. Given the grammar, our syntax-tree based mutation strategy parses a test case into an AST, traverses the AST, and randomly replaces several subtrees ($tree_{src}$) with candidate subtrees ($tree_{dst}$). We construct the simplest ASTs each representing a Markdown syntax and include them as candidates of $tree_{dst}$. We do not consider the combination of multiple Markdown syntaxes when constructing one candidate of $tree_{dst}$ because it can be achieved via replacing multiple $tree_{src}$. In this way, compared to mutation strategies that randomly flipping bits, our strategy preserves and extends the syntax of the original test case. Thus it can efficiently construct syntactically correct new test cases from the modified ASTs for later testing.

For example, given a test case of '**[demo](url) **' , we first parse it into the AST shown in Figure 3.6. We identify the basic subtrees (*i.e.*, ①, ②, ③) and randomly replace each of them to generate new test cases. For instance, we can replace the whole link (①) with an inline code span and produce a test case of '**` random`**'. The newly generated test case remains the text bold syntax but also exercises new syntax features. It is worth noting that the default mutation strategies of AFL can also be applied when the syntax-tree based one fails to parse a test case.

3.2.2.2 Fitness Function and Performance Bug Detection

MdPerfFuzz uses a fitness function to decide whether to favor a test case or not. We include both the coverage and the control flow graph (CFG) edge hit counts into the fitness function. As in other works [75, 93], the coverage feedback drives MdPerfFuzz to explore more newly discovered code. Only it, however, is not sufficient for our purpose as it does not consider loop iterations which are crucial for detecting high-complexity performance bugs [143]. The CFG edge hit counts, standing for the times a CFG edge is visited under a test case, enables MdPerfFuzz to explore *computationally expensive* paths. As stated in prior work [103], many programs (*e.g.*, PHP hash functions [103]) do have non-convex performance space. We thus do not use the number of executed instructions to guide MdPerfFuzz because it might fail to find the performance issues caused by local maxima. Therefore, as in the state-of-the-art work, PerfFuzz [103], we design MdPerfFuzz to favor those test cases that maximize certain CFG edge hit counts to better detect performance bugs. In this way, MdPerfFuzz tends to select test cases to either trigger new code or exhaust certain CFG edges. Note that we do not use run-time CPU usage or concrete execution time as the metric, because they show large variations affected by many uncontrollable factors, such as the fuzzer's concurrent features and the characteristics of the applications being tested.

Prior works [103, 143] rely on analysts to label performance bugs, which is time-consuming and does not scale. We thus design a statistical model to accurately identify performance bugs. Our statistical model first obtains the normal program execution behaviors, which help label abnormal ones as performance bugs. In particular, as in [103], we compute the total execution path length—the sum of the CFG edge hit counts—under a test case as the metric. We first prepare abundant random normal test cases; we feed each test case to the testing program and obtain the corresponding execution path length. We calculate the mean (l_μ) and the standard deviation (l_σ) of the execution path lengths (l_i). We label a case as a performance bug if its execution path exceeds the normal level to a certain extent. According to Chebyshev inequality (as shown in Equation 3.4), the probability of the random variable l_i that is k-standard deviations away from the mean (l_μ) is normally no more than $1/k^2$. Since only in rare cases

would the execution path length significantly deviate from the normal situations, we label a performance bug if its execution path length l_t is more than kl_σ away from the l_μ (see Equation 3.5).

$$P(|l_i - l_\mu| > kl_\sigma) \leq \frac{1}{k^2} \tag{3.4}$$

$$l_t > l_\mu + kl_\sigma \tag{3.5}$$

3.2.2.3 Bug De-Duplication

Though the execution path lengths under different test cases could all meet Equation 3.5, they could actually trigger the same performance bug. De-duplicating the bug reports is necessary for a more precise result, whereas prior works [103, 143] do not apply automated methods to de-duplicate the reports. Existing fuzzing works identify unique bugs using the call stack for memory corruptions (*e.g.*, crashes). However, it does not fit well our purposes for performance bug de-duplication. Though we can possibly collect the call stack as well (*e.g.*, by forcibly terminating the program at some point), the call stacks might not be accurate enough to represent unique bugs. This is because the exactly critical call stack for a performance bug can hardly be accurately exposed. Unlike memory corruption bugs that have deterministic call stacks when the bugs are triggered, performance bugs might exhibit diverse call stacks depending on when to obtain them. Therefore, a better bug de-duplication method is needed.

We propose a bug de-duplicating approach by merging reports with similar execution traces, similar to prior trace clustering methods [125, 177, 178, 192]. The high-level idea is that different exploiting inputs of the same performance bug should exhibit similar execution traces, *i.e.*, most CFG edges are visited in similar frequencies. In particular, we apply the test cases in the reports to the instrumented target software and obtain the CFG edge hit count (*i.e.*, number of times a CFG edge is visited in a test) for each edge. We summarize the unique CFG edges that are visited in all reports during fuzz testing into an n-dimensional vector space, where n is the total number of unique CFG edges being visited and each dimension in the vector space corresponds to a CFG edge. In other words, we construct an edge-hit-count vector, *e.g.*, $\vec{v} = (c_1, c_2, ..., c_n)$, for each report. Each dimension (c_i) represents the hit count of the ith CFG edge in that report.

To consider if two reports point to the same bug, we compute the cosine similarity [53] between their edge-hit-count vectors (*e.g.*, $\vec{v}$, $\vec{v}'$), as shown in Equation 3.6. Cosine similarity is based on the inner product of the two vectors and thus naturally assigns higher weights to the dimensions with larger values (*i.e.*, edges visited most). Therefore, we calculate the cosine similarity between every two reports and merge reports as the same bug if the cosine similarity between their corresponding edge-hit-count vectors exceeds a threshold.

$$cosine(\vec{v}, \vec{v}') = \frac{\vec{v} \cdot \vec{v}'}{|\vec{v}||\vec{v}'|} = \frac{\sum_{i=1}^{n} c_i c_i'}{\sqrt{\sum_{i=1}^{n} c_i^2}\sqrt{\sum_{i=1}^{n} c_i'^2}} \tag{3.6}$$

3.2.2.4 Implementation

We implemented the fuzzing part of MdPerfFuzz on top of an AFL-based fuzzer, PerfFuzz [103]. Specifically, we compiled a simplified Markdown grammar via ANTLR4 [139] into a Markdown parser; then we introduced the syntax-tree based mutation strategy as an extension that can be flexibly plugged in; we enhanced a C/C++ compiler to instrument the testing software and modified AFL's showmap functionality to trace the execution on the instrumented applications to obtain the CFG edge hit counts for bug de-duplication.

3.2.3 Detecting Performance Bugs via MdPerfFuzz

In this section, we investigate the prevalence of performance bugs in the wild. We apply MdPerfFuzz to detect performance bugs in several mainstream Markdown compilers.

3.2.3.1 Experimental Setup

Dataset. Since MdPerfFuzz employs an AFL-based fuzzer, it is only capable to analyze Markdown compilers implemented in C/C++. Therefore, we select all the 3 Markdown compilers in C in the recommended implementation list of CommonMark specification [37] and list them in the first column of Table 3.4.

Experiments. Each Markdown compiler is first instrumented using our enhanced C compiler. We then apply MdPerfFuzz to detect performance bugs on the instrumented

Table 3.4: Bug detection results of the Markdown compilers.

Software	Lang.	# Reported	# (Unique) Reported	# Confirmed
cmark (0.29.0)	C	1321	7	4
MD4C (0.4.7)	C	239	3	0
cmark-gfm (0.29.0)	C	981	4	3

Markdown compilers. We configure MdPerfFuzz to use a single process, a timeout of 6 hours, and an input size of 200 bytes. After our preliminary study, we empirically set k to 5 and the cosine similarity threshold to 0.91 for all testing software. All experiments described in this section are conducted on a server running Debian GNU/Linux 9, with an Intel Xeon CPU and 96GB RAM.

3.2.3.2 Results

We present the performance bug detection results in Table 3.4. Duplicate performance bug reports are naturally common during fuzzing. The fuzzing part of MdPerfFuzz reported $2{,}541$ cases in total and our de-duplicating algorithm merged them into 14 distinct reports. We observe that all the 14 cases did successfully slow down the Markdown compilers by from 2.31× to 7.28× compared to normal-performance cases.

We further manually check the reports to validate the performance bugs. Since MdPerfFuzz limits the input size like in other works [75, 103, 143] due to the concerns of large search space, our manual analysis attempts to identify the severity of the performance slowdown in more realistic scenarios, *e.g.*, larger input sizes of thousands of characters. To this end, we first identify the exploit input patterns in the reports that exhaust the run-time resources. With the patterns, we further construct larger test cases to verify the performance issues in practice. Finally, 7 cases in 2 Markdown compilers were confirmed as performance bugs, including 4 new bugs, after our manual analysis. We are in the process of reporting the new bugs to the concerned vendors. At the time of writing, 1 bug has been well acknowledged.

We found no bug in MD4C. The developers of MD4C explicitly mention that they seriously considered performance as one of their main focuses during the development. Therefore, the performance bugs could be avoided with domain knowledge and special care, which are often difficult for most developers.

We have investigated those false-positive cases to understand the reasons. We find

that the unique buggy cases reported by MdPerfFuzz indeed triggered performance issues, *i.e.*, those test cases led to longer execution paths. However, the larger attack inputs we constructed manually did not manifest such performance issues. This is because in our experiments we let MdPerfFuzz explore only small-size test cases (*e.g.*, hundreds of bytes) to limit the search space. This is because developers might choose to patch the performance bugs by enforcing certain limits (*e.g.*, **P1**). Such a strategy guarantees that there is no performance issue in large-size test cases; small-size test cases, however, can still trigger worst-case behaviors. We find all the false positives were caused because of this.

3.2.3.3 Comparison

We compare MdPerfFuzz with two state-of-the-art works, SlowFuzz [143] and PerfFuzz [103]. MdPerfFuzz and PerfFuzz are implemented above AFL whereas SlowFuzz is built on top of libFuzzer [114]. SlowFuzz (libFuzzer) uses in-process fuzzing, which is much faster as it has no overhead for process start-up; however, it is also more fragile and more restrictive because it traps and stops at crashes [114]. Nevertheless, we evaluate all the tools with the same dataset in Table 3.4 and run them for the same amount of time (6 hours), and the same input size (200 bytes) for a fair comparison. We failed to run SlowFuzz on MD4C because of some unexpected crashes after several minutes of the execution. To the best of our knowledge, there is no way to suppress such crashes. MdPerfFuzz and PerfFuzz—AFL-based fuzzers—do not suffer from this problem.

The results show that MdPerfFuzz outperformed PerfFuzz and SlowFuzz by detecting 3 and 5 more performance bugs, respectively. In particular, PerfFuzz reported 820/114/783 cases in cmark/MD4C/cmark-gfm, respectively; SlowFuzz reported 432/408 cases in cmark/cmark-gfm, respectively. These results also demonstrate the need of a bug de-duplication method. We applied our bug de-duplication algorithm to identify unique bugs and then manually confirmed the reports. Finally, PerfFuzz detected 2/0/2 real performance bugs in cmark/MD4C/cmark-gfm, respectively. SlowFuzz detected 1/1 real performance bugs in cmark/cmark-gfm, respectively. This demonstrates that our syntax-tree based mutation strategy can improve fuzzing efficiency by generating better inputs within the same resource budget.

Table 3.5: The performance slowdown and code coverage of MdPerfFuzz, PerfFuzz, and SlowFuzz.

Tool	Software	Best Slowdown	Line Coverage Rate	Function Coverage Rate
MdPerfFuzz	cmark	7.28×	71.90%	67.91%
	MD4C	2.31×	76.22%	58.11%
	cmark-gfm	6.54×	55.78%	57.35%
PerfFuzz	cmark	6.82×	56.21%	51.35%
	MD4C	2.21×	67.20%	50.20%
	cmark-gfm	5.05×	48.26%	44.31%
SlowFuzz	cmark	4.32×	40.28%	41.65%
	cmark-gfm	3.29×	38.30%	42.33%

The best slowdown across all tools is normalized over the baseline of the same random normal-performance case.

Performance Slowdown

Table 3.5 shows the performance slowdown caused by the inputs generated by MdPerfFuzz, PerfFuzz, and SlowFuzz. We use the maximum execution path length as the performance metric, and normalize the performance slowdown using a baseline obtained from the random normal-performance cases. We notice that, though all tools caused performance slowdown on the testing applications, MdPerfFuzz achieved a 14.71% higher average best performance slowdown over PerfFuzz, and 41.21% over SlowFuzz. Furthermore, we observe that MdPerfFuzz could generate inputs that slow down the compilers much faster than the other tools. For example, to reach a 4.32× performance slowdown on cmark, MdPerfFuzz took 3.2 hours, whereas PerfFuzz and SlowFuzz used 3.9 hours and 6.0 hours, respectively. This demonstrates the high efficacy of MdPerfFuzz in detecting performance bugs.

Code Coverage

We also evaluate the code coverage each tool achieves to measure the efficacy of our syntax-tree based mutation strategy. We collect the test cases generated by each tool and run on afl-cov [129], which detects the code coverage using the overall execution traces covered by the test cases. Though SlowFuzz is not based on AFL, we believe using its test cases on afl-cov can accurately reflect the code coverage under a fair metric.

We present the results of line coverage and function coverage in Table 3.5. The syntax-tree based mutation strategy of MdPerfFuzz was effective in reaching high code coverage. It enabled MdPerfFuzz to visit 67.97% of lines of code and 61.79% of functions

Table 3.6: Evaluation results on other Markdown compilers and applications.

	Software	Implementation Language	# Bugs
Other Markdown compilers	commonmark.js (0.29.3)	JavaScript	7
	markdown-it (12.0.4)	JavaScript	2
	marked (1.2.7)	JavaScript	25
	Snarkdown (2.0.0)	JavaScript	13
	commonmark-java (0.17.0)	Java	6
	flexmark-java (0.62.2)	Java	15
	commonmark.py (0.9.1)	Python	27
	php-commonmark (1.5.7)	PHP	19
	php-commonmark* (1.5.7)	PHP	0
	Parsedown (1.7.4)	PHP	8
	Parsedown* (1.7.4)	PHP	8
	php-markdown (1.2.8)	PHP	6
	markdown-go	Go	13
	Comrak (0.9.0)	Rust	11
	StackEdit	JavaScript	9
	DILLINGE	JavaScript	7
Applications	GitLab (13.7.3)	Ruby	6
	BitBucket (7.9.1)	Java	8
	Hugo (0.74.3)	Go	13
	Hexo (5.2)	JavaScript	14

$*$ denotes the security mode of the compiler.

on average. MdPerfFuzz achieved *higher code coverage* than PerfFuzz and SlowFuzz in *all* testing software. In the Markdown compilers, MdPerfFuzz outperformed PerfFuzz by 20.75% more lines of code and 13.17% more functions; MdPerfFuzz outperformed SlowFuzz by 28.68% more lines of code and 19.80% more functions. With the mutation strategy, MdPerfFuzz successfully fuzzed 8.02% of lines of code and 11.39% of functions that were not ever visited by other tools. As a result, 2 new performance bugs were identified within this proportion of code.

3.2.4 Testing More Compilers

Many Markdown compilers are implemented in languages other than C/C++, and they are not supported by MdPerfFuzz and other AFL-based fuzzers. To understand if and how these Markdown compilers suffer from performance bugs, we construct an extensive dataset and utilize the exploits generated by MdPerfFuzz in §3.2.3 to detect potential bugs in them.

3.2.4.1 Methodology

We construct a comprehensive dataset in Table 3.6, including a set of *other Markdown compilers* written not in C/C++ and another set of relevant real-world *applications*. We try to include popular Markdown compilers implemented in diverse programming languages to understand the effects of programming languages on performance bugs (if any). In particular, our dataset covers Markdown compilers written in Java, JavaScript, PHP, Python, Go, and Rust. We also include the first two Google search results (StackEdit and DILLINGE) in January 2021 into our dataset. StackEdit is also in the suggested application list for opening Markdown documents in Google Drive. We notice that some compilers (php-commonmark and Parsedown) provide options to enable additional security mode to mitigate certain bugs. We are interested in the effects of such security options, thus we present them separately with an asterisk suffix (*). Regarding real-world applications, we try to cover two main uses of Markdown compilers: 1) Markdown document rendering in code hosting software (*e.g.*, GitLab, and BitBucket); and 2) static web page generation frameworks (*e.g.*, Hugo and Hexo).

We downloaded the latest stable version of each Markdown compiler from its official website or GitHub repository in January 2021. We denote their actual software versions in the parenthesis if applicable. We install and configure them with the default settings.

Since MdPerfFuzz is not capable to detect performance bugs in the dataset in Table 3.6, we first collect the exploits generated from MdPerfFuzz in §3.2.3 and summarize them into 45 unique attack patterns. Each pattern exploits one Markdown syntax feature. We then apply them to evaluate the software in a black-box manner. Such an approach is practical and scalable, and enables us to analyze a diverse set of compilers.

We use the environment as in §3.2.3.1 to test standalone Markdown compilers. For the software in the *application* category, we empirically identify the entry points for triggering the Markdown compiler components (*e.g.*, command-line API or UI operations). For those that work in the server-client model and allow self-hosting (*i.e.*, GitLab and BitBucket), we deploy them on a computer running Debian GNU/Linux 9.12 with a 4-core Intel Xeon CPU and 16GB RAM. We use another computer in the same local area network as the client to send requests and measure the network response time, client-

side CPU time, and server-side CPU time after the client issues a request. We use such results to detect performance bugs and further understand whether the performance bugs appear in the server or the client.

It is hard to instrument the Markdown compilers implemented in diverse programming languages and the Markdown components in complex software. Hence we currently are unable to de-duplicate the reports for the software in the dataset. Nevertheless, as our test cases especially exploit distinct Markdown syntax features, we believe they are most likely to trigger different performance bugs. We will further discuss it in §3.2.6.

3.2.4.2 Results

We present the bug detection results in the last column of Table 3.6. The performance bugs in Markdown compilers are prevalent and might have been overlooked by the compiler developers. In particular, we successfully identified 168 performance bugs in the category of *other Markdown compilers*. We can observe that the number of detected performance bugs varies significantly among Markdown compilers. Some Markdown compilers were particularly vulnerable to performance bugs, whereas some did not have any performance issues. For instance, we detected 27 performance bugs in commonmark.py but only 2 bugs in markdown-it. We do not observe a distinguishable difference among programming languages in terms of the number of bugs. The performance bugs could substantially impact end-user experiences. For example, the performance bugs in StackEdit and DILLINGE could lead to data loss once the browser tab was unresponsive or was forcibly killed.

The Markdown compilers in popular applications are also vulnerable to performance bugs. We successfully detected 41 performance bugs—28 on the client-side and 13 on the server-side. In particular, GitLab and BitBucket suffered from server-side performance bugs, which could be exploited to significantly degrade the server performance. They can be exploited for launching DoS attacks (see §3.2.5 for more details).

Responsibly disclosing the bugs can greatly benefit the software users and the whole community. We are in the process of contacting the maintainers of the buggy Markdown compilers and reporting the newly detected performance bugs. At the time of writing,

24 performance bugs have been acknowledged. One performance bug in GitLab has been recognized in CVE-2021-22217 [43].

Though we mainly focused on performance bugs, we also detected memory corruptions in more than 5 Markdown compilers in our research. In particular, the Markdown compilers exhibited crashes when we fed them the test cases. The crashes happened in the Markdown compilers implemented in JavaScript, Java, and Python. For example, we particularly analyzed one crash in Snarkdown, and found the maximum call stack size was exceeded when using recursive function calls to process one type of our testing inputs. There were other memory errors (*e.g.*, segmentation faults) when our test cases triggered some illegal memory read or write.

We also detected several unexpected errors in the *application* category. We observed that GitLab and BitBucket could return HTTP 500 internal server errors when the test cases contained special Unicode characters, possibly because their Markdown compilers currently did not support compiling special Unicode characters. Though such errors usually affect only the user who sends documents containing such characters, they still lead to bad user experiences and shall be fixed.

3.2.4.3 Effects of Security Mode

php-commonmark and Parsedown introduce a security mode to mitigate certain bugs. Our results show that their security modes have different effects. As shown in Table 3.6, php-commonmark in its security mode (shown as php-commonmark*) was not vulnerable to any performance bugs, whereas the security mode of Parsedown (shown as Parsedown*) did not mitigate any performance bugs.

By reading the relevant documents and the source code, we find that the security mode of Parsedown mainly mitigates cross-site scripting (XSS) vulnerabilities but does not consider performance related issues. The security mode of php-commonmark, on the other hand, applies several strategies to mitigate performance bugs. For instance, it sets a threshold to limit the depth of nested structures, escapes HTML blocks in Markdown inputs, and disallows unsafe links. These strategies together could successfully mitigate all 19 performance bugs identified in its default mode. We will further discuss the countermeasures against performance bugs in §3.2.6.

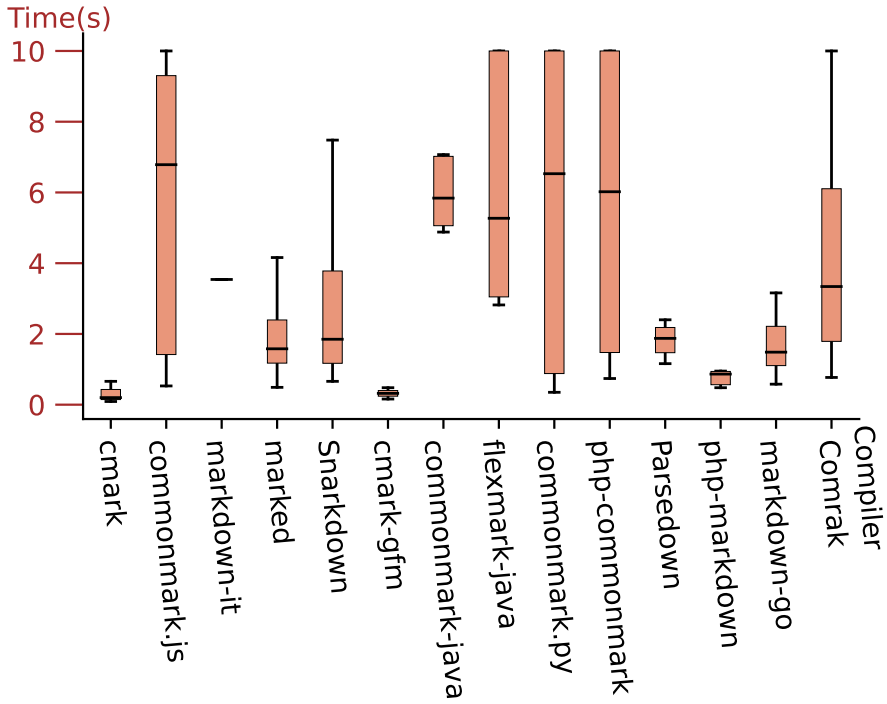


Figure 3.7: Compilation time of Markdown compilers under attack inputs of the size of 50,000 characters and a 10-second maximum threshold. The middle lines in the boxes represent the corresponding median values.

3.2.5 Impact on Performance

To better understand the performance degradation caused by performance bugs, we depict in Figure 3.7 the compilation time when the performance bugs are triggered by attack inputs of 50,000 characters. We use such a size as it can roughly represent the normal uses of Markdown compilers. The results show that performance bugs can cause significant performance degradation. In general, our attack inputs successfully exploited the performance bugs by causing over 3-second compilation time. Different performance bugs could result in different levels of performance degradation in a compiler. For example, the compilation time of commonmark.js ranged from 2 seconds up to 10 seconds (the maximum time threshold) due to the performance bugs. If a threshold was not set, the exploits would even cause the compilers to run for *several hours*.

The performance bugs can potentially affect many users if they reside in server-side applications. Specifically, we present two case studies about GitLab—a code hosting software, and Parsedown—a popular Markdown compiler module in PHP. We deploy the latest version of GitLab with its default NGINX web server; we develop a server-side PHP application that calls Parsedown to compile user-provided Markdown documents.

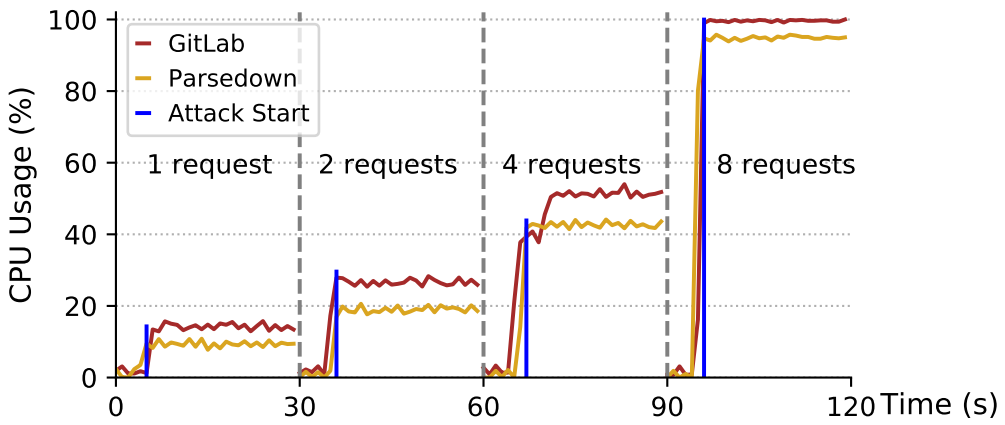


Figure 3.8: Server-side CPU usage over time under attacks on GitLab and Parsedown.

We randomly choose a common attack input that can trigger performance issues in both GitLab and Parsedown. We then test the applications with different numbers of concurrent attacks (requests). We try at most 8 concurrent requests because our server has only 8 logical CPU cores.

We depict the server CPU usage under the attacks in Figure 3.8. We clearly observe the increase of CPU usage when more concurrent attack requests were issued. In particular, when 8 requests were sent, the server CPU usage promptly reached almost 100%. Therefore, an attacker can send only a few attack requests at a very low rate to significantly degrade the performance of a vulnerable server application, making it unable to responsively serve other legitimate user requests.

3.2.6 Discussion

Mitigating performance bugs. Practical mitigation and defense techniques against performance bugs are necessary for protecting vulnerable Markdown compilers and applications. Several security strategies used in the security mode of php-commonmark (*e.g.*, enforcing the limits, escaping the HTML blocks, and disallowing unsafe links) are shown to be effective in mitigating performance bugs. However, they can break some functionalities, especially those related to the context-sensitive features. For example, some HTML blocks cannot be compiled as expected because of the security strategies. Longer legitimate Markdown documents might also be blocked because of the limits. A trade-off has to be made to balance functionality and security. In the future, we hope to port such mechanisms to mitigate attacks exploiting performance bugs in Markdown

compilers.

Bug de-duplication. Trace-based analysis is effective in triaging bugs [178]. MdPerfFuzz thus adopts an execution trace similarity approach to de-duplicate performance bugs. Besides the cosine similarity [53, 96] MdPerfFuzz employs, other algorithms (*e.g.*, those measuring the Euclidean distance [96]) can potentially be applied as well. Theoretically, the method can be applied to software implemented in diverse programming languages once the run-time CFG edge hit information is available. However, to the best of our knowledge, not all programming languages have available instrumentation tools exactly for such a purpose. It is also time-consuming and even infeasible to develop our own instrumentation tools within this work. We thus choose to not apply the method for the evaluation in §3.2.4. In the future, we plan to further investigate the feasibility of a language-agnostic method for de-duplicating performance bugs. For example, we want to explore transforming the software into certain intermediate representations (IRs) and obtain the necessary information from IRs to de-duplicate performance bugs.

Lessons. We have tested Markdown compilers implemented in languages other than C/C++ in §3.2.4 using the attack patterns generated by MdPerfFuzz. Each pattern corresponding to one Markdown syntax feature could exploit a group of bugs in different Markdown compilers. For example, one attack pattern was able to trigger similar performance bugs in 12 different Markdown compilers implemented in multiple programming languages. Based on our further analysis and the feedback from the developers, we learned that compiler developers often borrow implementation ideas from other similar or relevant projects. Such a design reuse is risky and has resulted in similar performance bugs across different implementations. We conjecture that similarly those compilers might commonly be subject to other types of bugs.

Prior works have studied performance bugs in the compilers of general-purpose languages (*e.g.*, GCC [86]) and domain-specific languages (*e.g.*, Node.js regex engine [54, 168]). However, performance bugs in the compilers of many other domain-specific languages such as Latex [29] and Wikitext [47] have not been systematically studied yet. We believe the design and implementation of MdPerfFuzz can shed some light on the following research on other domain-specific languages. To facilitate future research, we release the source code of MdPerfFuzz to help the development of fuzzers for those

languages. For example, researchers can replace our language model with their own ones in MdPerfFuzz. Further, MdPerfFuzz could also be easily extended to detect other types of bugs (*e.g.*, memory corruptions) in Markdown compilers.

3.2.7 Related work

Detecting performance bugs. The detection of performance issues has drawn significant attention from researchers over the past years. Prior studies focus on application-layer DoS vulnerabilities [59, 85, 122], algorithmic complexity DoS vulnerabilities [44, 163], and other general performance issues [86, 113]. Static methods analyze the source code of the applications and diagnose vulnerable bug patterns, for example, repeated loops [132, 133].

Dynamic methods are also applied to identify performance bugs. SlowFuzz [143], PerfFuzz [103], and HotFuzz [21] proposed fuzzing solutions to detect the worst-case algorithmic complexity vulnerabilities. SAFFRON [98] also used grammar-aware fuzzing to find worst-case complexity vulnerabilities in Java programs. Toddler [133] detected performance bugs by identifying loops with similar memory access patterns. Hybrid approaches [159] combining static analysis and fuzzing were proposed to detect ReDoS in Java programs. Our work is tailored for Markdown compilers and is equipped with a bug de-duplication method.

Grammar-aware fuzzing. To find security bugs in compilers, a number of grammar-aware fuzzing frameworks [15, 20, 74, 179, 183, 196, 200] have been proposed. Guided by the grammar, fuzzers can generate syntactically-correct inputs to test the compilers. In particular, LangFuzz [82] modified existing test cases by randomly combining JavaScript code fragments to generate new test cases. Superion [184] extended AFL to support additional grammar-aware mutation strategies via pluggable language parsers. Several learning-based [45, 101] fuzzers transformed inputs into ASTs and performed subtree replacement with neural network models. Inspired by these works, MdPerfFuzz also employs grammar-aware fuzzing to detect performance bugs in Markdown compilers.

□ End of chapter.

Chapter 4

Testing and Improving Symbolic Execution

Symbolic execution is a foundational program analysis technique that symbolically reasons about program behaviors. It has shown great promise and has been applied to various tasks such as bug detection [23, 27, 111, 150], vulnerability assessment [12], root cause analysis [194], *etc.* For example, recent symbolic execution frameworks are able to detect vulnerabilities in well-tested software like OpenJEPG, Chrome and Firefox [23, 150]. Software development often involves *internal functions*, which are provided along with the language systems. They include library functions and built-in functions that offer basic operations like string processing, arithmetics, bit manipulation, *etc.* Similar to normal concrete execution, symbolic execution also requires understanding the semantics of the internal functions.

We identify that the support of internal functions in symbolic execution is often erroneous. Incorrect support would make the symbolic analysis results unreliable. In this chapter, we test and improve symbolic execution, especially its internal function support.

4.1 SEDiff: Scope-Aware Differential Fuzzing to Test Internal Function Models in Symbolic Execution

4.1.1 Motivation

The internal functions however incur two important problems to symbolic execution: *scalability* and *compatibility*. First, internal functions are often frequently-invoked basic functions (*e.g.*, string processing and arithmetical operations). Our study shows that, in modern software, around 70% of internal functions are invoked at least twice and many are called for thousands of times (see §4.1.2.1 for more details). Therefore, symbolic execution easily becomes unscalable if it repeatedly runs them. Second, internal functions are often implemented in a language (*e.g.*, C) different from the one of the main programs (*e.g.*, PHP). Existing symbolic execution engines typically target only the language of the main program and are unable to handle internal functions in a different language [12, 193].

To solve the aforementioned problems, function modeling is the *go-to approach* that has been widely adopted in common symbolic execution tools. Modeling is to abstract the behaviors of the target function, so the analysis does not have to go through the internal details repeatedly. Prior works [12, 111, 160, 193] model the behaviors of internal functions and integrate the models to the underlying symbolic execution engines. In this work, we refer to such interpretation of internal functions in symbolic execution as *internal function models* [12, 111, 160, 193]. For example, a popular symbolic execution engine, Angr [160], defines SimProcedure [2] for modeling.

The correctness of the internal function models is critical to symbolic execution and its applications. Incorrect internal function models could lead to incorrect reasoning of the programs. A symbolic execution based bug detector would wrongly determine the path feasibility because of an incorrect internal function model, thus the results it outputs would be inaccurate and unreliable. Also, an error in symbolic execution based vulnerability assessment may miss critical vulnerabilities and delay the patching. Accordingly, the security analysts have to take excessive time to manually verify the results or filter out the false reports.

Despite the importance, to the best of our knowledge, testing internal function models remains an under-explored topic. Past practices largely rely on user reports to identify and fix bugs in them [1, 3], which is inefficient. Only a few recent research works have attempted to automatically test them through differential testing, which compares the behavior of symbolic execution to that of concrete execution. However, they have several inherent limitations. They focus on testing symbolic execution engines as a whole, whereas the internal function models are not thoroughly studied. In particular, Kapus and Cadar [91] checked whether the results of symbolic execution conform to concrete execution for programs generated by Csmith [197]. Their method explores more on the diversity of overall program syntax. It fails to probe the semantics of internal functions and their models, and as a result, does not detect any bugs in the models.

A fundamental problem with existing differential testing works is that they do not attempt to specifically target *modeled functionalities* that are interpreted in the internal function models. By its nature, modeling focuses on only the most important functionalities [111, 160, 193], and chooses to discard or ignore the rest unmodeled functionalities [2]. As existing works test symbolic execution engines as a whole, they simply report all inconsistencies (including the ones out of the modeling scope) as bugs. Most of the reported "bugs" would be false positives. We believe that a reasonable testing approach should rather focus only on the modeled functionalities. However, automatically identifying the modeled functionalities (*i.e.*, the scope) is challenging, as neither the developer intention nor the modeling logic is provided. Distinguishing the modeled functionalities from the unmodeled ones requires deep understanding of their semantics and thus is non-trivial.

In addition to the scope challenge, we identify two other major challenges in developing a differential testing framework for the models. First, it is challenging to scalably support diverse representations of the models in different symbolic execution engines. Symbolic execution engines take distinct ways to construct their internal function models. As a key component, the models closely coordinate with the rest of the engines. It is thus non-trivial to distinguish the models from the other symbolic execution components, which however is a required first step for inferring the modeled functionalities. Second, generating workloads to efficiently detect bugs within the scope is hard.

The workloads are expected to extensively exercise the modeled functionalities without wasting resources on testing other unmodeled functionalities. None of the existing works has ever attempted to consider the scope for generating workloads.

In this work, we design a *scope-aware* differential testing framework, SEDiff. It incorporates several new techniques to overcome the challenges. First, we observe that, though the internal function models are implemented diversely within the symbolic execution engines, most internal function models are passed to the SMT solvers (*e.g.*, Z3 [55]) in a uniform format, SMT-LIB language [19]. We propose *minimal-program synthesis* for generating the SMT-LIB expressions of internal function models, to help us uniformly and accurately extract the internal function models. Second, we find that every internal function has its original implementation that realizes all its functionalities, including the modeled functionalities. We define a program path in the *original implementations* of the internal functions as a functionality. We develop a mechanism that maps the paths in the models to their original implementations to identify the modeled functionalities and resolve the scope challenge. To realize the mapping, we develop a new technique to recover the data flows from disordered SMT-LIB formulas for the models. Third, we leverage the grey-box fuzzing technique on the original implementations to thoroughly drive differential testing. We design a new coverage metric and a feedback mechanism that help generate in-scope workloads. We also develop a tailored bug checker to accurately label bugs during testing.

We thoroughly evaluated SEDiff on several state-of-the-art symbolic execution engines (*e.g.*, Angr [160]) that employ internal function models. We first apply SEDiff to extract models and identify modeled functionalities. Our manual investigation of the modeled functionalities it reported reveals that SEDiff can accurately pinpoint modeled functionalities with high precision. We then use SEDiff to differentially fuzz the models. It successfully detected 46 new bugs in the 298 internal function models. Our characterization further demonstrates the importance of identifying modeled functionalities and confirms the benefits of SEDiff' awareness of scope. We believe that SEDiff's techniques are generic. It has huge potential for other application domains that have multiple implementations complying with similar specifications. We open-sourced our prototype implementation to facilitate future research.

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

- We propose the first comprehensive study on internal function models, and define several key concepts of this problem.
- We propose SEDiff with multiple generic techniques, including 1) automated and uniform model extraction through SMT-LIB and minimal-program synthesis; 2) automated identification of modeled functionalities with data-flow recovery in SMT-LIB formulas; and 3) scope-aware differential fuzzing for modeled functionalities with new input generation and feedback mechanisms.
- With SEDiff, we found numerous new bugs in the internal function models in the state-of-the-art symbolic execution engines.

4.1.2 Problem Statement

4.1.2.1 Internal Functions in Symbolic Execution

Following the code-reuse paradigm, programming language systems contain numerous functions that cover basic operations. They aim to facilitate the use of the language systems and significantly improve programming efficiency. In this work, we name the functions that provide basic operations like string processing, arithmetic and bit manipulation as internal functions. For example, the C/C++ language systems include a large number of standard library functions; the PHP system provides numerous built-in functions in the PHP interpreter implemented using C. As a necessary component of the programming language systems, internal functions are widely adopted and used by developers [123]. In a preliminary study, we measured the use of internal functions. We parsed a popular PHP application—WordPress and a widely-used C program suite—GNU CoreUtils,[3] and computed the frequency of direct (static) function invocations in their source code. The results showed that 35.25% (resp. 43.28%) of function invocations in WordPress (resp. GNU CoreUtils) were about internal functions.

Symbolic execution simulates the program execution in a symbolic manner and it naturally has to reason about the semantics and behaviors of internal functions. Two

[3]We used the latest WordPress v5.92 as of Mar 2022. We failed to configure the latest GNU CoreUtils and used a relatively old version v8.21. We believe the overall trend could naturally be ported to the latest version.

important problems arise when symbolic execution meets internal functions. The first is the *scalability* problem. Internal functions are often frequently invoked for basic operations such as string processing and arithmetics. Our preliminary study showed that 68.99% (resp. 75.97%) of internal functions in WordPress (resp. GNU CoreUtils) appeared at least twice. In both cases, many internal functions occurred hundreds to thousands of times, and some internal functions were among the 5 most frequently invoked functions. For example, WordPress invoked $\mathrm{substr}()$ for $2{,}211$ times, which accounted for 3.17% of all function calls; GNU CoreUtils called $\mathrm{strlen}()$ for $3{,}148$ times, which represented 3.89% of all function calls. Therefore, symbolically executing those functions repeatedly would significantly slow down the overall performance of symbolic execution.

Second, symbolic execution would raise *compatibility* issues due to the cross-language nature [109]. In particular, internal functions in a programming language are often implemented in a different language that is incompatible with the symbolic execution engine. As a result, a symbolic engine for one language system can hardly seamlessly analyze the target programs without additional interpretation of internal functions. For example, all PHP internal (built-in) functions are implemented in C in the PHP interpreter [5]; a typical PHP symbolic execution engine naturally analyzes the main language—PHP, while it is also necessary for the engine to support the internal functions. Interpreting the internal functions, however, is non-trivial [150].

4.1.2.2 Function Modeling as a Practical Solution

Function modeling, which abstracts the relevant semantics and behaviors of a target internal function, is the *go-to approach* to addressing the scalability and compatibility issues in common state-of-the-art symbolic execution engines [12, 111, 160, 193]. In particular, they construct a model for an internal function only once and symbolic execution can reuse it across the whole analysis phase, thus resolving the scalability problem [12, 111, 160, 193]; the constructed model can be seamlessly integrated into underlying symbolic execution engines thus tackling the compatibility issue. Many well-known symbolic execution engines (*e.g.*, Angr [160], Navex [12], *etc.*) employ modeling for bug detection and exploitation. In this work, we refer to such models of internal functions in symbolic execution as *internal function models*.

The modeling process requires the domain knowledge of language systems and symbolic execution engines, thus is often completed in a manual manner. Due to the excessive manual efforts modeling requires, as a practical implementation choice, developers thus choose to model *the most important internal functions*. We indeed observe that the modeled functions are normally among *the most frequently invoked ones*. For example, substr() and strlen() are generally modeled in popular symbolic execution engines such as Navex and Angr. For the same reason, developers model only *the most important functionalities* of a function instead of all functionalities. We refer to such functionalities in the models as *modeled functionalities* in this work.

4.1.2.3 Research Goals

As symbolic execution is a foundational analysis technique, depending on the applications, incorrect models can lead to many issues, such as failures to detect vulnerabilities [193], delaying the patching of critical bugs [190], or introducing regression bugs [117]. Also, the security analysts have to take excessive time to manually verify the results or filter out the false reports.

Testing the correctness of the models is thus of great importance. In this work, we aim to automatically apply differential testing to the internal function models. We want to identify what functionalities are included in the models and test if they are modeled correctly. More importantly, under the progressive evolution of symbolic execution, we hope to provide a systematic solution for developers to understand the models, probe any mistakes, and improve the state-of-the-art symbolic execution engines.

4.1.2.4 Research Challenges

We identify several research challenges that motivate us to propose new techniques.

Supporting diverse representations of models. As the first step, we need to uniformly extract the models from a given symbolic execution engine that may target a specific language. Symbolic execution engines for different language systems have a diverse set of internal functions, *e.g.*, the PHP built-in functions and the C/C++ standard library functions realize different functionalities. Manual analysis of the code to pinpoint the models certainly cannot scale and is costly, especially because symbolic

execution engines are quite complex—implemented with millions of lines of code. The models in them comprise only a small proportion in the huge code base. Besides, as a key component, the models closely coordinate with other engine components to achieve the overall functionality. For example, the models can closely co-work with the memory management component, constraint solving component, *etc.*, which are out of our research scope [6]. It is hard to distinguish the code of the models from the rest components especially when the model code shows little difference from the code of other parts.

Identifying modeled functionalities. As we mentioned earlier, developers would include only selected functionalities of the functions into the models (*i.e.*, modeled functionalities), and intentionally ignore the rest unmodeled functionalities. From our own experience and the communications with developers, most developers do not appreciate or accept reports concerning unmodeled functionalities because they do not plan to support those functionalities in their tools from the very beginning. Apparently, the testing of the models should focus on only the modeled functionalities. Therefore, we need to distinguish if the revealed behaviors are related to the modeled functionalities. It is challenging to identify the model scopes as it requires a deep comprehension of the semantics of the models.

Efficiently detecting bugs in modeled functionalities. Differential testing requires test cases to drive the targets for the testing. None of the existing work has attempted to generate test cases to test the internal function models. Blindly generating test cases in a black-box manner is inefficient. It is hard to design heuristics to thoroughly exercise the models to achieve high coverage. Besides, the input generator should be aware of the model scope and construct high-quality inputs to exercise the modeled functionalities.

4.1.3 Design of SEDiff

In this work, we design a scope-aware differential testing framework, SEDiff, to facilitate bug detection in internal function models in symbolic execution. The high-level architecture of SEDiff is depicted in Figure 4.1. SEDiff entails overcoming the technical challenges with several new observations and techniques:

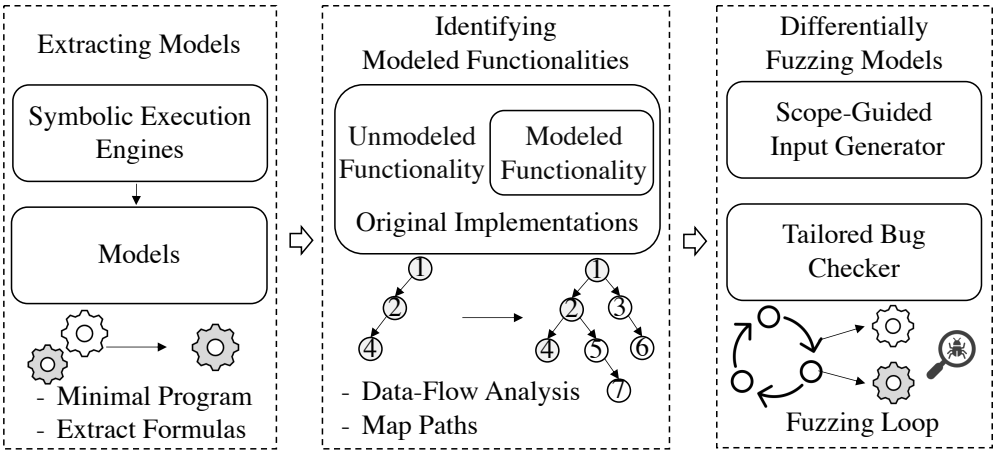


Figure 4.1: The architecture of SEDiff.

Automated and uniform model extraction. We observe that regardless of the language of a symbolic execution engine, the models are passed to an SMT solver at the end in the form of SMT-LIB language. We thus propose minimal-program synthesis to generate the SMT-LIB expressions for the models, which greatly helps us extract the models. We will explain how we extract the models in the uniform SMT-LIB expressions with minimum program synthesis §4.1.3.1.

Scope identification. By its nature, modeling is to abstract how a function should behave under different inputs, which are essentially the different execution instances (*i.e.*, code paths). To this end, we carefully define the functionalities as distinct program paths in the original implementations of the internal functions. Therefore, the task of identifying the model scope is transformed to mapping what program paths are realized in the models. We first propose new techniques to recover the data flow from disordered SMT-LIB formulas for the models. We then perform a data-flow analysis on both the models and their original implementations to identify common data paths.

Scope-aware differential fuzzing. We develop a scope-aware grey-box differential fuzzer to facilitate bug detection in modeled functionalities. We propose a new coverage metric to particularly guide exploration towards the identified modeled functionalities in the original implementations; we also design a tailored bug checker that can utilize the model scope to distinguish if or not the inconsistencies are associated with the modeled functionalities.

```php
<?php
$arg = $_POST["test"];
$ret;

if(abs($arg) == $ret) {
    // check point
}
```

Listing 4.1: A minimal program that invokes an internal function (model).

4.1.3.1 Model Extraction with SMT-LIB and Minimal-Program Synthesis

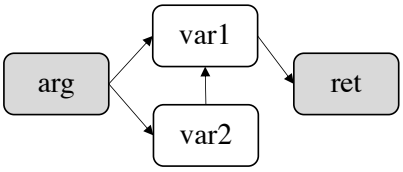
We extract the models based on a key observation—*the uniformity of SMT-LIB*. The SMT-LIB standard is a widely-adopted international initiative aiming at facilitating research and development in Satisfiability Modulo Theories (SMT) [19]. SMT-LIB specifies a general language for input formulas that SMT applications work with. We find that though the models are implemented diversely across engines, they are finally passed to SMT solvers in the same form of SMT-LIB expressions, where their operations are interpreted and captured. Therefore, the SMT-LIB language format enables a uniform and accurate analysis of diverse models across engines.

The next question is how to know which parts in the SMT-LIB expressions correspond to internal function models. Our idea is to minimize the SMT-LIB expressions and exclude those that are irrelevant to the models. To this end, we synthesize a *minimal program* whose purpose is solely to invoke the internal function models. This way, we can reliably extract the SMT-LIB expressions for the models by simply removing the irrelevant invoking expressions. For example, Listing 4.1 presents a minimal program that invokes the PHP built-in function abs(), which calculates the absolute value of a given argument. The minimal program invokes the internal function in a conditional statement (line 5). When a symbolic execution engine analyzes the minimal program to determine the path feasibility, it seamlessly interprets the internal function and calls its model for constraint solving, which is later recognized and extracted.

SEDiff automatically synthesizes minimal programs from function signatures. The step is currently assisted with a manual step where we manually obtain the corresponding function names of the models from the symbolic execution engines. This is manageable because the number of modeled functions is usually not large. Automatically extracting all the modeled internal functions is possible for one specific symbolic execu-

```
1 (assert (= ret var1))
2 (assert (= var1 (* (-1) arg)))
3 (assert (= var1
4     (ite (> arg 0) (arg)) (var2)
5     )
6 )
```

(a) Internal function model.



(b) Data dependencies of variables.

Figure 4.2: The simplified function model of PHP internal function abs() and the data dependency diagram of the variables.

tion engine, but requires a considerable amount of engineering efforts to support multiple engines. A function signature consists of the parameters and their types. SEDiff automatically constructs the code to call the internal function. The synthesized program includes the code of the function calls, together with the argument preparation code (*e.g.*, lines 2-3 in Listing 4.1). Note that the synthesized minimal program is required to be presented in the target language of the symbolic execution engine (*e.g.*, PHP for Navex [12]).

To obtain models in the SMT-LIB language representations, we first install the symbolic execution engine and use it to analyze the automatically synthesized minimal programs. Today, symbolic execution engines commonly use SMT solvers such as Z3 [55] and CVC4 [18]. The solvers provide options to dump the input constraint formulas. Therefore, we extend the SMT solvers to additionally save the constraint formulas when the solvers are invoked. As the output of this phase, the formulas describe the semantics and behaviors of the internal function models. Figure 4.2(a) is a simplified function model for PHP internal function abs() extracted from Navex. The model in the SMT-LIB language contains an argument (arg), a return variable (ret) and several intermediate variables (var1 and var2). It evaluates the value of the argument and accordingly returns the negated value or the original one. In particular, it uses a ternary operator (*i.e.*, ite) to evaluate if the argument is positive and returns a value based on the result. It uses the assertion operator (*i.e.*, assert) to enforce the relation (*e.g.*, equality =) between operands.

4.1.3.2 Identification of Modeled Functionalities

We identify the modeled functionalities automatically from the extracted models. We observe that a model is the abstraction of how a function should behave for different

inputs. It essentially represents the execution instances (*i.e.*, code paths) under different inputs. Therefore, we define a unique code path from the function entry to a return point in the *original implementation* of an internal function as a functionality. Such a definition has two benefits. First, it is fine-grained enough to capture the modeled functionalities because it can cover all possible execution instances. Second, the task of identifying the modeled functionalities is transformed into mapping the model's SMT-LIB formulas to the code paths in the original implementations.

We then propose a path mapping algorithm to identify the code paths of the functionalities included in the models. The extracted SMT-LIB formulas mostly describe the model's behaviors as the data relationship among variables using basic boolean and arithmetic operations. Thus a natural way to map the paths is utilizing the data flow information. A code path processes the function arguments. Because our correctness testing focuses on whether a model produces correct results in the return value or the function arguments, the data-flow relationship associated to a path with arguments can well represent the semantics and behaviors of the path. We thus refine the definition of *functionality* as the data-flow formula of the arguments in the code path. Therefore, our algorithm tries to map the data flow in a code path of the model to the one in its original implementation. The mapped data-flow pairs then indicate the functionalities included in the models. We particularly consider the data flows from the function arguments to the return values. The rationale is that the models preserve the semantics of the internal functions and the same data flow relationships between arguments and return values remain in the models.

Data-Flow Recovery and Analysis

Our data-flow analysis takes as inputs the source code of original function implementations and the extracted models represented in SMT-LIB. It infers the data-flow relationships between the function arguments and return values in both the original implementations and the models, and represents them in the same form. The uniform representation of the data-flow relationships enables SEDiff to link the original implementation of a functionality with its counterpart in the model.

Data-flow recovery and analysis on models. Performing data-flow analysis on the SMT-LIB representations of the models is non-trivial. To the best of our knowledge, no

Table 4.1: Data-flow representation system used by SEDiff.

Constant	$c \in Integer \vert String \vert Bool$
Function	$f \in \mathbb{F}$
Expression	$e := c \vert arg \vert f(e) \vert e_1\ op\ e_2 \vert op\ e_1$
Argument	$arg \in e$

existing works have ever attempted that. The SMT-LIB language describes the models as formulas, which, nonetheless, are disordered and do not contain explicit data flow. The obstacle for the data-flow analysis is that the uses of variables in the formulas do not convey explicit data-flow relationships. For example, $(\text{assert } (= \text{ ret var1}))$ only implies ret and var1 should hold the same value whereas their data dependency is unclear.

To solve this problem, we identify the data dependencies among variables in SMT-LIB formulas to recover the data flow. Our algorithm is based on the fact that data flow can only be propagated from *rvalue* to *lvalue* in assignment statements. lvalue stands for expressions that can be on the left-hand side of the assignment operator, or that refer to memory locations; rvalue represents all other expressions that can be on the right-hand side of the assignment operator. Our algorithm first identifies all assignment operations in a SMT-LIB formula. It then classifies the operands as lvalue and rvalue, respectively, and discovers the data dependencies between lvalue and rvalue.

SMT-LIB language only defines the equality operator (*i.e.*, $=$), which can actually imply either the assignment operation, *e.g.*, $(\text{ret}{=}\text{var1})$, or the equality operation, *e.g.*, (ret==var1), in the original implementations. We use several heuristics to infer the assignment operations and the data dependencies in their operands. First, there are three categories of variables in the formulas, namely, arguments, return variables and the rest intermediate variables. We can directly identify from the arguments and return variables by their symbolic names. The data flow in a path should start from arguments and end at return variables. For example, the data flows from arg to ret in Figure 4.2(a). Second, compound expressions can only appear in the equality operations or in the rvalues of assignment operations. If an equality operation has only one compound expression as an operand, we can determine that operand as its rvalue and the other as its lvalue. Line 2 in Figure 4.2(a) is such a case where lvalue is var2 and rvalue is $(*\ (\text{-1})\ \text{arg})$. Last, we conservatively consider that both operands can be either lvalue or

rvalue accordingly for the rest cases. The analysis thus outputs the data dependencies of variables as shown in Figure 4.2(b). While being simple, our experiments in §4.1.5.2 demonstrate that it has high precision of 73.94%.

After identifying the data dependencies, our algorithm chains the data dependencies to construct the full data flow of the paths. In particular, it seeks the data origin of rvalue (*i.e.*, arguments) and walks to lvalue next in each assignment. It explores possible paths that can reach the return variables from the arguments. At the end of the analysis, SEDiff represents the return values as formulas of the function arguments using the form shown in Table 4.1. The formulas are later compared to the ones from the original implementations. As shown in Table 4.1, the simplest form is an expression e, which can be either a constant value (c), a function argument (arg), a function call ($\mathrm{f(e)}$), or an operation above expressions ($\mathrm{e_1\ op\ e_2}$). Function arguments and return values can also be described as expressions. In the example of Figure 4.2(a), the two possible data-flow formulas are $\mathrm{ret{=}arg}$ and $\mathrm{ret{=}\text{-}arg}$, respectively.

Data-flow analysis on original implementations. SEDiff also performs a data-flow analysis on the source code of original function implementations. It constructs a control-flow graph (CFG) for a function and walks through the CFG from the function entry to the end (basically, return statements). It inlines callee functions and creates a single CFG to gather execution information across functions. Following the common practices [124], we set the maximum inlined basic blocks and functions to 50 and 32, thereby limiting the size of intermediate results in our analysis. Similarly, this data-flow analysis also outputs the return values using exactly the same form shown in Table 4.1.

Mapping Paths

SEDiff next maps the data-flow formulas of the return values to identify the modeled functionalities (code paths). Though represented in the same form, the two implementations use distinct intermediate variables and symbols, and simply comparing the data flow or variables in code paths would not necessarily work. To reduce the noises from the symbol names for mapping, SEDiff normalizes the symbol names in the representations. For example, an argument is turned into arg_i from its original identifier. SEDiff then cross-checks the normalized return value representations regarding arguments, and matches them from the models to the original implementations. In particular, SEDiff

checks if a return value is identically represented in both implementations, regardless the symbol names. If a match is found, it labels the corresponding path as a modeled functionality and the relevant code on the path as modeled code. In summary, this stage analyzes the data representations of return values in the coherent form shown in Table 4.1, and outputs modeled code locations in the original implementations.

4.1.3.3 Scope-Aware Differential Fuzzing for Models

After identifying modeled functionalities from the previous phase, SEDiff employs scope-aware differential fuzz testing to detect bugs in the models. It aims to check the function models' conformance to their original implementations.

Scope-Aware Exploration

Given the success of fuzzing, a natural choice is to use the coverage feedback to guide the exploration, which requires first instrumenting the testing targets—models. However, it is hard to design a strategy to instrument the models and capture their coverage information in the SMT-LIB formulas. It is also difficult to instrument them (which are identified yet in the inconsistent form of SMT-LIB) in the symbolic execution engines.

We take an alternative approach to collecting the coverage feedback by instrumenting the original implementations. Rather than directly fuzzing the models, we instead focus on the corresponding original implementations of the functions. We can naturally utilize existing fuzzing frameworks like AFL [202] and LibFuzzer [114] with their instrumentation tools to achieve this goal. A problem of this approach is that the original implementations cover not only modeled functionalities but also unmodeled ones; the latter is not our test target. Traditional fuzzers consider all edges between basic blocks and treat their coverage equally [114, 202]. As a result, they would try to fairly explore all code and waste much effort on the unmodeled functionalities.

Scope-aware input generation. Inspired by TortoiseFuzz [186], we propose a new coverage accounting approach that selects test cases based on the coverage of modeled functionalities. Our insight is that, we consider only modeled functionalities in the fuzzing coverage metric, and exclude the irrelevant unmodeled functionalities. In this work, we refer to the edges between basic blocks concerning modeled functionalities as *model edges* and their associated code coverage as *model coverage*, respectively. We

design SEDiff to exclusively instrument the model edges in the original implementations, and thus measure only model coverage during testing. Our input generator prioritizes test cases by the new model coverage metric and culls the prioritized test cases by the hit count of model edges.

The input generator explores the input space of the internal functions. It favors and saves both the inputs (arguments) and the results of a concrete test case during fuzzing if the test case is interesting. Specifically, it regards a test case as interesting if the test case brings new model coverage such as hitting new model edges or increasing model-edge hit-count. The rationale behind this is that such situations causing new model coverage are likely to reach new components in the models as well. The saved fuzzing results on original implementations are next applied to the models for differential testing (§4.1.3.3).

Feedback mechanisms. The fuzzing component employs two kinds of feedback mechanisms for seed selection and mutation. The feedback mechanism summarizes the importance of a test case and decides whether it deserves further exploration and mutation [92, 142]. Basically, our fuzzer tracks the visited code in a testing trial and uses the feedback from model coverage. It decides if a test case triggers new model coverage. Additionally, the input generator allows feedback from the checker to favor certain test cases. It currently uses the bug checking result—whether a bug is triggered or not—as the feedback. More energy for the mutation and exploration is allocated to the test cases with positive feedback. The feedback allows SEDiff to further lean its fuzzing efforts towards erroneous locations. The rationale is that a test case triggering a bug is likely to trigger other bugs near it because erroneous locations sometimes contain more than one bug. This has been evident in memory-safety bugs [186].

Differential Fuzzing

In each fuzzing trial of the original implementations, SEDiff simultaneously applies the fuzzing results to the models for differential testing.

Driver program generation. SEDiff constructs simple driver programs to help verify whether models comply with concrete fuzzing trials of original implementations. Suppose an interesting fuzzing case provides a tuple of $([\text{args}], [\text{ret}])$ for a function $\text{f}()$, SEDiff prepares a corresponding SMT-LIB formula in the form of $(\text{assert}(\text{op } \text{f}([\text{args}]) [\text{ret}])))$, where op denotes comparison operators such as $=$, $>$, *etc.* SEDiff then passes

the formula with the extracted model to the SMT solver and queries a satisfiability solution. The solution is used for the conformance check.

Tailored bug checker. We design a tailored bug checker to facilitate compliance checks. A bug checker can naturally be designed to cross-check the satisfiability solution from the solver and its expected value, and report any unexpected deviations as bugs. However, there are two categories of inconsistent behaviors that a bug checker needs to distinguish: inconsistent behaviors associated with modeled functionalities and unmodeled ones, respectively. A model can produce inconsistent results concerning the unmodeled functionalities as they are not fully supported by the developers. Such cases, nonetheless, are not our focus in this work and should be excluded.

Our bug checker probes if the inconsistent behaviors are associated with modeled functionalities. To do this, we use an instrumented version of the original implementation to mark the code paths of the modeled functionalities. During testing, the bug checker invokes the instrumented version to monitor whether a test case triggers the instrumented modeled functionalities and thus distinguishes the two types of inconsistencies. The checker also signals the bug checking result as the feedback for input generation and fuzzing.

4.1.4 Implementation

We implemented a prototype of SEDiff currently for models whose original implementations are written in C/C++. We developed the data-flow analysis for original implementations as an LLVM pass above JUXTA [124] with 1K lines of C++ code. We used 2K lines of Python code to parse the SMT-LIB language and realize its data-flow analysis. The grey-box fuzzing stage was built above AFL using around 1K lines of C code. The tailored bug checker was realized above AFL's LLVM instrumentation tool using 500 lines of Python code. The rest of the section describes some important implementation details.

Instrumenting original implementations. The instrumentation phase of fuzzing inserts additional instructions in each basic block. The fuzzer can leverage the instructions to dynamically perceive which particular basic blocks or edges are visited in a

fuzzing trial. To realize our tailored instrumentation against modeled functionalities, we first identify all basic blocks of the modeled functionalities discovered during the data flow analysis and path mapping steps. Besides the bitmap for overall coverage, we allocate an additional shared memory area for the model coverage. We also employ instrumentation to realize the bug checker. In particular, given a test case, we record the execution trace of the original implementations and check whether it triggers a code path in the modeled functionalities.

Feedback and seed selection. AFL maintains a favored seed queue and adds test cases triggering new coverage to the queue [186, 202]. We extend AFL by adding test cases that trigger bugs to the queue according to the checker feedback. In particular, we modified the save_if_interesting() function of AFL. The function could perceive if a test case causes positive checker feedback, new model coverage and new overall coverage. We modified the cull_queue() function of AFL to perform seed selection. The cull_queue() function is used to prune the test cases while maintaining the same amount of edge coverage. We select efficient test cases that cover all visited model edges using the model coverage information.

4.1.5 Evaluation

This section evaluates SEDiff on various aspects. We aim to answer the following questions:

- How effective is SEDiff in identifying modeled functionalities?
- Can SEDiff detect bugs in internal function models?
- How do our techniques contribute to SEDiff's bug detection capability?

4.1.5.1 Experimental Setup

We include diverse state-of-the-art symbolic execution engines that employ internal function models in our evaluation. Our dataset selection criteria are to include engines that are 1) implemented in different language systems, 2) used for diverse application scenarios and 3) tested by related works. As our prototype currently supports only C/C++, we limit the engine selection to those of which the modeled internal functions

Table 4.2: Experiment results of modeled functionality identification in representative symbolic execution engines.

Engine	Impl. Lang.	Target	# Model	# Func.	# M. Func	% M. Func.	% M. Code	Time
Angr [160]	Python	Binary	165	18,518	9,839	53.13%	29.90%	38
Navex [12]	Java	Web	35	13,021	3,671	28.19%	17.15%	20
KUBO [111]	C++	Kernel	52	3,832	1,328	34.66%	25.62%	13
Deadline [193]	C++	Kernel	46	2,053	946	46.08%	39.78%	8
Total	-	-	298	37,424	15,784	42.18%	27.83%	73

M. Func. and # Func. mean the number of modeled functionalities and the total functionalities among all models in an engine.
% M. Code means the basic block level code proportion.
Time stands for the static analysis time in minutes.

are implemented in C/C++. To this end, we include 4 representative symbolic execution engines shown in the first column of Table 4.2. The engines are implemented in different programming languages such as Python, Java and C++, and have application targets of binary, web applications and kernel. We currently do not include some popular symbolic execution engines in our evaluation, either because they are not open-sourced, or they support internal functions using approaches other than function modeling [27, 35]. For example, KLEE [27] does not employ function modeling in analyzing the LLVM IR of C/C++ programs, because the relevant internal functions (*e.g.*, library functions) are also implemented in C/C++ and can be compiled into LLVM IR for analysis. Nevertheless, we believe the high diversity of our dataset allows us to thoroughly evaluate the effectiveness and scalability of SEDiff.

We download the source code of each engine from its official website and configure it with the default settings in our experiments. We first manually identify the function names and signatures of the models from the engines for minimal program synthesis. As SEDiff relies on the original implementations of the models for its analysis, we also find the original implementations of each model. The original implementations can be found from different sources such as GNU C library [4], PHP interpreter [5], *etc.* The experiments were conducted on a server running Debian GNU/Linux 9.13 (stretch) with four 24-core Intel Xeon CPUs and 512GB RAM, and a desktop computer running Debian GNU/Linux 10 (buster) with a 4-core Intel Xeon CPU and 16GB RAM.

Table 4.3: Bug detection results of SEDiff.

Engine	FP	TP	$\text{TP}_{Str.}$	$\text{TP}_{Arr.}$	$\text{TP}_{Arith.}$	TP_{Other}
Angr [160]	2	6	2	2	0	2
Navex [12]	2	35	18	14	1	2
KUBO [111]	0	3	3	0	0	0
Deadline [193]	2	2	1	0	1	0
Total	6	46	24	16	2	4

4.1.5.2 Identification of Modeled Functionalities

We use SEDiff to extract the model and perform data-flow analysis. Here we evaluate its efficacy in identifying modeled functionalities.

Identification Results

Table 4.2 presents the results of our modeled functionality identification phase. In general, not all internal functions and functionalities are modeled in practice. Our dataset in total contains 298 internal function models and $37{,}424$ functionalities, out of which $15{,}784$ (42.18%) were identified by SEDiff as modeled ones. This is consistent with our previous claim in §4.1.2.2 and can be explained by the limited human resources and high requirement of domain knowledge. It further indicates the importance of our functionality identification technique for testing internal function models.

We further compute the proportion of the modeled functionalities at source code level. Each modeled functionality corresponds to a code path in the original implementation. Therefore, we find the relevant basic blocks in the code path, count the number of basic blocks, and calculate the proportion of the identified modeled functionalities. We found that the modeled functionalities comprised only a small proportion—27.83%—of the code base in the original implementations. The detailed proportion for each engine is presented in the 8th column of Table 4.2.

Precision of Identification

We study if SEDiff can precisely identify the modeled functionalities, *i.e.*, if it can precisely map the data flow pairs. We are particularly interested in understanding how many unmodeled functionalities are wrongly identified as modeled ones. A wrong identification is an incorrect positive mapping between the data flow in the SMT-LIB formula and the one in the source code.

We conduct a manual analysis to investigate the results SEDiff reported. Due to

the large number of reported functionalities, confirming all of them requires excessive human effort and is infeasible. Instead, we sample a total of 50 models in the engines according to the number of models in them. Specifically, we randomly selected 29, 6, 8 and 7 models from the 4 engines, respectively. We believe such a random sampling approach is sufficient for obtaining some general statistics about the precision of our technique. For each sampled model/function, we manually confirm true positives in the mapped functionality paths by understanding whether each pair conveys the same functionality. We specifically compare the semantics and usage by reading the code and descriptions.

The 50 models contain in total 1,984 functionalities in the original implementation. SEDiff identified 752 as modeled functionalities that were mapped to their original implementations. Among those, our manual investigation revealed that 556 out of 752 were true modeled functionalities, resulting in a precision of 556/752 = 73.94%. Considering the inherent challenge of understanding the modeled functionalities across diverse representations, we believe that this number is reasonable for guiding path exploration.

We have investigated the causes of the 196 incorrect mappings. The main causes can be categorized into three classes. First, the data flow in some cases could not be precisely reconstructed by our heuristics due to the lack of semantics in the formulas. Our conservative approach considers all possible data flow directions in the assignment statements. As a result, SEDiff mistakenly mapped several code paths. Such situations account for around 30% of the incorrect mappings. Second, around 50% of incorrect mappings were caused by our limited inter-procedural analysis. Our data flow analysis encounters inter-procedural calls. The model and its original implementation have different supports of function calls. We temporally treat all function calls equally and do not intercept their semantics. As a result, some data flow could not be captured. Third, our current implementation does not support complex parameter logic, precise point-to analysis, *etc.* Such cases account for 20% of the incorrect mappings.

Performance

SEDiff's static analysis is highly efficient and scalable. It statically analyzed all the models in the 4 complex symbolic execution engines within 73 minutes. The analysis time for each engine in detail is shown in the 9th column of Table 4.2.

4.1.5.3 Bug Detection

We further apply SEDiff to differentially fuzz the internal function models and detect bugs. Each engine contains multiple internal function models and we separately fuzz each model for 5 runs, each with 6 hours.[4] This experiment in total took over $9{,}000$ CPU hours.

Due to the fundamentally random nature of fuzzing [93], SEDiff could report diverse results in different runs even with the same configurations. Therefore, we aggregate all reported unique bugs and present the results in Table 4.3. We do not use the average because the aggregation allows us to quantitatively analyze the false positives among the reported bugs. SEDiff reports as a unique bug if the case exhibits behavior deviations concerning modeled functionalities and triggers a unique execution trace. Overall, SEDiff is highly effective and reported 46 new bugs in the symbolic execution engines. Specifically, SEDiff identified 6, 35, 3 and 2 bugs in Angr, Navex, KUBO and Deadline, respectively. The bugs span 36 (12.08%) out of 298 internal function models. This demonstrates that many internal function models in production symbolic execution engines are still error-prone.

We observe that the bug detection result varies per engine. In particular, the state-of-the-art PHP symbolic execution engine—Navex—has more bugs than enginies targeting binary analysis and kernel analysis. We suspect that Navex statically translates certain PHP internal (built-in) functions into SMT formulas and does not carefully consider the dynamic typing feature of PHP.

The results include a few false positives. Our manual investigation showed that the false positives caused inconsistent behaviors that concerned unmodeled functionalities, which were previously incorrectly identified as modeled ones (Table 4.1.5.2). In other words, all false positives come from the false positives in the static identification phase. Differential testing approaches usually have a large number of false reports, *e.g.*, 33.3% of bugs that R2Z2 [166] reported were false cases. Nevertheless, as shown in Table 4.3, the ratio of false positives to all bugs reported by SEDiff is not high—only 6 out of 52 cases (11.54%).

[4]Though Klees *et al.* [93] recommended to run the fuzzer for complex programs for 24 hours and 5 runs, our practice shows that 6 hours here are sufficient as we separately fuzz each model and function.

While these are not vulnerabilities that can be exploited for attacks, we reported the bugs to the developers of the symbolic execution engines, which are widely employed for security applications. At the time of writing, 7 bugs have been acknowledged and the others are still under review. We will continue working with the developers to understand and fix the bugs.

Bug Characterization

Correlation with function types. We classified the bugs by the types of functions they reside in. In particular, based on the data type the functions operate on, we categorized them into 1) string processing functions, 2) array related functions, 3) arithmetic functions and 4) others. The breakdown statistics can be found in Table 4.3. We find models of certain types of functions are especially buggy. For example, models of string processing functions (*e.g.*, $\mathrm{strip\_tags}()$) and array related functions (*e.g.*, $\mathrm{explode}()$) are the dominant buggy types compared to the other types. 52.17% and 34.78% of bugs occur in these two types, respectively.

Causes of bugs. The models did wrongly behave over certain inputs in our testing. Yet it is still hard to conclude the causes due to the nature of such correctness (logic) bugs. Unlike memory-safety bugs and crashes that have clear clues of misuses (*e.g.*, use-after-free vulnerabilities), such correctness bugs are mainly related to program logic and we cannot *confidently and objectively* label particular code locations as incorrect, *i.e.*, the root causes. With the source code and manual investigation, we currently could only conclude that 6 bugs were caused by the functionality simplification that certain checks or conditions were (intentionally) discarded or ignored in the models. This could also be confirmed in related descriptions in [193]. We will closely work with the developers to investigate the bugs.

4.1.5.4 Ablation Experiments and Comparison

We design experiments to understand how each technique in SEDiff contributed to the final bug detection results and compare it to the related work. We include two SEDiff's variants together with a related work into controlled experiments:

- SEDiff$_{AFL}$. To understand the benefits of SEDiff's awareness of scope, we replace

Table 4.4: Bug detection results of SEDiff$_{AFL}$, and SEDiff$_{NF}$.

Engine	SEDiff$_{AFL}$		SEDiff$_{NF}$	
	True-Positive	False-Positive	True-Positive	False-Positive
Angr [160]	1	725	4	2
Navex [12]	17	1,235	27	2
KUBO [111]	2	230	2	0
Deadline [193]	1	98	3	2
Total	21	2,288	36	6

SEDiff's fuzzing component with a vanilla AFL into SEDiff$_{AFL}$, which thus equally explores all code space. SEDiff$_{AFL}$ is customized to report any inconsistencies as bugs regardless in modeled or unmodeled functionalities.

- SEDiff$_{NF}$. SEDiff$_{NF}$ is a variant of SEDiff that uses only the basic model coverage feedback but not the feedback from the bug checker.

The differential fuzzing framework of SEDiff includes a grey-box fuzzing component. However, besides SEDiff$_{AFL}$, we do not construct variants with other grey-box fuzzing components. This is reasonable because SEDiff is a specially designed framework for differentially fuzzing internal function models whereas other fuzzers detecting corruptions, *etc.*, are orthogonal in terms of targets. Besides, some generic techniques proposed in other fuzzers are complimentary to SEDiff. For example, though not open-sourced, CollAFL's [64] path-sensitive approach to eliminating bitmap hash collision can be integrated to SEDiff and improve SEDiff from another angle. We believe the controlled experiments by comparing SEDiff with SEDiff$_{AFL}$, and SEDiff$_{NF}$ are sufficient to demonstrate the efficacy of SEDiff. We present the experiment results in Table 4.4.

Awareness of scope. Compared to SEDiff, SEDiff$_{AFL}$ is not scope-aware. After evaluating SEDiff$_{AFL}$ with the same resource budget, SEDiff$_{AFL}$ naively reported 2,309 cases as bugs, which include a large number of false positives. To filter out them, we first applied the tailored bug checker of SEDiff on the reports, which excluded 2,285 cases concerning unmodeled functionalities from the reported results. We then manually investigated the rest cases and further removed 3 false positives. This demonstrates the necessity of our tailored bug checker in detecting bugs associated with only modeled functionalities.

In total, SEDiff$_{AFL}$ detected 21 true positives in the models, whereas SEDiff outper-

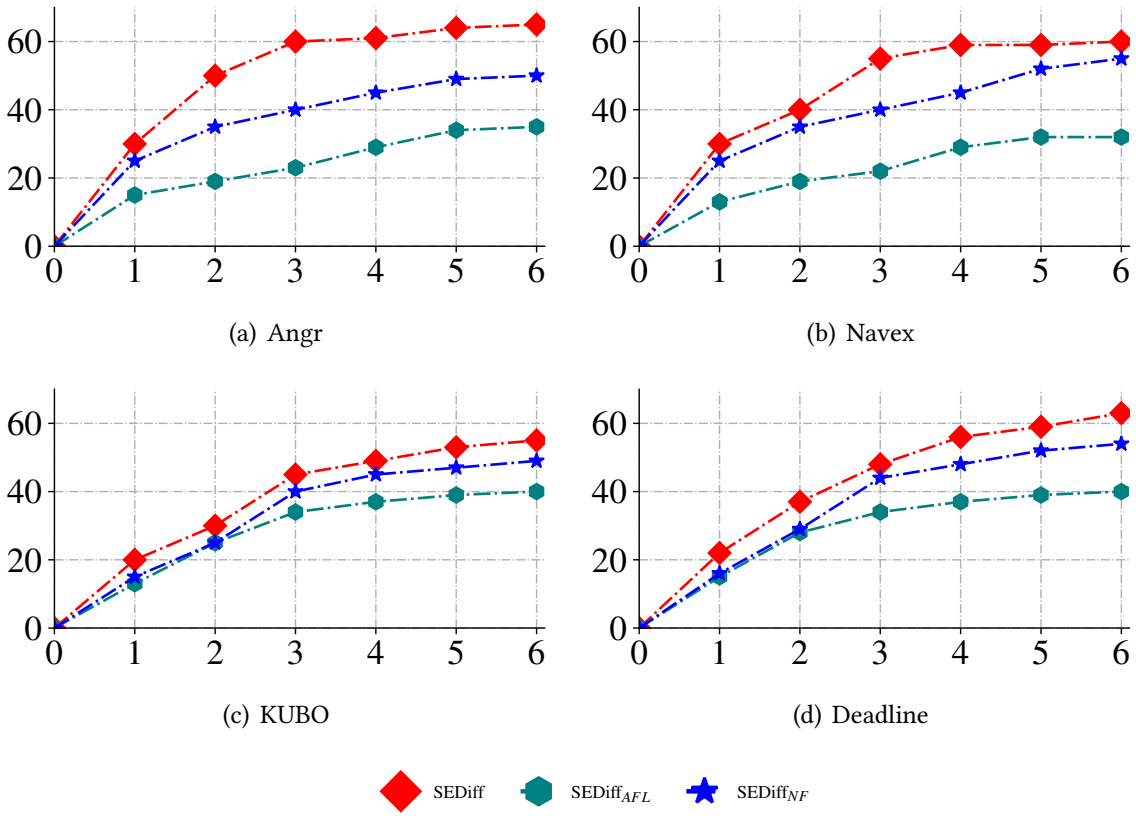


Figure 4.3: Model coverage (percentage %) over time.

formed it with 25 (108.70%) more bugs. This is because SEDiff$_{AFL}$ spent much effort on exploring unmodeled functionalities. We further characterized the bugs reported by both and found that SEDiff successfully identified *all* bugs SEDiff$_{AFL}$ reported. The results demonstrate that SEDiff's awareness of scope significantly improves the testing performance.

Feedback from bug checker. We compare SEDiff to SEDiff$_{NF}$ to investigate the benefits of the feedback from the bug checker. The results show that SEDiff$_{NF}$ detected only 36 bugs—10 fewer than SEDiff—in the modeled functionalities. The explanation lies in that the test cases receiving positive feedback from the checker are prioritized for more mutations and testing. Some erroneous code locations contain more than one bug. Therefore, by exploring more in that direction, SEDiff can potentially detect more bugs.

Code coverage. Besides bug detection, code coverage is another important measure of the effectiveness of fuzzing. Intuitively, the more execution paths are covered, the more thoroughly a target model is tested. We interpret *model coverage* concerning the

modeled functionalities instead of overall code coverage in original implementations because model coverage can better describe the exploration efficacy in the models. Each engine contains multiple models. We calculate the average model coverage among all models for an engine. We depict per engine the model coverage of SEDiff, SEDiff$_{AFL}$ and SEDiff$_{NF}$ over time during testing in Figure 4.3.

From Figure 4.3, we find that SEDiff achieves higher model coverage than its variants in all engines. At the end of the 6-hour period, SEDiff ultimately outperforms SEDiff$_{AFL}$ by 30%, 27%, 15% and 23% for Angr, Navex, KUBO and Deadline, respectively. The improvements mainly come from generating in-scope workloads. Besides, SEDiff outperformed SEDiff$_{NF}$ in all engines regarding model coverage. Additionally, Figure 4.3 shows the increasing trend of model coverage over time. It suggests that all variants usually could find a lot of paths at the beginning and then get stuck at some time. It also indicates that the model coverage in SEDiff constantly performed better than SEDiff$_{AFL}$ and SEDiff$_{NF}$.

Summary. Our comparison and characterization demonstrate the benefits of the awareness of scope and bug-checking feedback in SEDiff. The full-fledged SEDiff thus could outperform its variants in both bug detection and code coverage. In particular, *many bugs and paths would be missed without the awareness of scope.*

4.1.6 Discussion

Threat to validity. SEDiff relies on the SMT solver for the model extraction and testing. Our approach assumes the underlying SMT solvers can perform correctly during symbolic execution. We believe this is a valid assumption because the SMT solvers have been thoroughly tested before releases and bugs in SMT solvers can rarely be triggered. In our experiments, we do not observe any such cases. However, we admit that it is still possible that SMT solvers do not behave correctly, for example, having soundness bugs [116, 188]. Because of that, the results reported by SEDiff could be unreliable. To mitigate this problem, one approach is to leverage multiple SMT solvers for testing. The final result would be more reliable if all solvers give consistent solutions for a case.

Code availability of function implementations. SEDiff currently requires the source code of original implementations for its semantic mapping and grey-box fuzzing.

For the common usage of internal functions, most of them can be found directly from public channels, *e.g.*, official websites. Certain language systems do not publicize the internal functions in the form of source code but binary thus SEDiff currently is not capable to analyze them. We believe the techniques in SEDiff are generic. In the future, we plan to extend SEDiff to handle such binary forms by utilizing advanced binary analysis techniques [24, 160].

Soundness and completeness. SEDiff combines both static analysis and dynamic analysis. It is neither sound nor complete by design. On the one hand, as mentioned earlier in §4.1.5.2, the static analysis of SEDiff is not sound as it could report modeled functionalities incorrectly. This is caused by its imprecision in reconstructing the data flow and the complex parameter logic, *etc.* On the other hand, the differential testing part does not produce false positives; however, the fuzz testing nature means SEDiff is not complete and can miss bugs. Our primary goal in this work is to design an automated and scalable framework to test internal function models. Meanwhile achieving soundness or completeness is challenging. We will explore this direction in the future.

4.1.7 Related Work

Symbolic execution. Symbolic execution has been widely used for web security. SAFELI [63] performs symbolic execution on the instrumented bytecode of Java-based web applications for SQLi scanning. Kudzu [154], a JavaScript symbolic execution framework, was proposed to study the client-side vulnerabilities. Differently, SEDiff targets server-side PHP applications. Apollo [14] and Navex [12] use concolic execution for test generation. They either instrument the Zend engine or use xdebug to get runtime information for constraint solving. Compared with SEDiff, they rely on analysts to translate PHP built-in functions into the SMT-LIB language, which is found to be error-prone.

Testing symbolic execution engines. The correctness of symbolic execution is essential. To date, source code auditing is still the mainstream approach to testing symbolic execution engines. There is only a little research on testing symbolic execution engines. Kapus and Cadar [91] proposed the first study testing symbolic execution engines through differential testing. They checked the conformance of symbolic execution

against concrete execution. To the best of our knowledge, our work is the first study especially on testing internal function models in symbolic execution engines. Instead of blindly generating test cases, our work takes advantage of grey-box fuzzing to explore the input space, assisted with new concepts of modeled functionalities, new coverage guidance and a new bug checker. Rather than internal function models, some work tested SMT solvers to identify memory issues and soundness bugs [26, 188, 198]; other relevant works revealed bugs in static analyzers [46, 94], model checker [204], debugger [102], *etc.*

Differential testing. In some cases it is difficult to define a testing oracle without prior knowledge of expected behaviors. Differential testing addresses this problem by checking the behavior conformance among similar implementations. For example, Chen *et al.* employed differential fuzzing to find bugs across Java Virtual Machines [33]. Slutz proposed differential testing for database management systems [161]. Chen *et al.* used the asymmetric behaviors between testing programs to guide the fuzzer towards finding semantic bugs in SSL/TLS implementations [34]. Our work uses the function behaviors on original implementations as the ground truth, and differentially tests internal function models.

4.2 XSym: Automating Internal Function Modeling for Symbolic Execution

4.2.1 Motivation

As we introduced earlier, symbolic execution has been widely used in the domain of web applications. However, a general challenge in symbolic execution is handling language-specific built-in functions (also known as internal functions). Such built-in functions are commonly used to provide basic operations like string processing, arithmetic, and bit manipulation, *etc.* 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 requires precisely modeling their behaviors for constraints solving. As a common strategy, prior works model a small number of built-in functions into SMT-LIB language [164] for constraint solving and ignore the other ones [12, 84]. Such a modeling process is expensive, as it typically requires an excessive amount of human effort and the domain knowledge of the function behaviors. Manual models can also be error-prone, and lead to false results (both false positives and false negatives) in vulnerability detection and exploitation. For example, incorrect modeling can bring soundness problems that a true positive case can be classified as a negative [25, 188].

Unlike some languages (*e.g.*, C) whose built-in (library) functions are written in 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, some language features (*e.g.*, operators and type systems) are inconsistent between the two languages. Some operators in one language might not exist in the other, making it hard to understand the behaviors of built-in functions using such operators.

In this work, we aim to explore the feasibility of automatically modeling built-in functions for PHP symbolic execution. We face several challenges. First, the cross-language nature renders the modeling very hard. To the best of our knowledge, there exists no automated tool for modeling PHP built-in functions yet. Second, it is hard to

achieve a high *coverage* of the built-in functions. There are a large number of built-in functions in a programming language, with different function definitions. An automated method shall be able to support many of these built-in functions. Third, it is difficult to achieve an acceptable *correctness*. A built-in function can be designed for several tasks. Under different arguments, the behaviors and results can be different. Inaccurate modeling can lead to invalid execution results.

We propose a cross-language program synthesis method to automate the built-in function modeling. We make use of the C implementations of PHP built-in functions to understand their behaviors. In particular, we first 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 engine to solve the task. We propose a type inference algorithm and a syntax mapping method to overcome the challenges posed by the language feature inconsistency in the cross-language integration. Because the synthesized C program retains the semantics of the original PHP constraint solving task, the solutions of the C symbolic execution can be applied to the PHP symbolic execution. We thus achieve the goal of automatically modeling PHP built-in functions.

We implement our methodology into XSym and plan to release the source code. We successfully apply it to automatically model 287 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 experts. With the help of XSym, we can exploit 141 vulnerabilities in the evaluation dataset. Compared with XSym, a state-of-the-art manual modeling tool can exploit 133 vulnerabilities using the same constraints collected in PHP symbolic execution. XSym further 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 suggests that manual modeling can be error-prone and can cause soundness bugs.

We further thoroughly characterize our automated modeling and 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 *correct-*

ness. It can be easily applied to most of the in-scope built-in functions and achieve a high accuracy. We find that our automated method can even be used as a means for verifying the correctness of manual methods. We further show that the manual methods and our automated method complement each other as they have different strengths.

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

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

4.2.2 Problem Statement

4.2.2.1 Research Problem and Research Goals

Symbolic execution requires understanding 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 them through manual modeling. Manual methods usually take an excessive amount of human effort and require a deep understanding of the function behaviors to accurately model them. They can also be error-prone.

In this work, we first aim to explore the feasibility of automating the modeling of built-in functions for PHP symbolic execution. Second, we hope to systematically evaluate the automated models and compare with the state-of-the-art tools, in terms of function coverage, accuracy, and applications. Third, we aim to summarize the lessons we learn in our exploration of an automated method and provide some insightful sug-

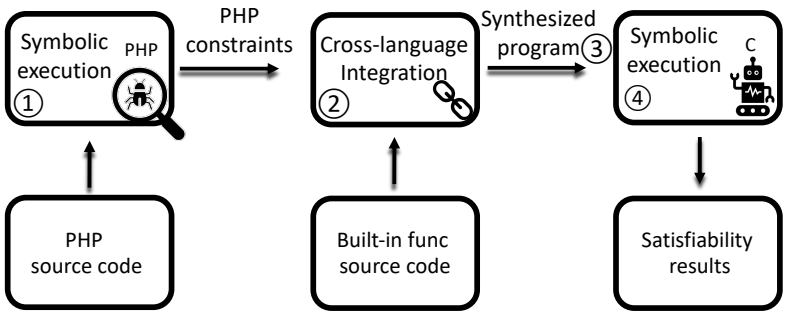


Figure 4.4: The overall methodology of XSym.

gestions to shed some light on future research.

4.2.2.2 Research Challenges

We face several challenges to automatically model built-in functions for PHP symbolic execution. First, modeling built-in functions requires understanding the behaviors of them. To automate such a 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 built-in function may contain diverse functionalities, under different input arguments, different features can be thus enabled or disabled. To achieve a high correctness, the models need to thoroughly support the entire functionality of the built-in functions.

4.2.3 Design of XSym

4.2.3.1 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 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 [81]. 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 a *synthesized program*. Afterward, 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 4.2, to exploit the XSS vulnerability in line 8, instead of converting all the 10 lines of the PHP program, we need to convert only the task of finding a possible solution for the control flow constraint, *e.g.*, strtolower ('phpbb_' . $_GET['uname']) == 'phpbb_root') && $_GET['passwd'] == 'mypassword' and the data flow constraint collected in PHP symbolic execution. As an example, the synthesized C program is illustrated in Listing 4.3. 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 concrete value assignments to the symbolic variables (*e.g.*, _GET_uname) can be provided. The solutions are then applicable to the original PHP constraints and PHP applications.

There are several technical challenges. First, the language inconsistency between PHP and C makes the constraint solving task conversion difficult. PHP is a weakly-

```php
<?php
$user = 'phpbb_' . $_GET['uname'];
$password = $_GET['passwd'];
$announcement = $_GET['ann'];

if(strtolower($user) == 'phpbb_root') {
    if($password == 'mypassword') {
        echo $announcement; // XSS
    }
}
```

Listing 4.2: An XSS vulnerability for demonstration.

```php
// 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 4.3: The synthesized program for code in Listing 4.2.

typed programming language that can initialize and use variables with assignments, while C, a statically-typed language, requires variable declarations 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 4.2) that are not defined in C. Second, PHP built-in functions are implemented inside the PHP interpreter and interact with other modules through complex PHP internal APIs. Identifying such APIs and extracting only the built-in functions 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 4.4. To the best of our knowledge, there does not exist a well maintained open-source symbolic execution framework for PHP applications. Many prior works design their own symbolic execution on their custom intermediate formats. Thus we first design a PHP symbolic execution framework in §4.2.3.2. We propose a type inference algorithm to infer the variable types, and a *light-weight syntax mapping* method to handle other PHP language features

(*e.g.*, operators and built-in functions) in §4.2.3.3. We seamlessly convert the constraint solving task and synthesize C programs in §4.2.3.4. Last, we construct symbolic inputs and leverage a C symbolic execution engine for the synthesized C program in §4.2.3.5. The results of the C symbolic execution can be applied to solve the constraint solving task in the original PHP programs.

4.2.3.2 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 [12, 135] to perform symbolic execution over our custom CFGs constructed from ASTs. Our symbolic execution creates symbolic variables for the inputs, and walks through the CFGs for constraint collections. Next, we briefly describe the key techniques in our symbolic execution.

Memory Space Management

Symbolic execution simulates the real execution with symbolic inputs. In our design, we treat all values that are not concrete 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 pairs of keys and the corresponding values. The keys are viewed as addresses of variables. We organize a dictionary, $\mathrm{Keydict}$, to store the mappings from PHP variables to keys in Memory. For assignment statements in the ASTs that initialize new variables, we first create a mapping of the variable name (left side value) and an unused key in $\mathrm{Keydict}$. We 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 general challenge of PHP analysis [49], thus we currently do not handle it.

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 relevant variables with some attack payloads, can be also collected to determine the exploitability when required. Once reaching interesting code locations (*e.g.*, concerned bugs), 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 [12, 135].

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 4.1.

$$\begin{aligned} Term \;\; t &:= c \mid v \mid f(t) \\ Formula \;\; F &:= true \mid false \mid t_1 \; op \; t_2 \mid F_1 \; op \; F_2 \end{aligned} \tag{4.1}$$

The simplest formula is a term (t), which can be either a constant value (c), a symbolic variable (v), or a PHP built-in function call ($f(t)$). 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 [152], which is applicable for the state-of-the-art solvers, *e.g.*, Z3 [55].

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 [17].

In sumamry, our symbolic execution is generic and can be applied to several tasks. We currently do not consider tackling other inherent challenges of symbolic execution (*e.g.*, path explosion) as they are orthogonal to our work in modeling built-in functions. In the example of Listing 4.2, it forks at the two if statements and explores three paths in total. It reasons about the sources of the values in the conditions, and replaces the variables in the conditions with the values in Memory. Therefore, the control flow constraint is collected. For the data flow, the critical variable $announcement in the echo statement is combined with additional attack payloads to constitute the final data flow constraint, *e.g.*, ($_GET['ann'] == " alert ('XXS')").

4.2.3.3 Cross-Language Integration

The PHP symbolic execution outputs the PHP constraints that are collected for certain analysis tasks. The constraints inherit 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 exist in another language. For example, PHP has the equality operator (*i.e.*, ==) that compares only the values (but not types) 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 method to overcome the syntax inconsistency. With these

techniques, we then synthesize a C program that equally represents the PHP constraints.

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 the 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 of variables. Locally, the operator behaves as the context of 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. From the observation, we first decide operand types based on operators, and the types of argument and return values at call sites based on the function 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 constraint in Listing 4.2 as an illustration example, in the control-flow constraint (strtolower (' phpbb_' . $_GET['uname'] == 'phpbb_root' && $_GET['passwd'] == 'mypassword'), because of the concatenation operator, $_GET['uname'] is inferred as in 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 finally.

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 4.1).

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 fewer 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 complex 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 () takes a pointer of hash tables to pass argument values. The special argument-fetching design requires the 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.

4.2.3.4 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 step, 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_pro()) in Listing 4.3, line 5-7 declares 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.

4.2.3.5 Symbolic Execution on Synthesized Programs

We perform symbolic execution on the synthesized C program. In this work, we use KLEE [27], 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.*, make_symbolic ()) to declare variables as symbolic inputs, *e.g.*, _GET_uname, that are to be solved. 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 advantage 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 solvers 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.

4.2.4 Implementation

We implement the aforementioned methodology into XSym and plan to release our prototype implementation. In particular, we implemented our PHP symbolic execution engine on top of the PHP-Parser [131] 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 [27] 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 a good performance.

4.2.5 Evaluation

In this section, we evaluate XSym in three aspects: 1) *coverage*, 2) *correctness*, and 3) *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 §4.2.5.2 and the correctness in §4.2.5.3. Next, in §4.2.5.4, we demonstrate the efficacy of XSym in exploiting real-world applications. Last, we characterize manual and automated methods in §4.2.5.5.

4.2.5.1 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 [12]. 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 [12]. 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 [11]) and vulnerability analysis (*e.g.*, RIPS [49]).

4.2.5.2 Coverage

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 [52], this set has covered more than 90% of the most popular functions.

Coverage for Built-in Functions

XSym achieves a high coverage. The evaluation results show XSym is able to automatically model 287 functions. In contrast, Navex modeled only 35 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.

```
1 // include built-in function
2 #include "php-built-in.h"
3
4 int syn_pro() {
5     php_string sol; //symbolic return value
6     if(php_is_equal(strtolower("TESTCASE"), sol) ) {
7             assert(0); // synthesized error
8     }
9 }
```

Listing 4.4: A synthesized C program for evaluating the correctness.

4.2.5.3 Correctness

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")== $sol), we synthesize the C program in Listing 4.4. We try to symbolize either the arguments ($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 the correctness, we leverage the test suite shipped with the PHP interpreter, which includes the expected return values for executing built-in functions with the provided concrete arguments.

Results

The evaluation results are shown in Table 4.5. Since string-related and arithmetical functions are the most prevalent in PHP, we divided these 287 functions into three classes: string-related functions, arithmetical functions, and the others. We observe that XSym

Table 4.5: 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%

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 [97].

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 correctness evaluation. First, to a 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.

4.2.5.4 Vulnerability Detection

To understand how our automated models help security applications, we apply XSym for the detection of SQL injection and cross-site scripting vulnerabilities, which are the dominant 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 [12] and are orthogonal to our work.

Overall Results

The evaluation results are shown in Table 4.6. We use the subscripts X and N to denote the results of XSym and Navex, respectively. Sinks, Sol, Rep, and TP in the column headings mean the number of tainted sinks, solvable sinks (including both satisfiable and unsatisfiable), reported bugs by a tool and manually confirmed true-positive bugs, respectively. We also show the false positives in the parenthesis. 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 4.6. 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 language 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. However, as we demonstrate next, Navex has a quite high number of false

Table 4.6: Statistics of vulnerability detection for SQLi and XSS.

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	9,090	25	12	12	7	7	7	2	2	2	1	1	2
WebChess (0.9)	29	5,219	13	6	8	4	7 (4)	5	16	11	12	8	9	8
WordPress (4.7.4)	699	181,257	0	0	0	0	0	0	0	0	0	0	0	0
HotCRP (2.1)	145	57,717	0	0	0	0	0	0	0	0	0	0	0	0
SchoolMate (1.5.4)	63	15,375	50	33	36	25	29 (4)	28	11	8	9	6	8 (2)	7
HotCRP (2.6.0)	43	14,870	0	0	0	0	0	0	4	4	4	2	3 (1)	2
Zen-Cart (1.5.5)	1,010	109,896	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	21,108	0	0	0	0	0	0	0	0	0	0	0	0
osCommerce (2.3.3)	541	49,378	1	1	1	1	0	1	47	28	33	18	21 (2)	21
osCommerce (2.3.4)	684	63,631	0	0	0	0	0	0	5	3	3	2	2	2
Drupal (8.3.2)	8,626	585,094	0	0	0	0	0	0	0	0	0	0	0	0
WeBid (0.5.4)	300	65,302	43	38	39	29 (1)	26 (2)	30	13	10	8	4	4	5
Gallery (3.0.9)	510	39,218	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	1,322	2	1	2	1	1	1	1	1	1	1	1	1
Joomla (3.7.0)	2,764	302,701	0	0	0	0	0	0	0	0	0	0	0	0
FAQForge (1.3.2)	17	1,676	17	5	8	3	4	4	7	4	4	3	3	3
LimeSurvey (3.1.1)	3,217	965,164	0	0	0	0	0	0	0	0	0	0	0	0
Collabtive (3.1)	836	172,564	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)	3,201	215,870	0	0	0	0	0	0	0	0	0	0	0	0
CPG (1.5.46)	359	305,245	3	2	3	2	2	2	11	7	9	5	5 (1)	6
MediaWiki (1.30.0)	3,680	537,913	0	0	0	0	0	0	1	0	1	0	0	0
PHPBB (3.0.11)	74	29,164	4	3	3	3	3 (1)	3	16	13	16	8 (1)	11 (6)	7
PHPBB (3.0.23)	387	158,756	5	4	4	1	1	1	0	0	0	0	0	0
Total	27,473	3,909,736	172	110	120	81 (1)	84 (11)	87	139	95	105	62 (1)	72 (12)	68

Sinks, Sol, Rep, and TP denote the number of tainted sinks, solvable sinks (including satisfiable and unsatisfiable), reported bugs by a tool, and true-positive bugs, respectively.
The subscripts X and N denote the results of XSym and Navex.
The numbers in parenthesis mean false positives.

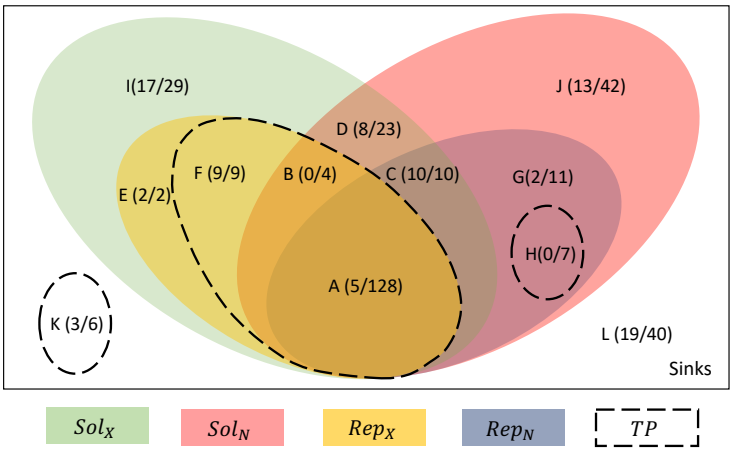


Figure 4.5: Distribution of vulnerability detection results. The alphabets (A-L) denote different situations. The numbers in parenthesis denote (number of cases including built-in functions supported by only XSym/ total number of cases).

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 4.6. 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.

Analysis

To clearly understand the capability of each tool, we depict the distribution of results in Figure 4.5. 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 reasons.

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 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 [62].

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 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 [12, 16], but not a limitation of our 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 *constraint relaxation*. The results suggest that Navex could have generated more accurate results if more built-in functions were

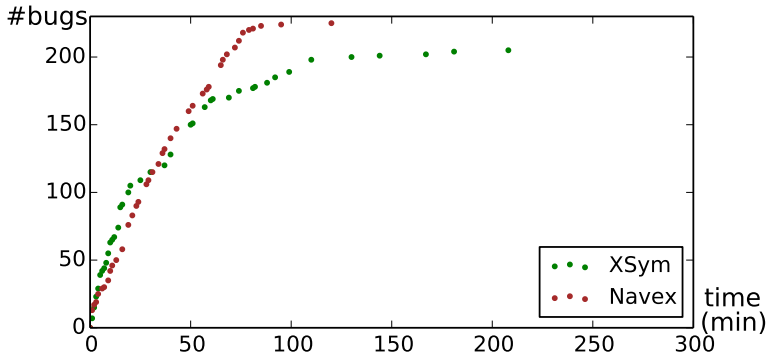


Figure 4.6: Number of solved bugs over time for XSym and Navex

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 a lower false-negative rate. Navex, because of its *constraint relaxation* and *functionality simplification*, produces wrong decisions or solutions for 27 cases, while XSym has only two.

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 4.6. 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.

4.2.5.5 Characterizing Manual and Automated Modeling Methods

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 complex logic, manual modeling typically cannot cover all built-in functions or provide 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 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 the 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 automated 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.

4.2.6 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 [35] 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 methods. Our methodology uses a C symbolic execution tool (*i.e.*, 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 manually modeled functions to their manual models, and seamlessly integrate the manual models into the execution context in the C symbolic execution. As a result, this might require some enhancement on the underlying C symbolic execution tool. We leave it as our future work.

4.2.7 Related Work

Program synthesis. Program synthesis as the task of generating programs from user intent, has been widely used for studying security problems [80]. Singularity [187] transforms the complexity testing problem to optimal program synthesis to identify performance bugs. Aspire [31] synthesizes application specifications from input-output examples to meet user intent and to guarantee the security. Many fuzzing works [82, 112, 138] 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 internal functions. Analyzing built-in functions is also common in static analysis. Pixy [90] configures 29, and RIPS [49] classifies and analyzes over 900 built-in functions in static analysis for taint propagation and sanitization. In symbolic execution and test generation, SMART [72] 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 [140] 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 [73]. However, they can only cover a very few function situations. Tools like Chainsaw [11], Navex [12], and UChecker [84] that manually model built-in functions are shown to be error-prone. In comparison, XSym employs symbolic execution on C implementations of PHP built-in functions to automatically model their behaviors.

□ End of chapter.

Chapter 5

Future Work and Conclusion

5.1 Future Work

We listed the several representative examples of software bugs in Table 1.1 regarding the three properties of software systems. In this thesis, we have presented novel automated bug detection techniques to solve these problems in three directions. However, software systems are evolving and the table is by no means complete. In the future, we plan to extend the presented research and develop new foundations, tools, and infrastructures to further advance software quality. Some of the potential research directions are as follows.

Extensive testing for collaborative development. Software development is progressive and evolutionary with periodical updates of bug patches and functionality addition from different developers. This introduces two sides of problems. On the one hand, patches in the software updates might not sufficiently fix the bugs, and the known issues will remain; newly added functionalities might introduce unknown bugs. On the other hand, one developer might misunderstand code written by others and therefore make mistakes in the setting of collaborative development. In the future, we will utilize the development history to identify the code changes in each software update and develop advanced techniques to find suspicious buggy code. We plan to develop novel algorithms to learn the patterns of developers' programming mistakes from the development history and use them to check new bugs in software updates.

Efficient symbolic execution for interpreted languages. Besides internal functions, there are several open problems for performing symbolic execution for interpreted languages such as PHP and Python. Prior symbolic execution solutions build their custom intermediate representations (IRs) and symbolically interpret the IRs for the analysis. However, the symbolic interpretation can hardly cover the complex language semantics and widespread native methods [107]. For instance, applications written in interpreted languages commonly use arrays in adaptive and changeable sizes; symbolically managing arrays on the custom IRs is challenging because it is non-trivial to evaluate the array access operations. We plan to instead perform the symbolic analysis on the internal IRs of language interpreters such as PHP operation code, just-in-time compilation IRs. Since the internal IRs natively model the complex language semantics and native methods, such a solution can overcome the challenges from the root.

Artificial intelligence for bug detection. The integration of artificial intelligence (AI) techniques with automated program analysis is a promising approach to enhance bug detection capabilities. By leveraging machine learning and deep learning models, it becomes possible to identify patterns and coding mistakes that often result in software bugs [8, 205]. This can significantly improve bug detection accuracy during the development process. Additionally, AI models have shown promise in detecting malware and vulnerabilities in binary programs, further highlighting their potential in bug detection [28, 182]. Moving forward, the objective is to integrate advanced AI techniques with program analysis solutions to enable intelligent bug detection and provide developers with more robust tools for ensuring software quality and reliability.

Embracing hardware-software co-designs. We are currently in the era of the technology revolution of both software and hardware. New technologies are being developed to revolutionize how we use and interact with computer systems. For example, trusted execution environments better protect data confidentiality and integrity; persistent memory improves the performance and scalability of conventional storage systems; IoT devices substantially change how we live at home and in the city. The research presented in this thesis is an example that solves different challenges in software analysis. In the future, we plan to further target hardware analysis and embrace its coordination with software technology. For instance, we hope to exploit the hardware capability to

improve the quality of computer systems. The research experience in software analysis has laid a good foundation to investigate these problems in the long-term future.

5.2 Conclusion

We all expect to develop or use software systems with correctness, security, and performance guarantees. However, it remains arguably impossible today due to routinely occurred software bugs. This thesis has made an essential step in taking us closer to achieving this expectation with automated bug detection techniques. In particular, we characterized software bugs in different categories, and presented tailored and generic approaches to detecting them. With the approaches, we revealed hundreds of new software bugs and vulnerabilities, which led to urgent software updates and tool enhancements in foundational software systems. We also open-sourced our prototype implementations of the proposed approaches to contribute to the community. We believe that the profound research efforts and the promising research results have demonstrated the significance of the contributions achieved in this thesis.

We envision a future where trustworthy software systems become a reality. However, software systems keep evolving and posing new challenges, making conventional software analysis techniques inadequate in realizing this vision. The content presented in this thesis can be generally helpful and serve as a foundation to tackle the new problems. Going forward, we would therefore expect more efforts to advance the research in this important yet challenging domain.

□ End of chapter.

Bibliography

[1] Angr pull request #84, March 2018. https://github.com/angr/angr/pull/889.

[2] Angr document: Simprocedure, December 2021. https://docs.angr.io/extending-angr/simprocedures.

[3] Angr pull request #2956, October 2021. https://github.com/angr/angr/pull/2956.

[4] The gnu c library (glibc), December 2021. https://www.gnu.org/software/libc/.

[5] The php interpreter, December 2021. https://github.com/php/php-src.

[6] Navex, March 2022. https://github.com/aalhuz/navex.

[7] Giovanni Agosta, Alessandro Barenghi, Antonio Parata, and Gerardo Pelosi. Automated security analysis of dynamic web applications through symbolic code execution. In *2012 Ninth International Conference on Information Technology-New Generations*, 2012.

[8] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding bugs using your own code: Detecting functionally-similar yet inconsistent code. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual event, August 2021.

[9] Alexander Aiken and Edward L Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional programming languages and computer architecture*, 1993.

[10] Alexa. Alexa top 1 million sites, 2021. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[11] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[12] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. Navex: Precise and scalable exploit generation for dynamic web applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[13] Vincenzo Arceri and Sergio Maffeis. Abstract domains for type juggling. *Electronic Notes in Theoretical Computer Science*, 2017.

[14] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, July 2008.

[15] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[16] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian 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, April 2017.

[17] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 2018.

[18] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011.

[19] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, 2010.

[20] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, June 2017.

[21] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[22] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[23] Fraser Brown, Deian Stefan, and Dawson 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, August 2019.

[24] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011.

[25] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. 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, 2006.

[26] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1459–1470, Seoul, Korea, June–July 2020.

[27] Cristian Cadar, Daniel Dunbar, and Dawson 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, December 2008.

[28] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, USA, November 2016.

[29] Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? or, use this latex class file to pwn your computer. In *3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2010.

[30] Hongxu Chen, Bihuan Chen, Yinxing Xue, Xiaofei Xie, Yang Liu, Yuekang Li, and Xiuheng Wu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[31] Kevin Chen, Warren He, Devdatta Akhawe, Vijay D'Silva, Prateek Mittal, and Dawn Song. Aspire: Iterative specification synthesis for security. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

[32] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering*, 2014.

[33] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, June 2016.

[34] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in ssl/tls implementations. In *Proceedings of the 10th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, August 2015.

[35] Vitaly Chipounov, Volodymyr Kuznetsov, and George 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, March 2011.

[36] Codiad. Codiad Web IDE, May 2020. http://codiad.com.

[37] commonmark.org. commark-spec, 2021. https://github.com/commonmark/commonmark-spec.

[38] The MITRE Corporation. Cve-2019-10231, May 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10231.

[39] The MITRE Corporation. Cve-2020-10568, May 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10568.

[40] The MITRE Corporation. Cve-2020-14450, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14450.

[41] The MITRE Corporation. Cve-2020-26409, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-26409.

[42] The MITRE Corporation. Cve-2020-8088, May 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8088.

[43] The MITRE Corporation. Cve-2021-22217, 2021. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22217.

[44] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, August 2003.

[45] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, July 2018.

[46] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 341–355, Berlin, Heidelberg, April 2012. Springer-Verlag.

[47] Brian W Curry, Andrew Trotman, and Michael Albert. Extricating meaning from wikimedia article archives. In *16th Australasian Document Computing Symposium*, 2010.

[48] Cve. Browse vulnerabilities by date, 2022. https://www.cvedetails.com/browse-by-date.php.

[49] Johannes Dahse and Thorsten 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, February 2014.

[50] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[51] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th USENIX Security Symposium (Security)*, Montréal, Canada, August 2009.

[52] Dams. Top 100 PHP functions, May 2018. https://www.exakat.io/top-100-php-functions/.

[53] Pratap Dangeti. *Statistics for machine learning*. Packt Publishing Ltd, 2017.

[54] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. 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, November 2018.

[55] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, March–April 2008.

[56] Henri Maxime Demoulin and Isaac Pedisich. Detecting application-layer denial-of-service attacks with finelame. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[57] Drupal. Drupal markdown module, 2021. https://www.drupal.org/project/markdown.

[58] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, May 2022.

[59] Veronika Durcekova, Ladislav Schwartz, and Nahid Shahmehri. Sophisticated denial of service attacks aimed at application layer. In *2012 Elektro*. Ieee, 2012.

[60] ffmpeg. A complete, cross-platform solution to record, convert and stream audio and video, 2022. https://ffmpeg.org/.

[61] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *International Static Analysis Symposium*, 2000.

[62] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 2006.

[63] Xiang Fu and Kai Qian. Safeli: Sql injection scanner using symbolic execution. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, 2008.

[64] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[65] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2019.

[66] GitHub. Github docs: About readmes, 2021. https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-repository-on-github/about-readmes.

[67] GitHub. Github markdown, 2021. https://docs.github.com/en/rest/reference/markdown.

[68] Github. Codeql, 2023. https://codeql.github.com/.

[69] GitLab. Gitlab flavored markdown, 2021. https://docs.gitlab.com/ee/user/markdown.html.

[70] GitLab. Gitlab markdown api, 2021. https://docs.gitlab.com/ee/api/markdown.html.

[71] Gnu. Backtraces, 2022. https://www.gnu.org/software/libc/manual/html_node/Backtraces.html.

[72] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)*, Nice, France, January 2007.

[73] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[74] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2008.

[75] Google. Oss-fuzz, 2021. https://google.github.io/oss-fuzz/.

[76] Google. Crash-reporting system., 2022. https://chromium.googlesource.com/breakpad/breakpad.

[77] Google. Google fuzzer test suite, 2023. https://github.com/google/fuzzer-test-suite.

[78] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, Long Beach, California, January 2005.

[79] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Atlanta, Georgia, October 1997.

[80] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 2017.

[81] Hhvm. Ending PHP Support, and The Future Of Hack. https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html, September 2018.

[82] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[83] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[84] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[85] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Computer Networks*, 2017.

[86] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012.

[87] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, June 2012.

[88] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium (Security)*, San Diego, CA, August 2004.

[89] Nenad Jovanovic, Christopher Kruegel, and Engin 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, May 2006.

[90] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 2010.

[91] Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL, October–November 2017.

[92] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[93] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, October 2018.

[94] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*, Beijing, China, July 2019.

[95] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Phantm: Php analyzer for type mismatch. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Santa Fe, NM, November 2010.

[96] Marzena Kryszkiewicz. The cosine similarity in terms of the euclidean distance. In *Encyclopedia of Business Analytics and Optimization*. IGI Global, 2014.

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

[98] Xuan-Bach D Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. Saffron: Adaptive grammar-based fuzzing for worst-case analysis. *ACM SIGSOFT Software Engineering Notes*, 2019.

[99] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[100] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual event, August 2021.

[101] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2019.

[102] Daniel Lehmann and Michael Pradel. Feedback-directed differential testing of interactive debuggers. 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, November 2018.

[103] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, July 2018.

[104] Penghui Li, Yinxi Liu, and Wei Meng. Understanding and detecting performance bugs in markdown compilers. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Online event, November 2021.

[105] Penghui Li and Wei Meng. Lchecker: Detecting loose comparison bugs in php. In *Proceedings of the Web Conference (WWW)*, Ljubljana, Slovenia, April 2021.

[106] Penghui Li, Wei Meng, and Kangjie Lu. SEDiff: Scope-Aware Differential Fuzzing to Test Internal Function Models in Symbolic Execution. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Singapore, November 2022.

[107] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. On the feasibility of automated built-in function modeling for php symbolic execution. In *Proceedings of the Web Conference (WWW)*, Ljubljana, Slovenia, April 2021.

[108] Vickie Li. PHP Type Juggling Vulnerabilities, September 2019. https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09.

[109] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. Polycruise: A cross-language dynamic information flow analysis. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[110] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019.

[111] Changming Liu, Yaohui Chen, and Long 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, February 2021.

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

[113] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May–June 2014.

[114] Llvm. libfuzzer, 2021. https://hammer-vlsi.readthedocs.io/en/stable/LibFuzzer.html.

[115] Changhua Luo, Wei Meng, and Penghui Li. SelectFuzz: Efficient directed fuzzing with selective path exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, USA, May 2023.

[116] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sacramento, CA, November 2020.

[117] Paul Dan Marinescu and Cristian Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *2012 34th International Conference on Software Engineering (ICSE)*. Ieee, 2012.

[118] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, 2010.

[119] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 2015.

[120] MediaWiki. MediaWiki, May 2020. https://www.mediawiki.org/wiki/MediaWiki.

[121] Sipke Mellema. Spot The Bug Challenge 2016 Write-up, January 2016. https://www.securify.nl/blog/SFY20170103/spot-the-bug-challenge-2016-write-up.html.

[122] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[123] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol)*, 2009.

[124] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[125] Kathryn Mohror and Karen L Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In *Proceedings of the conference on high performance computing networking, storage and analysis*, pages 1–12, 2009.

[126] Monstra. Github of Monstra, May 2020. https://github.com/monstra-cms/monstra.

[127] Sven Morgenroth. Detailed explanation of php type juggling vulnerabilities, August 2018. https://www.netsparker.com/blog/web-security/php-type-juggling-vulnerabilities/.

[128] Sven Morgenroth. Type Juggling Authentication Bypass Vulnerability in CMS Made Simple, September 2018. https://www.netsparker.com/blog/web-security/type-juggling-authentication-bypass-cms-made-simple/.

[129] mrash. afl-cov - afl fuzzing code coverage, 2021. https://github.com/mrash/afl-cov.

[130] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, San Sebastian, Spain, October 2020.

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

[132] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.

[133] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[134] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and practice of object systems*, 1999.

[135] Oswaldo Olivo, Isil Dillig, and Calvin 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, Colorado, October 2015.

[136] osCommerce. Website of osCommerce, May 2020. https://www.oscommerce.com.

[137] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. $parmesan$: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.

[138] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[139] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 1995.

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

[141] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dvul: Discovering 1-day vulnerabilities through binary patches. In *Proceedings of the 2019 International Conference on Dependable Systems and Networks (DSN)*, Portland, Oregon, June 2019.

[142] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. Ieee, 2017.

[143] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[144] Marcel Pham, Van Thuan, Manh Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[145] Marcel Pham, Van Thuan, Manh Dung Nguyen, and Abhik Roychoudhury. Github repository of aflgo, 2022. https://github.com/aflgo/aflgo.

[146] Php. PHP string in numeric contexts. https://www.php.net/manual/en/language.types.string.php#language.types.string.conversion, May 2020.

[147] Php. Type juggling, 2021. https://www.php.net/manual/de/language.types.type-juggling.php.

[148] Php-ai. PHP-ML – a machine learning library, May 2020. https://github.com/php-ai/php-ml.

[149] PHPLint. PHPLint, May 2020. http://www.icosaedro.it/phplint/.

[150] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.

[151] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, Florence, Italy, May 2015.

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

[153] Korosh Koochekian Sabor, Mohammad Hamdaqa, and Abdelwahab Hamou-Lhadj. Automatic prediction of the severity of bugs using stack traces. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, 2016.

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

[155] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

[156] Benoit Schweblin. Stackedit, 2021. https://stackedit.io/.

[157] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. Mc2: Rigorous and efficient directed greybox fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, November 2022.

[158] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1997.

[159] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, September 2018.

[160] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[161] Donald R Slutz. Massive stochastic testing of sql. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, New York, USA, August 1998.

[162] Chris Smith. PHP Magic Tricks: Type Juggling, 2015. https://owasp.org/www-pdf-archive/PHPMagicTricks-TypeJuggling.pdf.

[163] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a nids. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[164] Smt-lib. Smt-lib, October 2020. http://smtlib.cs.uiowa.edu/.

[165] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011.

[166] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2z2: Detecting rendering regressions in web browsers through differential fuzz testing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May 2022.

[167] Prashast Srivastava, Stefan Nagy, Matthew Hicks, Antonio Bianchi, and Mathias Payer. One fuzz doesn't fit all: Optimizing directed fuzzing via target-tailored program state restriction. In *Annual Computer Security Applications Conference*, Austin, TX, 2022.

[168] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[169] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, November 2016.

[170] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, 2016.

[171] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 2014.

[172] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, August 2011.

[173] Fangqi Sun, Liang Xu, and Zhendong Su. Detecting logic vulnerabilities in e-commerce applications. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[174] syoyo. Tiny openexr image library, 2023. https://github.com/syoyo/tinyexr.

[175] thorfdbg. libjpeg, 2023. https://github.com/thorfdbg/libjpeg.

[176] Tyler Borland (TurboBorland). Writing Exploits For Exotic Bug Classes: PHP Type Juggling, August 2013. https://turbochaos.blogspot.com/2013/08/exploiting-exotic-bugs-php-type-juggling.html?view=classic.

[177] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, September 2018.

[178] Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. Tracesim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020.

[179] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, 2016.

[180] W3Techs. Usage statistics of content management systems, May 2020. https://w3techs.com/technologies/overview/content_management.

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

[182] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jtrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, July 2022.

[183] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[184] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, Montréal, Canada, May 2019.

[185] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[186] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[187] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 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, November 2018.

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

[189] WordPress. Support–wordpress.com: Markdown block, 2021. https://wordpress.com/support/wordpress-editor/blocks/markdown-block/.

[190] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie 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, February 2020.

[191] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 International Conference on Dependable Systems and Networks (DSN)*, Lisbon, Portugal, June–July 2009.

[192] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. Identifying features of android apps from execution traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Ieee, 2019.

[193] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo 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.

[194] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. ARCUS: Symbolic root cause analysis of exploits in production systems. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual event, August 2021.

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

[196] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[197] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[198] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing smt solvers via two-dimensional input space exploration. In *Proceedings of the 30th International Symposium on Software Testing and Analysis (ISSTA)*, Online, July 2021.

[199] yguoaz/. Docker of beacon, 2022. https://hub.docker.com/r/yguoaz/beacon.

[200] Hyunguk Yoo and Taeshik Shon. Grammar-based adaptive fuzzing: Evaluation on scada modbus protocol. In *Proceedings of IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2016.

[201] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[202] Michal Zalewski. American fuzzy lop, 2023. https://github.com/google/AFL.

[203] zeux. Light-weight, simple and fast xml parser for c++ with xpath support, 2023. https://github.com/zeux/pugixml.

[204] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and understanding bugs in software model checkers. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, August 2019.

[205] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *Proceedings of 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021.

[206] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.