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

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

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

Kangjie Lu
University of Minnesota
Minneapolis, USA
kjlu@umn.edu

ABSTRACT

Symbolic execution has become a foundational program analysis technique. 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 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 paper, 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 equips 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.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; • Security and privacy → Software security engineering.

KEYWORDS

Differential Testing; Symbolic Execution; Internal Function Models

ACM Reference Format:
Penghui Li, Wei Meng, and Kangjie Lu. 2022. 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 '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549080

1 INTRODUCTION

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 [12, 15, 31, 38], vulnerability assessment [9], root cause analysis [48], *etc.* For example, recent symbolic execution frameworks are able to detect vulnerabilities in well-tested software like OpenJEPG, Chrome and Firefox [12, 38].

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.

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 §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 [9, 47].

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 [9, 28, 31, 40, 47] 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* [9, 31, 40, 47]. For example, a popular symbolic execution engine, Angr [40], defines SimProcedure [2] for modeling.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549080

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 [23] checked whether the results of symbolic execution conform to concrete execution for programs generated by Csmith [49]. 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. To the best of our knowledge, the most relevant work, XSym [28], leverages an existing regression test suite to check internal function models in a PHP symbolic execution engine—Navex [9]. However, its approach is restricted in both scalability and depth: it requires human efforts to extract the models and can only test part of the models. Thereby it could not precisely reveal bugs in internal function models, as we will show in §5.4.

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 [31, 40, 47], 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 paper, we design a *scope-aware* differential testing framework, SEDiff. It incorporates several new techniques to overcome the above-mentioned 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 [21]) in a uniform format, SMT-LIB language [11]. 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 [40]) 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. It significantly outperformed the related work XSym [28] by detecting 33 more bugs. 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 [8].

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

- **First in-depth study.** We propose the first comprehensive study on internal function models, and define several key concepts of this problem.
- **New techniques.** 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.

- **Numerous new bugs.** With SEDiff, we found numerous new bugs in the internal function models in the state-of-the-art symbolic execution engines.

2 MOTIVATION AND PROBLEM STATEMENT

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 [28]. 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 [35]. 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,[1] 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 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 `substr()` for 2,211 times, which accounted for 3.17% of all function calls; GNU CoreUtils called `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 [28, 29]. 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 [28, 38].

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 [9, 31, 40, 47]. 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 [9, 31, 40, 47]; the constructed model can be seamlessly integrated into underlying symbolic execution engines thus tackling the compatibility issue [28]. Many well-known symbolic execution engines (*e.g.,* Angr [40], Navex [9], *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 paper.

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 [28, 47], delaying the patching of critical bugs [46], or introducing regression bugs [34]. 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.

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

[1] 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.

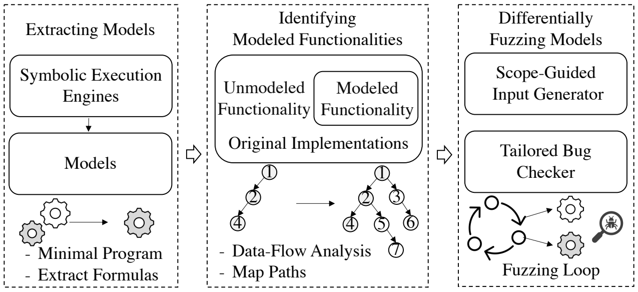


Figure 1: The architecture of SEDiff.

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 [7]. 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.

3 DESIGN OF SEDIFF

In this paper, 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 1. SEDiff entails overcoming the technical challenges with several new observations and techniques:

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 [28]. 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 §3.1.

```
1 <?php
2 $arg = $_POST["test"];
3 $ret;
4
5 if(abs($arg) == $ret) {
6     // check point
7 }
```

Listing 1: A minimal program that invokes an internal function (model).

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.

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

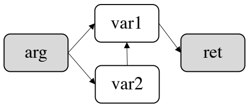
We extract the models based on a key observation—*the uniformity of SMT-LIB*. The SMT-LIB standard is a widely-adopted international initiative aiming at facilitating research and development in Satisfiability Modulo Theories (SMT) [11]. SMT-LIB specifies a general language for input formulas that SMT 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 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

```
(assert (= ret var1))
(assert (= var2 (* (-1) arg)))
(assert (= var1
    (ite (> arg 0) (arg)) (var2)
    )
 )
```

(a) Internal function model.



(b) Data dependencies of variables.

Figure 2: The simplified function model of PHP internal function abs() and the data dependency diagram of the variables.

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 execution 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 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 [9]).

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 [21] and CVC4 [10]. 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 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.

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.

Table 1: Data-flow representation system.

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	$arg \in e$

3.2.1 *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 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, (assert (= 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.,* `(ret=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 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 2(a) is such a case where lvalue is `var2` and rvalue is `(* (-1) 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 2(b). While being simple, our experiments in §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 1. The formulas are later compared to the ones from the original implementations. As shown in Table 1, the simplest form is an expression `e`, which can be either a constant value (`c`), a function argument (`arg`), a function call (`f(e)`), or an operation above expressions ($\texttt{e}_1$ `op` $\texttt{e}_2$). Function arguments and return values can also be described as expressions. In the example of Figure 2(a), the two possible data-flow formulas are `ret=arg` and `reg=-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 [36], 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 1.

3.2.2 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 $\texttt{arg}_i$ 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 1, and outputs modeled code locations in the original implementations.

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.

3.3.1 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 [51] and LibFuzzer [32] 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 [32, 51]. 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 [44], 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 (§3.3.2).

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 [24, 37]. 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 [44].

3.3.2 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 `([args], [ret])` for a function `f()`, SEDiff prepares a corresponding SMT-LIB formula in the form of `(assert(op f([args]) [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 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 [36] 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 prototype is available with the DOI: 10.5281/zenodo.6665380 [8]. 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 [44, 51]. 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.

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?

In the rest of this section, we first describe the experimental setup (§5.1). Then we evaluate the effectiveness of modeled functionality identification (§5.2) and bug detection (§5.3). Next, we present the ablation study and comparison (§5.4).

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 are implemented in C/C++. To this end, we include 4 representative symbolic execution engines shown in the first column of Table 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 [15, 19]. For example, KLEE [15] 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.

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.

5.2.1 Identification Results. Table 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 §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 2.

5.2.2 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.

5.2.3 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 2.

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.[2] This experiment in total took over 9,000 CPU hours.

Due to the fundamentally random nature of fuzzing [25], 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 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,

[2] Though Klees *et al.* [25] 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.

Table 2: Experiment results of modeled functionality identification in representative symbolic execution engines. # 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.

Engine	Impl. Lang.	Target	# Model	# Func.	# M. Func	% M. Func.	% M. Code	Time
Angr [40]	Python	Binary	165	18,518	9,839	53.13%	29.90%	38
Navex [9]	Java	Web	35	13,021	3,671	28.19%	17.15%	20
KUBO [31]	C++	Kernel	52	3,832	1,328	34.66%	25.62%	13
Deadline [47]	C++	Kernel	46	2,053	946	46.08%	39.78%	8
Total	-	-	298	37,424	15,784	42.18%	27.83%	73

Table 3: Bug detection results of SEDiff.

Engine	FP	TP	TP$_{Str.}$	TP$_{Arr.}$	TP$_{Arith.}$	TP$_{Other}$
Angr [40]	2	6	2	2	0	2
Navex [9]	2	35	18	14	1	2
KUBO [31]	0	3	3	0	0	0
Deadline [47]	2	2	1	0	1	0
Total	6	46	24	16	2	4

Table 4: Bug detection results of SEDiff$_{AFL}$, SEDiff$_{NF}$ and XSym.

Engine	SEDiff$_{AFL}$		SEDiff$_{NF}$		XSym	
	TP	FP	TP	FP	TP	FP
Angr [40]	1	725	4	2	-	-
Navex [9]	17	1,235	27	2	2	3
KUBO [31]	2	230	2	0	-	-
Deadline [47]	1	98	3	2	-	-
Total	21	2,288	36	6	2	3

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 (§5.2.2). 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 [42] reported were false cases. Nevertheless, as shown in Table 3, the ratio of false positives to all bugs reported by SEDiff is not high—only 6 out of 52 cases (11.54%).

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.

5.3.1 Bug Characterization. We present the characterization of the detected bugs.

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 3. We find models of certain types of functions are especially buggy. For example, models of string processing functions (*e.g.,* `strip_tags()`) and array related functions (*e.g.,* `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 [28, 47]. We will closely work with the developers to investigate the bugs.

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 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.
- XSym. We include the only relevant work, XSym, into our evaluation. XSym employs a regression testing suite to test Navex's internal function models.

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 [22] 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$_{AFL}$, SEDiff$_{NF}$ and XSym are sufficient to demonstrate the efficacy of SEDiff. We present the experiment results in Table 4.

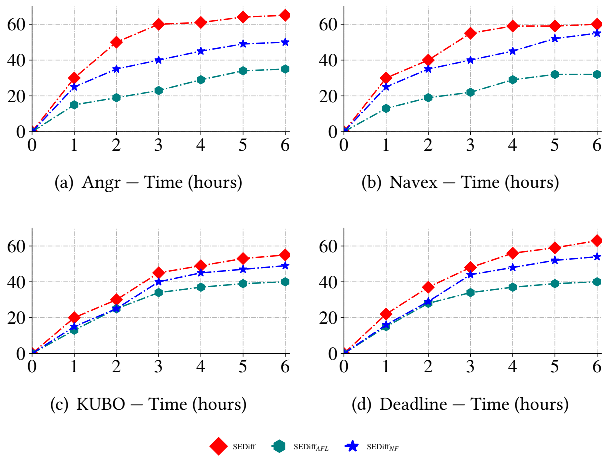
Awareness of scope. Compared to SEDiff, SEDiff$_{AFL}$ is not scope-aware. After evaluating SEDiff$_{AFL}$ with the same resource budget, SEDiff$_{AFL}$ naively reported 2,309 cases as bugs, which include a large number of false positives. To filter out them, we first applied the tailored bug checker 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 outperformed 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.

Comparison with regression testing. Since XSym does not support other engines, we evaluated XSym on only Navex's internal function models. XSym identified 2 true positive bugs out of 5 reports spanning 4 models. Note that XSym used a shorter CPU time to finish the tests. The 2 bugs were also successfully detected by SEDiff. The inherent limitation of XSym in using only the existing regression test suite in a black-box manner made it impractical to find bugs in the models. We further analyzed the 3 false reports of XSym. These 3 cases caused true program deviations, however, they concerned only unmodeled functionalities. Therefore, we conclude that identifying modeled functionalities and the awareness of scope allowed SEDiff to detect more bugs in Navex's models.

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 3. We do not include XSym because its method does not employ fuzzing and it finishes testing much early. It is not meaningful to depict its coverage-time diagram.



(a) Angr — Time (hours)

(b) Navex — Time (hours)

(c) KUBO — Time (hours)

(d) Deadline — Time (hours)

Figure 3: Model coverage (percentage %) over time.

From Figure 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 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.* The comparison with XSym further shows the high efficacy of our scope-aware differential fuzzing approach in exercising the models.

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 [33, 45].

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 [13, 40].

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 §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 complicated 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.

Portability. A key contribution of SEDiff is applying differential testing selectively on part of the functionalities via path mapping. It can be considered a generic mechanism to explore two different semantically equivalent implementations without domain-specific knowledge about their internals. Although this work particularly studies the internal function models, SEDiff is capable to test other function models as well, *e.g.,* user-defined functions, if the data flow in them can be constructed and mapped. Besides, SEDiff has huge potential for other application domains that have multiple implementations of common functionalities that comply with similar requirements or specifications. For example, database systems, data parsers, *etc.*

7 RELATED WORK

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 [23] proposed the first study testing symbolic execution engines through differential testing. They checked the conformance of symbolic execution against concrete execution. XSym [28] used the existing PHP regression testing suite to test Navex's internal function models. 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 [14, 45, 50]; other relevant works revealed bugs in static analyzers [20, 26], model checker [52], debugger [27], *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 [17]. Slutz proposed differential testing for database management systems [41]. Chen *et al.* used the asymmetric behaviors between testing programs to guide the fuzzer towards finding semantic bugs in SSL/TLS implementations [18]. Our work uses the function behaviors on original implementations as the ground truth, and differentially tests internal function models.

Fuzz testing. Many researchers have paid great attention to coverage-guided fuzzing to identify bugs. It has been applied to many aspects such as kernel [6, 24], binary [30], network protocols [53], *etc.* For example, CollAFL [22], TortoiseFuzz [44] and MUZZ [16] proposed new coverage metrics to guide the seed selection to achieve better code coverage. Static analysis is also useful in assisting fuzzing. For example, MUZZ [16] used static analysis to extract suspicious interleaving operations for concurrency bug detection in multi-threaded programs. Steelix [30] and VUzzer [39] analyzed magic values, immediate values and strings that could affect control flow to help input mutation. SEDiff first applies a static analysis to identify modeled functionalities. Based on the results, SEDiff employs a new coverage metric to guide the exploration. Besides, some techniques proposed in other fuzzers [22, 43] that generically improve fuzzing efficiency can complement SEDiff. Though orthogonal, we plan to explore integrating them to SEDiff in the future.

8 CONCLUSION

Symbolic execution is a foundational program analysis technique that typically models internal functions. The correctness of internal function models is critical as it would affect the broad range of applications of symbolic execution. In this paper, we proposed new concepts of modeled functionalities, and showed the importance of identifying them for bug detection. We designed SEDiff, a scope-aware differential testing framework. SEDiff employs new algorithms to automatically identify the modeled functionalities with the recovery of data flows in SMT-LIB formulas. After that, SEDiff takes advantage of grey-box fuzzing with a scope-aware input generator and tailored bug checker to efficiently detect bugs. In a thorough evaluation on several state-of-the-art symbolic execution engines, SEDiff was able to identify the modeled functionalities with high precision and found 46 new bugs. The evaluation results demonstrate that SEDiff is scalable and effective in finding bugs in internal function models in symbolic execution.

ACKNOWLEDGMENTS

The authors would like to thank our shepherd and the anonymous reviewers for their helpful suggestions. The work described in this paper was supported in part by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14210219). Kangjie Lu was supported in part by the NSF awards CNS-1931208 and CNS-2045478.

REFERENCES

[1] 2018. Angr Pull Request #84. https://github.com/angr/angr/pull/889.

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

[3] 2021. Angr Pull Request #2956. https://github.com/angr/angr/pull/2956.

[4] 2021. The GNU C Library (glibc). https://www.gnu.org/software/libc/.

[5] 2021. The PHP Interpreter. https://github.com/php/php-src.

[6] 2021. Syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[7] 2022. Navex. https://github.com/aalhuz/navex.

[8] 2022. Zenodo Archive of SEDiff. https://doi.org/10.5281/zenodo.6665380.

[9] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

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

[11] Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*. Springer.

[12] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: a Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

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

[14] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Seoul, Korea.

[15] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA.

[16] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[17] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. 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.

[18] Yuting Chen and Zhendong Su. 2015. 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.

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. 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.

[20] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*. Berlin, Heidelberg.

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

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

[23] Timotej Kapus and Cristian Cadar. 2017. 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.

[24] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. 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.

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

[26] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA)*. Beijing, China.

[27] Daniel Lehmann and Michael Pradel. 2018. 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.

[28] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2021. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *Proceedings of the Web Conference (WWW)*. Ljubljana, Slovenia.

[29] Wen Li, Ming Jiang, Xiapu Luo, and Haipeng Cai. 2022. POLYCRUISE: A Cross-Language Dynamic Information Flow Analysis. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[30] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Proceedings of the 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*. Paderborn, Germany.

[31] Changming Liu, Yaohui Chen, and Long Lu. 2021. 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.

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

[33] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. 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.

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

[35] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. 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)*.

[36] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. 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.

[37] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. 2017. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.

[38] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[40] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. 2016. 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.

[41] Donald R Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*. New York, USA.

[42] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. 2022. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. Pittsburgh, PA.

[43] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[44] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. 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.

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

[46] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. 2020. 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.

[47] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. 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.

[48] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis

of Exploits in Production Systems. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual event.

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. 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.

[50] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *Proceedings of the 30th International Symposium on Software Testing and Analysis (ISSTA)*. Online.

[51] Michal Zalewski. 2021. American Fuzzy Lop. https://github.com/google/AFL.

[52] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. 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.

[53] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. Virtual event.