# PathDiff: Systematic Differential Testing Using Symbolic Analysis

by

Weiqi Wang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2022 by Weiqi Wang

PathDiff: Systematic Differential Testing Using Symbolic Analysis

Weiqi Wang
Master of Applied Science

Graduate Department of Electrical and Computer Engineering
University of Toronto
2022

# Abstract

Discrepancies among software programs that implement the same specification frequently lead to bugs and vulnerabilities. Differential fuzzers, such as NEZHA, increase the efficiency of finding discrepancies by guiding the fuzzing process with domain-independent behavioural asymmetries. However, the random nature of fuzzing inevitably causes NEZHA to miss discrepancies, even if the pair of paths where the discrepancies lie has already been discovered. In this thesis, we propose PathDiff, a novel differential testing tool that leverages path asymmetry to systematically find all discrepancies for given path-pairs. PathDiff keeps the execution path in one program and iteratively negates each branch in the other program for discrepancies, exhaustively enumerating all discrepancies for given inputs. We have implemented PathDiff and evaluated it against NEZHA on 16 applications, including 12 applications used in the NEZHA paper as well as 4 newly selected ones. The results show that PathDiff finds 4.1× more discrepancies than NEZHA.

# Acknowledgements

I would like to give my heartfelt thanks to my supervisor, Professor David Lie, for giving me this opportunity to pursue graduate studies at University of Toronto. I greatly appreciate his guidance, support, and patience throughout the course of study for my Master's degree, and I am thankful for the constructive feedback he provided for my project and thesis. I would also like to thank Vitaly Chipounov for discussing the internals of the $S^2E$ platform with me, which lays the foundation of my prototype implementation. I am also thankful for the financial support provided by the Department of Electrical and Computer Engineering of the University of Toronto towards my studies. Lastly, I would like to thank my family and friends for their unconditional love, encouragement, and continuous help, especially during the COVID-19 pandemic.

# Contents


**1 Introduction** **1**
- 1.1 Contributions . . . 2
- 1.2 Thesis Structure . . . 3

**2 Background** **4**
- 2.1 Fuzzing . . . 4
- 2.2 Symbolic Execution . . . 5
- 2.3 Differential Testing . . . 7

**3 Related Work** **8**
- 3.1 Monolithic Testing . . . 8
  - 3.1.1 Fuzzing . . . 8
  - 3.1.2 Symbolic and Concolic Execution . . . 9
- 3.2 Differential Testing . . . 9
- 3.3 Semantic Bugs . . . 11

**4 Design** **13**
- 4.1 Overview . . . 13
- 4.2 Phase 1: Seed Selection . . . 15
- 4.3 Phase 2: Constraint Collection . . . 15
- 4.4 Phase 3: Candidate Generation . . . 16
- 4.5 Phase 4: Validation . . . 17
- 4.6 Spurious Discrepancies . . . 17

**5 Imprecision Mitigation** **19**
- 5.1 Symbolic Pointers . . . 19
- 5.2 Input-dependent Loops . . . 20

**6 Implementation** **22**
- 6.1 Constraint Collection . . . 22
- 6.2 Candidate Generation and Validation . . . 22
- 6.3 Optimizations . . . 23
  - 6.3.1 Modifying ctype.h . . . 23
  - 6.3.2 Deduplicating Constraints . . . 24

7 Evaluation 25
7.1 Environment Setup . . . . . 25
7.2 Comparison with Nezha . . . . . 26
7.3 Effectiveness of Imprecision Mitigation . . . . . 27
7.3.1 Discrepancy Detection . . . . . 30
7.3.2 Performance . . . . . 30
7.4 Case Studies of Discrepancies . . . . . 31
7.4.1 LibreSSL vs. GnuTLS . . . . . 31
7.4.2 Libjpeg vs. Libjpeg-turbo . . . . . 32
7.4.3 ImageMagick vs. GraphicsMagick . . . . . 33
7.4.4 BoringSSL vs. LibreSSL . . . . . 34

8 Limitations and Future Work 36
8.1 Constraint Collection . . . . . 36
8.2 Constraint Solving . . . . . 36
8.3 Seed Selection . . . . . 37
8.4 Implementation Limitations . . . . . 37

9 Conclusion 39

Bibliography 40

# List of Tables

6.1 Modification of ctype.h . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

7.1 Discrepancies reported by NEZHA and PathDiff . . . . . . . . . . . . . . . . . . . . . . . 28
7.2 Results of imprecision mitigation evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 29

# List of Figures

4.1 Overview of PathDiff . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.2 Validation phase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

7.1 Venn diagram of discrepancies reported by Nezha and PathDiff . . . . . . . . . . . . 27

# Listings


2.1 Symbolic execution example . . . . . 5

4.1 Design example function A . . . . . 15
4.2 Design example function B . . . . . 15
4.3 Spurious discrepancies example function A . . . . . 17
4.4 Spurious discrepancies example function B . . . . . 17

5.1 Symbolic pointer example . . . . . 19
5.2 Symbolic pointer example function A . . . . . 20
5.3 Symbolic pointer example function B . . . . . 20
5.4 Input-dependent loop example . . . . . 21

7.1 An example of the same error code for different error conditions . . . . . 26
7.2 Time field validity check in LibreSSL . . . . . 31
7.3 Time field validity check in GnuTLS . . . . . 32
7.4 Libjpeg's check on Se field . . . . . 32
7.5 Libjpeg-turbo's errorneous check on Se field . . . . . 33
7.6 GraphicsMagick's check on information header size . . . . . 33
7.7 ImageMagick's check on file size . . . . . 33
7.8 BoringSSL code for validating bitstrings . . . . . 34
7.9 LibreSSL code for validating bitstrings . . . . . 35

# Chapter 1

# Introduction

Implementing software products that conform to specifications is a difficult task. As many specifications (e.g. RFCs) are written in sometimes vague natural language, implementations might deviate from the specifications due to different interpretations among developers. Such differences may lead to unexpected and inconsistent behaviour, and in some cases, semantic bugs. In the security discipline particularly, semantic bugs frequently lead to serious security issues. For example, incorrect parsing logic has caused anti-virus software to skip scanning files [20, 21], and missing and incorrect checks in SSL/TLS libraries have caused a malformed certificate to be wrongly treated as valid [7, 14, 53].

Finding semantic bugs is a non-trivial task as they usually do not exhibit clear, externally visible misbehaviour, such as crashes. An intuitive way to find semantic bugs is to check implementations against specifications [30, 60]. However, the gap between code implementations and textual descriptions usually results in low precision.

Differential testing eliminates the need for specifications by having other programs as references. Given the same input, discrepancies in programs' output, if found, signify potential semantic bugs. Differential testing has been successfully applied in multiple domains such as code coverage tools [73], SSL/TLS libraries [7, 16, 53], anti-virus engines [34, 53], C compilers [72], Web Application Firewalls [3], and Java virtual machine implementations [15]. It is also able to identify regression bugs when applied to different versions of the same program [49].

However, most previously proposed methods for differential testing are undirected (i.e. the tool is blind to which inputs are more likely to trigger discrepancies). Instead, these tools simply optimize for code coverage in the hope that some of the inputs generated in that process find discrepancies. For example, frankencerts randomly generates inputs to find discrepancies [7]. Mucerts prioritizes the inputs that trigger high coverage in one program and apply it to all programs [16]. SymCerts optimizes coverage by using traditional symbolic execution on multiple programs and collects constraints for accept paths and reject paths. It then searches for inputs that can trigger an accept path in one program and a reject path in another program. However, because the initial search uses symbolic execution, which suffers from path explosion, SymCerts requires domain-specific knowledge and fine-tuning to reduce the number of states to make symbolic execution practical. Unfortunately, code coverage is not necessarily correlated with discrepancies. For example, an input is still useful if it triggers different functionality that wasn't previously correlated in two programs, even if that

functionality had been previously exercised before and thus does not increase coverage. As a result, optimizing for coverage wastes computational resources if the objective is to find discrepancies.

Targeting discrepancies requires the ability to reason about the *asymmetries* between two or more programs at the same time. NEZHA is the first attempt to build a differential testing tool that performs a guided search for discrepancies [53]. Rather than rewarding the input generation for finding inputs that generate new paths, NEZHA's search algorithm rewards the input generation for finding new path-pairs, even if the individual paths are not new. In this way, NEZHA is able to guide its fuzzing to focus on paths where discrepancies are likely to occur rather than randomly trying to find new paths. However, the random nature of fuzzing prevents NEZHA from systematically finding all discrepancies in a path-pair, even if the path-pair has already been discovered.

In this thesis, we propose PathDiff, a novel differential testing approach that uses symbolic analysis to systematically find all discrepancies for each given path-pair. For each given seed input, PathDiff executes the programs with the seed input and collects their path constraints (i.e. the constraints on the input that cause the path to be executed). Choosing one of the programs as the target and the other as the non-target, PathDiff negates one of the path constraints from the target and combines the new set of constraints with that of the non-target. A Satisfiability Modulo Theories (SMT) solver is then queried with the conjunction. If the query is satisfiable, the counterexample given by the solver is a concrete input that should cause the non-target to execute the original path while diverging the execution of the target at the branch that corresponds to the negated constraint, causing a discrepancy. By repeating the process for each of the target's constraints, PathDiff can conduct a systematic search for discrepancies for each path-pair defined by a seed input.

PathDiff requires seed inputs, which can be sourced using various methods (e.g. constructed manually, derived from common inputs). In our evaluation, we use NEZHA to generate the seeds, which both demonstrates the ability of PathDiff to integrate with an existing differential fuzzer (i.e. a program that injects randomly generated input into the programs under test to detect bugs), and simultaneously enables a fairer comparison between PathDiff and NEZHA.

The main challenge that PathDiff faces is choosing which constraints to include during constraint negation. Combining constraints from two different paths may lead to false conflicts due to imprecision in the constraints, which can prevent an SMT solver from finding a satisfying solution. In addition, constraints after the negated branch may not be on the newly executed path and should be removed. Thus, PathDiff carefully removes and refines the precision of constraints and maintains the order of constraints so that it can keep only the prefix (i.e. the constraints that are generated by execution before the negated branch) when negating constraints in the target. In this work, we identified 2 major sources of imprecision that lead to overly constrained bytes (i.e. over-constrainedness) and proposed corresponding techniques to mitigate them, which helps PathDiff find more discrepancies.

## 1.1 Contributions

This thesis makes the following contributions:

- **Systematic differential testing:** We proposed a novel approach to finding semantic bugs using symbolic analysis. Our approach searches all branches along the given path and is thus more systematic than previous works.

- **New understanding and techniques:** We identified 2 imprecision issues in applying the idea to real-world programs and presented corresponding mitigation techniques so as to find more discrepancies. Although they are used in a differential context, we believe that they can benefit general symbolic execution as well.
- **Implementation and evaluation:** We implemented PathDiff and evaluated it on 16 programs, including all 12 programs used by NEZHA as well as 4 newly selected ones. The results show that PathDiff found 4.1× more discrepancies than NEZHA.

## 1.2 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the relevant background on software testing techniques. Chapter 3 discusses related work. Chapter 4 describes the methodology for exposing discrepancies. Chapter 5 presents the challenges and solutions of applying the methodology to real-world programs. Chapter 6 describes the implementation of PathDiff. Chapter 7 evaluates PathDiff against NEZHA and presents several case studies of discrepancies. Chapter 8 outlines future directions. Chapter 9 concludes the thesis.

# Chapter 2

# Background

## 2.1 Fuzzing

Fuzzing is a widely used software testing technique. It starts from a set of seeds (i.e. a set of inputs to the program) and randomly mutates them to generate test cases (i.e. new inputs for testing the program), in the hope that one of the test cases causes misbehaviour of the program under test. Generally, the monitored misbehaviour is a *crash* since it is easy to detect and usually indicates problems in the program under test, such as causing a web server that runs the program to be vulnerable to Denial-of-Service attacks. The mutation can take many forms, such as bit flips; crossover; arithmetic operations; inserting, deleting, and reordering bytes.

A popular metric for evaluating the effectiveness of fuzzers is *code coverage* (or *coverage* for short) which measures how much percentage of the code is executed by the fuzzer-generated inputs, indicating the *opportunity* of finding bugs. This metric is proven to be useful by Miller's experimental results [45]—A 1% increase in code coverage increases the percentage of bugs found by 0.92%. However, fuzzing, in its most basic form, is not cost-efficient because the random mutation is blind to which bytes are more likely to trigger a crash. Researchers have proposed various methods to improve efficiency by guiding the fuzzing process with information collected during program execution. Based on the granularity of information used, the guidance can be categorized into three groups—black-box, grey-box, and white-box. Black-box fuzzers only interact with the input and output of the program while white-box fuzzers are the other extreme that systematically explore the program using information gathered from the source code and the execution, which is also called "concolic testing" as will be introduced in Section 2.2. Grey-box fuzzers are in the middle as they use partial information from the program to guide the fuzzing process. For example, the popular grey-box fuzzer AFL [2] uses *statistical* code coverage to guide the mutation of inputs, providing more effective input generation than the black-box method and lower overhead than the white-box method.

Seed selection is another interesting research area for improving the efficiency of fuzzing. As the number of inputs that a program can accept is unbounded, it is important to use a minimum set of seeds that triggers maximum code coverage. For instance, AFL uses logarithmic counters to record the hit count of each branch as the code coverage metric, and only regards a seed as interesting if its coverage differs from other seeds' in the orders of magnitude.

While fuzzers are effective at finding crashes, they cannot detect silent data corruption, such as pointer overwrite due to buffer overflow. Thus, researchers have proposed the use of sanitizers to detect such silent bugs. Sanitizers instrument the program under test to monitor unsafe behaviours such as out-of-bound access of an array and aborts the program so that fuzzers can detect them. To date, a variety of sanitizers have been implemented for diverse types of errors, such as memory corruption [28, 35, 58, 67], undefined behaviours [22, 62], and data races [59].

## 2.2 Symbolic Execution

Symbolic execution is a method of exploring a program by systematically generating an input for each execution path of a program, and is widely used to improve the code coverage during testing.

To symbolically execute a program, a symbolic execution engine is needed to maintain two pieces of information: a symbolic state which maps variables to their corresponding symbolic expressions by keeping track of how data is propagated from the input to each variable; and a conjunction of path constraints derived from the predicate of each branch. For each branch, the symbolic execution engine queries an SMT solver for inputs that can cause the branch to execute the `True` and the `False` branch, respectively. The inputs, if found, can then be injected into the program to fully explore each branch. By recursively performing the operation for each branch, symbolic execution systematically explores the program under test.

```
1  int square(int n) {
2      return n * n; // integer overflow ignored to keep the constraint
       simple
3  }
4
5  int main() {
6      int x = symbolic_input();
7      int y = symbolic_input();
8      z = x + y;
9      if (square(x) == y) {
10         if (z < 10) {
11             return SUCCESS;
12         }
13         else {
14             CRASH_SUM;
15         }
16     }else {
17         CRASH_SQUARE;
18     }
19 }
```

Listing 2.1: Symbolic execution example

For instance, when the program in Listing 2.1 is symbolically executed, $x$ and $y$ are the two symbolic inputs and $z$ is marked as derived from $x$ and $y$ upon the evaluation of Line 8, generating the symbolic state mapping $z = x + y$. At Line 9, the symbolic execution engine queries the SMT solver for inputs that satisfy $(x * x = y)$ and $\neg(x * x = y)$ to explore both sides of the branch.

Similarly, at Line 10, the symbolic execution engine explores both sides of the branch by querying the SMT solver with the constraint $(x * x = y) \wedge (x + y < 10)$ and $(x * x = y) \wedge \neg(x + y < 10)$ (Note that the $x * x = y$ is preserved because we are currently inside the `then` branch of Line 9). Suppose the solver can efficiently solve the satisfiability of these queries, it provides a concrete input that satisfies each query. As a result, all crashes and returns can be triggered by symbolic execution.

Despite providing good coverage, symbolic execution faces three main challenges in practice, hindering its scalability. First, symbolic execution recursively negates branches to generate new paths, which in turn have their branches negated to generate even more paths. The number of paths thus grows exponentially with the size of the program under test, making complete reasoning of large programs infeasible. Worse, the program under test can have unbounded loops (e.g. the program repeats until the user inputs the command "quit") that causes the number of paths to be infinite. This problem is referred to as *path explosion* and has been the key challenge of symbolic execution for decades. Second, constraint solving is expensive. Real-world programs often generate a large number of complex constraints that causes constraint solving to dominate the run time of symbolic execution. Sometimes, the constraints can be so complex (e.g. the non-linear constraints $(x*x = y)$ and $\neg(x*x = y)$ generated at Line 9) that the solver fails to find solutions due to reaching the limit of memory or number of iterations. As no concrete input is generated, symbolic execution cannot explore the corresponding branch, even if both sides of the branch are actually feasible in the original program. Finally, the generated constraints can be imprecise because of the interaction with the external environment such as library calls and complex statements that involves symbolic pointers and floating-point operations.

Concolic execution (a.k.a. white-box fuzzing) is a variant of symbolic execution which keeps track of the concrete state in addition to the symbolic state of the input. As the concolic execution engine has access to a starting concrete input, the workflow is slightly different from that of symbolic execution: Instead of querying the solver upon encountering each branch to explore both sides, concolic execution runs the program while gathering constraints from branches with respect to the concrete input. Once the program terminates, the concolic execution engine picks a constraint, negates it, and queries the solver for an input to explore the alternate execution path of the corresponding branch. This process is repeated for each constraint to systematically explore the program.

For the example in Listing 2.1, concolic execution starts with a concrete input, say $x = 2, y = 4$, runs the program and reaches Line 11 while recording the conjunction of constraints $(x * x = y) \wedge (x + y < 10)$. Then, the engine negates the first constraint (i.e. $\neg(x * x = y)$) and queries the SMT solver for an input that can execute the `else` side of the branch at Line 9. Next, the engine negates the second constraint $(x + y < 10)$ (i.e. queries the solver with $(x * x = y) \wedge \neg(x + y < 10)$) to explore the `else` side of the branch at Line 10, thus finding all paths in the program.

Keeping the concrete state of the input offers two key advantages. First, non-linear constraints can be simplified using the concrete value to enable the exploration of more branches. As aforementioned, SMT solvers cannot easily solve the non-linear constraint $(x * x = y)$ generated at Line 9, causing symbolic execution to fail to explore both sides of the branch and concolic execution to fail to explore the `else` side of the branch. Nevertheless, concolic execution can use the concrete value of $x = 2$ to simplify the negated constraint $\neg(x * x = y)$ to $\neg(4 = y)$. The solver can easily find a satisfying assignment of y (e.g. 3) to generate an input that has $x = 2, y = 3$ to explore the `else` side of the branch at Line 9. Second, the imprecision from interacting with the external environment

can be alleviated because concolic execution can concretely execute the external environment (e.g. libraries) with the concrete input and continue testing the rest of the program.

In this thesis, we leverage the constraint collection and negation from concolic testing to make our method systematic. But, instead of using the information for improving coverage, we analyze the collected path constraints using the newly proposed techniques for finding discrepancies. Thus, we call our method symbolic analysis.

## 2.3 Differential Testing

Differential testing [44] is a software testing method that simultaneously runs two or more similar programs with the same input to find discrepancies in their output. The insight is that if similar programs yield different outputs for the same input, some of them are faulty. In addition, assuming most of the programs are implemented correctly and enough similar programs can be found, a majority voting mechanism can be implemented to statistically pinpoint the faulty program(s). Differential testing has been successful at finding implementation deviations from the specifications (i.e. semantic bugs) since programs can serve as references to each other, thus eliminating the need to check code implementations against textual specifications.

As an example of semantic bugs, the XZ specification allows an archive to use a dictionary size between 4kB to 4GB. While the regular file archivers like XZ Utils implement the check correctly, ClamAV incorrectly adds a custom check that limits the dictionary size to 182MB. As a result, an XZ archive with a dictionary size greater than 182MB causes ClamAV to abort the parsing process and skip scanning the archive, while XZ Utils can successfully decompress the archive. Monolithic fuzzing cannot find this bug by testing ClamAV since there is no crash or memory corruption. However, this discrepancy leads to a serious security issue (assigned CVE-2017-6592 [20]) because it causes ClamAV to skip scanning such malformed archives which may contain malware, and security can be compromised since regular file archiver is able to decompress them.

Nezha [53] is the state-of-the-art domain-independent differential fuzzing tool that leverages path information observed during program execution to guide the differential fuzzing process. It proposes the concept of differential diversity ($\delta$-diversity) to prioritize the inputs that trigger previously unseen asymmetries in the programs under test—Unlike monolithic fuzzing that discards an input if it does not trigger new paths in the program under test, $\delta$-diversity regards the input as equally interesting to previous inputs as long as the path-pair is new. For example, if input $I_1$ triggered path $P_{\alpha 1}$ in program A and path $P_{\beta 1}$ in program B, input $I_2$ triggered path $P_{\alpha 2}$ in program A and path $P_{\beta 2}$ in program B, a monolithic fuzzer would discard input $I_3$ that triggers path $P_{\alpha 1}$ in program A and path $P_{\beta 2}$ in program B since both paths have been exercised before. But Nezha keeps $I_3$ since the path-pair $P_{\alpha 1}$-$P_{\beta 2}$ is new.

Nezha has been evaluated on multiple types of programs including archive utilities, ELF parser, anti-virus engines, PDF viewers, and SSL/TLS libraries. It discovered 27× to 52× more discrepancies than previous differential testing tools, and 6× more discrepancies than the state-of-the-art monolithic fuzzer AFL adapted to differential testing.

# Chapter 3

# Related Work

In this chapter, we discuss the existing works related to this thesis. We begin with standard monolithic testing. We then discuss the undirected and directed differential testing. Finally, we discuss semantic bugs.

## 3.1 Monolithic Testing

### 3.1.1 Fuzzing

In single application testing, a straightforward way is to randomly generate test inputs and feed them into the program under test. However, this method is cost-ineffective when the input is expected to have certain grammar (e.g. XML) or the input space is large. Thus, guidance is introduced to narrow search space or prioritize interesting inputs.

There are two major types of guidance: model-based and coverage-based. Model-based tools constrain the input space with user-specified models or specifications. For example, LangFuzz randomly combines and mutates code fragments from a given set of seeds to generate test cases [29]. Along with the specified grammar, it is able to produce syntactically correct code. Similarly, jsfunfuzz utilizes a grammar model to produce random yet syntactically correct test JavaScript code [36]. Unfortunately, writing the models requires domain-specific knowledge and is laborious. Deciding how to fuzz the input is another practical issue since there are many mutation operators and different types of fields. Picking the right mutation operators and the vulnerable fields for testing requires years of experience to achieve the optimal outcome. The generality and practicability of model-based tools are thus limited.

Coverage-based methods are domain-independent as they mutate existing seeds to search the input space. Different approaches have been proposed to guide the mutation process towards maximizing code coverage. For example, AFL [2] and libFuzzer [39] instrument the program and record coverage information which is then used to refine the inputs. VUzzer [55] and AFLFast [4] leverage control-flow and data-flow analysis to prioritize deep paths. Furthermore, Böhme et al. introduced directed grey-box fuzzing which targets specific locations in the program in order to focus on critical system calls and dangerous locations [5].

### 3.1.2 Symbolic and Concolic Execution

Symbolic/concolic execution is another single application testing method that achieves high code coverage by systematically exploring all branches. Although symbolic execution is powerful at increasing code coverage, its scalability is hindered by various issues such as path explosion, imprecise constraints, and expensive constraint solving. Several methods have been proposed to mitigate these issues. EXE [11] and KLEE [12] employ heuristics to simplify the generated constraints to optimize solver performance. UC-KLEE [54] directly invokes symbolic execution on a bounded space (e.g. functions) rather than the whole program, greatly reducing the test scope. However, ignoring the preconditions brings the false positive problem since the context of the functions is missing. While PathDiff shares the constraint collection and solving with the standard symbolic execution, it does not suffer from path explosion since PathDiff only negates the branches in the seed paths to search for path divergences. The number of branches to negate is thus finite.

Another form of symbolic execution, concolic execution, incorporates the concrete input to guide symbolic execution. DART [26] is the first concolic execution tool that systematically explores the program under test by iteratively negating and solving each branch along the execution path, while checking the code under each branch for typical errors such as assertion violation. CUTE [57] extends DART to handle concurrency. $S^2E$ [17] proposes selective symbolic execution that interleaves symbolic execution with concrete execution to efficiently analyze the program, and virtualization-based in-vivo analysis to take into consideration both the execution of the program and its environment, offering richer analysis by extending the symbolic execution from software stack to the hardware, Empowered by the two techniques, $S^2E$ is able to scale to large real-world systems such as Windows.

Researchers have also attempted to combine fuzzing and symbolic execution to take advantage of both methods. Driller [63] explores most parts of the programs using inexpensive fuzzing to avoid path explosion, and switches to selective concolic execution to pass the complex checks (e.g. magic number checks) that are hard for fuzzing. While finding more bugs in binaries from the DARPA Cyber Grand Challenge Qualifying Event, Driller's effectiveness on real-world programs is limited because it highly depends on the program structure. In many cases, the alternation between fuzzing and symbolic execution occurs so frequently that essentially devolves Driller to symbolic execution followed by path explosion, preventing Driller from progressing forward. In this thesis, we also use a form of hybrid method in the evaluation—a fuzzer produces the seeds and PathDiff consumes them to find discrepancies. However, instead of alternating between fuzzing and symbolic execution, we use fuzzing and symbolic analysis in a sequential fashion, which aims to find the discrepancies missed by the fuzzer. Thus, PathDiff does not devolve to either of the two techniques since there is no alternation.

## 3.2 Differential Testing

In differential testing, multiple similar programs are tested with the same input at the same time and their behaviours/outputs are monitored for discrepancies, which suggest potential bugs or non-conformance in some of the programs.

Differential testing was originally proposed to alleviate the cost of evaluating test results. A representative work is CSmith [72], which performs random differential testing on C compilers to identify faulty compilers. Governed by a grammar model, CSmith generates C programs that are

structurally valid to improve code coverage. In addition, CSmith avoids generating C programs that trigger undefined behaviours to ensure solid detection. However, CSmith is highly dependent on its grammar model—it can only generate C programs containing language features supported by the grammar model. Furthermore, CSmith cannot explore the discrepancies for invalid inputs since the grammar-based generation ensures that all inputs are valid C programs and undefined behaviours are avoided. Later use of differential testing utilizes a reference program to find bugs in the less tested ones. For instance, Jana et al. [34] compares the parsing result of applications and malware detectors to find evasion bugs in the detectors. DifuzzRTL [31] discovers CPU RTL vulnerabilities by comparing the execution result of RTL design against the ISA-level simulation result. TCP-Fuzz [74] compares multiple TCP stack implementations against a well-tested kernel-level TCP stack to find bugs in the relatively new stacks. However, the assumption that the reference program is guaranteed to be correct does not always hold, causing false positives and false negatives.

Recent research on differential testing has shifted to detecting non-conformance with the specifications. JEST [52] synthesizes conformance tests from the ECMAScript specification and performs differential testing on multiple JavaScript engines. The bugs are then attributed to either the specification or the implementations using majority voting.

Frankencerts [7] is the first tool that applies differential testing on SSL/TLS certificate validation. Different from CSmith, frankencerts collects real-world SSL/TLS certificates from the Internet and utilizes the genetic algorithm to generate test cases, which provides a variety of valid and invalid certificates for testing. However, the testing process is highly inefficient due to randomness—208 discrepancies at the cost of 8,127,600 test certificates.

Mucerts [16] optimizes the input generation of frankencerts. It uses one SSL/TLS library (i.e. OpenSSL) as the reference and adopted Markov Chain Monte Carlo (MCMC) sampling to retain representative certificates that have high coverage in OpenSSL. In this way, mucerts filters out the certificates that do not go deep into the programs, improving cost-effectiveness. Unfortunately, the code coverage information that mucerts uses to guide the sampling process only considers one application (i.e. OpenSSL), which does not fully exploit the asymmetries among programs. Therefore, mucerts is also not directed at triggering discrepancies.

SymCerts [14] approaches differential testing from the symbolic execution point of view. It uses traditional symbolic execution to search for accept set and reject set of paths for the given symbolic certificates. The accept and reject sets of paths are then cross-checked for discrepancies. However, due to the path explosion issue in symbolic execution, SymCerts requires domain-specific knowledge to craft partially symbolic certificates and abstract away complex functions.

NEZHA [53] took the first step to guide differential fuzzing with domain-independent behavioural asymmetries observed among programs. Using the programs' path information, NEZHA mutates the seeds in such a way that triggers previously unseen asymmetries, driving the fuzzer to find more discrepancies. However, the mutation by the fuzzer still introduces randomness to the testing process, preventing NEZHA from finding all discrepancies for a path-pair it has already discovered. In contrast, PathDiff directs (rather than guides) the differential testing process using symbolic analysis, targeting discrepancies by keeping the path in the non-target and diverging path in the target. Also, diverging path in the target is done by iteratively negating each constraint, allowing PathDiff to systematically explore all branches to exhaustively find discrepancies for each given seed. Finally, NEZHA's coarse-grained guidance is statistical and cannot help bug localization, while

PathDiff can pinpoint the root cause of discrepancies by correlating the negated constraint back to the source code.

A similar work to ours is Brumley et al.'s research on automatically detecting discrepancies among network protocol implementations of the same specification [8]. They convert execution traces of two programs to symbolic formulas $f1$ and $f2$ and solve $(\neg f1 \wedge f2) \vee (f1 \wedge \neg f2)$ using a constraint solver, which intends to find an input that keeps the execution in one program while diverging the execution in the other program. However, unlike PathDiff's fine-grained constraint-level negation, they negate the entire formula (i.e. the conjunction of all constraints in the path). Since the constraint solver can only present one satisfying input for each query, their approach can only find at most one discrepancy for a path-pair defined by an input, while PathDiff can find all discrepancies by exhaustively trying every constraint in the path-pair. Moreover, negating the entire formula fails to reason about which constraint negation causes the discrepancy, requiring more manual effort to pinpoint the root cause of the discrepancy.

Apart from non-conformance discovery, several works have proposed new applications of differential testing. DiffFuzz [48] focuses on discovering side-channel vulnerabilities via differentially fuzzing two versions of the same program. The goal is to determine, for the same input but different secret values, whether the program exhibits different resource consumption (time or space), and if yes, how serious is the difference. Using a novel resource-based feedback channel for grey-box fuzzing, DifFuzz aims to find inputs that maximize the difference in resource cost.

Shadow Symbolic Execution (SSE) [51] uses differential testing in different software versions to focus testing on the new behaviour introduced by a patch. The two versions are firstly unified into one program. Then, at the branch where the condition is changed by the patch, SSE performs a four-way forking to explore the four results: $\neg new \wedge \neg old$, $\neg new \wedge old$, $new \wedge \neg old$, and $new \wedge old$. The $\neg new \wedge old$ and $new \wedge \neg old$ are two scenarios where the behaviour of the new version differs from the old version. By querying the solver for inputs that satisfy the two conditions, subsequent testing can focus on the divergence between the two versions.

HyDiff [49] integrates the resource consumption feedback from DifFuzz and four-way forking from SSE to systematically explore programs. The two components run in parallel, with the search-based fuzzing inexpensively generating inputs and symbolic execution systematically targeting divergence. Both components perform individual differential analysis and exchange interesting inputs, achieving better results than each component alone.

## 3.3 Semantic Bugs

Semantic bugs prevail in software products that are implemented based on specifications. They are harder to detect than regular errors such as crashes. However, the consequence is comparable to that of the regular errors. In the security discipline, it can cause serious vulnerabilities such as evasion attack [3, 34, 53, 61, 70] and man-in-the-middle attack [19]. In non-security disciplines, it also leads to subtle bugs that are hard to detect such as incorrectly updating files' timestamps or missing permission checks in file systems [46].

Many works have used semantic information to find such bugs. iComment [65] leverages Natural Language Processing (NLP) and static analysis to analyze comments and code to find inconsistencies, which helps identify either bugs in the code or bad comments. tComment [66] extracts comments

from Javadoc APIs as rules and checks the dynamic execution results against them. SymbexNet [65] utilizes symbolic execution to generate an exhaustive set of input packets and compare them against rules filtered from RFCs to find implementation errors. CHIRON [30] builds a finite state machine (FSM) from code implementation and compares that against the FSM described in the specification. SFADiff [3] infers Symbolic Finite Automata (SFA) from programs using SFA learning and checks the equivalence of the inferred models.

As bridging the gap between the natural language and code implementation is still an unsolved problem in NLP, these works usually take advantage of the fact that part of the specification has fixed structures for easy rule extraction. Consequently, these methods depend heavily on the quality of the specifications and frequently suffer from false positives due to inaccurate rule extraction. In contrast, PathDiff leverages differential testing to detect semantic bugs, and thus is not affected by false positives.

# Chapter 4

# Design

To use PathDiff, the user supplies two programs and a set of seed inputs. PathDiff then finds discrepancies in 4 phases as shown in Figure 4.1. PathDiff is agnostic to how the seeds are generated. However, as mentioned in Chapter 1, we use the state-of-the-art differential testing tool NEZHA in our evaluation to generate the seeds, thus provided both tools with a similar starting point, facilitating a fairer comparison between PathDiff and NEZHA. Next, we first give an overview of each of the phases and then explain each phase in detail with code examples.

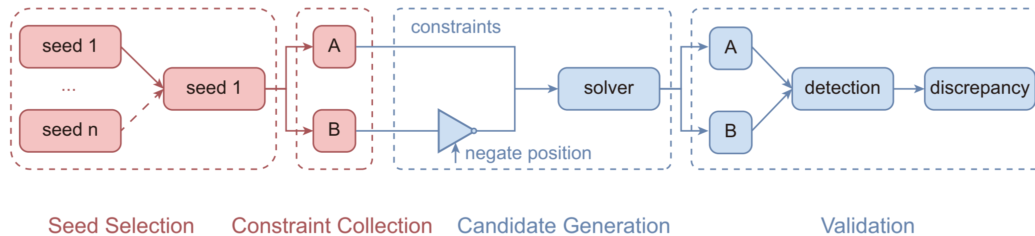


Figure 4.1: Overview of PathDiff. $B$ is chosen as the target in this example. Note that the candidate generation and the validation phase (i.e. the blue part) repeat with different negate positions until all of the target's constraints have been tried.

## 4.1 Overview

PathDiff requires at least one initial seed input to start its search for discrepancies. Because PathDiff exhaustively searches each seed for discrepancies, which can be computationally expensive, it is necessary to rank seeds by the likelihood it will find more discrepancies per seed. Then, the remaining three phases run from the highest rank seed to the lowest rank seed, each time consuming one seed.

For each seed, the two programs are executed with the seed input and their path constraints are collected in the constraint collection phase. This phase outputs a conjunction of symbolic constraints (sorted in execution order) for each program, which are gathered from the evaluation results of the predicates in branch statements. We call the conjunctions from the two programs a path-pair, capturing the fact that each conjunction represents the branch-level execution path of a program

for the given input.

Given a path-pair, PathDiff then executes the candidate generation and validation phases multiple times, generating several candidate discrepancy inputs by iteratively solving conjunctions with different constraints negated, and then validating the inputs by executing them to see if they actually cause discrepancies. In the candidate generation phase, PathDiff first chooses one of the programs as the *target* (we call the other program the *non-target*). PathDiff then negates one of the constraints in the target's conjunction. The negated constraint along with its prefix constraints (i.e. the constraints that are generated before the negated constraint in execution order) are combined with the non-target's conjunction to form a new conjunction, generating an SMT solver query. If the query is satisfiable, the solver presents a concrete input, which we call a *candidate input*. Barring imprecision in the collected constraints, the candidate input should cause the non-target to execute the original path and the target to execute a new path by diverging the target's execution at the negated constraint. As both programs return zero (i.e. no discrepancy) in the original path, this divergence has the potential to make the target return a non-zero value, indicating a discrepancy.

In the validation phase, the candidate input is fed back to the two programs and their actual return codes are examined to see if a discrepancy was found. This phase eliminates false-positive candidates because a divergence in path is not guaranteed to lead to a new return code. By repeating the candidate generation and the validation phase for each constraint of the target, PathDiff can exhaustively find all discrepancies for the given path-pair.

As an example, consider the code in Listings 4.1 and 4.2.[1] For ease of reference, we use $x, y, z$ to refer to the first, second, and third byte in the input, respectively.

The two programs implement the same specification:

- Check the validity of bytes in the input according to the constraints: $(10 < x < 20) \wedge (y \geqslant 1) \wedge (z \neq 8)$.
- Return 0 if all bytes are valid. Otherwise, use `-i` to indicate the index of the invalid byte, where `i` is the index (1-based indexing) of the byte in the input.
- If multiple errors exist in the input, implementations can return any one of the corresponding error codes at their own choice.

According to the constraints of the input, $A$ is implemented correctly and there are two problems with $B$:

**Incorrect check on $x$:** At Line 6 in Listing 4.2, $B$ implements an incorrect check that constrains $x$'s value to the range (10, 19), while the correct range should be (10, 20). To detect this bug, PathDiff negates the constraint of Line 6 in Listing 4.2 to generate an input that has $x = 19$ so that $A$ returns 0 and $B$ returns -1.[2]

**Missing check on $z$:** Corresponding to Line 15 in Listing 4.1, $B$ is missing a check for $z$. As a result, $B$ would treat the invalid $z$ value (i.e. 8) as valid and return 0 (assuming $x$ and $y$ are valid). To detect this bug, PathDiff negates the constraint of Line 15 in Listing 4.1 to generate an input that has $z = 8$ so that $A$ returns -3 and $B$ returns 0.

[1] The functions $A$ and $B$ are abstractions of real-world programs. We use functions here for ease of illustration as the inputs can be shown in the function arguments.

[2] Line 10 in Listing 4.2 is also an incorrect check that can be found in the same way as Line 6. We use Line 6 as an example here to illustrate the workflow of PathDiff and Line 10 to show the need for keeping the prefix.

```c
int A(uint8_t* input){
    uint8_t x = input[0];
    uint8_t y = input[1];
    uint8_t z = input[2];
    if (x <= 10 || x >= 20){
        return -1;
    }


    if (y < 1){
        return -2;
    }


    if (z == 8){
        return -3;
    }
    return 0;
}
```

Listing 4.1: Function A

```c
int B(uint8_t* input){
    uint8_t x = input[0];
    uint8_t y = input[1];
    uint8_t z = input[2];
    if (y >= 1){
        if (x > 10 && x < 19){
            // missing check
            // if (z == 8)
            //     return -3;
            if (x < 18)
                return 0;
            else
                return -1;
        }else{
            return -1;
        }
    }
    return -2;
}
```

Listing 4.2: Function B

In the following sections, we walk through how PathDiff collects constraints and performs negation to find these two bugs.

## 4.2 Phase 1: Seed Selection

In our current design, we use a simple heuristic that ranks the seeds based on the sum of the coverage generated by running the seed on the two programs. This heuristic is both easy to implement and intuitive—seeds that execute more code provide PathDiff with more opportunities to negate path constraints, potentially leading to more discrepancies. We believe there could be other heuristics that could work as well, and we discuss these as future work in Chapter 8. For the remainder of this section, we will assume that the seed selection phase has selected this example: `input[3]={15,5,7}`, which is a seed input that doesn't cause discrepancy.

## 4.3 Phase 2: Constraint Collection

Let $C^X$ denote the conjunction of path constraints of a program $X$. Given two programs $A$ and $B$ that implement similar functionality, PathDiff executes them with the same input and records the path constraints (sorted in execution order) $C^A = \bigwedge_{i=1}^{m} c_i^A$ and $C^B = \bigwedge_{j=1}^{n} c_j^B$ where $c$ is every individual constraint in the conjunction $C$, $m$ and $n$ are the number of path constraints from $A$ and $B$, respectively and $i$ and $j$ are the index of each constraint in $C^A$ and $C^B$, respectively. Note that we record the path constraints rather than, say, predicates in the if statements. Therefore, the constraints reflect the actual value in the seed input.

Given the seed input `input[3]={15,5,7}`, both programs return 0. The constraints collected

for $A$ and $B$ are shown in Equation 4.1 and Equation 4.2.[3]

$$C^A = (10 < x < 20) \wedge (y \geqslant 1) \wedge (z \neq 8) \tag{4.1}$$

$$C^B = (y \geqslant 1) \wedge (10 < x < 19) \wedge (x < 18) \tag{4.2}$$

## 4.4 Phase 3: Candidate Generation

In the candidate generation phase, one of the programs becomes the target, say $B$, and the other $A$, the non-target. PathDiff negates one of the target's constraints and keeps the prefix (i.e. $C^B_{\neg k} = \bigwedge_{j=1}^{k-1} c^B_j \wedge \neg c^B_k$) where $k$ is the negate position. The new conjunction is then merged with $A$'s conjunction $C^A$ (i.e. $C^A \wedge C^B_{\neg k}$) and an SMT solver is queried to find the satisfiability. If a solution is found, it is regarded as a candidate input that can potentially trigger a discrepancy.

For example, when $k = 2$, the second constraint of $B$ is negated and its prefix is kept. The new conjunction is shown in Equation 4.3.

$$C^B_{\neg 2} = (y \geqslant 1) \wedge \neg(10 < x < 19) \tag{4.3}$$

Combining Equation 4.3 with Equation 4.1, PathDiff obtains the new conjunction in Equation 4.4. One possible solution for such a query is `input[3]={19,1,0}` (the discrepancy-triggering byte is marked in red), which causes $A$ to return 0 while $B$ returns -1. We can see the importance of keeping only the prefix here. If all constraints from $B$ were kept, $\neg(10 < x < 19)$ would contradict with $(x < 18)$, causing the query to be unsatisfiable.

$$\begin{aligned} C^A \wedge C^B_{\neg 2} = {} & (10 < x < 20) \wedge (y \geqslant 1) \wedge (z \neq 8) \\ & \wedge (y \geqslant 1) \wedge \neg(10 < x < 19) \end{aligned} \tag{4.4}$$

The above process can also run with $A$ as the target. Negating constraint $z \neq 8$ in $C^A$ would generate a solver query as shown in Equation 4.5. One possible solution may be `input[3]={11,1,8}`, which causes $A$ to return -3 while $B$ returns 0.

$$\begin{aligned} C^A_{\neg 3} \wedge C^B = {} & (10 < x < 20) \wedge (y \geqslant 1) \wedge \neg(z \neq 8) \\ & \wedge (y \geqslant 1) \wedge (10 < x < 19) \wedge (x < 18) \end{aligned} \tag{4.5}$$

As shown in the two examples above, target/non-target designation is arbitrary and is not relevant to the location of bugs. To find the incorrect check on $x$, PathDiff needs to choose $B$ as the target, and to find the missing check on $z$, PathDiff needs to choose $A$ as the target. Thus, in practice, PathDiff will try both programs in a pair as target and non-target. We can also see from the example that our method does not require both programs to *have* the check and only differ in range restriction (so that it can find a non-empty conjunction by negating the more constrained one). Rather, PathDiff can also find missing checks.

[3] For ease of illustration, we use each predicate as a unit for showing constraint. In real-world situations, each comparison (as opposed to each predicate) generates an individual constraint. We also slightly tweak the expression to avoid double negation. Lastly, we ignore short-circuit evaluation here to be consistent with the code so that it is easier to follow.

## 4.5 Phase 4: Validation

Even if the negated constraint diverges the original path, it is possible that the target does not generate a new return code. An example is shown in Figure 4.2, the program only prints a warning if $x$ is not greater than 10 rather than generating a new return code. Suppose the false branch was taken in the original path and PathDiff chose to negate the constraint of this branch, the path would diverge temporarily (to print the warning) but return to the original path later on, leading to the same return code, resulting in a false positive (i.e. no discrepancy). Therefore, as the last step, PathDiff runs the two programs with the candidate input and checks whether they actually generate different return codes.

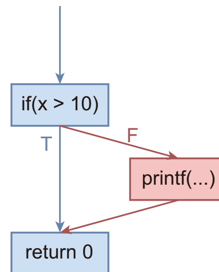


Figure 4.2: Validation phase

## 4.6 Spurious Discrepancies

As aforementioned, negating a constraint does not always lead to a discrepancy. The same reason can also cause spurious discrepancies (i.e. they have the same root cause but appear multiple times because the negated constraint does not actually trigger the discrepancy). Consider the example in Listing 4.3 and Listing 4.4, $A$ does not have any checks while $B$ only generates a different return code if $z = 8$. Thus, negating Line 9 in Listing 4.4 triggers the discrepancy. Suppose the seed input has $z \neq 8$ so both $A$ and $B$ return 0. When negating Line 5 or Line 7 in Listing 4.4, the constraint $z \neq 8$ is removed as it is not in the prefix for either Line 5 or Line 7. If PathDiff then naïvely searched for a solution to the combined constraints, the solver may find a solution with $z = 8$, as $z$ is completely unconstrained, but coincidentally also generates a discrepancy. This could result in PathDiff incorrectly finding multiple *spurious discrepancies* by negating Lines 5 or 7.

```
1  int A(uint8_t* input){
2
3
4
5
6
7
8
9
10
11     return 0;
12 }
```

Listing 4.3: Function A

```
1  int B(uint8_t* input){
2      uint8_t x = input[0];
3      uint8_t y = input[1];
4      uint8_t z = input[2];
5      if (x > 10)
6          printf("Warning");
7      if (y > 7)
8          printf("Warning");
9      if (z == 8))
10         return -3;
11     return 0;
12 }
```

Listing 4.4: Function B

To prevent this, PathDiff keeps track of the bytes that are related to the negated constraint (in the remainder of this thesis, we call such bytes *affected bytes*, the bytes that are not related to the negated constraint are thus *non-affected bytes*). After PathDiff obtains an input from the SMT solver, it only applies the new values from the input to the affected bytes to make sure that the discrepancy is indeed caused by the negated constraint. In the example then, PathDiff would only use new values returned by the SMT solver for $x$ when negating Line 5 and $y$ when negating Line 7, and would leave $z$ at its original value in both cases. Thus, it wouldn't trigger any discrepancies when negating Line 5 or Line 7.

# Chapter 5

# Imprecision Mitigation

In general, a scalable symbolic analysis must be tolerant of some amount of imprecision. However, imprecision when solving for path constraints from two programs simultaneously introduces a new set of problems: when negating a constraint, imprecision in the constraints that the negated constraint is combined with can lead to over-constrained bytes for which the solver cannot find a solution when in fact, a more precise set of constraints would have led to a solution. PathDiff deals with this by retroactively relaxing and refining constraints using 2 heuristics so that the solver can find more solutions when negating constraints, increasing the probability of discovering more discrepancies.

## 5.1 Symbolic Pointers

Symbolic pointer constraints are generated when the program dereferences a pointer whose value is derived from the symbolic input. An example is shown in Listing 5.1 where symbolic input bytes are used as indexes to a map to determine whether the values are valid. Since all dereferenced values of the input bytes are 1, the input array is considered valid.

```
1  // 1 means valid, 0 otherwise
2  unsigned int isValid[5] = {0, 1, 1, 0, 1};
3  // input[3]={2, 1, 4};
4  int checkValid(uint8_t* input, size_t size){
5      for (int i = 0; i < size; i++){
6          if (!isValid[input[i]]){
7              return -1;
8          }
9      }
10     return 0;
11 }
```

Listing 5.1: Symbolic pointer example

Without special handling, these cases can cause complex constraints to be fed to the solver [11, 13, 26, 57], substantially increasing solving time, or even breaking the solver when it tries to construct expressions. In traditional symbolic execution, a simple way to reduce the burden on the solver is to concretize the pointers at the point of dereference, generating simple equality constraints as shown

in Expression 5.1.

$$(input[0] = 2) \wedge (input[1] = 1) \wedge (input[2] = 4) \tag{5.1}$$

This helps make constraints solving practical but sacrifices completeness [13]—any byte in the input can be 1, 2, or 4 but the concretization makes it look like only `input[3]={2,1,4}` (i.e. the same value as the concrete input) is allowed.

In differential testing, we need to remove such over-constrainedness. Otherwise, it would prevent the solver from finding a discrepancy-triggering input by negating a constraint. Consider the example in Listing 5.2 and Listing 5.3. Both of them implement the functionality that checks whether $p$ is in the range [0, 2]. The implementation difference is that $A$ uses an array while $B$ uses an if statement. In addition, $B$ incorrectly set the range to [0, 2). Thus, $p = 2$ would trigger the discrepancy.[1]

Given an input where $p = 1$, PathDiff gets constraints $p = 1$ for $A$ and $p < 2$ for $B$. The constraint from $A$ is a concretization because $p$ is used as an index to the `isValid` array. To find the discrepancy, PathDiff would try quering solver with $(p = 1) \wedge \neg(p < 2)$ but find that it is unsatisfiable. Thus, PathDiff removes the symbolic pointer constraint $p = 1$ when negating $p < 2$ so that solver can choose a new value for $p$.

Another problem is that after removing the symbolic pointer constraints, the only constraint left is $\neg(p < 2)$, for which the solver could pick a value (e.g. 3) that does not trigger the discrepancy. In this case, PathDiff adds an inequality constraint based on the concrete value in the input (i.e. $p \neq 3$) and queries the solver again to get a new value for $p$. The process is repeated several times (the limit can be configured by users). Finally, the discrepancy would be triggered when the solver gives 2 for $p$.

```
1 unsigned int
2 isValid[6] = {1,1,1,0,0,0};
3
4 int A(uint8_t p){
5     if(!isValid[p])
6         return -1;
7     return 0;
8 }
```

Listing 5.2: Function A

```
1 int B(uint8_t p){
2
3     if (p<2){
4         return 0;
5     }else{
6         return -1;
7     }
8 }
```

Listing 5.3: Function B

## 5.2 Input-dependent Loops

As shown in Listing 4.1 and 4.2, if statements are a common source of constraints and negating them triggers new paths. On the contrary, loop statements also generate constraints but negating them sometimes does not affect the path. Consider a simplified example adapted from GnuTLS time parsing code in Listing 5.4. Here, `year` is derived from the symbolic input (i.e. the input certificate). Suppose the `year` in the input is 2021, as the loop executes, PathDiff would get constraints as shown in Equation 5.2. From the constraints' perspective, it would look like the program only reaches the remaining code in the path if the `year` is equal to 2021, while actually, any value greater than 1970 will work.

[1] Assume $p$ has been restricted to [0, 5] before being passed to the functions so there is no out-of-bound array access in $A$.

```
// year is symbolic with concrete value 2021
for (i = 1970; i < year; i++)
    result += 365 + ISLEAP(i);
```

Listing 5.4: Input-dependent loop example

$$
\begin{aligned}
C = (1970 < \mathit{year}) \wedge (1971 < \mathit{year}) \wedge \cdots \\
\wedge (2020 < \mathit{year}) \wedge (2021 \not< \mathit{year})
\end{aligned}
\tag{5.2}
$$

To mitigate such over-constrainedness, we observe that such constraints follow a specific pattern:

- Each constraint contains three elements: a compare operator, a constant operand, and an expression that is constructed to represent the value read from the symbolic input.
- The constants of these constraints are at the same operand position and form a monotonic sequence across constraints.
- The affected bytes of these constraints are the same.

Thus, PathDiff removes the subsequence of constraints that satisfy the above pattern to remove such over-constrainedness.

# Chapter 6

# Implementation

## 6.1 Constraint Collection

We implemented a plugin of the S$^2$E selective symbolic execution engine [56] (git commit 0be5c89) to collect constraints. For regular constraints, S$^2$E emits an `onStateForkDecide` signal before forking states to explore a branch. We extended the signal to also contain the constraint information of the branch and implemented a callback of the signal to log the constraints to a file. For symbolic pointer constraints, we took advantage of the fact that S$^2$E emits an `onSymbolicAddress` signal before emitting the `onStateForkDecide` signal when handling symbolic memory access. Our plugin uses the `onSymbolicAddress` signal as an indicator for the subsequent invocation of `onStateForkDecide` callback to distinguish symbolic pointer constraints from regular constraints, allowing the callback to determine whether to record the symbolic pointer constraints. After the execution finishes, the plugin generates a file containing the constraints in the KQuery format [38]. Additionally, the plugin collects the metadata as follows for the candidate generation phase:

- Affected bytes of each constraint. It is used to determine which bytes to use the new values from the concrete input provided by the SMT solver. Since the symbolic variable name for each byte is assigned by S$^2$E and contains the offset. We used regular expression to extract the indexes of the affected bytes.
- The module where each constraint originates from. Some typical types include: (1) the binary that is executed, (2) shared libraries that the binary calls, and (3) the Linux kernel. This information can be used by the candidate generation phase to choose which constraints to include.
- Native address of the instruction where each constraint originates from. If source code is available, this information can be used with debug information to map the address of the negated constraint to line number (e.g. addr2line [1]) to help debug the discrepancy.

## 6.2 Candidate Generation and Validation

We implemented a Python package for the candidate generation and the validation phase. It takes two sets of constraints and metadata as input. Then, it negates and merges the constraints and

invokes Kleaver [37], a wrapper for multiple backend SMT solvers, to get a candidate input. We chose STP v2.3.3 [64] as the backend SMT solver of Kleaver as recommended by KLEE's documentation [9]. Finally, the two programs are executed with the candidate input inside docker containers to preserve the environment where their constraints were collected from.

## 6.3 Optimizations

### 6.3.1 Modifying ctype.h

In standard symbolic execution, function models are used to reduce path explosion by replacing typical libc functions (e.g. `memcmp()`) with a symbolic expression that represents the function's return value. Similarly, we observed that general macros and functions such as `isdigit()` in ctype.h use locale array implementation and are frequently used by the programs, causing many symbolic pointer constraints that are not precise representations of the branches. If we left them as is, the generated symbolic pointer constraints would be removed and suffer under-constrainedness as discussed in Section 5.1. As these macros and functions are frequently used and are general to all programs, we modified ctype.h to use arithmetic rather than locale array implementation to get precise constraints. The modification involves 16 lines in total and is straightforward. The new implementation of each modified macro/function shown in Table 6.1.

Table 6.1: Modification of ctype.h

| **Name** | **Type** | **Implementation** |
|---|---|---|
| `isalnum(c)` | Macro | `(('0' <= c && c <= '9') \|\| ('a' <= c && c <= 'z') \|\| ('A' <= c && c <= 'Z'))` |
| `isalpha(c)` | Macro | `(('a' <= c && c <= 'z') \|\| ('A' <= c && c <= 'Z'))` |
| `iscntrl(c)` | Macro | `((0 <= c && c <= 31) \|\| c == 127)` |
| `isdigit(c)` | Macro | `('0' <= c && c <= '9')` |
| `islower(c)` | Macro | `('a' <= c && c <= 'z')` |
| `isgraph(c)` | Macro | `(33 <= c && c <= 126)` |
| `isprint(c)` | Macro | `(32 <= c && c <= 126)` |
| `ispunct(c)` | Macro | `((33 <= c && c <= 47) \|\| (58 <= c && c <= 64) \|\| (91 <= c && c <= 96) \|\| (123 <= c && c <= 126))` |
| `isspace(c)` | Macro | `((9 <= c && c <= 13) \|\| c == 32)` |
| `isupper(c)` | Macro | `('A' <= c && c <= 'Z')` |
| `isxdigit(c)` | Macro | `(('0' <= c && c <= '9') \|\| ('a' <= c && c <= 'f') \|\| ('A' <= c && c <= 'F'))` |
| `isblank(c)` | Macro | `(c == 9 \|\| c == 32)` |
| `tolower(c)` | Macro | `(isupper(c) ? c+32 : c)` |
| `toupper(c)` | Macro | `(islower(c) ? c-32 : c)` |
| `tolower(__c)` | Function | `return isupper(__c) ? __c+32 : __c;` |
| `toupper(__c)` | Function | `return islower(__c) ? __c-32 : __c;` |

### 6.3.2 Deduplicating Constraints

As the plugin just naïvely records all constraints, it is possible that the constraints of each program contain duplicates because the same check happens on the same bytes multiple times. The duplicates would prevent the solver from finding a solution when negating one of them because the duplicates contradict the negated constraint. Thus, PathDiff deduplicates constraints for each program before the candidate generation phase.

# Chapter 7

# Evaluation

In this chapter, we evaluate PathDiff along three aspects: (1) comparison with Nezha, (2) effectiveness of imprecision mitigation, and (3) case studies of discrepancies.

We analyzed the discrepancies found in the evaluation. We have either verified that they have been fixed in the latest versions (since the programs used in our evaluation contain old versions as used in the Nezha paper) or reported them to the developers. All reported discrepancies have been acknowledged by the developers as cases where the program silently accepts malformed inputs. We note, however, that not all developers consider such cases as bugs as they may not lead to memory corruption or crashes. We document some of these cases in Chapter 7.4.

## 7.1 Environment Setup

Our evaluation corpus consists of all 12 applications used in the Nezha paper as well as 4 newly selected ones. Specifically, we use Binutils (v2.26-1-1_all) [24], XZ Utils (v5.2.2) [71], ClamAV (v0.99.2) [18], OpenSSL (v1.0.2h) [50], LibreSSL (v2.4.0) [42], BoringSSL (git commit f0451ca) [6], wolfSSL (v3.9.6) [68], mbedTLS (v2.2.1) [43], GnuTLS (v3.5.0) [25], Evince (v3.22.1) [23], MuPDF (v1.9a) [47], and Xpdf (v3.04) [69] from the Nezha paper. We also added 4 image processing libraries: libjpeg (v9d) [40] and libjpeg-turbo (v2.1.1) [41] for parsing JPEG images and ImageMagick (v7.1.0-12) [32] and GraphicsMagick (v1.3.36) [27] for parsing BMP images. Nezha only collects path information from the programs, as opposed to the entire execution environment (i.e. libraries, the OS kernel). Thus, for a fair comparison, we also configure PathDiff to only use constraints from the program code. For the seeds of Nezha, we followed common practice for choosing the seeds for fuzzing (i.e. randomly select from the repositories' test suite or the Internet, and create with common tools such as OpenSSL). Our evaluation is performed on Ubuntu 18.04 machine with 32 CPUs and 64GB memory. Both PathDiff and Nezha run with 32 workers. For PathDiff, we use an SMT solver timeout of 30 minutes. The limit of number of solver trials for a discrepancy-triggering input (Chapter 5.1) is set to 2.

Unfortunately, $S^2E$ crashed when collecting constraints from MuPDF. As the problem is inside the $S^2E$ engine, we exclude MuPDF from the evaluation and leave investigating the cause of the crash to future work.

## 7.2 Comparison with Nezha

In this section, we compare the number of discrepancies reported by PathDiff and the state-of-the-art differential testing tool NEZHA with a 24-hour time budget for each tool. Specifically, we ran NEZHA for 24 hours. Then, we transferred the seeds that were generated in the first 30 minutes of the NEZHA run to PathDiff (we made sure that these seeds do not trigger discrepancies). Finally, PathDiff was given the remainder of the time budget (i.e. 23 hours and 30 minutes) to find discrepancies. We limited PathDiff to only run the top 3 ranked seeds from the seed selection phase because our current implementation of PathDiff cannot automatically run the next seed after it finishes running the current one. We limited the number of seeds to 3 so that PathDiff does not exceed the 24-hour time budget, making the results an underestimate of the number of discrepancies that could be found by PathDiff. We leave the implementation work required to automatically load additional seeds to future work. In summary, this experiment demonstrates: (1) PathDiff's ability to integrate with a standard off-the-shelf fuzzer (2) for the same budget of CPU resources, how many discrepancies PathDiff finds compared to NEZHA.

To compare which discrepancies are uniquely found by PathDiff and NEZHA, respectively, and which can be found by both tools, a bucketing criterion is needed for grouping the discrepancies. Ideally, the criterion for bucketing should be the root cause of each discrepancy. However, this is analogous to automated debugging in a differential context which is non-trivial [53]. Existing works have made several attempts to solve this problem. For instance, frankencerts and mucerts use the programs' return codes to bucket the discrepancies [7, 16]. This approach is very sensitive to the return code space of the programs: if the two programs only used 0 for success and 1 for failure, all discrepancies would be bucketed together, even though they actually have different root causes. NEZHA mitigates this issue by manually exposing the return codes at different function levels to generate a combined return code for bucketing. However, NEZHA's bucketing is imprecise because it lacks path information. Specifically, we observed that real-world programs often use the same return code for different error conditions inside a function as shown in Listing 7.1. In such cases, NEZHA's bucketing cannot differentiate the different error conditions at Line 3 and Line 6 since the error codes are the same.

```
1 WOLFSSL_LOCAL int GetLength(...){
2     ...
3     if ( (i+1) > maxIdx)
4         return BUFFER_E;
5     ...
6     if ( (i+length) > maxIdx)
7         return BUFFER_E;
```

Listing 7.1: An example of the same error code for different error conditions

To bucket the discrepancies more accurately, we refined NEZHA's bucketing to also track the path to differentiate the two error conditions. However, including the full path information in bucketing can cause the discrepancies that are due to the same error to be considered as different ones. For example, a longer input may cause extra loop iterations but does not result in a different error. In this case, we need to ignore the branches that are not related to the error. Thus, we use the following

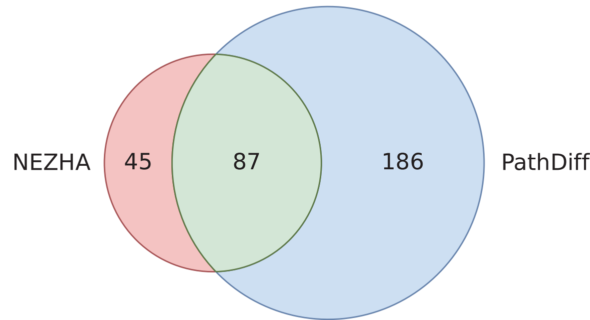


Figure 7.1: Venn diagram of discrepancies reported by Nezha and PathDiff

heuristic to only capture the path variation that makes the errors different. We report the cases we observed and used for bucketing below:

1. A return keyword followed by a macro containing strings like "error" or "err", etc. or a number which refers to error as indicated in the code comment.

2. A goto keyword followed by a label containing strings like "error" or "err", etc. The statement located at the error label is usually an error return which satisfies the first case. We further locate the original goto statement in this case for more fine-grained line information.

3. A function call to the programs' custom error functions that logs errors or throws exceptions, where the function name contains strings like "error", "err", and "Throwexception", etc.

The results of this experiment are shown in Figure 7.1 and a detailed breakdown is tabulated in Table 7.1. Apart from the 87 discrepancies that were found by both tools, PathDiff reported 186 discrepancies not found by Nezha while Nezha reported 45 discrepancies not found by PathDiff. In summary, PathDiff reported 4.1× more discrepancies than Nezha, showing that PathDiff's systematic approach to finding discrepancies can find significantly more discrepancies than random fuzzing. We examined the 45 discrepancies that Nezha found but were missed by PathDiff and found that the main reason was that the seeds required to find these discrepancies had not been in the top 3 used by PathDiff. This does show that similar to fuzzers, the set of discrepancies found by PathDiff does depend on the initial seeds provided to it.

## 7.3 Effectiveness of Imprecision Mitigation

To demonstrate the effectiveness of our imprecision mitigation techniques, we evaluate one pair of programs for each program type: Binutils vs. ClamAV for ELF parsing, XZ Utils vs. ClamAV for XZ archive parsing, libjpeg vs. libjpeg-turbo for image parsing, Evince vs. Xpdf for PDF parsing, LibreSSL vs. GnuTLS for SSL/TLS certificate validation. The time budget is set to 24 hours. Each program pair is evaluated under 5 setups as follows:

- Naïve: The results achieved by naïvely collecting, negating constraints and solving them (i.e. the naïve version of our idea – negate constraint, keep prefix and solve).

- Base: The results achieved after applying (1) avoiding counting spurious discrepancies, (2) optimizing ctypes.h and (3) deduplicating constraints. This serves as a baseline for the three

Table 7.1: Discrepancies reported by NEZHA and PathDiff. The two rows for each program pair refer to the first and second program returns non-zero, respectively (i.e. setting the first and second program as target in PathDiff, respectively). N=NEZHA, I=Intersection, P=PathDiff.

| **Program pair** | **N** | **I** | **P** | **Program pair** | **N** | **I** | **P** |
|---|---|---|---|---|---|---|---|
| Binutils | 0 | 1 | 0 | LibreSSL | 0 | 2 | 4 |
| ClamAV | 7 | 1 | 0 | BoringSSL | 0 | 1 | 0 |
| XZ utils | 2 | 4 | 6 | LibreSSL | 0 | 6 | 12 |
| ClamAV | 0 | 0 | 1 | wolfSSL | 0 | 2 | 3 |
| libjpeg | 0 | 0 | 1 | LibreSSL | 1 | 5 | 11 |
| libjpeg-turbo | 0 | 0 | 0 | mbedTLS | 7 | 5 | 4 |
| ImageMagick | 0 | 1 | 1 | LibreSSL | 1 | 5 | 9 |
| GraphicsMagick | 0 | 1 | 1 | GnuTLS | 2 | 1 | 3 |
| Evince | 0 | 0 | 0 | BoringSSL | 0 | 3 | 13 |
| Xpdf | 0 | 5 | 0 | wolfSSL | 0 | 1 | 6 |
| OpenSSL | 0 | 1 | 0 | BoringSSL | 1 | 1 | 10 |
| LibreSSL | 0 | 3 | 3 | mbedTLS | 2 | 7 | 9 |
| OpenSSL | 0 | 0 | 0 | BoringSSL | 0 | 2 | 7 |
| BoringSSL | 0 | 0 | 0 | GnuTLS | 0 | 3 | 3 |
| OpenSSL | 0 | 4 | 12 | wolfSSL | 1 | 0 | 2 |
| wolfSSL | 2 | 1 | 4 | mbedTLS | 3 | 5 | 12 |
| OpenSSL | 1 | 1 | 11 | wolfSSL | 0 | 2 | 6 |
| mbedTLS | 7 | 1 | 8 | GnuTLS | 1 | 1 | 3 |
| OpenSSL | 1 | 2 | 9 | mbedTLS | 4 | 4 | 6 |
| GnuTLS | 0 | 4 | 2 | GnuTLS | 2 | 1 | 4 |

setups below.

- Loop: Based on the Base setup, add imprecision mitigation for input-dependent loop.
- Symptr: Based on the Base setup, add imprecision mitigation for symbolic pointer constraints.
- Both: Based on the Base setup, apply both imprecision mitigation techniques i.e. Loop+Symptr.

The discrepancy detection results are shown in Table 7.2, focusing on three metrics:

1. The percentage of satisfiable queries (S) is defined as:

$$S = \frac{\text{Number of satisfiable queries}}{\text{Total number of queries}} \times 100 \tag{7.1}$$

   It measures the improvement in the *quality* of the constraints generated by the imprecision mitigation techniques, which seek to relax over-constrainedness and enable more queries to be satisfiable. This metric can be seen as analogous to the code coverage in fuzzing.

2. Number of discrepancies measures the number of discrepancies reported by PathDiff, which is the end result users would expect from a differential testing tool.

Table 7.2: Results of imprecision mitigation evaluation

| Program pair | Metric | Naïve | Base | Loop | Symptr | Both |
|---|---|---|---|---|---|---|
| ¬Binutils ClamAV | Satisfiable queries (%) | 12.95 | 33.61 | 33.41 | 55.87 | 55.59 |
| | Discrepancies | 0 | 0 | 0 | 0 | 0 |
| | Run time (s) | 1192 | 984 | 984 | 1408 | 1404 |
| Binutils ¬ClamAV | Satisfiable queries (%) | 0.23 | 0.70 | 0.82 | 0.84 | 1.01 |
| | Discrepancies | 4 | 2 | 2 | 1 | 1 |
| | Run time (s) | 640 | 460 | 454 | 431 | 426 |
| ¬XZ Utils ClamAV | Satisfiable queries (%) | 2.21 | 3.78 | 4.08 | 6.58 | 7.12 |
| | Discrepancies | 13 | 13 | 13 | 21 | 21 |
| | Run time (s) | 195 | 198 | 192 | 164 | 163 |
| XZ Utils ¬ClamAV | Satisfiable queries (%) | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| | Discrepancies | 0 | 0 | 0 | 0 | 0 |
| | Run time (s) | 175 | 185 | 188 | 152 | 152 |
| ¬libjpeg libjpeg-turbo | Satisfiable queries (%) | 0.02 | 0.04 | 0.04 | 0.07 | 0.07 |
| | Discrepancies | 1 | 1 | 1 | 1 | 1 |
| | Run time (s) | 13129 | 10182 | 10334 | 4680 | 4532 |
| libjpeg ¬libjpeg-turbo | Satisfiable queries (%) | 0.02 | 0.03 | 0.03 | 2.57 | 2.59 |
| | Discrepancies | 0 | 0 | 0 | 0 | 0 |
| | Run time (s) | 17994 | 10312 | 10746 | 6326 | 3739 |
| ¬Evince Xpdf | Satisfiable queries (%) | 2.01 | 2.80 | 2.82 | 15.56 | 15.68 |
| | Discrepancies | 0 | 0 | 0 | 0 | 0 |
| | Run time (s) | 992 | 1290 | 1138 | 1137 | 1143 |
| Evince ¬Xpdf | Satisfiable queries (%) | 7.79 | 10.40 | 10.40 | 33.08 | 33.13 |
| | Discrepancies | 23 | 17 | 17 | 43 | 43 |
| | Run time (s) | 982 | 1390 | 1026 | 1341 | 1339 |
| ¬LibreSSL GnuTLS | Satisfiable queries (%) | 0.91 | 2.88 | 2.99 | 7.11 | 7.81 |
| | Discrepancies | 159 | 38 | 38 | 39 | 41 |
| | Run time (s) | 47920 | 15796 | 14812 | 8046 | 10305 |
| LibreSSL ¬GnuTLS | Satisfiable queries (%) | 0.09 | 0.36 | 0.33 | 2.16 | 1.91 |
| | Discrepancies | 35 | 15 | 15 | 15 | 14 |
| | Run time (s) | 86400 | 23849 | 30150 | 21929 | 35487 |

3. Run time is the time taken for PathDiff to run.

We now analyze the results regarding the discrepancy detection and performance.

### 7.3.1 Discrepancy Detection

We observe that, as more techniques are applied (i.e. from the Base setup to the Both setup), the percentage of satisfiable queries, as well as the number of discrepancies, generally increase. On average, the percentage of satisfiable queries increases by 7.03% and the number of discrepancies found increases by 3.5, showing that our imprecision mitigation techniques successfully remove over-constrainedness and help PathDiff find more discrepancies.

The only exception is the Binutils vs. ¬ClamAV case, where the number of discrepancies decreases while the percentage of satisfiable queries increases. The reason is that the techniques incorrectly removed constraints that are not over-constrained ones, causing a loss of execution consistency (i.e. previously constrained bytes are now incorrectly under-constrained). The solver happens to choose a value that causes the target to take a path that still returns zero, preventing discrepancy-triggering negations from finding the discrepancies when more techniques are applied. The decrease of one discrepancy from Symptr to Both in the LibreSSL vs.¬GnuTLS case is due to an SMT solver timeout, which is set to 30 minutes. We have verified that, for that negate position, the solver does not timeout (takes ∼12 minutes) and PathDiff can find the discrepancy when the worker for the negate position is the only one that runs on the machine (as opposed to 32 workers run simultaneously). Thus, the decrease in this case is due to the contention of computing resources, not incorrect constraints removal by PathDiff.

Comparing the number of discrepancies change from the Base setup to the Loop setup and from the Symptr setup to the Both setup, one might ask why, in the ¬LibreSSL vs. GnuTLS case, there is no change from Base to Loop but an increase from Symptr to Both since the only difference is the addition of the loop technique. The reason is that the same byte may be over-constrained by input-dependent loop and symbolic pointer constraints simultaneously. Therefore, removing one of the over-constrainedness still results in an unsatisfiable query. The discrepancy can only be found when both techniques are applied.

Comparing the number of discrepancies change between the Naïve setup and the Base setup, we can see that the number of discrepancies sometimes decreases. For example, in the LibreSSL vs. ¬GnuTLS case, the number of discrepancies decreases from 35 to 15. This is because Naïve does not track affected bytes and thus generates spurious discrepancies as described in Chapter 4.6. These are real discrepancies, but their root cause is attributable to another discrepancy, resulting in the same discrepancy being counted multiple times.

### 7.3.2 Performance

We can see that the run time of PathDiff generally decreases as the number of applied techniques increases. This is because the number of constraints that PathDiff needs to negate decreases. In the ¬libjpeg vs. libjpeg-turbo case, particularly, the run time of PathDiff decreases from 3.65 hours to 1.26 hours (∼2.9× gain) and the number of discrepancies is not impacted. This shows that although applying the imprecision mitigation techniques sometimes does not find more discrepancies, it can improve performance while maintaining accuracy.

Sometimes we observed that the performance is worse in the Symptr and Both setups. This is because when the symbolic pointer technique is enabled, PathDiff may repeatedly invoke the solver as it searches for a value that triggers a discrepancy (see Section 5.1), resulting in a greater number of solver invocations. Also, the overhead of the validation phase contributes to the increase in run time. As more techniques are applied, more queries become satisfiable, increasing the number of invocations of the validation phase. Since we implemented the validation phase as invocations of docker containers (for keeping the environment the same as $S^2E$'s constraint collection environment), the overhead of starting a docker container becomes significant when the number of satisfiable queries increases dramatically (e.g. the number doubled in the ¬Binutils vs. ClamAV case). This could be solved by keeping the container running as a service, which we leave to future work.

## 7.4 Case Studies of Discrepancies

In this section, we detail our analysis of some discrepancies found by PathDiff. 3 of these discrepancies are new discrepancies found by PathDiff while 1 of these is a discrepancy that was also found by NEZHA but for which PathDiff was able to better identify the root cause.

### 7.4.1 LibreSSL vs. GnuTLS

As shown in Listing 7.2, LibreSSL explicitly checks the validity of all fields inside a time field. However, GnuTLS only checks the month and year fields as shown in Listing 7.3. Consequently, a certificate with an invalid time field (i.e. out-of-range day, hour, minute, or second) is incorrectly accepted by GnuTLS. We have verified that the bugs have been fixed in the latest version of GnuTLS.

```
1  int asn1_time_parse(...)
2      ...
3      if (lt->tm_mon < 0 || lt->tm_mon > 11)
4          return (-1);
5      ...
6      if (lt->tm_mday < 1 || lt->tm_mday > 31)
7          return (-1);
8      ...
9      if (lt->tm_hour < 0 || lt->tm_hour > 23)
10         return (-1);
11     ...
12     if (lt->tm_min < 0 || lt->tm_min > 59)
13         return (-1);
14     ...
15     if (lt->tm_sec < 0 || lt->tm_sec > 59)
16         return (-1);
17     ...
```

Listing 7.2: Time field validity check in LibreSSL

```
1 static time_t mktime_utc(...){
2     ...
3     if (tm->tm_mon < 0 || tm->tm_mon > 11 || tm->tm_year < 1970)
4         return (time_t) - 1;
5     ...
```

Listing 7.3: Time field validity check in GnuTLS

### 7.4.2 Libjpeg vs. Libjpeg-turbo

PathDiff uncovered a discrepancy where libjpeg-turbo successfully parses the JPEG while libjpeg complains about the input JPEG format, which is due to a bug that has existed in libjpeg-turbo for 11 years.

As required by the JPEG standard [33], the `Se` (End of spectral selection) field can only be set to 63. However, libjpeg implements a non-standard feature called SmartScale, which allows variable DCT block size by making the `Se` field variable as shown in Listing 7.4. As libjpeg-turbo developers chose not to support this non-standard feature, they removed the SmartScale-related checks when forking from libjpeg, not realizing that one of the checks is guarding the standard-conforming value 63 (Line 7 in Listing 7.4). As a result, a JPEG file that has an invalid `Se` field is accepted by libjpeg-turbo as shown in Listing 7.5. This subtle discrepancy has stayed hidden for 11 years. Yet, by presenting the corresponding check in libjpeg (automatically obtained by PathDiff based on the negated branch) to libjpeg-turbo developers, they quickly identified that the cause was related to SmartScale and fixed the bug within 3 days. We believe such mistakes happen frequently when forking and refactoring codebases. Therefore, this example demonstrates that PathDiff can help identify such subtle discrepancies and improve protection against invalid values.

```
1  LOCAL(void) initial_setup (...){
2      ...
3      switch (cinfo->Se) {
4          case (1*1-1):
5              ...
6          ...
7          case (8*8-1):
8              ...
9          ...
10         case (16*16-1):
11             ...
12         default:
13             ERREXIT4(...);
14             break;
15     ...
```

Listing 7.4: Libjpeg's check on Se field

```
METHODDEF(void) start_pass (...){
    ...
    if(cinfo->Ss != 0 || cinfo->Ah != 0 || cinfo->Al != 0
        ||(cinfo->Se < DCTSIZE2 && cinfo->Se != DCTSIZE2-1))
        WARNMS(cinfo, JWRN_NOT_SEQUENTIAL);
    ...
```

Listing 7.5: Libjpeg-turbo's errorneous check on Se field

### 7.4.3 ImageMagick vs. GraphicsMagick

PathDiff discovered that GraphicsMagic only accepts certain values as valid information header sizes as shown in Listing 7.6 while ImageMagick does not. We presented the corresponding checks in GraphicsMagick to ImageMagick developers and they have fixed the bug. PathDiff also found a case where ImageMagick complains about the mismatch between the claimed size in the header and the actual amount of data read, as shown in Listing 7.7, while GraphicsMagick does not. GraphicsMagick developers acknowledged the discrepancies but chose to leave it as is to tolerate slightly malformed inputs. This suggests that different implementations have design choices to tolerate slightly malformed inputs. We believe that by providing such discrepancy information, PathDiff can benefit real-world situations when one wants to use a similar library as a drop-in replacement for another one. Because she might subjectively assume that both libraries function in the same way while they actually do not. Knowing the discrepancies between libraries helps users understand the potential impact of subtle functionality changes so as to make better decisions.

```
static Image *ReadBMPImage(...){
    ...
    if ((bmp_info.size != 12) && (bmp_info.size != 40) &&
        (bmp_info.size != 108) && (bmp_info.size != 124) &&
        (!(bmp_info.size >= 12 && bmp_info.size <= 64)))
        ThrowBMPReaderException(CorruptImageError,ImproperImageHeader,
        image);
    ...
```

Listing 7.6: GraphicsMagick's check on information header size

```
static Image *ReadBMPImage(...){
    ...
    if ((MagickSizeType) bmp_info.file_size != blob_size)
    {
        ...
        if (IsStringTrue(option) == MagickFalse)
            (void) ThrowMagickException(exception,GetMagickModule(),
            CorruptImageError,"LengthAndFilesizeDoNotMatch","`%s'",
            image->filename);
    ...
```

Listing 7.7: ImageMagick's check on file size

### 7.4.4 BoringSSL vs. LibreSSL

This bug is reported by Nezha in their paper. However, their report is imprecise, and PathDiff's symbolic analysis is able to provide a more complete understanding of the root cause of the discrepancy.

In the Nezha paper, only Line 3 to Line 6 of Listing 7.9 is presented. The authors state that LibreSSL masks the padding (stored in variable `i`) with 0x07 and continues processing (Line 6 in Listing 7.9), while BoringSSL compares `padding > 7` and rejects the value, resulting in a discrepancy. Given the description in the Nezha paper, one might incorrectly conclude that any padding value `> 7` will generate a discrepancy. However, this is not the case—at Line 13 in Listing 7.9, 0xFF is shifted left based on the value of the padding value byte and then used to mask the last byte in `s` (when parsing the KeyUsage extension, the `len` is 1 upon reaching the bit shift operation). If an arbitrary padding value `> 7` is used, this will cause `s[0]` to become 0, which also causes LibreSSL to reject the certificate as invalid, resulting in no discrepancy.

When running PathDiff, the input certificate has padding value 1 which does not trigger a discrepancy. By negating the constraint `padding <= 7` generated at Line 5 in Listing 7.8, PathDiff discovers a candidate input where the padding is set to 33, triggering a discrepancy. To understand how PathDiff is able to find the specific value of 33, we examined the collected constraints and found that a constraint `padding & 0x1f != 0` is generated at Line 13 in Listing 7.9 (remember that the constraints we collect are path constraints. Since the padding value in the input is 1, the operator here is "not equal"). This constraint is not explicit in the code, but is added by the CPU model in $S^2E$ (which precisely models the shift instruction based on real CPU behaviour) because the CPU masks shift amounts to 5 bits (i.e. maximum shift count is 31). By considering both the negated constraint ¬`(padding <= 7)` from BoringSSL and the constraint `padding & 0x1f != 0` from LibreSSL, PathDiff generates the value 33 for the padding, which is equivalent to a shift amount of 1. Thus, such value allows LibreSSL to treat the padding byte as if it was 1 when performing the shift and parse the input without error, while BoringSSL's padding check fails, resulting in an invalid input error, triggering the discrepancy. We can see from this example that PathDiff can capture discrepancy-triggering input more reliably based on complete reasoning of the behaviour of the programs and their execution environment as opposed to depending on randomness.

```
1  ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp){
2      ...
3      p = *pp;
4      padding = *(p++);
5      if (padding > 7) {
6          OPENSSL_PUT_ERROR(ASN1, ASN1_R_INVALID_BIT_STRING_BITS_LEFT);
7          goto err;
8      }
9      ret->flags&=~(ASN1_STRING_FLAG_BITS_LEFT|0x07);
10     ret->flags|=(ASN1_STRING_FLAG_BITS_LEFT|padding);
11     ...
```

Listing 7.8: BoringSSL code for validating bitstrings

```
1  ASN1_BIT_STRING *c2i_ASN1_BIT_STRING(..., char **pp){
2      ...
3      p = *pp;
4      i = *(p++);
5      ret->flags&=~(ASN1_STRING_FLAG_BITS_LEFT|0x07);
6      ret->flags|=(ASN1_STRING_FLAG_BITS_LEFT|(i&0x07));
7      ...
8      if (len-- > 1)
9          {
10             s = malloc(len);
11             ...
12             memcpy(s, p, len);
13             s[len - 1] &= (0xff << i);
14             p += len;
15         }
16     ...
```

Listing 7.9: LibreSSL code for validating bitstrings

# Chapter 8

# Limitations and Future Work

The limitations of PathDiff are mostly performance-wise and are attributable to the challenges in the symbolic execution domain. PathDiff follows a modular design and allows the components to be replaced with minimal effort. Thus, future advances in the symbolic execution domain can be easily installed to PathDiff for better scalability.

## 8.1 Constraint Collection

PathDiff currently relies on $S^2E$ to collect constraints and thus inherits its limitations. We observed that concolic execution in $S^2E$ takes much longer than regular execution of programs, which is due to the tracking of symbolic variables, translating and interpreting each instruction, etc. In addition, $S^2E$ crashed when collecting the constraints from MuPDF in our experiment, suggesting bugs inside the $S^2E$ concolic execution engine. If there is a future concolic execution engine that addresses these issues, PathDiff will benefit from it as well.

## 8.2 Constraint Solving

Constraint solving is the key bottleneck of PathDiff. On the one hand, constraint solving dominates the run time of PathDiff. On the other hand, security-sensitive applications usually implement encoding functions (e.g. encryption/decryption, checksum) to enforce confidentiality and integrity. Solving constraints from such functions is computationally hard because it is analogous to attacking cryptographic hash functions. In our experiments, we had to manually patch the programs under test to remove the encoding functions such as the signature check in SSL/TLS libraries. The patching is manual and requires access to the source code which may not always be available.

Fortunately, constraint solving is one of the major backbone technologies behind symbolic execution and there have been advances in solving the aforementioned problems, which PathDiff can benefit from as well. For example, KLEE [12] optimizes the solver performance using incremental solving. The insight is that by caching the solutions of previous queries, KLEE can easily answer new queries that involve the supersets/subsets of the cached queries. For instance, if KLEE has queried the solver with $(x > 0) \wedge (x < 10)$ and obtained solution $x = 5$, the next time when KLEE wants to query the solver with the constraints $(x > 0) \wedge (x < 10) \wedge (x < 20)$, it can quickly validate

that the previous cached solution $x = 5$ is still valid and thus do not need to query the solver. Caballero et al. proposed a method to automatically identify the encoding functions constraints and query the solver with the remaining constraints for a partial input [10]. The complete input is then generated by concretely executing the encoding function. We believe that these solutions can be integrated into PathDiff to provide better performance and fully automatic analysis of security-sensitive applications.

## 8.3 Seed Selection

PathDiff currently ranks the seeds based on the rank of the coverage sum of the two programs with the intuition that more coverage provides larger space to explore, potentially leading to more discrepancies. However, this scheme has redundancy because usually most of the coverage overlaps for two seeds that have consecutive ranks. As a result, PathDiff has to spend resources running a seed only to find the same discrepancies it has already discovered when running with the previous seed. As running all seeds may be impossible due to a large number of seeds, it is important to concentrate computing resources on new coverage to find more new discrepancies. Thus, a good ranking scheme should try to minimize the redundancy, meaning that it should re-rank the remaining seeds based on the coverage overlap with the inputs that PathDiff has already run with, which requires more fine-grained coverage tracking as well as a better algorithm to reduce the computational complexity. We plan to design a dedicated seed selection scheme to provide seeds that have high new coverage per seed for PathDiff to explore.

## 8.4 Implementation Limitations

In this section, we discuss the 3 major limitations of our prototype implementation of PathDiff and outline solutions for future work.

First, $S^2E$ executes the program in a virtual machine environment called QEMU, which has a different Operating System (OS) environment from our host machine. Thus, we implemented the validation phase as invocations of docker containers that have the same OS environment as the QEMU image. When the number of satisfiable queries increases, the invocation of the validation phase increases, causing the overhead of starting docker containers to become significant. We plan to keep the container up as a service to address this problem.

Second, when dumping constraints of a program, $S^2E$ assigns labels to expressions for reference at their later occurrence. This improves the performance of the solver (e.g. less time taken to load the constraints and less memory footprint) when $S^2E$ concolically executes a program to explore it. PathDiff also benefits from the labeling for each individual program. However, PathDiff currently does not redo the labeling when combining the constraints from the two programs, causing redundancy in the generated conjunction of constraints (i.e. an expression is dumped in its original form again while it could have been a reference to an existing expression), resulting in performance degradation of the solver. An intuitive solution is to make PathDiff re-label the expressions after combining the constraints. However, it is analogous to implementing a parser for the solver query language, which is non-trivial and introduces overhead since PathDiff's candidate generation phase is implemented in Python. To address the two issues, we plan to explore the feasibility of extract-

ing the query language parser from the existing solver implementation to a C/C++ library and let PathDiff offload the label reconstruction to the library, which simplifies implementation and provides lower overhead than a Python implementation.

Finally, an implementation artifact in $S^2E$ makes it difficult to automatically start it with a different seed each time, preventing PathDiff from automatically running one seed after another. We believe with more engineering effort, this can be fixed by enhancing $S^2E$'s launch script.

# Chapter 9

# Conclusion

Despite the recent advance in differential testing that guides the testing process with the asymmetries among programs under test, differential testing remains unsystematic and thus misses many discrepancies that could have been found if the collected information is examined more carefully. In an effort to make differential testing directed and, more importantly, systematic, we present PathDiff, a novel domain-independent differential testing tool that utilizes symbolic analysis to systematically find all discrepancies for the given path-pairs. PathDiff collects path constraints from the two programs under test and negates one of the programs' constraints. Combining the new set of constraints with that of the other program, PathDiff generates inputs that execute the original path in one program and diverges the path in the other, directly targeting discrepancies. By repeating the process for each constraint, PathDiff conducts a systematic search for discrepancies around the path-pair, enabling it to discover more discrepancies than random fuzzers given the same resources.

We have performed extensive evaluation of PathDiff by comparing it with the state-of-the-art differentialt testing tool Nezha using all 12 applications used in the Nezha paper and 4 newly selected ones. PathDiff reported 4.1× more discrepancies than Nezha. We have also evaluated the imprecision mitigation techniques by incrementally applying each technique. The results show that the techniques improve the percentage of satisfiable queries by 7.03% and number of discrepancies by 3.5% on average, proving the effectiveness of the techniques.

Based on our experience, we believe that PathDiff can help discover non-compliance with the specifications, pinpoint newly introduced bugs in regression testing and fingerprint applications.

# Bibliography


[1] addr2line(1). `https://linux.die.net/man/1/addr2line`.

[2] American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`.

[3] George Argyros et al. "Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1690–1701.

[4] Marcel Böhme et al. "Coverage-based greybox fuzzing as markov chain". In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506.

[5] Marcel Böhme et al. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[6] BoringSSL. `https://boringssl.googlesource.com/boringssl/`.

[7] Chad Brubaker et al. "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 114–129.

[8] David Brumley et al. "Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation." In: *Proceedings of the 16th USENIX Security Symposium*. 2007, p. 15.

[9] Building KLEE with LLVM 9. `https://klee.github.io/build-llvm9/`.

[10] Juan Caballero et al. "Input generation via decomposition and re-stitching: Finding bugs in malware". In: *Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security*. 2010, pp. 413–425.

[11] Cristian Cadar et al. "EXE: Automatically generating inputs of death". In: *ACM Transactions on Information and System Security* 12.2 (2008), pp. 1–38.

[12] Cristian Cadar et al. "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*. Vol. 8. 2008, pp. 209–224.

[13] Cristian Cadar et al. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[14] Sze Yiu Chau et al. "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations". In: *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 503–520.

[15] Yuting Chen et al. "Coverage-directed differential testing of JVM implementations". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 85–99.

[16] Yuting Chen et al. "Guided differential testing of certificate validation in SSL/TLS implementations". In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 793–804.

[17] Vitaly Chipounov et al. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *ACM SIGPLAN Notices* 46.3 (2011), pp. 265–278.

[18] ClamAV. `http://www.clamav.net/`.

[19] CVE-2014-0224. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0224`.

[20] CVE-2017-6592. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=2017-6592`.

[21] CVE-2017-6593. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6593`.

[22] Will Dietz et al. "Understanding integer overflow in C/C++". In: *ACM Transactions on Software Engineering and Methodology* 25.1 (2015), pp. 1–29.

[23] Evince. `https://wiki.gnome.org/Apps/Evince`.

[24] GNU Binutils. `https://www.gnu.org/software/binutils/`.

[25] GnuTLS. `https://www.gnutls.org/`.

[26] Patrice Godefroid et al. "DART: Directed automated random testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005, pp. 213–223.

[27] GraphicsMagick Image Processing System. `http://www.graphicsmagick.org/`.

[28] Istvan Haller et al. "TypeSan: Practical type confusion detection". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 517–528.

[29] Christian Holler et al. "Fuzzing with code fragments". In: *Proceedings of the 21st USENIX Security Symposium*. 2012, pp. 445–458.

[30] Endadul Hoque et al. "Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs". In: *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2017.

[31] Jaewon Hur et al. "Difuzzrtl: Differential fuzz testing to find cpu bugs". In: *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. IEEE. 2021, pp. 1286–1303.

[32] ImageMagick. `https://imagemagick.org/index.php`.

[33] Information technology – digital compression and coding of continuous-tone still images – requirements and guidelines. `https://www.w3.org/Graphics/JPEG/itu-t81.pdf`.

[34] Suman Jana et al. "Abusing file processing in malware detectors for fun and profit". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 80–94.

[35] Yuseok Jeon et al. "Hextype: Efficient detection of type confusion errors for c++". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2373–2387.

[36] jsfunfuzz. https://github.com/MozillaSecurity/funfuzz.

[37] Kleaver. https://klee.github.io/docs/kleaver-options/.

[38] KQuery. https://klee.github.io/docs/kquery/.

[39] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[40] libjpeg. http://libjpeg.sourceforge.net/.

[41] libjpeg-turbo. https://libjpeg-turbo.org/.

[42] LibreSSL. https://www.libressl.org/.

[43] mbedTLS. https://tls.mbed.org/.

[44] William M. McKeeman. "Differential Testing for Software". In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107.

[45] Charlie Miller. "Fuzz by number: More data about fuzzing than you ever wanted to know". In: *Proceedings of the 2008 CanSecWest* (2008).

[46] Changwoo Min et al. "Cross-checking semantic correctness: The case of finding file system bugs". In: *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*. 2015, pp. 361–377.

[47] MuPDF. https://mupdf.com/.

[48] Shirin Nilizadeh et al. "DifFuzz: differential fuzzing for side-channel analysis". In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE. 2019, pp. 176–187.

[49] Yannic Noller et al. "HyDiff: Hybrid differential software analysis". In: *Proceedings of the 42nd International Conference on Software Engineering*. IEEE. 2020, pp. 1273–1285.

[50] OpenSSL. https://www.openssl.org/.

[51] Hristina Palikareva et al. "Shadow of a doubt: testing for divergences between software versions". In: *Proceedings of the 38th International Conference on Software Engineering*. 2016, pp. 1181–1192.

[52] Jihyeok Park et al. "Jest: N+1-version differential testing of both javascript engines and specification". In: *Proceedings of the 43rd International Conference on Software Engineering*. IEEE. 2021, pp. 13–24.

[53] Theofilos Petsios et al. "Nezha: Efficient domain-independent differential testing". In: *Proceedings of the 2017 IEEE Symposium on Security and Privacy*. IEEE. 2017, pp. 615–632.

[54] David A Ramos et al. "Under-constrained symbolic execution: Correctness checking for real code". In: *Proceedings of the 24th USENIX Security Symposium*. 2015, pp. 49–64.

[55] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: *Proceedings of the 24th Network and Distributed Systems Symposium*. Vol. 17. 2017, pp. 1–14.

[56] S2E. https://github.com/S2E/s2e.

[57] Koushik Sen et al. "CUTE: A concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[58] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Annual Technical Conference*. 2012, pp. 309–318.

[59] Konstantin Serebryany et al. “ThreadSanitizer: data race detection in practice”. In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. 2009, pp. 62–71.

[60] JaeSeung Song et al. “SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications”. In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 695–709.

[61] Nedim Šrndić; et al. “Practical evasion of a learning-based classifier: A case study”. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 197–211.

[62] Evgeniy Stepanov et al. “MemorySanitizer: fast detector of uninitialized memory use in C++”. In: *Proceedings of the 2015 International Symposium on Code Generation and Optimization*. IEEE. 2015, pp. 46–55.

[63] Nick Stephens et al. “Driller: Augmenting fuzzing through selective symbolic execution.” In: *Proceedings of the 23rd Network and Distributed Systems Symposium*. Vol. 16. 2016. 2016, pp. 1–16.

[64] STP. `https://github.com/stp/stp/releases/tag/2.3.3`.

[65] Lin Tan et al. “/* iComment: Bugs or bad comments? */”. In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. 2007, pp. 145–158.

[66] Shin Hwei Tan et al. “@ tcomment: Testing javadoc comments to detect comment-code inconsistencies”. In: *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 260–269.

[67] Caroline Tice et al. “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM”. In: *Proceedings of the 23rd USENIX Security Symposium*. 2014, pp. 941–955.

[68] wolfSSL. `https://www.wolfssl.com/`.

[69] Xpdf. `https://www.xpdfreader.com/`.

[70] Weilin Xu et al. “Automatically evading classifiers”. In: *Proceedings of the 23rd Network and Distributed Systems Symposium*. Vol. 10. 2016.

[71] XZ Utils. `http://tukaani.org/xz/`.

[72] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011, pp. 283–294.

[73] Yibiao Yang et al. “Hunting for bugs in code coverage tools via randomized differential testing”. In: *Proceedings of the 41st International Conference on Software Engineering*. IEEE. 2019, pp. 488–499.

[74] Yong-Hao Zou et al. “TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing”. In: *Proceedings of the 2021 USENIX Annual Technical Conference*. 2021, pp. 489–502.