Master's Thesis

Symbolic Execution Groundwork for Distinguishing Automatically Generated Patches

Nguyễn Gia Phong

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2023

Symbolic Execution Groundwork
for Distinguishing Automatically Generated Patches

Nguyễn Gia Phong

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

Symbolic Execution Groundwork for Distinguishing Automatically Generated Patches

A thesis submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of Master of Science

Nguyễn Gia Phong

2024-01-12
Approved by
Advisor

Jooyong Yi

Symbolic Execution Groundwork
for Distinguishing Automatically Generated Patches

Nguyễn Gia Phong

This certifies that the thesis of Nguyễn Gia Phong is approved.

2024-01-12

Advisor: Jooyong Yi

Mijung Kim

Yuseok Jeon

Abstract

In recent decades, automated program repair (APR) has been advancing consistently according to benchmarks. However, its use in practice is still limited due to the difficulty in choosing a desired patch among the generated pool.

This work introduces a method to logically differentiate between patches through symbolic execution. The technique generates a tree of decisions for developers to reason between patches based on the program's inputs and semi-automatically captured outputs. Its implementation PSYCHIC based on KLEE is evaluated on patches automatically generated for toy programs in the INTROCLASS benchmark, showing promising preliminaries.

Contents

List of Figures

Technical Terms and Abbreviations

APR automated program repair. 1, 7, 10, 12

bug is a software defect that results in undesired behaviors. 1

differential test is an input and a collection of outputs of two or more programs of the same purpose, such that each output is pair-wise different. 1, 4, 8, 9

FFI foreign function interface. 7

IR intermediate representation. 3

SAT satisfiability. 2, 3, 8

SMT satisfiability modulo theories. 3, 4, 7, 8, 12, 13

state or *symbolic process*, includes a program counter, an execution environment (a register file, a stack frame, and an address space for heap and global variables), and path constraints. Variables in an execution environment are all symbolic expressions. 2, 4

symbolic output is an output or intermediate value of a program under symbolic execution that reflects behaviors of the program relevant to the analysis. 4, 7, 8

UB undefined behavior. 8, 10, 11

I Introduction

Modern software development is largely an iterative and incremental process [1]. Due to practically commonly weak specifications and thus incomplete verification, each refinement may involve discovering, eliminating, and creating new software defects, or bugs. Like for other software engineering tasks, efforts have been put into automating this debugging process, including detecting and locating software faults [2], and generating fixes [3].

Progress on APR is greatly hindered by the weak specification issue as well [3]. Due to the criteria for correctness being incomplete, patches generated may over-fit such test suites, resulting in an incomplete fix and/or new regression bugs [4]. For the same reason, more than one patch fitting the specification can be found. Contemporary APR tools do not give insight on the difference between these patches, but only a ranking on potential correctness [3].

While this ranking is beneficial in evaluating the tools themselves, together with the lack of certainty of correctness, it is an insufficient guide for choosing *the* desired patch, if any. APR users must then verify each patch to decide which to apply. Recognizing the tedious nature of this process, we work on methods to highlight the semantic difference between patches, in the form of differential tests. Existing automation techniques for differentiation such as black- or gray-box fuzzing [5] and symbolic execution [6] has shown promise on pairs of program revisions. However, there is a lack of precedents in doing so at scale like for multiple APR-generated patches.

Symbolic execution is observed as one promising direction as it works directly with path constraints, which can be combined and manipulated to reveal semantic differences. In this research, we explore symbolic execution for mass differential testing with the ultimate goal of making deciding on automatically generated patches easier for developers. The main contributions of this work are:

1. Introduction of semantic difference mining from multiple program revisions as a semi-automated pipeline, for communicating the reasoning behind each patch in form of a decision tree
2. A tool named PSYCHIC implementing this process for C programs and handling platform specificities for applicative generalization
3. Preliminary results of its performance on patches automatically generated for small programs

II Background

As mentioned earlier, our technique is based on the symbolic execution, an engine named KLEE in particular. Their general principles and some specifics are laid out here to make it less ambiguous to describe our approach later on.

2.1 Symbolic execution

At a high level, symbolic execution is an interpreter loop. Every iteration, it selects a symbolic process, also known as a *state*, in whose context an instruction is then executed [7]. As shown in algorithm 1, each path with possible satisfiability (SAT) of the given program is explored. The number of paths grows at an exponential rate, so the interpreter loop also breaks at the exhaustion of the time budget [7], which is omitted from the algorithm for brevity.

Algorithm 1 Symbolic execution

```
Input: Program ($P$)
Output: Set of test cases ($T$)
 1: $T \leftarrow \varnothing$
 2: $S \leftarrow \textsc{InitialState}(P)$                ▷ set of states
 3: while $S \neq \varnothing$ do
 4:     $s \leftarrow \textsc{Select}(S)$
 5:     $S \leftarrow S \setminus \{s\}$
 6:     $(i, e, \Phi) \leftarrow s$                ▷ instruction, environment, and path constraints
 7:     $(i', e') \leftarrow \textsc{Execute}(i, e)$
 8:     if $\textsc{IsHalt}(i)$ then
 9:         $t \leftarrow \textsc{GenerateTest}(\Phi)$
10:         if $t \neq \bot$ then $T \leftarrow T \cup \{t\}$
11:     else if $\textsc{IsBranch}(i)$ then
12:         $(\varphi, i'_c, i'_a) \leftarrow i'$                ▷ condition, consequent, and alternative
13:         if $\textsc{sat}(\Phi \land \varphi)$ then
14:             $s'_c \leftarrow (i'_c, e', \Phi \land \varphi)$
15:             $S \leftarrow S \cup \{s'_c\}$
16:         if $\textsc{sat}(\Phi \land \neg\varphi)$ then
17:             $s'_a \leftarrow i'_a, e', \Phi \land \neg\varphi$
18:             $S \leftarrow S \cup \{s'_a\}$
19:     else
20:         $s' \leftarrow (i', e', \Phi)$
21:         $S \leftarrow S \cup \{s'\}$
```

A SAT problem involving background theories like arithmetic in computer programs is known as a satisfiability modulo theories (SMT) problem, a generalization of the Boolean SAT problem [8], which is already NP-complete. Under the exponential time hypothesis, the cost of solving an arbitrary SAT problem grows exponentially with the number of variables [9].

2.2 Usage of KLEE

The symbolic execution engine KLEE analyses the LLVM intermediate representation (IR) of a program, also known as its bitcode [10], starting from values marked as symbolic. Marking can be communicated either through command-line options for files (including standard streams) and command-line arguments or via in-source annotations [11].

As shown in algorithm 1, the path constraints are only expanded at branching instructions, so only values affecting the control flow are tracked, since KLEE targets the exploration of execution paths. In order to track other values of interest, i.e. non-inputs, a condition involving such value must be artificially injected. One common idiom for doing this condition injection [12–14] is shown in figure 1. There, a temporary symbolic avatar `var` is created for the sole purpose of being fixed to the value `val` in the path constraints. From here, this mechanism shall be referred to in abstract descriptions as SYMBOLICOUTPUT(VALUE, CONTEXT).

```
#define KLEE_OUTPUT_GEN(T)                         \
T __klee_output_##T(T val, ...)     { T var; \
  klee_make_symbolic(&var, sizeof(T), ...); \
  klee_assume(var == val);      return val; }
KLEE_OUTPUT_GEN(int) ...
#define KLEE_OUTPUT(T, val, ...) \
  __klee_output_##T((val), ...)
```

(a)

```
 int foo;
 ...
-bar(foo);
+bar(KLEE_OUTPUT(foo));
```

(b)

Figure 1: Condition injection (a) implementation in C and (b) example annotation.

III Technique

In order to generate test cases to distinguish between multiple patches, our tool extends symbolic execution in a number of ways. First, instead of taking in a single program, all of its revisions are passed to the tool. Since each patch only modifies a small portion of the program, most of the code is shared among these revisions, hence they are combined into a *meta program* to avoid repeating execution.

Second, we combine the path constraints in execution states of different revisions in a certain way before feeding to a SMT solver to generate differential tests. Third, the new conditions the combined SMT formula includes (aside from the conjunction of its parents) are assertions on the distinction of corresponding values across revisions, so we offer multiple options to capture such *symbolic outputs*.

Next, we introduce concrete execution to reduce the theorem proving load, and produce differential tests for clustering the patches and finally produce a decision tree. One may observe that the clustering information can help with selecting more "promising" states, although such scheduling enhancement is not covered by this work.

We implemented these extensions on top of KLEE in a tool named Psychic, whose high-level architectures are compared in figure 2. The overall process of Psychic is denoted in algorithm 2.

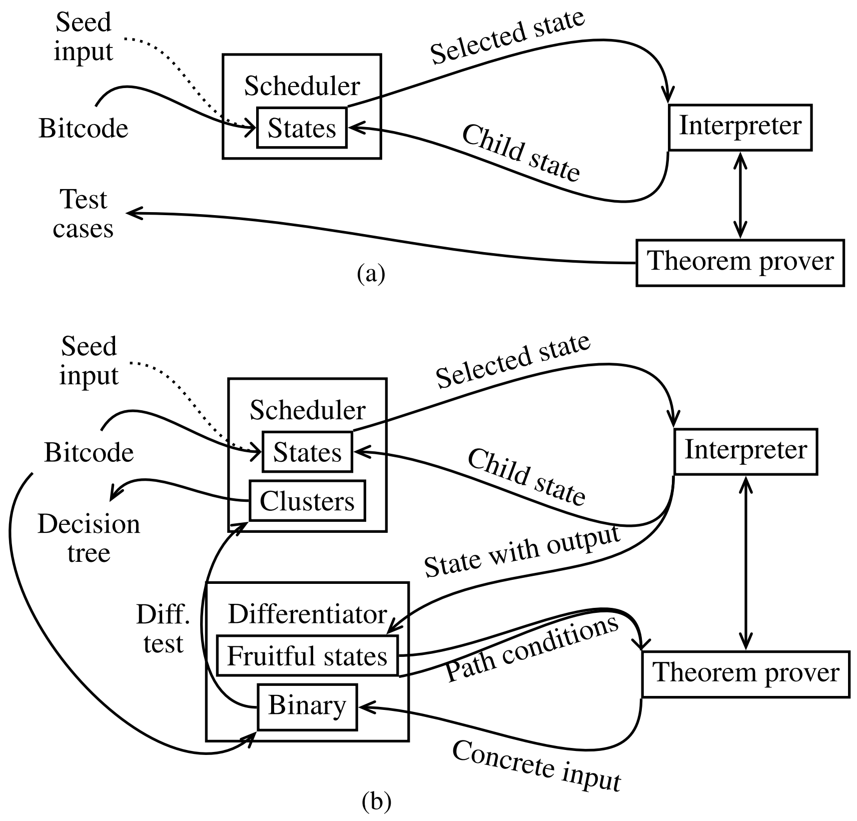

Figure 2: Architecture overviews of (a) KLEE and (b) Psychic extension.

Algorithm 2 Overall execution of PSYCHIC

Input: Meta program P

Output: Set of differential tests T

```
1:  $T \leftarrow \varnothing$
2:  $D \leftarrow \varnothing$                                                         ▷ Distinguished revision pairs
3:  $H \leftarrow \varnothing$                                                         ▷ Halt states
4:  $S \leftarrow$ INITIALSTATE$(P)$
5:  while $S \neq \varnothing$ do
6:      $(i, e, \Phi, n) \leftarrow$ SELECT$(S)$     ▷ instruction, environment, path constraints, and revision number
7:      $S \leftarrow S \setminus \{(i, e, \Phi, n)\}$
8:      $(i', e') \leftarrow$ EXECUTE$(i, e)$
9:      if ISHALT$(i)$ then
10:         for $(\Phi', n') \in H$ do
11:             $t \leftarrow$ DIFFTEST$(D, \Phi, n, \Phi', n')$
12:             if $t \neq \perp$ then
13:                 $T \leftarrow T \cup \{t\}$
14:                 $D \leftarrow D \cup$ DISTINGUISHPAIRS$(t)$
15:         $H \leftarrow H \cup \{(\Phi, n)\}$
16:     else if ISBRANCH$(i)$ then
17:         $(\varphi, i'_c, i'_a) \leftarrow i'$                          ▷ condition, consequent, and alternative
18:         if ISMETA$(\varphi)$ then
19:             $S \leftarrow S \cup$ APPLYMETABRANCH$(\varphi, i'_c, i'_a, e', \Phi, n)$
20:         else
21:             if SAT$(\Phi \wedge \varphi)$ then $S \leftarrow S \cup \{(i'_c, e', \Phi \wedge \varphi, n)\}$
22:             if SAT$(\Phi \wedge \neg\varphi)$ then $S \leftarrow S \cup \{(i'_a, e', \Phi \wedge \neg\varphi, n)\}$
23:     else
24:         if ISRETURN$(i)$ then
25:             $\Phi' \leftarrow \Phi \wedge$ FUNCTIONOUTPUTS$(i)$
26:         else
27:             $\Phi' \leftarrow \Phi$
28:         $S \leftarrow S \cup \{(i', e', \Phi')\}$
```

```
--- dfasearch.c c1cb19fe
+++ dfasearch.c 8f08d8e2
@@ -360,2 +360,1 @@
-beg = match;
-goto success_in_len;
+goto success;
```

(a)

```
--- dfasearch.c c1cb19fe
+++ dfasearch.c YmI4MGM3ND
@@ -360,2 +360,2 @@
 beg = match;
-goto success_in_len;
+goto success;
```

(b)

Figure 3: Unified format of (a) a fix in `grep`'s commit `8f08d8e2`, and (b) an alternative patch.

```
int __klee_meta2 = __klee_meta(2);
if (__klee_meta(2) == 0) { // original buggy version
  beg = match;
} else if (__klee_meta2 == 82) { // upstream patch
} else if (__klee_meta2 == 43) { // alternative patch
  beg = match;
}
int __klee_meta3 = __klee_meta(3);
if (__klee_meta(3) == 0) {
  goto success_in_len;
} else if (__klee_meta3 == 82) {
  goto success;
} else if (__klee_meta3 == 43) {
  goto success;
}
```

Figure 4: Meta branches with the fixes in `grep`'s upstream and alternative patches in figure 3.

3.1 Meta program

The meta program format takes inspiration from program unification used for shadow symbolic execution. Unlike SHADOW which tests only one patch [6] to reveal its regressions, though, the goal of PSYCHIC is to find the difference in behavior between multiple patches. Therefore, the former's annotation at the level of expressions does not meet our expectation on expressivity. Instead, we implement higher-order branching. Annotating patches at block level simplifies the generation process and improves the readability of the meta program.

For instance, given the two unified diffs [15] of a bug fix from upstream `grep`, and a patch in DBGBENCH [16] in figure 3,[*] one can construct the relevant part of the meta program in figure 4, where `__klee_meta` parses the environment for the choice of revision when executed concretely [17] and returns a new symbolic value in symbolic evaluation. This way, the meta program can be used for both executions to share metadata such as the patch identifier (82 and 43 in this case).

One may notice that the meta program may have any level of granularity. PSYCHIC utilizes the meta program to maximize instructions and path constraints that are shared, which effectively means to defer the divergence point deep into the control flow. For its intended use on automatically generated patches whose hunks are commonly short, optimizing the meta program construction is not a major issue.

[*]These snippets have been outdented to fit within the width of this page.

In algorithm 2, ISMETA cheaply detects higher-order branching patch identifier, via basic tree pattern matching [18] and skip adding its condition to the children states. Instead, those states carry the revision number outside of their path constraints for constant-time access when pruning paths going through multiple patches in algorithm 3.

Algorithm 3 Construction of possible states after a patch location

```
1: procedure ApplyMetaBranch($\varphi, i_c, i_a, e, \Phi, n$)
2:     $n' \leftarrow$ RevisionNumber($\varphi$)
3:     $s_a = (i_a, e, \Phi, n)$   ▷ assume the lack of meta branch nesting and always follow alternative paths
4:     if $n = 0$ then return $\{(i_c, e, \Phi, n'), s_a\}$   ▷ $i_c$ enters patch $n'$
5:     if $n' \neq 0 \wedge n \neq n'$ then return $\{s_a\}$   ▷ at most one simultaneous patch
6:     return $\{(i_c, e, \Phi, n), s_a\}$
```

3.2 Symbolic output selection

PSYCHIC is based on KLEE and thus inherits its support for the diverse set of programs written in languages with C foreign function interface (FFI) for calling KLEE intrinsics and a compiler using LLVM as a backend. One kind of LLVM instruction may be used for difference purposes, so for better applicability we try to accommodate for a range of usages. For capturing output values, PSYCHIC offers the following methods.

1. Source code annotation
2. Output files (including standard output)
3. Function return values
4. Pointers to mutable data as function arguments

Method 1 is handled similar to previous works described in section 2.2. With KLEE support for symbolic files, the same avatar idiom is trivially applied for 2. However, since files are byte buffers, their application in symbolic execution is limited due to the scalability of SMT solvers, which treat each byte as a variable. For human-readable files, this is particularly inefficient due to their low information entropy [19]. On the other hand, because of imperfect fault localization either by human [16] or automatically [20], output annotation has limited exhaustiveness.

To balance between manual annotation and computational scalability, methods 3 and 4 are introduced to automatically capture output more precisely at the end of every interested function. Debugging information forwarded from compiler frontend are kept in bitcodes. They are used to limit the subroutines to be monitored, for example in algorithm 4, the file that is patched by an APR tool.

Each symbolic output then is given a unique name for later comparison. For the sake of simplicity, we name them after the capturing method and associated identifier, i.e. variable or subroutine.

Algorithm 4 Extraction of symbolic outputs of a function

```
1: procedure FunctionOutputs($i$)
2:     $f \leftarrow$ Function($i$)
3:     if $f$ is not in the patched file then return $\top$
4:     $\Phi \leftarrow$ SymbolicOutput(ReturnValue($i$), $f$)
5:     for $a \in$ FunctionArguments($i$) do
6:         if IsPointer($a$) $\land \neg$IsConstPointer($a$) $\land \neg$IsFunctionPointer($a$) then
7:             $\Phi \leftarrow \Phi \land$ SymbolicOutput(Dereference($a$), $a$)
8:     return $\Phi$
```

3.3 Input generation

When two states monitor some commonly named symbolic outputs and are at *parallel* positions in their path, their path constraints can be used to generate a differential test. Two states are said to be parallel if and only if their program counter points to the same instruction and the conjunction of their path conditions is satisfiable.

Currently PSYCHIC only compare halt states to minimize memory usage and SMT solving cost. Such SMT formula is constructed in algorithm 5 by first append a suffix to each output name, then for the common ones assert for a possible distinction between two paths. If the path constraints of both states are SAT with at least one difference in outputs, a differential test can be generated from the SMT model.

Halt states are reached naturally at the program's termination or an error. The latter category also includes sanitizer traps for undefined behavior (UB).

3.4 Decision tree construction

From each differential test, a set of clusters of revisions with the same output is constructed. We then superimpose these clustering sets to create hierarchical clusterings or a decision tree. A minimal tree, i.e. a tree with the shortest height, is found by searching through the subsets of the set of clustering sets.

Algorithm 5 Differential test generation

```
1:  procedure $\textsc{DiffTest}(D, \Phi, n, \Phi', n')$
2:      if $n \neq n' \vee \{n, n'\} \in D$ then return $\perp$
3:      $\Psi, N \leftarrow \textsc{RenameOutputs}(\Phi, n)$
4:      $\Psi', N' \leftarrow \textsc{RenameOutputs}(\Phi', n')$
5:      $\Psi_{\text{diff}} \leftarrow \bigvee_{a \in N \cap N'} \textsc{Distinct}(a, n, n')$
6:      if $\neg\textsc{sat}(\Psi \wedge \Psi' \wedge \Psi_{\text{diff}})$ then return $\perp$
7:      $m \leftarrow \textsc{Model}(\Psi \wedge \Psi' \wedge \Psi_{\text{diff}})$
8:      $I \leftarrow \textsc{InputValues}(m)$
9:      return $(m, \{\textsc{Execute}(I, r) \mid r \in \textsc{revisions}\})$
10: procedure $\textsc{RenameOutputs}(x, n)$
11:     $X, k \leftarrow x$                                   ▷ subexpressions and kind
12:     if $\textsc{IsRead}(t)$ then
13:         $a, i \leftarrow X$                               ▷ array name and index
14:         if $\textsc{IsOutput}(a)$ then
15:             $a' \leftarrow \textsc{AppendName}(a, n)$
16:             $x' \leftarrow ((a', i), k)$
17:             return $x', \{a\}$
18:         else
19:             return $x, \varnothing$
20:     $X' \leftarrow X$
21:     $N \leftarrow \varnothing$                            ▷ seen output names
22:     for $i \in 1..\textsc{Length}(X)$ do
23:         $X'_i, N_+ \leftarrow \textsc{RenameOutputs}(E_i, n)$
24:         $N \leftarrow N \cup N_+$
25:     return $(X', k), N$
```

IV Experiments

To evaluate our approach, we perform experiments with our implementation PSYCHIC on C meta programs to generate decision trees of patches.

4.1 Experiment setup

Experiments are run on a machine with AMD Ryzen 3700X at 3.6 GHz and 16 GB of memory. Software dependencies and tooling are provided by a declarative Nix environment [21] for reproducibility.

We evaluate PSYCHIC on patches generated by MSV [17], a fork of the pattern-based APR tool PROPHET [22] that adds extra repair templates and generate meta programs. Patches are generated for INTROCLASS, a set of attempts for solving homework problems [23].

In addition to the test cases provided by INTROCLASS, which are non-exhaustive of the input domains, we use property tests [24] with the respective ground truth program[†] to select buggy programs. MSV then generate fixes in the form of meta programs, which are then post-processed into PSYCHIC-compatible format.

Due to the limitation of KLEE's POSIX runtime, `libc` calls such as `scanf` has to be replaced by more direct access of standard input or command-line arguments. Unless the logic is outside of the `main` function, an output value is manually annotated.

A selection of 10 meta programs with at least two semantically different revisions (including the original buggy version) are then fed to PSYCHIC to generate differential tests and decision trees with a timeout of 10 minutes. The vast majority of these bugs are either simple logical mistakes or missing initializations that trigger UBs.

4.2 Results

Experiment results on INTROCLASS is summarized in table 1. In case of `digits 1b31@2` and `smallest d25c@1`, PSYCHIC failed to generate differential tests. In other cases, except for the timeout in `digits 0cdf@3`, the tool successfully differentiate all semantically different patches.

PSYCHIC took orders of magnitude more time on `digits 0cdf@3` and `smallest d25c@1`, which respectively include a loop and heavily-nested conditions.

To demonstrate the decision tree, we examine the result for `grade 317a@3`. The task is to calculate the letter grade, given the threshold for each grade from A to D, followed by a percentage grade. This attempt is buggy because it misses the initialization the result grade variable, which is inserted in various locations by MSV. The decision tree generated from these patches is illustrated in figure 5, from which patch number 10 can be deduced to be the most correct one.

[†] `https://github.com/McSinyx/IntroClass`

Task	Attempt	Bug type	n	Tree height	Largest cluster	Time (s)
`checksum`	`3b23@3`	Arithmetic	2	1	1	1.2
`checksum`	`cb24@3`	Logic	3	1	2	1.7
`digits`	`0cdf@3`	Logic	16	1	5	600.0
`digits`	`1b31@2`	Logic	5	0	5	0.9
`grade`	`1b31@3`	UB	2	1	1	0.8
`grade`	`317a@3`	UB	8	3	5	1.0
`grade`	`b192@3`	UB	2	1	1	0.5
`median`	`97f6@3`	UB	7	1	6	1.6
`smallest`	`0704@2`	UB	5	1	3	87.0
`smallest`	`d25c@1`	UB	5	0	5	1.3

Table 1: Decision tree quality on the patch pool generated by MSV for IntroClass programs. Respective to each meta program with n revisions are the height of the patch decision tree and the maximum number of undistinguished revisions.

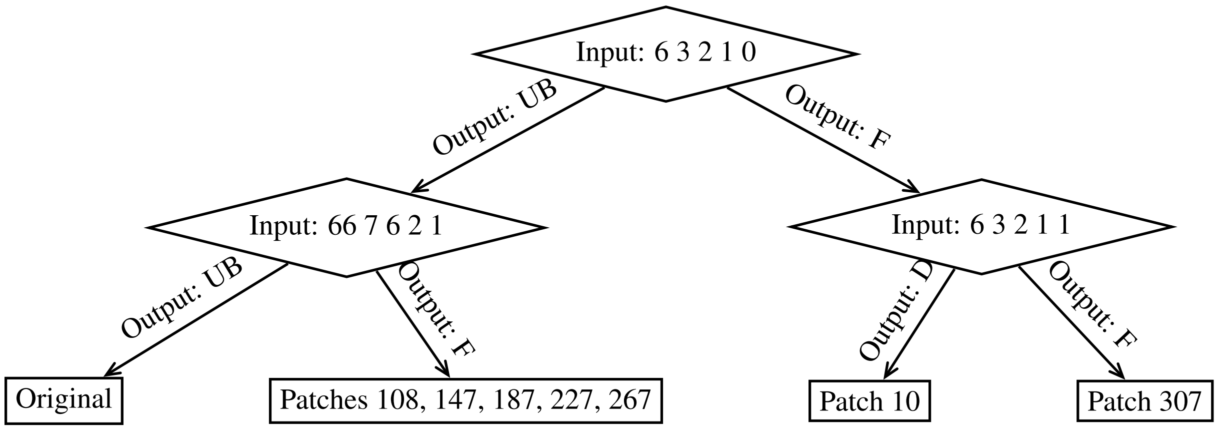

Figure 5: Decision tree generated from patches of `grade` `317a@3`, with outputs minimized.

V Discussion

The experiment on INTROCLASS reveals some limitations of PSYCHIC. First, the tool has trouble scaling with growing path spaces. While this is inherent to the path explosion problem, not all paths need to be explored. Therefore, more aggressive path pruning [25,26] and scheduling heuristics [27–29] could prove to be beneficial. The use of DIFFTEST on any (undifferentiated) state pair in algorithm 5 further hurts performance.

Next, while in theory, KLEE can execute any LLVM bitcode, the programming interface sets the limit on logical interpretation. More specifically, to leave rooms for optimization, C and POSIX standards leave a lot of behaviors unspecified. For example, function `atoi(3)`, which converts string to integer, may return any number upon a parsing error [30], lumping this implementation-defined number with error cases.

Some others like `scanf(3)` and `printf(3)` are prohibitively expensive due to heavy pattern matching, and since they operate on input/output, their calls are likely dominators or post-dominators in the control-flow graph, requiring manual interventions. Failing to replace these calls could either significantly degrade efficiency or force the theorem prover to give up on SMT solving. In case of INTROCLASS, ignoring `printf(3)` calls disabled the use of output files capturing.

On more positive notes, capturing local function return values and mutable arguments has proved its usability. Together with the use of eager concrete execution to skip symbolic test generation, promising preliminaries have been obtained.

5.1 Threats to Validity

The small scope of experiments on toy programs challenges the practical generalizibility of the tool. Since the PROPHET-based APR tool MSV mines for tokens within the codebase [22], patches are not diverse in semantics and the overall result is not statistically strong.

5.2 Related Works

There exists multiple differential test generators for a pair of program revisions, such as NEZHA which uses black- or gray-box fuzzing [5], SHADOW based on symbolic execution [6], and HYDIFF combining both [26]. Some recent work targets the opposite problem of detecting semantic code clone [31].

For interpreting multiple program variants symbolically at the same time, alternative to PSYCHIC keeping independent states, other approaches may choose shadow execution [6], or to join [32] or merge [33] states at higher-order branches.

VI Conclusion

This work introduced a differential testing technique for multiple program variants at once, implemented as an extension named PSYCHIC on top of the symbolic execution engine KLEE. It took into account edge cases to lay out a groundwork to be easily used for a diverse set C programs to help developers with choosing automatically generated patches. That being said, experiment results indicates a large room for improvements.

In the future, other techniques need to be incorporated to improving the performance and generalizibility. Sophisticated scheduling and path pruning could help with efficiency, and further works on symbolic execution runtimes would also be beneficial. Parallel SMT symbolic execution [34] should also be considered to make better use of modern hardware. Finally, a more comprehensive evaluation is needed to prove the applicability of the approach.

References

[1] C. Larman and V. R. Basili, “Iterative and incremental developments: A brief history,” *Computer*, vol. 36, no. 6, pp. 47–56, 2003, doi:10.1109/MC.2003.1204375.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng. (TSE)*, vol. 42, no. 8, pp. 707–740, 2016, doi:10.1109/TSE.2016.2521368.

[3] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019, doi:10.1145/3318162.

[4] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” in *Proc. Joint Meet. Found. Softw. Eng. (FSE)*. ACM, 2015, pp. 532–543, doi:10.1145/2786805.2786825.

[5] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, “Nezha: Efficient domain-independent differential testing,” in *IEEE Symp. Secur. and Priv. (SP)*, 2017, pp. 615–632, doi:10.1109/SP.2017.27.

[6] T. Kuchta, H. Palikareva, and C. Cadar, “Shadow symbolic execution for testing software patches,” *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 27, no. 3, Sep. 2018, doi:10.1145/3208952.

[7] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *USENIX Symp. Oper. Syst. Des. and Implement. (OSDI)*, Dec. 2008. [Online]. Available: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems

[8] L. de Moura and N. Bjørner, “Satisfiability modulo theories: An appetizer,” in *Formal Methods: Foundations and Applications*. Springer Berlin Heidelberg, 2009, pp. 23–36, doi:10.1007/978-3-642-10452-7_3.

[9] R. Impagliazzo and R. Paturi, “Complexity of k-SAT,” in *Proc. IEEE Conf. Comput. Complex. (CCC)*, 1999, pp. 237–240, doi:10.1109/CCC.1999.766282.

[10] LLVM Project, *LLVM 13.0.1 documentation*, Feb. 2022, ch. LLVM Bitcode File Format. [Online]. Available: https://releases.llvm.org/13.0.1/docs/BitCodeFormat.html

[11] KLEE *documentation*, the KLEE team. [Online]. Available: https://klee.github.io/docs

[12] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 691–701, doi:10.1145/2884781.2884807.

[13] R. S. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proc. ACM SIGPLAN Int. Conf. Program. Lang. Des. and Implement. (PLDI)*, 2021, pp. 390–405, doi:10.1145/3453483.3454051.

[14] N. Parasaram, E. T. Barr, and S. Mechtaev, "Trident: Controlling side effects in automated program repair," *IEEE Trans. Softw. Eng. (TSE)*, pp. 4717–4732, 2022, doi:10.1109/TSE.2021.3124323.

[15] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files for Diffutils 3.10 and `patch` 2.5.4*. GNU Project, Jan. 2023, ch. `diff` Output Formats, p. 13. [Online]. Available: https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html

[16] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? An experiment with practitioners," in *Proc. Joint Meet. Eur. Softw. Eng. Conf. and ACM Symp. Found. Softw. Eng. (ESEC/FSE)*, 2017, doi:10.1145/3106237.3106255.

[17] Y. Kim. MSV. [Online]. Available: https://github.com/Suresoft-GLaDOS/MSV

[18] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *J. ACM*, vol. 29, no. 1, pp. 68–95, Jan. 1982, doi:10.1145/322290.322295.

[19] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, doi:10.1002/j.1538-7305.1948.tb01338.x.

[20] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proc. ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 609–620, doi:10.1109/ICSE.2017.62.

[21] E. Dolstra and A. Hemel, "Purely functional system configuration management," in *USENIX workshop on Hot top. oper. syst. (HotOS)*, May 2007. [Online]. Available: https://www.usenix.org/conference/hotos-xi/purely-functional-system-configuration-management

[22] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2016, pp. 298–312, doi:10.1145/2837614.2837617.

[23] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Softw. Eng. (TSE)*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015, doi:10.1109/TSE.2015.2454513.

[24] D. R. MacIver, Z. Hatfield-Dodds, and many other contributors, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019, doi:10.21105/joss.01891.

[25] S. Cha and H. Oh, "Making symbolic execution promising by learning aggressive state-pruning strategy," in *Proc. Joint Meet. Eur. Softw. Eng. Conf. and ACM Symp. Found. Softw. Eng. (ESEC/FSE)*, 2020, pp. 147–158, doi:10.1145/3368089.3409755.

[26] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "HyDiff: Hybrid differential software analysis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng. (ICSE)*, 2020, pp. 1273–1285, doi:10.1145/3377811.3380363.

[27] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. Int. Conf. Static Analysis (SAS)*, 2011, pp. 95–111, doi:10.5555/2041552.2041563.

[28] D. Qi, D. H. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," in *Proc. ACM SIGSOFT Symp. and Eur. Conf. Found. Softw. Eng.*, 2011, pp. 278–288, doi:10.1145/2025113.2025152.

[29] S. Cha, S. Hong, J. Bak, J. Kim, J. Lee, and H. Oh, "Enhancing dynamic symbolic execution by automatically learning search heuristics," *IEEE Trans. Softw. Eng. (TSE)*, vol. 48, no. 9, pp. 3640–3663, 2022, doi:10.1109/TSE.2021.3101870.

[30] IEEE and The Open Group, *The Open Group Base Specifications*, 2018, ch. atoi. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/functions/atoi.html

[31] K. Takemoto and S. Takada, "Applying symbolic execution to semantic code clone detection," in *Int. Conf. Softw. Eng. and Knowl. Eng. (SEKE)*. KSI Research Inc., 2023, pp. 118–122, doi:10.18293/SEKE2023-070.

[32] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *Runtime Verification*. Springer Berlin Heidelberg, 2009, pp. 76–92, doi:10.1007/978-3-642-04694-0_6.

[33] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012, doi:10.1145/2345156.2254088.

[34] E. Rakadjiev, T. Shimosawa, H. Mine, and S. Oshima, "Parallel smt solving and concurrent symbolic execution," in *IEEE Trustcom/BigDataSE/ISPA*, vol. 3, 2015, pp. 17–26, doi:10.1109/Trustcom.2015.608.

Acknowledgements

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A2C1009819) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001).

The experiments would not be possible without YoungJae Kim's work on meta program format and the automatic patch generator MSV, and integration efforts by Trương Huy Trung.

I would also like to thank my academic advisor, Dr Jooyong Yi, and other lab mates for being nothing but supportive both in life and at work, even though I could be antisocial at times. Last but not least, I am grateful of my family and friends back in Việt Nam, and the new friends I met here in UNIST, especially in the shared kitchen, for their consistent support and encouragements.