KOMPARE: Symbolic Execution for Assured Patching

by

Musa Haydar

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer Science and Engineering)
in the University of Michigan
2023

Master's Thesis Committee:

Professor Baris Kasikci, Chair
Professor Kevin Leach
Professor Manos Kapritsos

Musa Haydar
musah@umich.edu

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my thesis advisor, Professor Baris Kasikci, to whom I can attribute considerably my academic and professional development for the opportunities and support he has given me these past two years.

I would also like to thank my mentors: Marina Minkin, who provided a great deal of support and guidance throughout the process of this thesis, and Ian Neal, who contributed invaluable feedback and technical expertise to this project. I would also like to thank this master's thesis committee for their time and feedback.

Finally, I'd like to thank my brothers, Ibrahim and Eissa, who constantly inspire me, and my parents, family, and friends for all their unwavering encouragement.

TABLE OF CONTENTS

ABSTRACT

Patches are often applied to systems in deployment to quickly resolve bugs and remove vulnerabilities. However, patch verification is a challenge: ideally, a patch should fix the intended issue without introducing new bugs or changing the program's behavior in unintended ways. Symbolic execution is a testing technique which can be used to verify these proprieties of the patch by considering the behavior of the program along the many possible execution paths.

This thesis presents KOMPARE, a tool which leverages symbolic execution to compare two versions of a program for patch verification. We reason that two programs which behave the same for a given input will either produce identical outputs or fail on the same errors. Thus, to verify a patch, KOMPARE performs symbolic execution on a patched program to generate program inputs. Then, concrete executions are performed using the generated inputs on both the patched and original versions of the program. During these concrete executions, KOMPARE records and compares the externally visible outputs of the program to determine if the behaviors of the two versions match. We reason that the code paths we are interested in analyzing are those which execute or lead to the execution of code modified by the patch. Thus, to increase the efficiency of KOMPARE's analysis, we present Patch-Directed Symbolic Execution, a technique used by KOMPARE to drive symbolic execution towards modified code. On initialization, KOMPARE performs a static analysis to determine the differences between the two versions, computing priorities for execution states which are used when selecting states during symbolic execution.

We evaluate KOMPARE on an example of a custom patch to the COREUTILS program Echo to demonstrate its comparison analysis and the effectiveness of Patch-Directed Symbolic Execution. We then use KOMPARE to analyze some benchmarks from the DARPA Assured Micropatching program, containing code for the embedded systems on autonomous vehicles. We find that KOMPARE is successfully able to verify that these patches both correct the vulnerabilities while preserving the original functionality of the benchmarks.

CHAPTER 1

Introduction

When software bugs and vulnerabilities are found in critical systems, fixing them quickly can be crucial. Often, this involves the application of a software patch—a set of modifications to a program—to a system in deployment. Ideally, a correct patch should maintain the following properties: first, the patch should fix the intended bug or vulnerability or otherwise cause the intended change to the program's functionality. Secondly, the patch should not introduce any new instruction-level bugs or failures in the program. Thirdly, the patch should not alter the program's behavior in unintended ways, causing regression bugs or other program behavior bugs. However, verifying that a patch is correct on all three of these criteria is a challenge. It's possible, for instance, to test a program under concrete test cases which achieve high or total branch coverage without exploring the new behavior introduced by the patch [8].

Symbolic execution [1] is a software analysis technique which may be leveraged to verify the correctness of a patch in these regards. Under symbolic execution, a program is executed using symbolic variable values in the place of concrete values. During the program's execution, as each path is explored, constraints are collected which express the possible values that each symbolic value may hold. Then, when a potentially failing instruction is reached or an execution path terminates, these constraints may be solved to determine a concrete input that would cause the particular path to be executed or failure to be produced. Symbolic execution thus enables the exploration and analysis of all paths in a program and automates the generation of inputs which do so.

Many tools have been created which leverage symbolic execution to verify that patches do not introduce failures to the program or to expand code coverage of an existing test suite to include the added code [6, 8, 9, 12, 13], which is the first of our desired properties. Some of these techniques have been further applied to cross-check functions or differing versions of a program [5, 11]. However, this work is limited, either to the function level or in requiring manual comparison by the developer. Furthermore, there are some challenges in symbolic execution which need to be addressed. For instance, path explosion occurs when the number of paths to explore grows exponentially with the number branches in the program. Prior work addresses this issue in various problem domains by directed symbolic execution towards code that is more likely to give useful

results [6, 7, 9, 10, 14]. Similarly, in verifying that the patch has not changed the program behavior, we are especially interested in exploring regions of code which have been modified by the patch.

Symbolic execution enables the analysis of each path through a program which can then be used to reason about the patch's impact on the possible program behaviors. This thesis presents KOMPARE, a tool built upon the KLEE [2] symbolic execution engine which leverages symbolic execution for automated program comparison. KOMPARE first symbolically executes a program to generate concrete inputs which exercise the program's behaviors or cause it to fail. Then, KOMPARE performs concrete executions on both versions of the program (the patched/modified version and the orignal version), recording all externally visible outputs to determine if the behavior of the program has been altered by the patch.

By providing KOMPARE with an output comparison function which accounts for the expected differences caused by a patch or which verifies the correctness of the output, KOMPARE is able to verify that the patch causes the correct change in the program's behavior (the first criteria). Additionally, in exploring all paths through the program, KOMPARE is able to find new bugs and program failures introduced by the patch (the second criteria). Finally, the key contribution of KOMPARE is that, by comparing the outputs of the program along each execution path discovered under symbolic execution, KOMPARE is able to verify that the patch does not alter the program's behavior in any unintended ways (the third criteria).

This thesis additionally presents Patch-Directed Symbolic Execution, a technique to prioritize the execution and analysis of code modified by the patch. KOMPARE uses Patch-Directed Symbolic Execution to drive symbolic execution towards the desired code by selecting states which have executed or will lead to the execution of modified code. We find that, for patches which cause differences to occur in the outputs of a program, Patch-Directed Symbolic Execution allows those differences to be discovered by KOMPARE sooner, as compared with KLEE's default exploration strategy.

The remainder of this thesis is organized as follows: in §2, we introduce symbolic execution and its associated challenges, as well as examine previous work which has applied symbolic execution towards patch verification. In §3, we present the design of KOMPARE's program comparison analysis. We also present the Patch-Directed Symbolic Execution algorithm, which performs a static analysis of program difference and then assigns priorities to code which has been modified by the patch. In §4, we discuss the implementation of KOMPARE as a new driver for the KLEE symbolic execution engine and the implementation of Patch-Directed Symbolic Execution as an extension to KLEE. In §5, we evaluate KOMPARE using a custom example patch to the COREUTILS program Echo. Then, we use KOMPARE to analyze some benchmarks from the DARPA Assured Micropatching program. Finally, in §6, we discuss some of the limitations, possible improvements, and future work for KOMPARE, and in §7 we conclude.

CHAPTER 2

Background

This chapter provides a brief introduction to symbolic execution, followed by a discussion of prior work which applies symbolic execution towards patch verification. We find that many of the tools which use symbolic execution for patch verification either verify that no new failures are caused by the patch or simply use symbolic execution to generate test cases which exercise the modified code. This motivates the design of KOMPARE, a new tool which additionally verifies that the patch does not unexpectedly alter the program's behavior.

2.1 Symbolic Execution

Symbolic execution is a technique by which we execute a program using symbolic values as inputs to the program in the place of concrete ones. These symbolic values express the possible values the variables may hold throughout the program's execution. During symbolic execution, a *symbolic store* maintains a mapping from variables to their respective symbolic expressions. As the program is executed, these symbolic expressions are updated per the program's instructions which operate on them. Then, at each branching instruction, *constraints* are formulated which describe the conditions that must have been satisfied by the symbolic values along each execution path, and each path is executed in turn. Eventually, these constraints are solved to determine what concrete input would cause a particular path or be executed and to determine if any concrete input to the program may cause errors in potentially dangerous instructions or violate program assertions [1].

Consider the example program in Listing 2.1. Under symbolic execution, this function will may be executed with a symbolic value for the input x, for instance, $x = \lambda$. After executing line 2, the symbolic store will be updated to contain $z = \lambda + 10$, since the value of z is computed using a symbolic value. Then, the execution will reach the branching condition on line 3, for which we have two possible states to explore. In the first, the execution state will maintain the constraint that $z > 100$ (or $\lambda + 10 > 100$), and in the second, that $z \leq 100$. These branches are then executed independently. Both lines 4 and 6 contain potentially dangerous instructions: if the

```
1 int func (int x) {
2     int z = x + 10;
3     if (z > 100) {
4         return 2 / z;
5     } else {
6         return 10 / z;
7     }
8 }
```

Listing 2.1: A simple function in C.

value of z is zero, a division-by-zero error will occur. For the first of these, solving the constraints determines that z cannot have the value 0 along that execution path, and so this error will never occur. However, solving constraints for z collected up to and including line 6 will reveal that the value of λ being -10, or the input $x = -10$, will produce this error. In contrast, an input with the value $x = -8$ will not produce an error.

There are some challenges in classical symbolic execution, and much work has been done to alleviate these challenges generally or within specific problem domains. One such challenge is *state space explosion*, also called *path explosion*. Under symbolic execution, each branch in the program may fork the execution, such that there are two or more possible paths with their own path constraints to consider, and the number of paths to be explored may grow exponentially [2]. Some techniques mitigate this challenge by directing symbolic execution towards desirable paths or by pruning undesirable ones [1]. For example, AGAMOTTO [7] is a tool which uses symbolic execution to search for bugs in persistent memory systems. To do so more effectively than classical symbolic execution, AGAMOTTO directs symbolic execution towards locations in the code which may access persistent memory and may therefore contain this kind of bug.

A technique which has been used to effectively test large, real world systems which are more likely to incur path explosion, is *under-constrained symbolic execution*. By analyzing individual functions under symbolic execution instead of entire programs, UC-KLEE [11] is able to effectively verify patches to functions in libraries such as OpenSSL and in the Linux kernel. However, missing preconditions on entry to these functions causes UC-KLEE to report false positives when analyzing a function.

Another major obstacle in symbolic execution is constraint solving: the satisfiability problem is known to be NP-Hard. Although much work has gone into improving the state-of-the-art SMT solvers used by symbolic execution engines such as KLEE [2], complex constraints, such as those involving non-linear arithmetic, continue to pose a challenge [1]. When execution becomes stuck on a constraint which takes too long to solve, a technique known as *concretization*, where some portion of the symbolic input is instead made concrete, may be employed to enabled the execution to proceed. Concretizing symbolic values reduces the complexity of the constraints possibly at the

expense of soundness, as certain paths through the program will not be explored.

The technique of guiding the symbolic execution to desired paths by substituting symbolic values for concrete ones is known as *dynamic symbolic execution*. The DART [4] engine, for instance, executes a program both concretely (under randomly generated test inputs) and symbolically, using the values discovered during concrete execution to direct symbolic execution to explore all paths. Similarly, Execution Reconstruction (ER) [14] reproduces system failures under symbolic execution by iteratively concretizing values when execution stalls due to the symbolic constraints becoming too complex for the solver. In each iteration, ER identifies a candidate constraint to be concretized, traces an execution of the program to record concrete values, and repeats the analysis until the constraints no longer stall execution and a test case is generated.

Another challenge in symbolic execution involves code which interacts with the environment, such as through operating system calls, file input/output, or network code. A straightfoward approach which some symbolic execution techniques take is to execute the external system calls directly, using the concrete results during the subsequent symbolic execution [1]. Ideally, these interactions should be considered symbolically as well, so as to consider all possible resulting values. KLEE addresses this issue by modelling these interactions in C code that "understands the semantics of the desired action well enough to generate the required constraints" [2].

2.2 Symbolic Execution for Patch Verification

When verifying the correctness of a patch, we seek to demonstrate that the patch makes the intended change to the program's behavior, that the patch does not introduce any new bugs or failures, and that the patch does not alter the program's behavior in any unintended ways. Some work has been done in the domain of applying symbolic execution towards patch verification, much of which in seeking to verify that no new code bugs are introduced by the patch by generating test cases which explore the modified code.

One work which applies symbolic execution towards patch verification is Directed Incremental Symbolic Execution (DiSE) [9, 13]. DiSE extends classical symbolic execution with insights about software patching to increase the efficiency of analyzing a program. The authors present a static analysis which characterizes the differences between two versions of a program by discovering the set of nodes in the control flow graph (CFG) which have been impacted by the change. Specifically, they consider all CFG nodes which have been added or removed by the change as impacted, and nodes which are control flow or data dependant on an impacted node are marked as impacted as well. DiSE then uses the results of this analysis to generate path constraints for the impacted code, leveraging them so as to only explore paths in the program that are impacted by the change. Finally, in considering these generated path constraints, DiSE is also able to prune execution paths which

differ only in their sequence of unimpacted nodes, thereby reducing the total cost of the symbolic analysis. By directing symbolic execution towards impacted code, DiSE is able to effectively generate test cases which exercises the modified code, which can then be added to a regression suite, thus verifying that the patch does not introduce any new bugs.

Another tool, called KATCH [6] similarly applies symbolic execution with the goal of generating test cases which increase the coverage to include patched code in an existing test suite. Like DiSE, KATCH directs execution towards "target" lines of code, targeting reachable code not already covered by the system's existing regression test suite. They observe that only one target is needed per basic block (containers of sequential instructions), since all the code in a basic block must execute together. Then, beginning execution with a seed input, KATCH selects paths to explore in the program by estimating which paths have the shortest distance to the target code. The novelty in KATCH's approach is in *informed path regeneration* and *definition switching*. If a path towards the target becomes infeasible during symbolic execution, and it depended on a symbolic value, then KATCH backtracks to the branch which added the constraint and continues exploration with the corrected branch conditions. If the path became infeasible due to a concrete value, KATCH attempts to find an alternate definition for the offending variable. Thus, KATCH is able to effectively and automatically increase the coverage of a regression suite to include code modified by a patch.

KPSec [10] is another tool which takes a similar approach in the domain of patch security testing. Many patches introduce vulnerabilities in code systems due to, for instance, potential memory leaks, dangling pointers, or use of initialized memory. To verify a patch for these kinds of errors, KPSec generates execution paths through the program via a data-flow analysis, using these results to drive symbolic execution towards potential "security points" in the program.

The tools and techniques discussed so far improve the efficiency of their symbolic program analyses by targeting execution towards patch code and thereby demonstrating that the patch does not introduce instruction-level or program assertion failures. While this is an important aspect towards patch verification, we also seek to determine that a patch does not alter program behavior in unintended ways, a problem which requires analysis beyond the code modified by the patch itself. To achieve this, we might consider how two versions of a program can be cross-checked, such that, on each input, the versions produce the same behavior.

Under-constrained symbolic execution is a technique which has been shown to effectively verify patches in real-world code. By operating at a function level instead of a program level, UC-KLEE [11] is not only able to find new crashes introduced by a patch within each function, but to reason about function equivalence. In particular, they perform *cross-checking* of two differing implementations of a function, where the two versions share the same interface and output formats. Furthermore, the authors reason about *error equivalence*: they find that real-world programs tend to crash on the same illegal inputs between the versions [12]. This is useful for ensuring

identical behavior between different implementations of functions which share the same interface or between an optimized program against a reference implementation. While limiting their cross-checking to the function level enables UC-KLEE to effectively analyze large, real-world libraries and systems, their technique cannot determine if the intended changes to a given function's output affects the behavior of a program which uses this function in unintended ways. Furthermore, UC-KLEE produces false-positive bug reports due to missing input preconditions when evaluating functions independently [11].

Another symbolic execution technique for patch verification is *shadow symbolic execution*, implemented in the tool SHADOW [5, 8]. SHADOW takes the two versions of the program and unifies them into a single binary, with annotations inserted in the code locations where the versions differ. Then, SHADOW symbolically executes this unified binary, using a combination of symbolic and concrete inputs. Until the annotations are reached, the path constraints and symbolic stores for both versions of the program are identical, so they are maintained once for both versions. Then, upon reaching the annotated branches, the execution forks in what SHADOW calls "four-way forking," which explores both the divergent path conditions as well as the divergent code between the two program versions. This enables shadow to effectively consider all paths up to the modified code only once, and then thoroughly explore all the paths which differ between the versions. Furthermore, by unifying the symbolic execution, SHADOW is able to both effectively share execution state data between these divergent paths and prune irrelevant paths, thus improving the space and time efficiency of their analysis, respectively.

Once the test cases are generated by SHADOW, the tool is able to cross-check the two versions of the program to find any differences in their externally visible outputs. This cross-checking requires manual inspection by the developers to reason whether the differences in output are expected or occurrences of newly introduced bugs. Additionally, SHADOW's cross-checker requires the programs be executed natively. This requires more overhead from the user; one of the advantages of simulating a program in symbolic execution, such as in the KLEE [2] engine, is that it does not require any specific hardware where native execution might.

These techniques discussed are able to effectively generate tests which exercise the modified code and verify that no new failures are introduced by the patch. However, these techniques can not verify that the patch does not unexpectedly alter the program's behavior. Works which additionally cross-check function or program versions are limited. This inspires the design of KOMPARE: our goal is to automate the cross-checking of program behavior on all paths, using symbolic execution to generate concrete inputs which enable this. To increase the efficiency of its analysis, KOMPARE also prioritizes the execution of modified code similar to previous work. Unlike DiSE, KOMPARE completes symbolic execution on all subsequent paths after having executed modified code to verify that the paths behave as expected.

CHAPTER 3

Design of KOMPARE

Prior work has explored techniques by which symbolic execution can be used to effectively seek out new bugs introduced by a patch [6, 8, 9, 11]. While such techniques may ensure that new bugs (such as memory errors or program assertion failures) are not caused by the modified code, they do not assure that the patch has not altered the program's behavior in unintended ways. In this chapter, we discuss the design of KOMPARE, a tool which builds upon the KLEE [2] symbolic execution engine, utilizing symbolic execution to cross-check the patched and original versions of the program to verify the aforementioned properties for a patch.

The UC-KLEE authors reason about using symbolic execution to verify patches at the granularity of functions [12]. When comparing varying implementations of the same function, UC-KLEE verifies that, given identical inputs, the two routines produce identical outputs. Thus, they reason that the two functions behave the same on each input. Specifically, they ensure that both routines "write the same values to all escaping memory locations" or otherwise "terminate with the same errors." Then, to cross-check two implementations of a function, UC-KLEE considers the inputs and outputs directly, expecting that the differing versions of the function maintain the same interface and write to the same memory locations or return variables.

Analogously, when reasoning about program-level equivalence, we consider the *externally visible* outputs of a program. A change to a program's code may result in a change to the program's internal state along particular execution paths. However, if the externally visible outputs are identical, we conclude that the patch did not change the program's behavior. In particular, we inspect the externally visible system calls and seek to verify that, excepting the intended changes by the patch, both programs exhibit identical externally visible outputs or fail on the same errors along each execution path.

To accomplish this, KOMPARE provides a new driver program for KLEE which performs symbolic execution over a selected version of the program. Constraints are gathered during execution, with which KLEE generates concrete inputs that exercise each execution path in the program. KOMPARE than executes both versions of the program, providing these concrete values as inputs and recording any externally visible outputs for comparison.

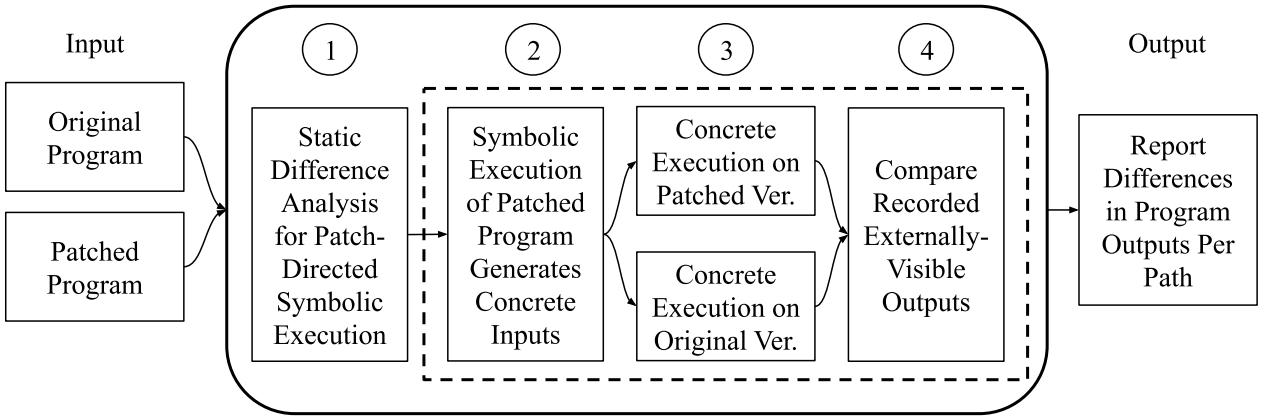

Figure 3.1: Overview of KOMPARE functionality.

Under symbolic execution, the number of potential execution paths to explore grows quickly as branches and loops are encountered, possibly resulting in path explosion, where a prohibitively large number of states are generated. We reason that behaviors of the program altered by the patch are those for which modified code is executed. Then, to mitigate this issue, we additionally present Patch-Directed Symbolic Execution, an approach for prioritizing the exploration of modified code under symbolic execution, thereby prioritizing the analysis of behaviors possibly affected by the patch. This enables KOMPARE to generate inputs which explore modified code and collect results sooner and may also prevent the symbolic execution from becoming stuck while exploring some uninteresting subset of the program's paths.

To direct symbolic execution towards modified code, we devise a static analysis which compares the patched and original versions of the program. Like KLEE, this analysis operates on programs which have been compiled down to LLVM bitcode representation. An overview of our Patch-Directed Symbolic Execution approach is illustrated in Figure 3.2. First, we perform a static analysis which compares the two versions of the program, assigning weights to basic blocks which differ between the versions (step 1). Then, these weights are used to generate priorities for instructions, which are back-propagated through the program in a similar manner to AGAMOTTO [7] (step 2). These priorities are then used by KOMPARE to direct symbolic execution towards code with the highest priority, i.e. code that will either execute or lead to the execution of modified code (step 3).

If we exhaustively explore paths under symbolic execution, we will have determined that for all inputs whether the patch has changed the behavior of the program. This may not be possible, as mentioned, due to difficulties in path explosion or constraint solving. However, the more time KOMPARE is given to run, the closer its analysis will be to complete, since KOMPARE tests paths as the respective concrete inputs are generated. Furthermore, by directing symbolic execution

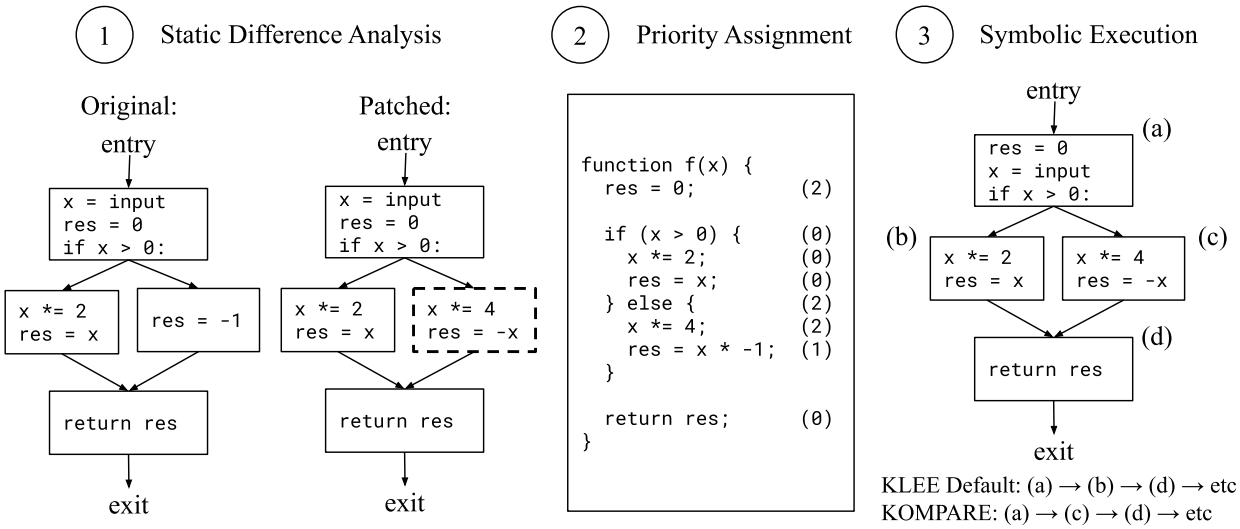

Figure 3.2: Overview of Patch-Directed Symbolic Execution. Step (1) represents the programs as control-flow graphs. In step (2), instructions are annotated with their priorities. Note that the priority assignment is given in psuedocode, whereas this process operates on LLVM code in KOMPARE. Step (3) compares KLEE's default state exploration strategy to KOMPARE's patch-directed strategy, where (c) is prioritized over (b).

towards patch-modified code, this approach is more likely to expose the differences between the versions of the program earlier in the analysis.

An overview of KOMPARE's design is provided in Figure 3.3. KOMPARE takes as input the two program versions in LLVM bitcode format and outputs a report of which concrete test cases exhibited differences in the program output. On itialization, KOMPARE performs a static analysis to determine the differences between the programs (step 1, elaborated in §3.1), which is used to direct symbolic exeuction towards the modified code. Then, KOMPARE performs it's program comparison, described in §3.2. First, KOMPARE begins symbolic execution of the patched version of the program (step 2). Whenever a test case is generated, KOMPARE begins concrete executions of this test case on both versions of the program (step 3) and compares the outputs to determine if the programs behaved the same for this input (step 4). Steps 2 through 4 occur concurrently: KOMPARE performs it's concrete executions and comparisons while symbolic execution is ongoing.

3.1 Patch-Directed Symbolic Execution

In symbolically executing the program to generate inputs for the comparison, we are primarily interested in exploring the code which has been directly modified by the patch, reasoning that the program behaviors which are affected by the patch occurs along paths which execute directly

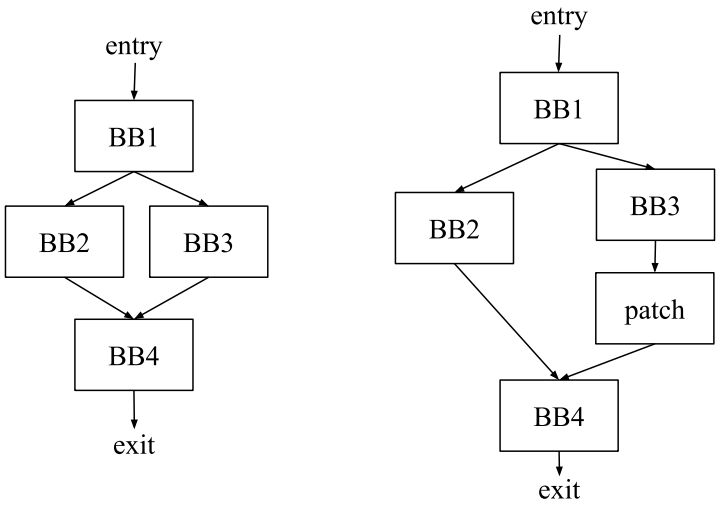

Figure 3.3: Example control-flow graphs of an original (left) and patched (right) program, where the patch adds a new basic block labelled "patch" between the block BB3 and the exit block, BB4.

modified code. The goal is then to prioritize the exploration of states which execute or lead to the execution of modified code. Memory aliasing may pose a challenge this assumption, discussed further in §6.2. We take an approach similar to DiSE's [9] static analysis to determine the differences between the programs and direct execution accordingly. A key difference is that DiSE prunes paths for which a particular sequence of impacted nodes have already been covered, whereas, with KOMPARE, we're interested in analyzing all paths subsequent to modified code.

To achieve this, we devise a static analysis which compares the patched and original versions of the program. As before, the choice of which version should be considered patched and original, respectively, is arbitrary, and in order to maintain completeness in our program comparison, we will perform this analysis on both, considering each the "patched" version in turn. This analysis first assigns a weight value to each instruction in the patched version of the program (i.e. the target version for symbolic execution) such that non-zero weight is given to the code that differs between the two versions.

In order to compare the two versions of the programs, we must consider both the contents of the basic blocks as well as the control flow of the program. Since the instructions of each basic block must execute sequentially and entirely, the analysis can assign weights at the granularity of basic blocks instead of instructions. Consider the example control flow graphs in Figure 3.3. Here, a new basic block "patch" is inserted between the basic blocks named "BB3" and "BB4," and the other basic blocks remain unchanged. The goal of the analysis is to assign non-zero weight only to the basic block "patch." If, for instance, weight is assigned to the exit block, "BB4," KOMPARE may prioritize the path through BB2, delaying its analysis of the code introduced by the patch.

```
%4 = alloca i32, align 4
store i32 100, i32* %4, align 4, !dbg !18
...
%7 = alloca i32, align 4
store i32 100, i32* %7, align 4, !dbg !22
```

Listing 3.1: Two equivalent LLVM instructions which store the constant value 100 to an allocated stack variable.

At a high level, the analysis operates as follows: for each function in the patched version of the program, it attempts to find a function of the same name in the original program. If no such function exists, the entire function is marked as differing. Otherwise, each basic block in the patched version is compared against each basic block in the original. First, they are compared for equivalence while ignoring control flow (i.e. by skipping branching instructions). In this step, two basic blocks are considered equivalent if all their non-branching instructions are equivalent and occur in the same order. This necessitates a notion of instruction equivalence, which we describe in §3.1.1. Next, the equivalent basic blocks are compared for control flow differences. Once weights are assigned, we compute priorities for each instruction by back-propagating weights through the program, such that instructions which execute or lead to the execution of patch code are given highest priority.

3.1.1 Instruction Equivalence

First, we define a notion of instruction equivalence between the two versions of the program. We consider two LLVM instructions to be equivalent if they perform the same operations on equivalent operands. Constant operands, such as string literals, can be compared directly. Additionally, we consider non-constant operands to be equivalent between the two programs if they are output by two equivalent instructions. LLVM's single-static assignment form guarantees that each variable is only assigned once, so that for each operand, we need only consider the instruction which assigns it a value. Furthermore, metadata (such as debug locations) can be ignored as they do not affect the outcome of the instruction.

Consider, for example, the store instructions in Listing 3.1. Both of these instructions store the constant value 100 the stack variable of their operand, each of which has allocated to it a 32-bit integer by the previous, equivalent `alloca` instructions. Although the operand variables have different labels and the instructions have different metadata, we can still consider these store instructions equivalent.

In implementing KOMPARE, we define a function which compares two instructions for equivalence. First, it checks that the two instructions are identical, ignoring metadata and operand labels.

Then, it recursively checks that the instructions which define each non-constant operand are also equivalent. To prevent repeating recursive queries, the function memoizes, for any two instructions, its previous decision on their equivalence.

3.1.2 Assigning Weights and Priorities

Given a notion of equivalence between two instructions, the analysis iterates through each function of the patched program which appears in both versions of the program. Each basic block in a given function is then assigned a weight of 0 if it has an equivalent basic block in the original function, and 1 otherwise. This analysis occurs in two steps: first, the analysis compares each instruction in the basic block up to and excluding the terminating instruction, thus ignoring the control flow of the function's basic blocks. Each basic block in the patched version is compared to each basic block in the original version. The comparison operates by iterating through the instructions in each basic block in order. If any pair of these instructions are not equivalent, or one basic block contains fewer instructions than the other, the analysis has determined that the basic blocks are not equivalent and continues to the next basic blocks to be compared.

Since this step ignores control flow, it is possible that the basic block in the patched program is equivalent to multiple basic blocks in the original version. This occurs more often for small basic blocks performing simple tasks, such as those which load a return value and branch to the function's exit block. Therefore, for each basic block in the patched function, we maintain a set of equivalent basic blocks from the original function.

Next, for each basic block in the patched version, we compare the control flow for all equivalent basic blocks. This check is simple: for each pair of equivalent basic blocks, the analysis verifies that each of the successor blocks are equivalent along each of the branch conditions. If any of the successors are not equivalent, the basic block weights are updated to mark them as differing. It would be apt to mark the successor with no equivalent basic block as differing as opposed to the predecessor block, since the code subsequent to the modified branch is the target for the symbolic execution. However, it is sufficient (and much simpler in implementation) to mark the parent basic block as differing, such that the symbolic execution will prioritize reaching the parent basic block and then explore all subsequent paths according to their respective weights.

At this point, each differing basic block is assigned a weight value of 1, and the blocks which have an equivalent in the original program are assigned a weight of 0. Once weights are computed for each basic block, priorities are assigned to each instruction in the program. Like AGAMOTTO [7], we perform back-propagation using the computed weights so that each instruction is given a priority equal to the number of reachable modified instructions. This is done by iterating through the program's control flow from the end points to the entry point. The priority assigned to each

instruction is equal to the weight of the basic block which contains it plus the maximum priority of all reachable instructions from that point. These priorities can then be used by KLEE's simulator to prioritize executing modified code by selecting the state with the highest priority when determining which state to execute. In §4.2, we discuss the extension we implement in KLEE to accomplish this.

3.2 KOMPARE's Comparison Analysis

Steps 2 through 4 of KOMPARE's functionality, outlined in Figure 3.3, perform KOMPARE's program comparison analysis, which we describe in this section. Before this analysis is performed, KOMPARE performs the static analysis for determining the difference between the versions, described above, and the symbolic execution performed in Step 2 is patch-directed. However, this comparison analysis does not depend on Patch-Directed Symbolic Execution and can be performed without it.

The approach we take in KOMPARE to compare the two versions of the program is as follows: first, we begin symbolic execution of the patched version of the program. As the symbolic execution reaches the end of a path or a potential error, the constraints are solved to generate a concrete input which would exercise that particular execution path or expose that error. Then, we perform concrete execution of both versions of the program for each generated program input, during which we record all externally visible outputs for comparison. After all symbolically-explored paths have been compared, KOMPARE reports which test cases exposed differences between the versions of the program.

3.2.1 Removed Program Paths

As a consequence of only symbolically executing one version of the program, in particular the version of the program with the patch applied, this approach can only consider the execution paths added or changed by the patch. It cannot consider those paths removed by the patch. Consequently, any differing program behaviors caused by a removed execution path may be missed, unless the input generated by the constraints for the patched version happens to expose this difference.

Consider, for example, the functions in listings 3.2 and 3.3. Here, the patch to the function `func` has removed the check for the input `x == 10`. If we explore the patched version of `func` symbolically, we will find that there is only one possible execution path. Then, for any concrete input we generate such that `x` is not equal to 10, we will conclude that both functions have the same behavior (and indeed, for that particular input, they would). If the input we generate happens to be 10, we will observe this difference. However, if we consider the function with the branch to be

```c
int func (int x) {
    if (x == 10) {
        return 1;
    }
    return 0;
}
```

Listing 3.2: A simple example function in C with a branch.

```c
int func (int x) {

    // if branch removed by patch

    return 0;
}
```

Listing 3.3: The same example function with the branch removed, as though by a patch.

the patched version (i.e. that the patch added the path instead of removed it), then we will discover both paths during symbolic execution and determine that the functions behave differently only on the input `x == 10`.

Clearly, symbolical execution of only the patched version of the program introduces a challenge to the soundness of our analysis. Thus, a straightforward approach we take to enable the exploration of the removed program behaviors is to perform this technique by symbolically executing both the original and patched versions of the program in turn. In §6.1, we discuss additional techniques that may be applied to improve the performance of this analysis, as this will result in the analysis of many paths which have been analyzed previously.

3.2.2 Concrete Executions and Output Comparison

Throughout the symbolic execution, KOMPARE collects the generated inputs executes both versions of the program with them, during which outputs are collected for comparison. The specific outputs to be compared depends on the program being analyzed. For instance, we may be interested in inspecting data sent along particular buses for an embedded system. As mentioned, the outputs we consider in KOMPARE are those which occur in output system calls made by the program (such as `write`, `printf`, `fputs`, etc.). In §4.1, we discuss how KOMPARE records the outputs sent to these system calls. These outputs are compared to determine if the two versions of the programs exhibit the same behavior or fail on the same error for that particular input.

The output comparison performed by KOMPARE is defined in a function, elaborated with respect to KOMPARE's implementation described in §4.1. The default comparison strategy we take is to check that the outputs are entirely identical. This comparison is sufficient for patches which we expect not to change the program's functionality along all or most paths. However, patches may affect a program in various ways and the outputs of certain programs may require closer inspection to compare. Therefore, the output comparison function used by KOMPARE should be specified with respect to the program and patch to be analyzed. Furthermore, this comparison function may be used to verify that the output is itself correct for the patched version of the program. In §6.1.1, we discuss some such cases and how the output comparison may account for them.

CHAPTER 4

Implementation of KOMPARE

In this chapter, we discuss the implementation of KOMPARE, our extension to the KLEE [2] symbolic execution engine described in §3. KOMPARE implements a new driver program which creates instances of KLEE to symbolically execute the target and collects the results to perform the output comparison. Additionally, we implement the patch-direct symbolic execution algorithm in KLEE, which performs a static analysis to determine the difference between the two versions and then drives symbolic execution towards the modified code.

4.1 KOMPARE Driver

We implement a new driver for the KLEE engine which we call KOMPARE. Since this is built upon KLEE, which symbolically executes a program that has been compiled to LLVM bitcode, we require LLVM bitcode representations of the target program. KOMPARE takes as input two programs: the original program and the patched program. The patched version is the one which will be explored symbolically under KLEE, an instance of which is spawned by the KOMPARE driver. As mentioned in §3.2.1, in order to ensure the soundness of KOMPARE's analysis and explore all paths removed by the patch, we run the KOMPARE driver twice, considering each version of the program to be the "patched" version in turn.

As the instance of KLEE which is symbolically executing the patched version of the program reaches the end of an execution path or a potentially failing instruction, it outputs test cases which would exercise the path or produce the failure under a concrete execution. These test cases are collected by the KOMPARE driver, and for each test case, KOMPARE spawns two additional instances of KLEE which replay the test case on both the patched and original versions of the program, collecting any externally-visible outputs (particularly system calls which write data) for comparison. Simulating the concrete executions using KLEE instead of comparing the outputs of native program executions makes analyzing two versions of the program convenient, as the versions only need be compiled to LLVM bitcode for the entire analysis to take place. Furthermore, it also simplifies the

process of capturing the externally visible outputs for comparison.

To capture the externally visible outputs, we implement wrappers for the system calls of interest, including system calls such as `write`, `fwrite`, `printf`, `fputs`, and so on. These wrapper functions are implemented within KLEE's symbolic POSIX runtime, and have the string `kcmp_` prepended to their name (as in `kcmp_write`). Then, as an additional step when linking the target program in KLEE, the symbols for these system call functions have the string `kcmp_` prepended to them, such that, during execution, the wrapper functions are called instead. When called, these wrapper functions append the output they receive to a file before proceeding as normal.

Once both versions have completed execution for the given input and their outputs are collected, KOMPARE executes a function to compare the results and determine if they match. By default, this function compares both output files line-by-line, reporting a mismatch if the files are not identical. This function can easily be replaced to account for expected differences in output formats or to support smarter output comparisons. For example, in §5.2.1, we evaluate a patch which replaces one encryption function with another, providing an output comparison function which accounts for this difference. Finally, KOMPARE produces a report which lists for which generated test cases the outputs of the two programs match or differ.

In order to evaluate a patch with KOMPARE, the program must have its two versions—the version with the patch applied and the version without—compiled to LLVM bitcode format, with both versions provided as input to KOMPARE. Then, wrapper functions should be added to the KOMPARE POSIX runtime for all functions whose outputs we are interested in capturing and comparing. If we expect the target to have differences in the output format, a smarter comparison function may also be provided to the KOMPARE driver such that only paths whose outputs differ in unexpected ways will be reported.

4.2 Patch-Directed Symbolic Execution

To increase the efficiency of this comparison analysis, as discussed in §3.1, we implement an additional extension to the KLEE engine which performs a static analysis to determine the locations of patch-modified code and drive symbolic execution towards those locations. KLEE provides an interface for a "searcher" class, which determines the order in which KLEE will execute possible states. The implementation of Patch-Directed Symbolic Execution is in two components: a class which performs the program analysis and a new searcher in KLEE which uses the results of the analysis which determine which states to execute. We call this new searcher class the Patch-Priority Searcher.

The static analysis is performed upon initialization of the Patch-Priority Searcher (and thus only once on KLEE's initialization). It iterates through the control flow of the symbolic execution target

program, computing weights and priorities for instructions using the algorithm described in §3.1.2. This class then maintains two data structures which are used by the searcher class: a mapping from instruction pointers to priorities and a set of instruction pointers which have been identified as modified from the original version of the program.

A searcher class in KLEE maintains a data structure of current states, and at each iteration of the program's execution, KLEE's interpreter queries the searcher for the next state to execute. Each state that is added or updated is assigned a weight by the Patch-Priority Searcher based on the priority of the instruction it will execute next, computed by the initial program difference analysis. Additionally, whenever an execution state executes some modified code for the first time, it is marked as having done so. The Patch-Priority Searcher than maintains a priority queue of candidate states for execution. States which have already executed modified code are prioritized over states which have not yet, and ties between state priorities are broken by prioritizing older states (i.e. states which have executed more instructions). Since priority is assigned as the number of patch-modified instructions reachable from a given instruction, paths which execute a greater number of modified instructions are prioritized.

Effectively, when current no state has reached modified code, the searcher will prioritize following a patch which executes modified code as quickly as possible. However, once modified code is reached, the searcher prioritizes exploring all possible subsequent execution paths, reaching the end of each path as quickly as possible so the comparison analysis has test cases to consider. This state prioritization scheme effectively causes KOMPARE to compare the outputs on all paths which follow the execution of some patch code, thus demonstrating that the patch does not modify the program behavior on any subsequent paths.

CHAPTER 5

Evaluation

To demonstrate the effectiveness of KOMPARE, we first take the COREUTILS program Echo and create an example of a patch which adds a new command line option. This contrived example, discussed in §5.1, demonstrates the effectiveness of KOMPARE's analysis in determining the inputs which cause the outputs of the original and patched versions to differ, as well as demonstrating the effectiveness of patch-directed symbolic execution. We find that using patch-directed symbolic execution enables KOMPARE to explore the modified code sooner thereby sooner exposing the differences between the versions.

Then, we use KOMPARE to analyze some benchmarks from DARPA's Assured Micropatching (AMP) program. These benchmarks come from embedded system code for autonomous vehicles which receives messages on its control area network (CAN) bus, updates the system's state, and logs the messages. We consider two kind of patches in our evaluation: patches to the system's logging functionality and patches which correct vulnerabilities in the transport protocol. For the former, we are interested in demonstrating that the outputs on all paths match besides the expected change. For the latter, we want to demonstrate that the vulnerability no longer occurs and that the outputs match on all other paths.

Before these benchmarks may be evaluated in KLEE, they must be compiled down to LLVM bitcode. KOMPARE makes the analysis of these particular benchmarks convenient by simulating the concrete executions in KLEE as well, since compiling and running these embedded system benchmarks natively requires more effort and specific hardware.

We first discuss some of the changes to the benchmarks that must be made to enable their symbolic execution. Then, we provide output comparison functions which enable our analysis of the outputs for these benchmark scenarios. For the logging functionality patches, we find that we are able to successfully verify the patch on all paths. For the transport protocol vulnerability patches, we are able to verify that the execution paths for which these vulnerabilities occur in the original program no longer occur in the patched version (that is, the outputs differ for those paths as expected) and that the patch does not cause any unintended changes to the program behavior (that is, the outputs match on all remaining paths).

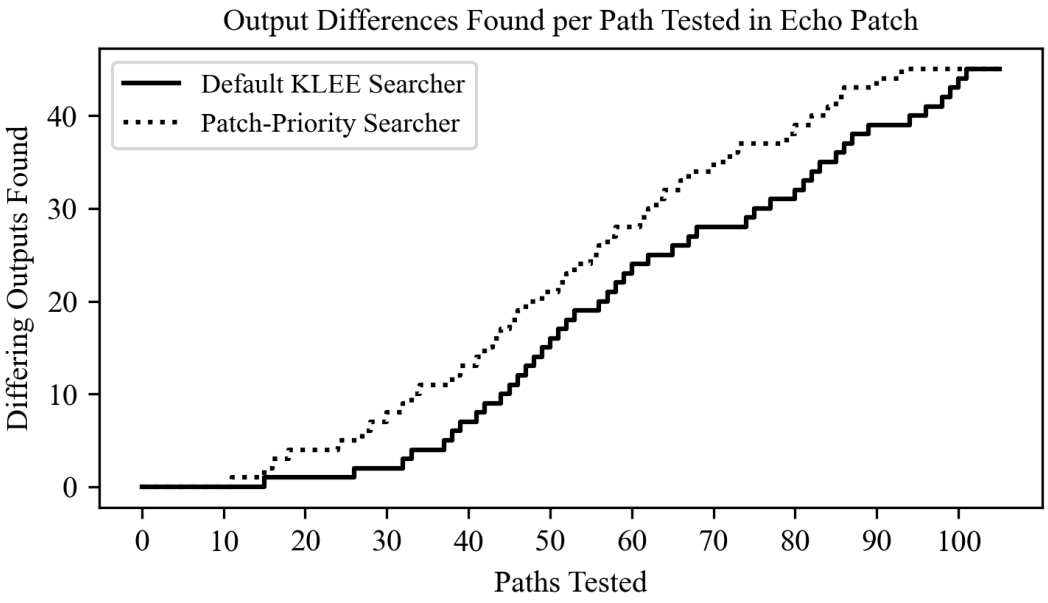

Figure 5.1: A graph comparing the number of paths with differing outputs found per the number of paths tested between KLEE's default (random-path interleaved) searcher and KOMPARE's Patch-Priority Searcher.

5.1 Contrived Example

To demonstrate the capabilities of KOMPARE, we'll consider the the contrived example of a custom patch to the COREUTILS program Echo. We take Echo as an example since it is a simple program with a simple output format, which can demonstrate the effectiveness of KOMPARE's comparison analysis as well as the impact of patch-directed symbolic execution. The patch we create adds a command line option which capitalizes all lowercase letters in the output when enabled. For instance, before the patch is applied, the command `echo -c Hello` outputs `-c Hello`. With the patch applied, this same input will output `HELLO`. A selection of the code added by this patch is presented in Listing 5.2. Since this patch only adds paths to the program and does not remove any existing code paths, we only need to symbolically execute the patched version of the program for a sound analysis of the differences between the two versions.

We analyze this patch in KOMPARE using a symbolic input of four characters, expecting the input `-c a` (for any lowercase character `a`) to expose the difference in outputs between these two versions of the program. During symbolical execution, KOMPARE explores 105 paths through the program, and of them discovers 45 concrete inputs with differing outputs. KOMPARE takes 3 minutes and 10 seconds to run and compare all paths. These experiments were performed on a server equipped with a Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz and 256 GB of DRAM. The time it takes to compare all paths is limited by the execution time of the program on the

```
int main (int argc, char **argv) {

  [...]

  while (argc > 0) {









  fputs (argv[0], stdout);

  argc--;
  argv++;
}
```

Listing 5.1: Selections from the unmodified source code of Coreutils Echo program.

```
int main (int argc, char **argv) {
  bool capitals = false;
  [...]

  while (argc > 0) {
  if (capitals) {
    const char *s = argv[0];
    unsigned char c;
    while((c = *s++)) {
      if (c >= 'a' && c <= 'z') {
        c -= 32;
      }
      putchar (c);
    }
  } else {
    fputs (argv[0], stdout);
  }
  argc--;
  argv++;
}
```

Listing 5.2: The patch applied to Listing 5.1, which capitalizes letters in input. Not shown is the patch code which checks for the new command line option, setting "capitals" to true.

concrete input plus the time overhead of simulating the program in KLEE; it takes only a few seconds to simulate Echo on each concrete input.

In its analysis, KOMPARE exposed an unexpected behavior as a consequence of the patch: any 4-character input which contains some combination of possible command line options including the `-c` option, such as `-ce` or `-cec` will differ in output between the two versions. In the original version, the output will simply echo the input, and in the patched version, Echo will output nothing on such inputs. This is because when the Echo program receives an invalid option, it will ignore all the options and simply echo the input as any other string. With the addition of the `-c` option, these previously invalid option strings are now accepted by Echo and not included in the output string. As a consequence of this, we find a rather high proportion of paths differing (45 paths out of the 105 explored) as KLEE explores the various combinations of command line options provided to Echo.

Enabling patch-directed symbolic execution during this analysis adds an additional 2 minutes and 30 seconds of initialization time during which the static analysis is performed to find the differences between the programs. As shown in Figure 5.1, using patch-directed symbolic execution with the Patch-Priority Searcher in KLEE results in finding the the differences in the program sooner than KLEE's default random-path interleaved searcher.

Notably, it seems that the Patch-Priority searcher finds some differing paths sooner (after around 15 explored paths) and then maintains some nearly-constant delta between the two searchers. This is a consequence of the Patch-Priority Searcher's prioritization of paths which have already executed modified code. Effectively, the searcher prioritizes reaching modified code as quickly as possible and then explores all subsequent paths, therein prioritizing paths which contain some additional modified code themselves, exhausting all subsequent paths before backtracking to explore previously encountered paths.

5.2 Assured Micropatching Benchmarks

We use KOMPARE to analyze some benchmarks from DARPA's Assured Micropatching (AMP) program. These benchmarks are for an embedded system for autonomous vehicles, which receives messages on its CAN bus in accordance with the J1939-21 transport protocol. After receiving a message, the system will update its state accordingly, and then encrypt and log the messages. In addition to patching out a particular vulnerability in the system, each of the benchmarks presents some additional challenges towards symbolically executing and analyzing the program.

To run these benchmarks with symbolic input, we create a wrapper program which first initializes symbolic CAN frames, which will be received by the benchmark program, before calling the benchmark's main function. Additional changes to the benchmarks are also required to enable their symbolic execution. For instance, the benchmarks all run a loop until they receive a termination signal, and this loop must be bound to a finite number of iterations so that the symbolic execution eventually terminates. As with any symbolic program exploration, increasing the number of main loop iterations executed in (or CAN frames sent to) the program will cause the number of possible paths to explore to grow exponentially.

Moreover, we provide function stubs for certain library functions. This includes system library functions, such as network socket functions, as functions which interact with an external environment pose a challenge for KLEE. KOMPARE already provides wrappers for system output functions (e.g. `fwrite` and `fputs`) so that the externally visible outputs can be captured and compared. In addition, we provide a definition for the `read` function, which is called by the benchmarks at each iteration of the main loop, and which feeds the symbolic CAN frame into the program.

The benchmark also uses OpenSSL functions, which, even with their LLVM bitcode provided to KLEE, are challenging to symbolically execute. Cryptographic functions are unfeasible to symbolically execute as that would require constraint solvers to invert values which, by design, is hard to do [3]. To enable the analysis of these programs, we create simple versions of these functions which behave the same (sharing the same interface and output format) as their OpenSSL counterparts and that can be efficiently symbolically executed. This is similar to KLEE's modeling of

external environment calls: although these functions are not as robust nor secure as OpenSSL encryption functions, they enable us to reason about the effect of the patch on the program as they maintain the same input and output formats.

5.2.1 Logging Functionality Patches

One patch we analyze is applied to the benchmark's logging system: all traffic received on the CAN bus will be encrypted and then logged. The vulnerability occurs in a deprecated DES3 decryption function, and the patch resolves this by exchanging DES3 for AES-256 encryption. After each CAN message is received, the program will log the current state of the system in addition to the encrypted CAN message. These writes to the log file will be recorded by KOMPARE and then compared to ensure the patch does not affect the program behavior, such as by causing the program to unexpectedly fail or produce incorrect changes to the system's state.

For this benchmark, we expect the output to differ consistently for every input: where it previously logged the result using DES3 encryption, we will find the result of AES-256. All other outputs should be unaffected. Therefore, we must provide a new comparison function to KOMPARE. We expect both encryption schemes to output the same number of bytes, so our comparison function first ensures that all outputs are identical up to the encrypted CAN message, and then ensures both encrypted CAN messages share the same length. This function is implemented in about 20 lines of code.

We analyze the benchmark with KOMPARE to verify that the patch does not alter the program's behavior beyond this. With the execution bounded to two iterations, KOMPARE discovers 257 paths through the program. To simulate and compare all 257 paths on concrete inputs takes KOMPARE 10 minutes and 34 seconds, and we find that the outputs pass the comparison function on every path, thus verifying that the patch did not alter the program's behavior except where desired. Since we find that the outputs on all paths match as expected, enabling patch-directed symbolic execution shows no difference in output; in both cases, all paths through the program are analyzed. The static analysis for patch-directed symbolic execution took an additional minute.

A similar AMP benchmark adds additional functionality to the logging system: now, messages are hashed and the hashes logged as an integrity check for logged data. To increase the robustness of the integrity check, a patch replaces a CRC-32 hash function with SHA-256. To evaluate this benchmark, we must account for the differences in the output formats, as SHA-256 outputs a larger hash than CRC-32. Additionally, as hash functions cannot be symbolically executed, new simplified hash functions should be provided to model the behavior of SHA-256 and CRC-32, respectively. Then, the analysis performs similarly to the previous benchmark discussed, verifying that the outputs match as expected for all inputs.

5.2.2 Transport Protocol Vulnerability Patches

Some of the AMP benchmarks provide patches which address vulnerabilities in the provided implementations of the J1939-21 transport protocol. As with the other AMP benchmarks, the goal of our analysis is to verify that these patches do not change the behavior of the program in unintended ways by verifying that the externally visible outputs match on all expected paths.

One such vulnerability in this implementation is that, when copying packets into the receiving buffer, if more frames are received than anticipated, the data will continue to be copied to the buffer causing an overflow. To solve this, a patch is added which checks the number of packets being copied against the sequence number of the packet received. For this benchmark, we find that symbolic analysis becomes stuck on a difficult constraint unless some of the bytes in the CAN frame are made to be concrete. These values were recorded manually during a concrete execution which exercises the paths through the program we are interested in. As a consequence of concretizing a portion of the input, the number of paths which can be explored symbolically is limited. Additionally, for this benchmark, output comparison was simple: since we expect both versions of the program to maintain the same output for any input that does not cause the buffer overflow, we can compare them exactly using the default comparison function in KOMPARE.

Here we have a patch which removes vulnerable behavior, and as discusses in §3.2.1, we need to run KOMPARE twice for a complete analysis, symbolically executing both versions in turn. Symbolically executing the patched version with the partially concretized input reveals twenty paths through the program, for which the the output matched on all of them, successfully verifying that the patch resolved the buffer overflow error without affecting the behavior of the program. Symbolically executing the vulnerable version reveals twenty-eight paths through the program, the additional eight of which differed between the two versions in output as, in the vulnerable version, they did not complete execution in KLEE on encountering the buffer overflow error.

Another AMP benchmark corrects a vulnerability related the J1939-21 transport protocol's connection management: when a connection is made, the size of the data is provided as a parameter, and a patch is made to correct a vulnerability where the anticipated data size is not checked against the maximum allowed by the J1939 standard. This benchmark also adds code to the system which, when run under KLEE with a symbolic CAN frame, requires indexing into a large array using a symbolic value. KLEE cannot handle this path explosion, as each valid index in this large array creates a possible execution path. The execution proceeds prohibitively slow, therefore requiring further concretization of the input, namely the entire CAN identifier. Again, further increasing the amount of concretized input limits symbolic exploration of the program. As with the previously discussed benchmark, we are able to only about to analyze the program on explored paths and verify that the outputs match on both versions of the program.

CHAPTER 6

Discussion

In this chapter, we discuss some of the limitations, potential improvements, and future work on KOMPARE. Many of the limitations in KOMPARE's analysis, as demonstrated in analyzing the real-world AMP benchmarks, occur either as limitations of the KLEE engine or of the constraint solver. In this section, we propose techniques to alleviate these challenges, including those which involve the techniques of existing symbolic execution patch verification tools discussed in §2.2. Another potential improvement discussed is path pruning, which may reduce the cost of KOMPARE's analysis. Although KOMPARE does have a conservative path-pruning approach implemented, we find it to be ineffective in practice, and discuss how this might be made useful in the future. We additionally discuss potential improvements for KOMPARE's output comparison functionality.

6.1 Improvements to KOMPARE's Analysis

A potential improvement to the scalability of KOMPARE's analysis is to consider partial program executions instead of complete ones. Concrete inputs may be generated along paths which stall during symbolic execution, solved from the constraints gathered up to such a point, and these inputs may then be used to compare the two versions of the program for differing behavior. While such partial executions may not result in a complete analysis of the program, doing so may allow for a greater number of paths to be compared in the face of symbolic execution challenges.

Techniques discussed in previous work might be applicable to improve the performance of KOMPARE's comparison analysis. One such example is the automated concretization of input for dynamic symbolic execution, as used in DART [4] and ER [14]. In evaluating some of the AMP benchmarks, we discovered that, as these limitations occur in KLEE, KOMPARE was unable to analyze paths which gathered constraints too complex for the solver. These works have shown that it is possible to concretize values which direct symbolic execution towards target code in the program, and such techniques might be applicable to KOMPARE to more efficiently reach modified code or to enable the exploration of additional paths in the program where the constraints are too

difficult to solve.

Another performance consideration, discussed in §3.2.1, is that, as a consequence of only symbolically executing a single version of the program, code paths removed by the patch will be missed. To ensure the completeness of our analysis, we run KOMPARE twice on each benchmark. However, this results in KOMPARE analyzing paths which were analyzed previously, wasting time and computation. A more efficient approach may involve shadow symbolic execution, as presented in SHADOW [5, 8], wherein a single, unified program binary is explored symbolically, and where execution forks both on branches in the program and divergences between the versions. This would effectively enable KOMPARE to generate inputs that explore removed behaviors during its symbolic execution, with which it can perform the output comparison as before for a more cost-effective complete analysis.

6.1.1 Output Comparison

The default output comparison provided by KOMPARE checks that the outputs of both versions of the program are identical and reports a difference otherwise. This is undesirable for programs for which the output occurs in a nondeterministic order, relies on randomness, or changes between executions (such as by including a timestamp). For such programs, comparison functions may be provided which can account for expected differences in the outputs by comparing the outputs at a finer granularity. For instance, the comparison function may check for correctly formatted output with regular expressions or ignore some portions of the output. In §5.2.1, we provide a comparison function which accounts for the expected difference in the outputs when comparing them, since the patch changed the format of the program's output along every path. Currently, these output comparison functions must be implemented within the KOMPARE driver, however it would be a simple and convenient change to allow the users to specify their comparison function externally and provide it to KOMPARE.

These output comparison functions may also be used to assert certain properties of the output. In doing so, KOMPARE can verify not only that the patch does not alter the program's behaviors in unexpected ways, but that the patch correctly solves the problem at hand. For example, in the aforementioned benchmark which exchanges DES3 enryption for AES-256, the comparison function might collect and decrypt the logged output and report an error if the output for the patched version is not correct. Thus, a more intelligent comparison function could additionally verify that the patch indeed behaves correctly with regard to the vulnerability it has patched.

6.2 Pruning Execution Paths

As the program is being symbolically executed, it is possible that constraints gathered along a path become too complex for the solver, thus stalling the program's execution. Furthermore, the symbolic execution of unmodified code (i.e. code that may not be interesting for this problem domain) may be obstructively slow, preventing the comparison analysis from reaching code which exposes differences in the versions' outputs. By directing symbolic execution towards patch code, we explore modified code and its effects sooner than unmodified code. Running the analysis for a fixed amount of time would then effectively prune paths which are not reached later in the analysis. Nevertheless, it may be advantageous to prune uninteresting execution paths as they are encountered, thus decreasing the cost of comparison analysis [9].

We reason that states which do not execute, have not executed, and will not lead to the execution of modified code will exhibit the same behavior for both versions of the program. Thus, we can conservatively consider any such states as candidates for pruning. Once modified code has been executed along a particular path, we are interested in analyzing all subsequent paths through the program to determine if the change in the code has caused any unexpected changes to the program's behavior. Therefore, once modified code has been executed along a path, it will no longer be considered a candidate for pruning by this approach. If the paths we prune do not affect the soundness of KOMPARE's analysis, we expect that, while pruning uninteresting paths, KOMPARE will report the same number of paths with differing outputs while evaluating fewer paths overall.

In KOMPARE, we implement this approach for pruning paths, taking advantage of the existing instruction priority calculations. After back-propagating priorities, an instruction which cannot lead to the execution of modified code will have a priority of 0. Therefore, whenever a state is added or updated in the Patch-Priority Searcher, KOMPARE determines if that state may be discarded, effectively pruning the path. KOMPARE first checks that the state's previous instruction was a branch (that is, this state is one that followed a particular path from a branch). Then, if the state has a priority of zero and no modified code has previously been executed to reach this state, the state is not added to the priority queue. Additionally, when a new state is added whose parent has been marked as having executed modified code, the new state will similarly be marked and not be considered for pruning.

We find that this pruning strategy is ineffective in practice. On the example of a patch which adds a new command line argument to the COREUTILS Echo program, discussed in §5.1, we find that KOMPARE is not able to prune any paths. This is a consequence of how the patch is implemented: line 2 of Listing 5.2 shows that the patch adds an additional variable to the main function of program. Changes to the program's control flow similarly pose a challenge for this simple path pruning. Consider line 16 in Listing 5.1. Here, the call to `fputs` becomes wrapped

in a branch condition on a variable introduced by the patch. Consequently, the paths which would maintain the same behavior would need to still be explored fully, even if the modified variable initialization did not occur in the program's main function. Then, all paths through main (thus all paths through the program) will execute modified code, so none may be considered candidates for pruning by this strategy. Similarly, no paths are pruned in the AMP benchmarks which modify the system's logging or connection-handling functionality, as every CAN frame which is received by the program is subsequently logged.

These examples demonstrate that this conservative method is ineffective when modified code is reached early during execution or along each path, motivating a more intelligent approach towards path pruning. For instance, a data-flow analysis, similar to that used by DiSE [9], can be used to determine which variables in the program are modified by the patch and which paths subsequently use the modified variables. Then, paths which are affected by the modified code but which do not operate on modified variables might be considered for pruning. However, memory aliasing presents a challenge for such an approach. Program behavior on such a seemingly unmodified path may have it's behavior affected by the patch if modified code elsewhere in the program writes to memory which is aliased by the unmodified code.

CHAPTER 7

Conclusion

In this thesis, we presented KOMPARE, a tool which uses symbolic execution for assured patching. KOMPARE cross-checks the externally visible outputs of two versions of a program (one version with a patch applied and the original version) to verify that the patch does not alter the behavior of the program in any unintended ways. Additionally, when provided with an output comparison function that accounts for the differences in outputs or asserts properties in the output, KOMPARE can also verify correct behavior of the patch itself. To increase the efficiency of its comparison analysis, this thesis also presented Patch-Directed Symbolic Execution, a technique to prioritize the symbolic execution and analysis of code modified by a patch. Given the two versions of the program as input, KOMPARE performs a static analysis to determine the programs' differences and uses these differences to drive symbolic execution towards the modified code as quickly as possible.

We evaluated KOMPARE with an example patch to demonstrate its operation on a patch which changes the behavior of the program along certain paths. Then, we used KOMPARE to analyze some benchmarks from the DARPA Assured Micropatching program, which included patches to an embedded control system's logging functionality and corrections to particular transport protocol vulnerabilities. We found that for the former, KOMPARE is successfully able to demonstrate that the patches do not change the behavior except where intended. KOMPARE additionally verifies, for the latter, that the patches also remove the vulnerabilities in the system. We found that KOMPARE's analysis is limited by some of the known challenges in symbolic execution, and discuss some techniques which may be applied to mitigate the impact of these challenges.

BIBLIOGRAPHY

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[3] Ricardo Corin and Felipe Andrés Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In Úlfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, pages 58–72, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[5] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. Shadow symbolic execution for testing software patches. *ACM Trans. Softw. Eng. Methodol.*, 27(3), sep 2018.

[6] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 235–245, New York, NY, USA, 2013. Association for Computing Machinery.

[7] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.

[8] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: Testing for divergences between software versions. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1181–1192, 2016.

[9] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 504–515, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Weizhong Qiang, Yuehua Liao, Guozhong Sun, Laurence T. Yang, Deqing Zou, and Hai Jin. Patch-related vulnerability detection based on symbolic execution. *IEEE Access*, 5:20777–20784, 2017.

[11] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 49–64. USENIX Association, 2015.

[12] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 669–685, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] Neha Rungta, Suzette Person, and Joshua Branchaud. A change impact analysis to characterize evolving program behaviors. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 109–118, 2012.

[14] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1155–1170, New York, NY, USA, 2021. Association for Computing Machinery.