University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research

Computer Science and Engineering, Department of

Summer 7-31-2015

Discovery Over Application: A Case Study of Misaligned Incentives in Software Engineering

Eric F. Rizzi
University of Nebraska-Lincoln, eric.rizzi@huskers.unl.edu

DISCOVERY OVER APPLICATION:
A CASE STUDY OF MISALIGNED INCENTIVES IN SOFTWARE ENGINEERING

by

Eric F. Rizzi

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Matthew B. Dwyer and Sebastian Elbaum

Lincoln, Nebraska

August, 2015

DISCOVERY OVER APPLICATION:
A CASE STUDY OF MISALIGNED INCENTIVES IN SOFTWARE ENGINEERING

Eric F. Rizzi, M. S.

University of Nebraska, 2015

Advisers: Matthew B. Dwyer and Sebastian Elbaum

In this thesis, we present evidence that there is an under-emphasis on the application of software systems in Software Engineering research, affecting the advancement of the field as a whole. Specifically, we perform a case-study on KLEE, a tool with over 1000 citations. We made improvements that consisted of fixing performance bugs and implementing optimizations that have become common practice, increasing KLEE's performance by 2-11X. To understand how techniques proposed in the literature would be affected by these improvements, we analyzed 100 papers that cited the original KLEE paper. From this analysis we found two things. First, it is clear that coherence to the principles of replication is lacking; it was often very difficult to understand how a particular study used KLEE, and therefore to understand how our improvements would affect the study. Second, when conservatively estimating how the studies relied on KLEE, we believe that seven of the 21 papers that we investigated could have their conclusions significantly strengthened or weakened. Upon closer investigation, six of these seven papers involved studies that directly compared a KLEE or a KLEE dependent tool to some other tool. The potential for mis-application within these competing techniques makes it difficult to understand which observations are true, a situation that potentially leads to wasted effort and slowed progress. To conclude, we examine several recent proposals to address this under-emphasis, using KLEE as an exemplar to understand their likely effects.

COPYRIGHT

Acknowledgements

First and foremost, I would like to thank my adviors, Prof. Matthew Dwyer and Prof. Sebastian Elbaum. They have been supportive in every way despite my eccentricities and long winded explanations. I have come to consider them not only my mentors, but my friends, and I wish them nothing but the best.

I would like to thank Myra Cohen for agreeing to serve on my committee.

I would like to thank my family for at at least feigning interest and passing along much more exciting versions of my work to people who ask what I'm doing.

I would like to thank Jenn Schanz for bearing with me through this whole process. I cannot express how grateful I am for her support.

I would like to thank my series of lab mates at UNL, including Josh Branchaud, Mitchell Gerrard, Katie Stolee, John Saddler, Corey Jergensen, Rafael Leano, Mikaela Cashman, and Bakhtiar Kasi. You helped turn what would have been an endlessly stressful situation into an endlessly stressful situation with a fun lunch break. While you all have different goals for you careers, I know you each will end up doing something meaningful.

I would like to thank Brady Garvin for his suggestion to advance upon this line of research. Any of the mis-executions I can only blame on myself.

Finally, I would like to thank Cristian Cadar for his patience and understanding in acquainting me with KLEE and its development community. His willingness to provide

his work for inspection and analysis shows the highest commitment to the ideals of Science. I, as well as the community at large, have benefited greatly from his efforts.

GRANT INFORMATION

This work was partially supported by Air Force Office for Scientific Research, Award #FA9550-10-1-0406. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The pursuits of scholarship are often broken down into four distinct phases. These phases are discovery: the investigation of new knowledge, integration: uniting theories in meaningful ways, application: using knowledge to solve real-world problems, and teaching: enabling future scholarship [1]. In this thesis, we present evidence that there is an under-emphasis on the application phase of scholarship in Software Engineering.

We contend that this under-emphasis stems from academia's reliance on publication as the main avenue for advancement [2]. "[P]ublish or perish" is a common refrain, summing up the pressures on faculty [3]. During the review process, papers that claim something new are much more likely to be published [2, 4, 5, 6]. For example, one paper surveyed 79 program-committee and editorial-board members about their views on replication [5]. The authors concluded "[m]ost participants wish to see more replications but, at the same time, are reluctant to conduct, read, or accept them ... [showing] a lack of incentives for conducting replication studies". Instead, the incentive structure links discovery to advancement, with everything else relegated to a secondary role [1].

While these misaligned incentives are pervasive throughout the scientific community [4, 7], they are more problematic for Software Engineering than for many other disciplines because we create the systems we study [3, 8, 9]. The close relationship between creation and evaluation often makes it difficult to distinguish discovery from mis-application [10]. In some cases, newly proposed techniques (discovery) may compensate for buggy implementations (application), leading to unnecessary complexity and false conclusions. In other cases, new techniques are compared against systems lagging behind the current state-of-the-art, artificially inflating the technique's effectiveness. In the end, the noise from the escalating complexity and competing techniques makes it difficult to understand which observations are true and which are a byproduct of implementation deficiencies. This uncertainty leads to wasted effort and slows the progress of the community as a whole.

To illustrate the problems that are associated with the current incentive structure, we examine the Symbolic Execution (SymExe) [11] engine KLEE [12]. We chose KLEE because it is a well thought of, well maintained tool in the Software Engineering research community. This is illustrated by the fact that the paper that presented KLEE, *KLEE: Unassisted and automatic generation of high-coverage tests* [13], has garnered over 1000 citations [14] and is at the foundation of many other influential research tools [15, 16, 17, 18]. Since the source code's release in 2009, the creators of KLEE have met or exceeded the expectations for maintaining and improving a foundational research tool. They have released the source code [19]. There is an active community of contributors from both academia and industry [20]. The core developers are welcoming of bug-fixes and optimizations to their code.

Despite its importance and widespread use, however, we were able to make several significant improvements to KLEE's code-base. Some of these improvements were bugs that have undermined KLEE's performance since its release. Others were optimizations that have since become common practice. In the end, the time required to run KLEE was reduced by up to 11X, a speedup that dwarfs many of the new techniques proposed in the literature. Given the fact that many other tools do not exhibit the same level of continued development and maintenance as KLEE (discussed in Section 2.3), we find it likely that other tools suffer from similar problems.

In order to understand how our improvements might affect the state-of-the-art, we analyzed 100 papers [21] that were published by international conferences and journals and cited the original KLEE paper [13]. Of these 100 papers, 26 used KLEE in their experiments. In an ideal world, we would be able to easily re-run each of the experiments in these 26 papers in order to understand the impact of our improvements. Unfortunately, our analysis showed us is that papers rarely live up to the principles of replication. While some papers (three) made it easy to closely follow their KLEE-based experiments, others (five) provided just enough information to make it clear that re-running their experiments would be very difficult. Most papers (18), however, left out critical information required to reliably replicate their experiments.

In order to proceed with the analysis about how our improvements affected the state-of-the-art, we decided to focus on the 21 papers that either made closely following their KLEE-based experiments easy or left out critical information on how to replicate their KLEE-based experiments (we left the five papers that had provided enough information to make it clear that replicating their experiments would be difficult for future work). We felt that including the first group of papers would provide valuable insights because we could closely follow their original experiments with our improvements included and examine the effect on their results. We felt that including this second group of papers was a necessary step. Since so many of them had excluded critical information for replication, the only way to understand how they would be affected is through approximation. Therefore, we examined each of these papers and conservatively estimated the KLEE setups they were most likely to have used.

Having made these decisions, we examined how our improvements would likely affect the outcomes of these 21 papers. In the end, we believe that seven of them could potentially have their conclusions significantly strengthened or weakened. These seven papers range in impact from a single citation [22, 23, 24] up to 16 citations [25] and appeared in venues such as ICSE [25], OOPSLA [26], ICST [22], and CAD [24]. These papers illustrate the potential negatives associated with the current incentive structure. They represent potentially wasted effort by authors who mistook mis-application for discovery. They also represent potentially wasted effort on behalf the community as these papers may provide conflicting signals of which techniques are effective and which techniques are not.

Upon closer examination, six of these seven papers directly compared the ability of different options to solve a particular problem and declared one to be *better* than the others. Indeed, 50% (6 of 12) of the papers that involved evaluations of this type we believe could be questioned. The lack of robust conclusions seen in papers that investigate which of several options is *better* is a symptom of our current incentive structure. Improvements are forsaken as more and more complexity is built into systems, leading to wasted effort and incorrect conclusions [27]. The difficulty is that we cannot simply exclude studies that examine which of several options is *better*, since they are important for the advancement of the field. *Better* options lead to faster implementations, in turn making more things become *possible*.

In order to address these problems, we examine how to best re-align the relationship between application and discovery. One option is to change how work on tools such as KLEE is cited and understood [28, 29, 7]. For example, if KLEE's functionality is being used, the tool itself, rather than the original paper that presented it, should be cited [7]. This change, combined with an increased appreciation for this type of work by the community at large, would mean that all contributions to a project would receive credit, rather than just a static list of the tool's original authors. Another option is to increase the role of funding bodies. In this case, money would be given to tools that achieve sufficient importance to merit a devoted staff of developers [30, 31, 32]. This funding would provide a staff whose primary motivation is to improve the tool. Another suggestion is to expand the role of competitions more broadly [33, 34, 35]. Here, the esteem of winning provides the incentive to apply ideas correctly. Finally, the inclusion of a maintenance track in conferences and journals has the potential to reward application within the current model for academic recognition. This track would provide a forum to both highlight improvements to important tools and to identify papers whose conclusions might be affected by these improvements. Each of these four options attempts to reward the work of application within the context of scholarship. Accomplishing this would acknowledge a truth that has long gone unrecognized in our discipline: quality engineering and correct application of state-of-the-art ideas is both difficult and important [36].

Finally, we believe that combining a re-alignment of the incentive structure to reward the scholarship of application with efforts to increase the community's conformance to replication principles [37] has the potential to yield benefits greater than those that would be achieved by each individually. This is because the effects of improvements to a tool underlying a set of papers could be quickly understood. This would allow researchers to identify techniques that are byproducts of incorrect implementations, providing the community with a refined view of the current state-of-the-art. By continually improving and vetting the systems and techniques that we rely on for research, there would be fewer false starts and fewer mixed signals, leading in turn to more consistent progress by the community as a whole [27]. For this reason, we believe that both efforts should be advanced simultaneously.

It is important to note that our analysis of the different ways to address the underemphasis on application leaves many questions unanswered. For example, there is a lot of debate about how to handle closed source and non-released tools within the context of scientific inquiry [38]. Since KLEE is an open source tool, we provide few insights on how to proceed on this front. Problems such as these will likely require their own specific solutions and paradigms.

It is clear that no amount of effort will completely remove all questions about the foundation upon which new ideas are built. The unknowns of Science lie both ahead and behind. What we can do is acknowledge our sources of error, and move to address them. This will lead to greater sustained progress by the community at large.

Chapter 2

Tip of the Iceberg

In this chapter we present KLEE and discuss how it works. After this, we present the improvements that we made to KLEE and evaluate their effect on its performance.

2.1 What is KLEE

KLEE is a powerful SymExe (Symbolic Execution) tool for automaticaly testing C programs. There exist a wide variety of SymExe tools, including ones for binary code [39, 40], Java [41, 42], C# [43], C++[44], and C [45, 46, 47, 48, 25, 49, 50, 51, 52]. Each has been used to find a wide variety of bugs and flaws in programs.

What made KLEE unique from the tools that preceded it was the sheer scale of programs it was able to test. Before KLEE, most SymExe tools were used to test small toy programs or pieces of larger systems. KLEE broke new ground, testing 89 of Unix's core-utils programs, exposing new bugs and achieving higher coverage in a few hours than test suites developed by experts over the course of decades [13].

There are several reasons that KLEE was able to make this important advance. First, it utilized the newly created LLVM framework [53], that allowed large C programs to be compiled into a format that a SymExe tool could handle. Second, KLEE relied on uClibc to model key system calls [54], allowing it to handle unaltered versions of the core-utils. Third, it was the work of some incredibly talented researchers who combined some of their own optimizations with those presented in both the literature and other tools into one fully functional tool. Finally KLEE relies on search strategies that focus on code coverage rather than a complete exploration. This tradeoff allows it to be more useful in practice.

Because of its success, KLEE has been used in a wide variety of ways since its codebase was released in 2009. These include instances where KLEE is used for testing [55, 56], including GPU "kernels" [57], networks [58], and file system checkers [59]. There are groups that have used KLEE for document synthesis [60], stress testing [26], and aiding debugging [61, 62]. In addition, it is often used as a baseline to show the efficacy of a new technique [63, 64], the implication being that if KLEE can't do it, then it is unlikely any other tool could.

2.2 How KLEE Works

Instead of executing a program in the normal manner, where the inputs control a path through the program, KLEE (and SymExe in general) works by representing inputs symbolically. This means that whenever a branch is encountered, the tool forks, ostensibly exploring both sides of the branch. In order to focus towards generating meaningful inputs, KLEE utilizes the STP SMT solver [65] to determine if a particular path is "feasible." A path is "feasible" if there exists some input that would be able to follow the path in question. Additionally, the solver can be used to create concrete inputs, allowing a normal execution to follow these same "feasible" paths. In contrast, "infeasible" paths are ones that cannot possibly be executed and therefore can be removed from the exploration.

In theory, given enough time, every "feasible" path in a program could be traversed, proving it to be free of common bugs (such as assertion violations) as well as generating a comprehensive test suite that exercises all of these paths. In practice, however, KLEE relies on search methods that allow it to target "important" paths sooner in the execution, increasing the chances that bugs will be found in a realistic amount of time.

For the majority of configurations, a KLEE execution can be split into two separate phases: *program exploration* and *test generation*. The *program exploration* phase is tasked with traversing various paths through the program, attempting to cover as much of the program being tested as possible. This phase utilizes two different parts of KLEE. The first part, called the *Executor*, chooses which paths to follow through the program, generating sets of constraints that represent each particular path traversal. The second part, called the *Solver Chain*, decides whether the constraint sets and the paths they represent are feasible (SAT/UNSAT).

Upon completion of the *program exploration* phase (due to timing out, reaching a termination metric, or having fully explored the program), KLEE enters its *test generation phase*.[1]Here, constraints that were generated during the *program exploration* phase are passed to the *Solver Chain* in order to generate concrete tests. These concrete tests allow for a native execution to follow the same paths that were traversed by KLEE.

During normal use, the *Solver Chain* is the most expensive part of an execution. This is because the *Solver Chain* is often forced to use the SMT solver: an extremely costly operation. In an attempt to mitigate these costs, KLEE uses three filters to reduce the size and number of constraint sets that reach the solver. In practice, these filters are very helpful, preventing on average 99.8% of the constraint sets from ever reaching the solver.[2]

[1] It practice, KLEE may also create tests during the *program exploration* phase. This occurs whenever a path being traversed reaches a program exit.

[2] Data generated across all 89 core-utils programs using *none* instance of KLEE configured using the *Generic Regression* setup. Both of these are discussed in Section 2.5.

The first filter looks for “constraint independence”, factorizing the incoming constraint sets into smaller independent subsets (see Appendix A for details). These smaller subsets are much easier for the latter parts of the *Solver Chain* to handle. Additionally, since KLEE builds these constraint sets incrementally, it is possible to identify the newest constraint from the constraints that have been a part of the path for a long time. With this information, KLEE can then “slice” the constraint set. This is useful because, in most cases, just the subset of constraints that contains the newest constraint needs to be forwarded further down the *Solver Chain*, leading to a significant reduction in the number of constraints that need to be handled [66]. Only when a solution for the entire state is required does the entire constraint set need to be solved.

The second filter caches previously determined solutions and checks whether the answer to an incoming constraint set has already been computed. It does so by maintaining a simple map of constraint sets and their solutions. If the constraint set is SAT, then its solution represents a point that, if plugged back into the constraint set, would evaluate to *True*. If, instead, the constraint set is UNSAT, this shows that no points exist that would evaluate to *True*.

The third filter uses a UBTree [67] to store a mapping between a previously solved constraint set and its solution. When a constraint set reaches the third filter, the UBTree is queried to see whether there exists a subsuming relationship between the incoming constraint set and a previously solved constraint set. Should there exist a constraint set that was previously determined to be UNSAT and that is a subset of the incoming constraint set, then the incoming constraint set must be UNSAT as well. If, instead, there is a constraint set that was previously determined to be SAT and is a superset of the incoming constraint set, then the incoming constraint set must also be SAT. Finally, should both of these checks fail, KLEE finds all the solutions of the constraint sets that are subsets of the incoming constraint set. It then plugs every one of these solutions into the incoming constraint set with the hope that one of the solutions evaluates to *True*. This final step relies on the fact that evaluating a solution is much cheaper than generating one via a solver.

2.3 How KLEE is Maintained

KLEE was first released on GitHub [19] in March of 2009. Since then, 24 different contributors have had at least one commit accepted into the main trunk. This effort translates into 989 total commits over the course of the project, including 177 total commits in the past year, and 49 commits in the last month [19]. When compared with other SymExe projects, KLEE's development community is among the strongest. For example, CREST [46], FuzzBALL [49], jCute[42], CAUT [51], and SPF [41] have equivalent or smaller development communities [20].[3]Other SymExe tools, such as Pex [43], do not provide source code for outside researchers to examine or build on. Yet others, such as SAGE [39], CREST-BV [25], and Oasis [52], don't even provide binaries.

To aid new users of KLEE, there is a website with a series of tutorials on how to get KLEE up and running [68]. On this website there is a trove of information that not only allows new users to replicate the experiments presented in the original 2008 KLEE paper, but also provides instructions on how to set up KLEE for development. Should these resources not be sufficient, there is also a klee-dev email list. All of this goes to show that the KLEE community is as responsive, diverse, and active as many other important tools'.

[3] as of 7/6/2015

CREST developers: 1, commits in past month: 0

FuzzBALL developers: 4, commits in past month: 7

jCute developers: 1, commits in past month: 0

CAUT developers: 1, commits in past month: 0

SPF developers: 12, commits in past month: 5

Improvement Description	Location	Pull #	Size in LOC	Status
Complete Solution Re-computation Optimization	Solver Chain	198	215	Pending
Quick Cache Bug	Solver Chain	206	85	Pending
Array Factory Bug	Memory Internals	202	135	Accepted
Default Enabled no-prefer-cex Bug	Exploration Termination	241	24	Accepted
Unnecessary UBTree Superset Computation Bug	Solver Chain	250	15	Accepted
Inequality Concretization Optimization	Constraint Storage	229	231	Pending

Table 2.1: Brief descriptions of the six improvements we made to KLEE. The *Improvement Description* column shows the name of each improvement. The *Location* column describes where the improvement was made within KLEE. The *Pull #* shows the pull request associated with each improvement on KLEE's GitHub page [12]. *Size in LOC* shows how many lines of code were changed in order to accomplish the improvement. Finally, the *Status* column, shows the current status of the pull request as of 7/24/2015.

2.4 Improvements to KLEE

Despite all of these efforts, there remained opportunities for improvement in KLEE. In this section, we go over the six different improvements that we made. Three of the improvements occurred in KLEE's *Solver Chain*. Of the other three, one has to do with how KLEE terminates, one has to do with KLEE's internal memory structures, and one has to do with how KLEE stores constraints. None of the improvements we made can be considered a contribution to the state-of-the-art. Instead, four address a performance bug in KLEE and two add simple optimizations that are now common practice. Table 2.1 provides a brief description of these different improvements. In the sections below, we examine each of them in-depth.

2.4.1 Complete Solution Re-computation Optimization

During KLEE's *program exploration* phase, the first filter of the *Solver Chain* minimizes the size of every new constraint set by factorizing it. Unfortunately, during the *test generation* phase, the original implementation of KLEE would no longer attempt to factorize incoming constraint sets. Therefore, instead of using the cached solutions for its individual subsets, a complete constraint set would pass unrecognized through the *Solver*

Chain and end up at the SMT solver.

To fix this, we changed KLEE so that a constraint set generated during the test generation phase would be fully factorized.[4]Once each subsets' solution is retrieved, all of them are stitched together to form a single, complete solution. This is faster than the original implementation since all of the pieces of the constraint set have already been solved for and combining them is very inexpensive. This approach for generating a single complete solution from many smaller solutions has been considered common practice since 2004 [69].

2.4.2 Quick Cache Bug

This bug occurred at the interface of the second and third filters in the *Solver Chain*. The second filter is designed to catch constraint sets that have been seen before. This is important for two reasons. From one perspective, it can be viewed as a means to quickly decide the status of an incoming constraint set. Another way of viewing it, however, is as preventing the third filter from using its increased power, and therefore increased costs, on constraint sets that could be handled more efficiently.

The misses happened when the second filter was invoked to check whether a particular branch could be traversed along both edges. If the first edge of the branch had not been seen before, then the entire constraint set was immediately forwarded to the third filter. This meant that the other edge of the branch was never checked in the second filter, even though its solution was often already cached. In practice, this meant that the second filter allowed thousands of equivalent constraint sets through. Therefore, the more expensive UBTree data structure in the third filter ended up being invoked when its power wasn't required. We decided that the least intrusive way to fix this problem was by implementing a simple hashmap in the third filter.[5]Similar to the second filter, the hashmap maintains

[4] Full description of improvement can be seen at `https://github.com/klee/klee/pull/198`

a simple map of constraint sets and their solutions. This hashmap is checked prior to invoking the UBTree.

2.4.3 Array Factory Bug

This bug lay deep within the low-level workings of KLEE. During the exploration phase, every time KLEE encounters an array, a data structure that tracks all of the reads and writes to that array is created. In the original implementation, each time one of these arrays was encountered, an entirely new instance of the data structure was created. The problem occured when the same array was encountered along a different path. In this case, two different instances ended up representing the same array. This manifested itself in solutions that were equivalent but not recognized as such due to referencing issues.

To fix this bug, we created a factory method that examined every incoming request for an array. If the request was for an array that had already been created, we simply returned a reference to that array.[6]Fixing this bug enables solutions to be stored and used much more flexibly than in the original implementation of KLEE. For instance, this improvement enabled the *Complete Solution Re-computation Optimization* and *Quick Cache Bug* improvements, discussed above, to be implemented without triggering errors. These errors would occur when solutions were evaluated on a particular constraint set. Despite the fact the solutions should have been recognized as valid, the aliasing problems made them seem otherwise.

2.4.4 Default Enabled no-prefer-cex Bug

In the original implementation, when a user executes KLEE's help command, a menu with over 160 command-line options was displayed. While most of the options are

[5] Full description of improvement can be seen at https://github.com/klee/klee/pull/206

[6] Full description of improvement can be seen at https://github.com/klee/klee/pull/202

clearly explained, there was a command-line option called *no-prefer-cex* without any accompanying description. If you looked through the code, there was no documentation on what this option does. An online search was similarly unhelpful.

Eventually, through a combination of trial and error and the klee-dev email list, we discovered that *no-prefer-cex* is an option that attempts to make each byte in a test case an ASCII character. This results in test cases that contain as many human readable characters as possible. During our exploration, we couldn't find any configuration that degraded KLEE's performance as much as when *no-prefer-cex* was enabled. Unfortunately, *no-prefer-cex* was enabled by default.[7]

We considered *no-prefer-cex* a bug because the costs far outweigh the benefits.[8]None of the papers that we examined mentioned that they wanted human readable output for their test cases. In our experiments with KLEE, we rarely even looked at a test case. While it seems possible that there is a place for this option, such as introducing new users to KLEE, making it the default behavior reduces the performance of KLEE for the average user.

2.4.5 Unnecessary UBTree Superset Computation Bug

This bug was the result of what turns out to be an unnecessary "optimization" added to KLEE. In the original implementation, the UBTree checked for both supersets and subsets of an incoming constraint set. The subset query was very efficient, taking on average less than 1% of the overall computation.[9]The superset query, however, was much less efficient. Over the course of a run, it could account for up to 79% of the computation while on

[7] The *no-prefer-cex* option is actually a double negative. When it is turned "off", it actually caused additional computation. Its default position was "off". Due to the difficulty discussing double-negatives, we will consider *no-prefer-cex* to be "enabled" if it causes the extra computation that produces human readable test cases. We will consider the *no-prefer-cex* option "disabled" if this extra computation is skipped.

[8] Full description of improvement can be seen at `https://github.com/klee/klee/pull/241`

average taking up 44% of the computation.

The reason for this inefficiency is that when more and more constraint sets are placed into the UBTree, the response time for a superset query increased at a much faster rate than the subset query. These escalating computational costs eventually outweighed the costs of bypassing the check and sending the query directly to the solver. Upon removing the superset check[10], our experiments show that, on average, 80% of the constraint sets that would have been caught by the superset check end up being decided by the much less expensive subset check.[11] Even when the subset check does miss, the costs associated with an SMT call end up being significantly less than a superset check.[12]

2.4.6 Inequality Concretization Optimization

Concretization is an important optimization in all SymExe tools. If a symbolic variable can be proven to be a single value, then it no longer needs to be symbolic. This logic is used to simplify and even eliminate many of the constraint sets that need to be solved.

KLEE's original implementation had a simple form of concretization. If a symbolic value was ever explicitly set to a single value $(x = 4)$, then it would be treated concretely. This simplification makes it easier for the SMT solver to return an answer. For example, the constraint set $(x = 4) \wedge (x + y < 10)$ would be simplified to $(y < 6)$.

KLEE did not, however, have a mechanism to handle inequalities, which can constrain a variable to the point where it can only be a single value. For example, the constraints

[9] Data generated across all 89 core-utils programs using *none* instance of KLEE configured using the *Generic Regression* setup. Both of these are discussed in Section 2.5.

[10] Full description of improvement can be seen at `https://github.com/klee/klee/pull/250`

[11] Data generated across all 89 core-utils programs comparing the *all* instance of KLEE against the *no-disable* instance of KLEE. Each instance is configured using the *Generic Regression* setup. Each of these is discussed in Section 2.5.

[12] The authors of CREST-BV noticed this problem when they were comparing their tool directly against KLEE [25]. Unfortunately, they turned off the entire UBTree, removing the highly effective subset checks. By turning off just the superset checks, the efficiency of the UBTree stage increases while simultaneously providing almost all of the filtering capabilities of the original implementation.

$(3 < x) \wedge (x < 5)$ can be used to show that $(x = 4)$. In order to implement this optimization, we simply created a series of maps that hold the maximal and minimal values that a variable can possibly be.[13]Should the maximal and minimal values ever become equal, then the variable would be concretized. This optimization has been common practice since at least 2007 [48, 66, 70].

2.5 Experimental Setup

The goal of this evaluation is to examine how the six improvements explained above might affect the state-of-the-art. In order to do this, we examined the configurations of the 21 papers from our analysis in Section 3.2.2. Of these 21 papers, three made it very clear how they were using KLEE. For the remaining 18, we often had trouble determining their exact configurations due to lack of documentation within the papers. After a close reading of these papers, however, we came to believe that each of these 21 papers could be mapped onto one of three different configurations. Table 3.1 shows the final configuration we determined for each paper.

The first setup, called *Core-Utils Coverage*, covers papers that re-run the experiments in the original KLEE paper. The goal of these original experiments was to find bugs within the core-utils and to create a test suite that only included tests that increased line coverage. In this case, various optimizing command-line options were turned on to allow KLEE to target specific patterns in the core-utils. The second setup, called *Generic Coverage*, covers experiments that did not involve core-utils, and therefore didn't use any optimizing command-line options, but still only outputs tests that increased coverage. The final option, called *Generic Regression*, mimics papers that attempted to generate as many tests as possible. The first two columns of Table 2.2 shows how the papers were

[13] Full description of improvement can be seen at https://github.com/klee/klee/pull/229

Setup	# of Papers	Command-Line Arguments
CC	2	klee –libc=uclibc –posix-runtime –max-forks-terminate=$FINISH –exit-on-error=0 –simplify-sym-indices –write-cvcs –write-cov –output-module –max-memory=1000 –disable-inlining –optimize –use-forked-solver –use-cex-cache –allow-external-sym-calls –only-output-states-covering-new –environ=test.env –run-in=/tmp/sandbox –max-sym-array-size=4096 –max-instruction-time=30. –max-memory-inhibit=false –max-static-fork-pct=1 –max-static-solve-pct=1 –max-static-cpfork-pct=1 –switch-type=internal –randomize-fork –search=random-path –search=nurs:covnew –use-batching-search –batch-instructions=10000 ls.bc –sym-args 0 1 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdout
GC	11	klee –libc=uclibc –posix-runtime –max-forks-terminate=$FINISH –exit-on-error=0 –only-output-states-covering-new=1 ls.bc –sym-args 0 1 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdout
GR	8	klee –libc=uclibc –posix-runtime –max-forks-terminate=$FINISH –exit-on-error=0 ls.bc –sym-args 0 1 10 –sym-args 0 2 2 –sym-files 1 8 –sym-stdout

Table 2.2: The number of papers and the command-line arguments associated with a particular setup. The three setups were the ones used by the papers in our analysis seen in Section 3.2.1. The *Setup* column shows each of the three setups we tested with CC meaning *Core-utils Coverage*, GC meaning *Generic Coverage*, and GR meaning *Generic Regression*. The *Papers* column shows the number of papers from our analysis that used that particular setup. The *Command-Line Arguments* shows the arguments used to run KLEE using that particular setup.

distributed across these three setups.

For test artifacts, we used the same version of the core-utils programs that was used in the original KLEE paper [13]. For each test artifact, we used the same number and size of symbolic arguments as suggested on the KLEE website [68].

We had to design the experiments that assessed the improvements to KLEE carefully. This is because KLEE does not have a traditional timeout. Instead, between the time when KLEE is started and the "timeout," KLEE is in its *program exploration* phase. Once the "timeout" is reached, KLEE switches to the *test generation* phase. This means a one hour "timeout" can result in a 10 hour KLEE run.[14]Simply terminating KLEE at the one hour mark would result in an unequal comparison between configurations. Because of this, we chose to generate a "finish line" and time how long it took the various configurations of KLEE to reach it *and* create all requisite tests. The "finish line" we selected was the number of forks our fastest configuration of KLEE was able to traverse in one hour. We then

[14] This huge disparity between the "timeout" and the actual termination of KLEE run is most common when many tests are being created. This is the case with the *Generic Regression* setup.

recorded how long it took each setup to reach this “finish line” and then generate all of its requisite tests. We did this five times for each program, averaging the execution times across all five runs. If a particular execution took longer than 24 hours, we terminated it.

The third column of Table 2.2 presents each of the three different configurations we evaluated, using the core-utils program *ls* as an example. The $FINISH variable represents the number of forks we used to terminate execution and would change depending on the program being tested. In addition, as previously discussed, the symbolic arguments passed into KLEE would change depending on the program as well.

Finally, for each of these three KLEE setups, we tested seven different instances of KLEE.[15]The first instance had all of our improvements enabled (denoted as *all*). The second instance was untouched by any of our improvements (denoted as *none*). The remaining five instances were versions that had *all but one* of our improvements enabled. We tested versions that were missing the *Complete Solution Re-computation Optimization* (denoted as *no-recompute*), the *Quick Cache Bug* fix (denoted as *no-quick*), the *Unnecessary UBTree Superset Computation Bug* fix (denoted as *no-disable*), the *Inequality Concretization Optimization* (denoted as *no-concrete*), and the *Default Enabled no-prefer-cex Bug* fix (denoted as *no-prefer*). By testing in this *all but one* fashion, we hoped to show the importance of each improvement. The one bug we could not directly test in this *all but one* fashion was the *Array Factory Bug* since both the *Complete Solution Re-computation Optimization* and the *Quick Cache Bug* required it in order to function. Turning off the *Array Factory Bug* would have required that we turn off these optimizations as well.

Setup	none	no-prefer	no-quick	no-recompute	no-disable	no-concrete
CC	11.6 (13.2)	1.2 (1.0)	1.2 (1.0)	1.2 (1.0)	1.6 (1.2)	8.5 (7.5)
GC	2.4 (1.9)	1.4 (1.0)	1.0 (1.0)	1.2 (1.0)	1.9 (1.5)	1.1 (1.0)
GR	11.0 (9.1)	4.5 (3.6)	1.0 (1.0)	1.8 (1.3)	1.9 (1.5)	1.1 (1.1)

Table 2.3: Average (median) improvement of the *all* instance relative to the other six instances on the set of runnable core-utils programs for a particular setup. The *Setup* column shows each of the three setups we tested with *CC* meaning *Core-utils Coverage*, *GC* meaning *Generic Coverage*, and *GR* meaning *Generic Regression*. The *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, and *no-concrete* columns show to how much faster the *all* instance executes all runnable core-utils programs relative to the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively.

2.6 Results

Figure 2.3 shows the results of our experiments. The columns *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, and *no-concrete* refer to how much faster the *all* instance executes all the setup of runnable core-utils programs is relative to the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively. Each cell contains the average improvement of the *all* instance with the median program's improvement in parentheses.

In addition, Figures 2.1, 2.2, and 2.3 are provided to show more detail of how our improvements affected the set of runnable core-util programs. Each figure shows a series of box-plots. The labels *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, and *no-concrete* refer to how much faster the *all* instance is relative to the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively. The box-plots show the minimal, first quartile, median, third quartile, and maximal improvements across the set of all core-util programs that

[15] Each of our test instances was based off commit number 3bd3789 from KLEE's GitHub repository [12]. For example, the *none* instance explained above is a build of commit number 3bd3789 while the *all* instance is commit number 3bd3789 with all of our improvements added on top. Complete execution information and data from our experiments can be found at http://bit.ly/1Ir86EI

could be run for a particular setup. Each graph is displayed using the same logarithmic scale with a maximum value of 100.

We generated each data point within the box-plots by dividing the time required to execute a particular program for a particular instance by the time required to execute that same program for the *all* instance. For example, the minimal point in Figure 2.1 for the box plot labeled *none* represents the one program (*ptx*) that was slower for the *all* instance to evaluate than the *none* instance. We calculated the value of this point by dividing the average time required to finish the core-utils program *ptx* for the *none* instance (768 seconds) by the average time required to finish *ptx* for the *all* instance (2925 seconds) to show that our improvements made analyzing *ptx* 0.26 times as fast.

2.6.1 Core-utils Coverage Setup

For this setup we were able to execute 86 of the 89 core-util programs. We couldn't test three of the programs (*date*, *touch*, and *fmt*) due to preexisting assertion violations that were encountered by every one of our KLEE instances, including the *none* instance.

After running each instance with the *Core-utils coverage Setup*, the average improvement by *all* over *none* across the set of all runnable core-utils programs was 11.6X, as can be seen in Figure 2.3. In addition, as can be seen by the left-most box-plot in Figure 2.1, the median performance of the *all* instance was 13.2X faster than the *none* instance. The program *ptx*, as discussed previously, was the only core-utils program where *all* was slower than *none*.

The improvement that has by far the largest effect on the overall performance of KLEE after our optimizations is the *Inequality Concretization Optimization*. This can be seen in the fact that the *no-concrete* instance had an average runtime that was 8.5X worse with a median value that was 7.5X worse than the *all* instance. In addition, the *Unnecessary*

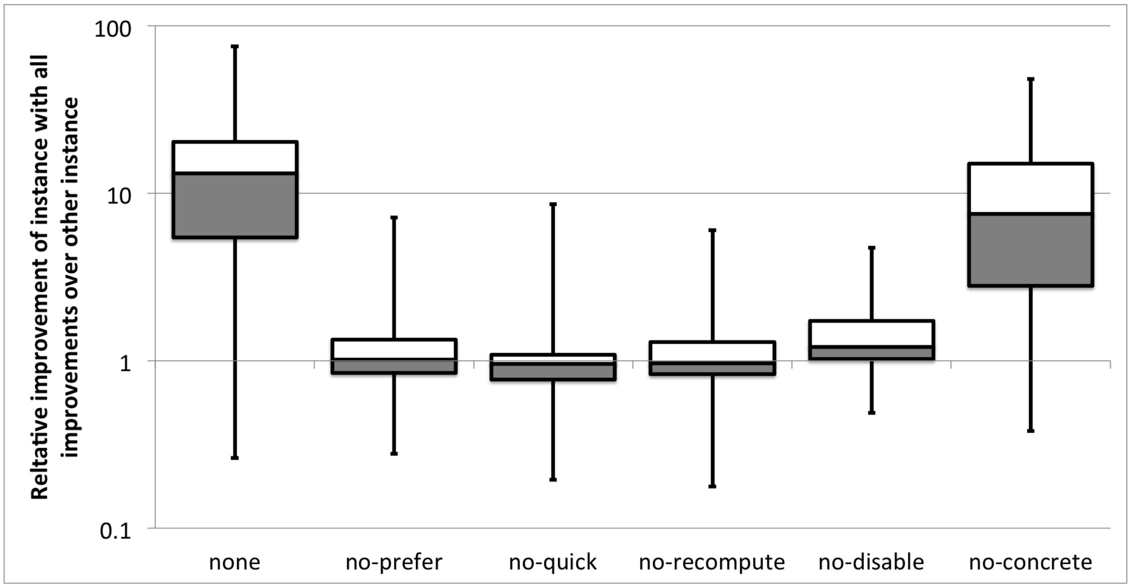


Figure 2.1: Results for executing KLEE with *all*, *none*, and *all but one* of our improvements on the *Core-utils Coverage* setup on all of the programs of core-utils we were able to run. The *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, *no-concrete* box-plots show how much faster the *all* instance is relative the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively.

UBTree Superset Computation Bug is very important, as can be seen by the fact that *no-disable* instance's average runtime is 56% worse with a median value that is 21% worse than the *all* instance. The *Default Enabled no-prefer-cex Bug* was helpful. Without it, the *no-prefer* instance ran, on average, 20% percent slower with a median value that was 1% slower.

Finally, the *Complete Solution Re-computation Optimization* and the *Quick Cache Bug* fix are also useful. The *no-recompute* and the *no-quick* instances had an average runtime that were both 21% and 20% worse than the *all* instance with median runtimes that were 3% and 4% *better* respectively. The fact that the average runtime was faster but the median runtime was sightly slower shows that these optimizations were particularly helpful for programs that took longer to execute.

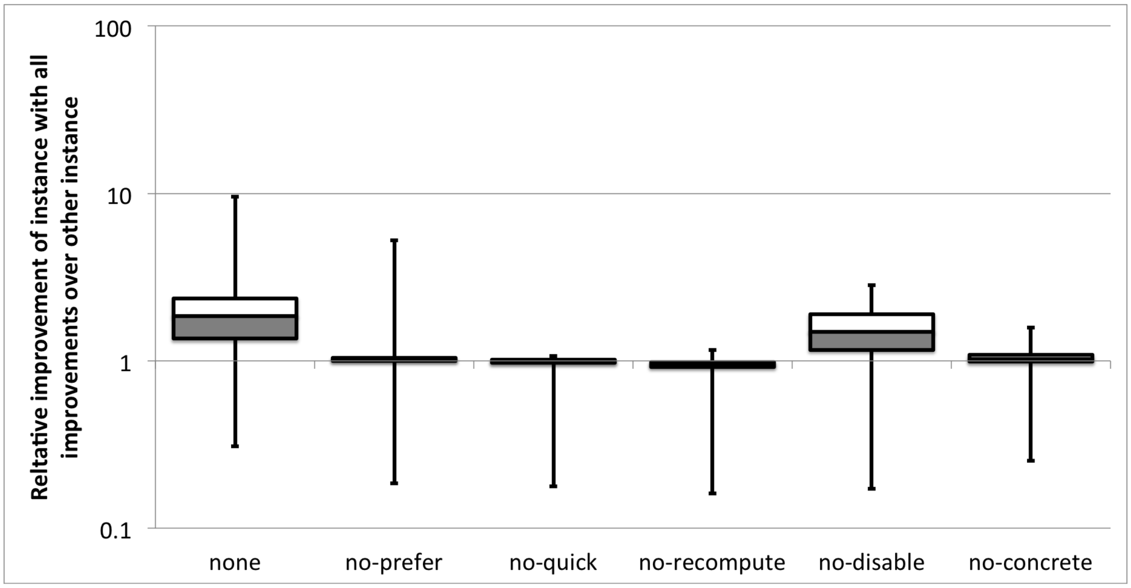


Figure 2.2: Results for executing KLEE with *all*, *none*, and *all but one* of our improvements on the *Generic Coverage* on the full set of core-utils programs. The *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, *no-concrete* box-plots show how much faster the *all* instance is relative the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively.

2.6.2 Generic Coverage

For this setup, we were able to execute all of the 89 core-utils programs. As can be seen in Figure 2.3, the average improvement by *all* over *none* was 2.4X. As can be seen by the left-most box-plot in Figure 2.2, the median program for the *all* instance was 1.9X faster than the *none* instance. The only program that was slower for the *all* instance was *yes*, which went from 257 seconds to 834 seconds. It should be noted that we omitted *sort* from the graph since it was over 1000X faster with the *all* instance than with the *none* instance.

The improvement that was by far the most important for this setup was the *Unnecessary UBTree Superset Computation Bug*. This can be seen by the fact that the *no-disable* instance had an average runtime that was 89% worse with a median runtime that was 49% worse than the *all* instance. The *Default Enabled no-prefer-cex Bug* was the next most important.

The average runtime for *no-prefer* was 40% worse while the median was was 1% worse. In addition, the *Complete Solution Re-computation Optimization* had a positive impact, as seen by the fact that the *no-recompute* instance's average value was 24% worse than the *all* instance, although its median value was actually 5% better than *all*. When examining the data more closely, it turns out that only a few programs benefitted from these improvement, but they benefitted dramatically enough to affect the average execution time.

Fixing the *Inequality Concretization Optimization* was the next most important optimization. The *no-concrete* instance had an average runtime that was 7% worse and a median value that was 3% worse than the *all* instance. The reason that this improvement was so effective in the *Core-utils Coverage* setup but of relative little importance in this setup is because of the difference in search strategies of these two setups. The *Core-utils Coverage* is very aggressive, making large jumps in an attempt to exercise unexplored branches. These jumps produce many more constraints, often bounding a variable to a single value. The *Generic Coverage Setup*, on the other hand, moves much more slowly through the program. This slow progress means that the paths followed are much shorter and therefore less constrained, meaning that the opportunity for concretization occurs much less frequently.

Finally, the *Quick Cache Bug* actually hurt execution times slightly. Both the average and median value for *no-quick* was 1% better when compared with the *all* instance.

2.6.3 Generic Regression

For this setup, we were also able to execute all 89 core-utils programs. As can be seen in Figure 2.3, the average improvement for *all* over *none* was 11X. In addition, as can be seen by the left-most box-plot of Figure 2.3, the median execution time of the *all* instance was

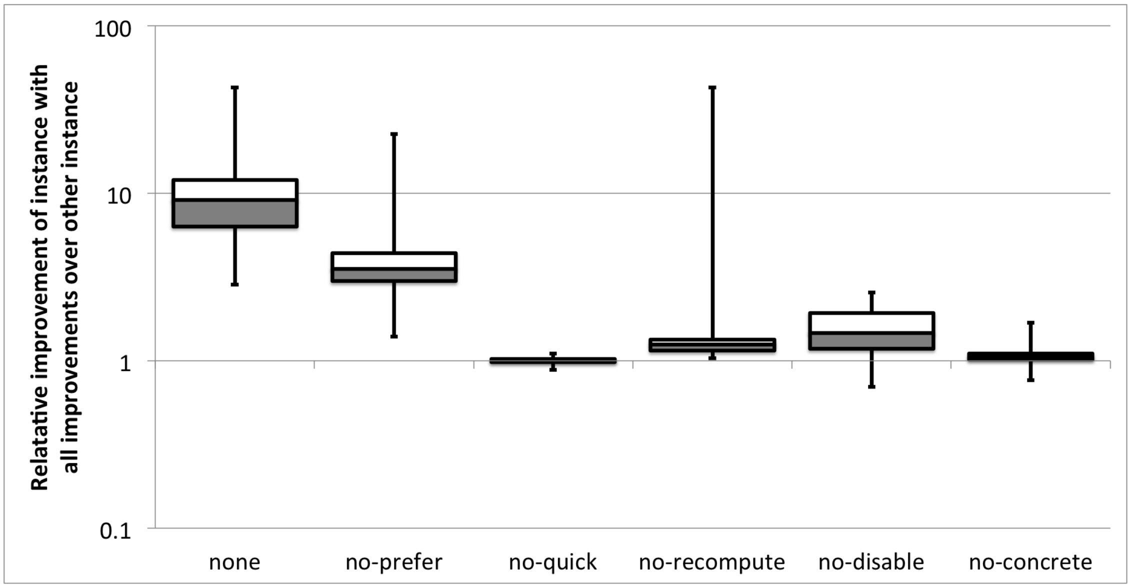


Figure 2.3: Results for executing KLEE with *all*, *none*, and *all but one* of our improvements on the *Generic Regression* on the full set of core-utils programs. The *none*, *no-prefer*, *no-quick*, *no-recompute*, *no-disable*, *no-concrete* box-plots show how much faster the *all* instance is relative the *none* instance, the *no-prefer* instance, the *no-quick* instance, the *no-recompute* instance, the *no-disable* instance, and the *no-concrete* instance respectively.

9.1X faster than the *none* instance. There were no programs that were slower when using the *all* instance. We again omitted *sort* from our graphs since the *all* instance was over 1000X faster than the *none* instance.

The most important improvement for this setup was the *Default Enabled no-prefer-cex Bug* fix, as seen by the fact that the average execution time for the *no-prefer* instance was over 4.5X worse with a median execution time was over 3.5X worse than the *all* instance. The reason that this improvement was so much more effective for this setup than for the previous setups is because this setup produces many more test cases. This means that reducing the costs of producing test cases has a much larger impact.

The next most important improvement was the *Unnecessary UBTree Superset Computation Bug*: the average execution time using the *no-disable* instance was, on average, 88% worse with a median value that was 47% worse than with the *all* instance. The next most important improvement was the *Complete Solution Re-computation Optimization*. The *no-recompute* instance had an average runtime that was 78% worse with a median value that was 25% worse when compared with the *all* instance. As with the *Default Enabled no-prefer-cex Bug* fix, the reason that this improvement was so much more effective for this particular setup is because this setup produces many more test cases and therefore a reduction in the costs of producing a test case has a larger impact.

The *Inequality Concretization Optimization* had a smaller effect since the *no-concrete* instance had an average value that was 9% and a median value that was 5% worse than the *all* instance. Finally, the *Quick Cache Bug* had minimal impact, as seen by the fact that the *no-quick* instance had an average and median runtime than was almost identical to the *all* instance.

2.7 Discussion

The first thing that the data shows is that a particular improvement can have different effects on different setups. As can be seen in Table 2.3, in the case of the *Core-utils Coverage* setup, the average and median increase was over 10X. For the *Generic Regression* setup, the improvements were almost as large, with an average improvement also over 10X and with a median improvement of 9.1X. The *Generic Coverage* setup, however, was much less influenced by our improvements. In this case, the average and median improvement were both around 2X.

The reason for these differences is that the importance of a particular improvement changes depending on KLEE's setup. When the original implementation of KLEE is run with the *Generic Regression* setup, test generation was by far the most expensive operation. It is for this reason that the *Complete Solution Re-computation Optimization*, and the *Default Enabled no-prefer-cex Bug* fix come to the fore. In the other two setups, *Core-utils Coverage* and *Generic Coverage*, there is much less time spent creating tests and therefore the costs of exploration become relatively larger. Therefore, the improvements associated with exploration become more important.

2.8 How We Got Here

In addition to understanding the effects of the improvements we made, examining how we stumbled upon them is useful as well, illustrating many of the challenges that are associated with the current under-emphasis of the application phase of scholarship in Software Engineering.

Over the past few years, we worked extensively with Green [66]. Green is a tool designed to be an intermediary between SymExe tools and SMT solvers. It works similarly to KLEE's *Solver Chain*, first factorizing, then simplifying and concretizing a constraint set to increase the chance it will be matched to a previously calculated solution. Only when all of its attempts fail is a constraint set passed forward to a solver.

One of the goals of this research was to add the ability to handle the bit-vector constraints generated by KLEE to Green. We hoped to eventually discover a new technique that would reduce the number of calls making it to the solver. As such, we ran a series of experiments with KLEE in *Generic Regression* mode. We captured the output that KLEE was sending to the SMT solver and fed it through Green. From this, we made two observations. First, KLEE was intermittently outputting constraints that could be factorized and handled solely through caching. This led us to the *Complete Solution Re-computation Optimization* discussed in Section 2.4.1. The other observation, however, was much more interesting. We noticed that it was possible to divide the constraint sets much more precisely than KLEE's first filter was able to (see Appendix B for details). From this intuition, we developed a simple heuristic that could quickly target these opportunities for factorization that KLEE's approach missed (see Appendix C for details).

As mentioned before, we knew that there was a problem somewhere in the factorization part of KLEE. As we began to implement the heuristic, however, we soon encountered other potential improvements, such as the *Array Factory Bug* and the *Quick Cache Bug*. Since these improvements were seemingly orthogonal to our line of research, we simply marked them for future work. When we eventually got our heuristic working, we achieved over a 2X increase in KLEE's performance.

At some point, however, we noticed our technique only worked when KLEE was attempting to generate a test case. Curious, we eventually stumbled onto the existence of the *no-prefer-cex* option. We realized that when *no-prefer-cex* was disabled, the performance increase enabled by our technique completely vanished (see Appendix D for details). It is fair to say we were disappointed; the technique that we had spent months on was rendered obsolete with the flip of a switch. It was apparent to us that, rather than discovering a technique that would help advance the state-of-the-art, we were really only improving an inefficient and unnecessary behavior.

All of this effort and eventual failure is illustrative of a larger problem. What began as an attempt to improve the state-of-the-art led us down a rabbit hole of implementation details and debugging issues. It was fairly lucky that we even discovered the *no-prefer-cex* option. We had started preparing to publish our technique, presenting it as a universal improvement for all SymExe tools. This realization of how close we came to publishing faulty results caused us to examine how this might have affected other areas of research. We hoped (perhaps cynically) to use our experience to frame the difficulties that can come from working within the current incentive structure, showing that without major changes, similar situations will either lead to faulty publications or wasted effort.

Chapter 3

Shaky Foundations

In Chapter 2, we discussed how several improvements we made increased the performance of a popular research tool within Software Engineering. The key insight is that none of these improvements could be considered contributions to the state of the art: some were performance bugs in KLEE and others were optimizations that have become common practice. In this chapter, we extend our analysis to examine how these realizations could affect publications that rely on KLEE.

3.1 Method

In order to see how our improvements might affect the state-of-the-art, we conducted an analysis of 100 papers. While our analysis was not meant to be a systematic review [21, 71], we attempted to follow the suggested guidelines for conducting one in order to make our setup and analysis complete and transparent. The goal of our analysis was to understand how the work in these papers built and depended on KLEE, and how their results and conclusions are affected by our improvements made to KLEE.

3.1.1 Research Questions

The research questions addressed by this study are:

RQ1 How is KLEE cited and used by the research community?

RQ2 How well do studies enable the understanding of how our improvements to KLEE affect their results and conclusions?

RQ3 How do our improvements to KLEE affect the results and conclusions of these studies?

3.1.2 Search Process

In order to collect the papers used for our analysis, we used *Google Scholar* [14] to identify all the papers that cited the original KLEE paper [13]. We decided to use *Google Scholar* because it aggregates data from many publications and professional organizations.

Our search parameters allowed for all papers that were available on or before 3/6/2015 (the date of our search). We began our search by locating the original KLEE paper in *Google Scholar*. We then clicked on the *Cited by* button to access a list of all of the papers that *Google Scholar* has identified as citing the original KLEE paper. We then scraped the contents of all of the resulting pages to create a local repository of files. From this point, we iteratively and randomly selected a file and evaluated it based on our inclusion/exclusion criteria until we had 100 papers for analysis. The author of this document was the original judge of all of the papers, with two members of the masters panel verifying a sample of the results.[1]

[1] All of the papers that were evaluated against the inclusion/exclusion criteria are available at http://bit.ly/1LTb4s4. In addition, the website offers a full justification for each of the decisions we made in our review.

3.1.3 Inclusion and Exclusion Criteria

There were three criteria for inclusion. First, the article had to cite the original KLEE paper. While our search process would seem to imply this, there were a few papers returned by *Google Scholar* that did not actually cite KLEE and were therefore removed from the study. Second, the papers had to be full, peer-reviewed articles that were published by international computer science conferences or journals. Finally, we excluded papers that were duplicates of papers that had already been included in the analysis.

The reason that we chose to include only international conference and journal articles, while excluding work such as technical reports, theses, and workshops, is because we believe that investigating these papers will allow us to take the most conservative and fair view possible of the state-of-the-art.

3.1.4 Quality Assessment

A typical systematic review contains a section that assigns different weights to results based on the quality of the study and venue in which they appear. These varying weights are then used to synthesize the results should any disagreements occur. This section is an instance where the guidelines specified by Kitchenham [21] did not suite our purpose. Instead, in our analysis we treat each paper in the same way. The reason for this is two-fold. First, our *Inclusion and Exclusion Criteria* sets a high bar for a paper to be included in our analysis. Second, our research questions address the general state-of-the-art rather than attempting to distill a particular field down to a single conclusion.

3.1.5 Data Collection

The data collected from each paper was:

DC1 The conference or journal where the paper was published

DC2 The author(s) of the paper

DC3 How KLEE was used in the paper

DC4 To what degree a study that used KLEE depends on execution time

DC5 How the results of a study that used KLEE would likely be affected by our improvements

DC6 Configuration details for KLEE or KLEE dependent tools if KLEE was used in the study

DC7 Version information for KLEE or KLEE Dependent tools if KLEE was used in the study

DC8 System setup information if KLEE was used in the study

DC9 Information on where additional resources (such as code and test artifacts) could be located if KLEE was used in the study

DC10 Whether the authors believe their proposed technique will map to KLEE if a SymExe tool other than KLEE was used in the study

Some of the data we collected from the papers, such as the author(s), is straightforward. Several of the pieces of information that we gathered, however, were more open to interpretation. Therefore, to aid the data collection process, we created three separate rubrics to formalize these data points.

The first rubric we created is called the *KLEE Usage Rubric* and is intended to classify how a particular paper used KLEE (DC3). The first category in this rubric we called the *Superficial Citations* category. Papers in this category cite the original KLEE paper, but do not use it or any other form of SymExe in their experiments or implementation.

The second category in the rubric we called *SymExe Dependent*. Papers in this category cite the original KLEE paper but use some other SymExe tool in their evaluation or implementation. The final category in the rubric we called *KLEE Dependent*. Papers in this category both cite the original KLEE paper and use KLEE in some part of their study.

The second rubric, called the *Time Dependence Rubric*, was created to classify papers based on the degree to which their evaluation depends on execution time (DC4). This rubric consists of three categories representing an escalating dependence on execution time. The first category, called *Time Reported*, contains papers that only report the time required to execute the technique. A second category, denoted *Timeout Used*, contains papers that use a timeout in their evaluation. Finally, the third category, named *Time Compared*, contains papers that use time to directly compare two or more techniques.

The final rubric we created, called the *Effects of Improvements Rubric*, was designed to identify how studies might be affected by our improvements to KLEE (DC5). The first category, called *Unaffected*, contains papers whose experimental results we believed would not be affected by our improvements to KLEE. A second category, called *Positively Affected*, contains papers whose experimental results would likely improve due to the improvements we made to KLEE. Finally, the third category, called *Negatively Affected*, contains papers whose experimental results would likely worsen due to the improvements we made to KLEE. It is important to note that with this rubric we do not investigate the magnitude of the improvements or how the improvements would affect the conclusions within the paper. Instead, we examine this question qualitatively in later sections.

The author of this document extracted and classified all the information. Particularly difficult decisions were brought before two of the members of the masters panel where they were discussed until a consensus was reached.

3.2 Results

This section presents the results of our study.

3.2.1 RQ1: How is KLEE cited and used by the research community?

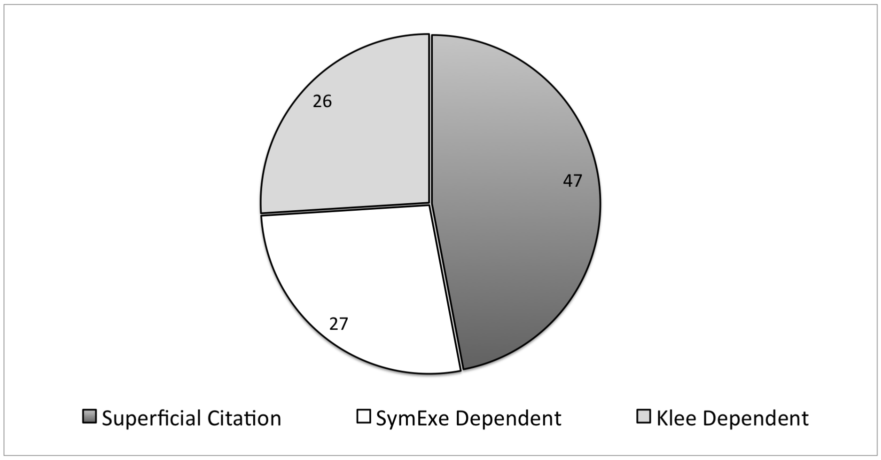


Figure 3.1: How the 100 papers in the analysis were classified by the *KLEE Usage Rubric*

In order to understand how the general research community uses and cites KLEE, we used the *KLEE Usage Rubric* as a coarse filter. From the initial 100 papers, we classified 47 as *Superficial Citations*, 27 as *SymExe Dependent*, and 26 as *KLEE Dependent*. These results can be seen in Figure 3.1. With this initial understanding, we examined the papers in each category.

3.2.1.1 Superficial Citations

Papers in this category include configuration space testing [72], software upgrades [73], accurate bug reports [74], and performance testing [75]. In some cases, KLEE acts as a potential goal [76, 77], providing motivation for the work. For example, one paper "offers a [technique for the] formalization of translation rules for C language that allows a ... test generator to validate properties, like KLEE" [76]. In other cases, KLEE is simply a naïve option that can be easily ruled out [78, 79, 80]. An example of this occurs in a paper that addresses "compile-time configurability" on large systems. They cite the fact that "the most powerful symbolic execution techniques (such as Klee ...) would currently not scale to the size of the Linux kernel" as the reason they don't include it in their study [78]. KLEE is also often cited as a potentially orthogonal tool that, when combined with a newly proposed technique, will lead to further advances [81, 82]. For example, one such paper says their technique "can take advantage of the high coverage achieved by ... KLEE. Specifically, we can reuse the test cases produced from ... KLEE to uncover vulnerabilities in the program" [83].

The diversity of papers in this category shows that KLEE is recognized well beyond the realm of software testing. In many instances, it provides an avenue for future work and advancement. Even when it isn't well suited for a particular problem, however, it is both popular and powerful enough to at least merit discussion.

3.2.1.2 SymExe Dependent

Papers in this category are mostly focused on the testing and debugging domains of Software Engineering. Some papers propose new ways to make the results of SymExe testing more meaningful for developers [84, 85]. Other papers talk about supporting SymExe in general through changes to the underlying theories [86], techniques [87, 88, 89, 90], or tools [66] of SymExe. Finally, papers in this category often create their own hybrid techniques to either target new languages [91], or to get around the path explosion problem in SymExe [92, 93, 94, 95].

Papers in this category are interesting because they were written by SymExe researchers who are aware of KLEE but chose to use a different tool. As we looked through the papers in this category, it became clear that a majority of them (62%) believe that their proposed techniques could be mapped onto KLEE and achieve similar results to those presented in the paper. In some cases this belief is explicit. For example one paper says "our algorithms are applicable in the context of other languages for which symbolic execution tools exists (e.g., Klee ... for C)" [85]. At the end of another paper the authors say "KLEE ... combines several search strategies in a round robin fashion to avoid cases where one strategy gets stuck. [Our technique] selects branches in a new context and this can help prevent the continuous selecting of the same branch" [90].

More often, however, the possibility of extending the proposed technique to KLEE is implied. This most often occurs when a paper identifies a problem with SymExe as a whole and then fixes the problem on a particular version of the tool. An example of this occurs when a paper asserts "[o]ther concolic and symbolic testing tools could integrate our algorithm to solve complex path conditions without having to sacrifice any of their own capabilities, leading to higher overall coverage" [87]. The reader is left with the impression that all SymExe tools, including KLEE, would similarly benefit. Another example occurs in a paper where the authors report "[w]e developed a scalable distributed concolic algorithm that ... can alleviate the limitations of the concolic approach caused by heavy computational cost" [89]. The reader is again left with the impression that similar approaches applied to KLEE will yield similar results. In Section 3.3 we analyze the implications of these common assumptions.

3.2.1.3 KLEE Dependent

When examining the papers in this category, the one thing that was immediately obvious was the versatility of KLEE. Several papers extend KLEE to test GPU's [57], Networks [58], and File Systems Checkers [59]. There are also papers that used KLEE to help a developer fix a buggy line [61] and recreate field failures in the lab [62]. Two other papers argue that KLEE has become such a powerful tool that it may be necessary to rethink altogether how we address certain problems [96, 64]. For example, one of these papers argues for the adoption of a particular programming paradigm since it "preserves performance" while also "enabl[ing] verification" through its amenability to KLEE [96]. Finally, KLEE is also used as a baseline to prove the efficacy of new techniques [22, 56, 63]. For example, one paper creates a front-end analysis to help KLEE "create valid test cases for programs that accept highly structured string inputs" [56]. Its evaluation shows how much more effective KLEE is when used in conjunction with their tool.

These papers show KLEE's influence within software testing and analysis research. In some cases, KLEE is used as the foundation for new techniques, enabling new problems to be tackled. In other cases, KLEE is simply used as a baseline against which other techniques are judged, the implication being that if KLEE can't do it, then the proposed technique must be powerful.

3.2.2 RQ2: How well do studies enable the understanding of how our improvements to KLEE affect their results and conclusions?

This question examines how well papers in the *KLEE Dependent* category present information that would allow future researchers to understand how changes to their implementation would affect their results. This question is important for our purposes because we are attempting to understand how our improvements to KLEE affect the state-of-the-art. Only by understanding how the state-of-the-art uses KLEE will we be able to answer this question.

In a perfect world, it would be possible to simply plug our improvements into each of the KLEE dependent implementations and re-run the experiments they are involved in, examining the effects on the results. Unfortunately, this is not the case. Instead, each paper in the *KLEE Dependent* category provided different amounts and types of information. As an example of the difficulties that can arise, it is not at all clear what

Paper Title	Setup	Config.	Ver.	Sys.	Code	Art.	Time	Eff.
KleeNet [58]	GR	No	No	No	Yes	Yes	TR	P
Modeling firmware as service functions and its application to test generation [97]	GR	No	No	No	No	No	TR	P
A SOFT way for OpenFlow switch interoperability testing [98]	GC	No	No	Yes	No	Yes	TR	P
Symbolic software model validation [99]	GC	No	No	No	No	Yes	TR	P
Automatic detection of floating-point exceptions [100]	GC	No	No	No	No	Yes	TO	P
Control flow obfuscation using neural network to fight concolic testing [64]	GC	No	No	No	No	Yes	TO	N
Detecting problematic message sequences and frequencies in distributed systems [26]	GR	No	No	No	No	Yes	TO	P *
MintHint [61]	GC	No	No	Yes	No	Yes	TO	P
Static analysis driven cache performance testing [101]	GC	No	No	Yes	No	Yes	TO	P
Automated software testing of memory performance in embedded GPUs [23]	GC	No	No	No	No	Yes	TC	N *
Automatic concolic test generation with virtual prototypes for post-silicon validation [24]	GR	No	No	Yes	No	No	TC	N *
Body armor for binaries: preventing buffer overflows without recompilation [102]	GC	No	No	Yes	No	Yes	TC	U
Chaining test cases for reactive system testing [55]	GR	No	Yes	No	No	Yes	TC	N *
Directed symbolic execution [103]	CC	Yes	Yes	Yes	Yes	Yes	TC	N
GKLEE [57]	GC	No	No	Yes	Yes	Yes	TC	P
HAMPI [56]	CC	Yes	No	Yes	Yes	Yes	TC	N
Industrial application of concolic testing approach [25]	GR	Yes	Yes	Yes	Yes	Yes	TC	N *
Postconditioned symbolic execution [22]	GR	No	No	Yes	No	Yes	TC	N *
Scalable testing of file system checkers [59]	GR	No	No	Yes	No	Yes	TC	P *
Software dataplane verification [96]	GC	No	No	No	No	No	TC	N
A synergistic analysis method for explaining failed regression tests [104]	GC	No	No	Yes	No	Yes	TC	P
Docovery [60]	N/A	No	No	Yes	No	Yes	TO	P
Craxweb [105]	N/A	No	Yes	Yes	No	Yes	TO	P
BugRedux [62]	N/A	Yes	No	No	Yes	Yes	TC	N
Selecting peers for execution comparison [63]	N/A	No	No	Yes	No	Yes	TC	N
Reproducing field failures for programs with complex grammar-based input [106]	N/A	Yes	No	No	Yes	Yes	TC	N

Table 3.1: Analysis of papers in *KLEE Dependent* category. *Setup* shows how setups tested in Section 2.5 relate to KLEE's usage in paper. *CC*, *GC*, *GR*, and *N/A* stand for *Core-utils Coverage*, *Generic Coverage*, *Generic Regression*, and *Not Applicable* respectively. *Config.* shows whether all necessary configuration details for KLEE are provided. *Ver.* and *Sys.* show if paper provided version of KLEE (or KLEE dependent program) or system information used in study. *Code* shows whether all code necessary to replicate experiments involving KLEE is provided. *Art.* shows if all artifacts used in study can be located. *Time* shows how we classified paper with *Time Dependence Rubric* with *TR*, *TO*, *TC* meaning *Time Reported*, *Timeout Used*, and *Time Compared* respectively. *Eff.* shows how we classified paper with *Effects of Improvements Rubric* with *P*, *N*, and *U* meaning *Positively Affected*, *Negatively Affected*, and *Unaffected*. A * next to the classification shows we believe the paper could have its conclusions significantly strengthened or weakened.

should be considered KLEE's default setting. The command-line inputs suggested on the website lead to very different results from what would traditionally be considered a default setting: no additional command-line arguments at all. This shows the level of detail that is required to fully understand the experiments involving KLEE.

In addition to these tricky configuration details, there were several instances of papers that provided *unofficial outside resources* in the form of repositories that were not referenced anywhere in the original papers. We found these extra repositories after doing a web-search on 7/5/15.[2]We included the results of this search because it was clear that several papers were presenting a tool to the community, but the norms of providing an internet address hadn't yet been established. For example, one paper says "[o]ur main contribution is a symbolic virtual machine (VM) to model the execution of GPU programs" [57]. The fact that the authors considered the tool to be their main contribution makes it clear that they intended for it to be considered a part of their overall work.

In our attempts to answer this research question, a number of special cases arose. Therefore, we will continually refer to Figure 3.2 to show how different sets of papers apply to our overall analysis for *RQ2*. For example, the input to the figure shows the initial 100 papers in the analysis. Of these initial 100 papers, 47 are *Superficial Citations* (*Superficial* in Figure 3.2) and 27 are *SymExe Dependent* (*SymExe Dep.* in Figure 3.2), and are therefore removed from this part of the analysis. Only the 26 papers in the *KLEE Dependent* category (*KLEE Dep.* in Figure 3.2) were examined while addressing this research question.

We decided to break our analysis of *RQ2* down into three parts. First, we examine the 21 papers for which we could find *no outside resources* (*N.O.R.* in Figure 3.2). Second, we examine the one paper that explicitly referenced *official outside resources* where the code and data from the study could be found (*O.O.R.* in Figure 3.2). Finally, we examine the four papers that have *unofficial outside resources* (*U.O.R.* in Figure 3.2).

[2] For each paper, this web-search involved Googling [107] the name of the tool presented, looking on GitHub for projects with the same name as the tool presented, and examining web pages associated with the authors.

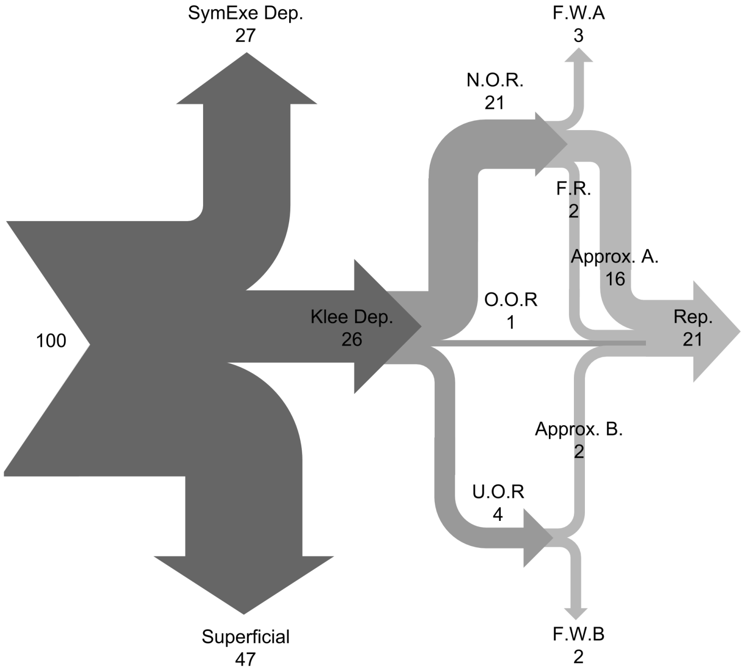


Figure 3.2: Breaking down the 100 papers in analysis. *Superficial*, *SymExe Dep.*, and *KLEE Dep.* denotes papers in the *Superficial Citations*, *SymExe Dependent*, and *KLEE Dependent* categories respectively. *N.O.R.* denotes papers with *no outside resources* in the form of repositories that contained code and artifacts associated with study. *O.O.R.* denotes papers with *official outside resources* in the form of repositories referenced in paper. *U.O.R* denotes papers with *unofficial outside resources* in the from of repositories not referenced in papers but found during a web-search. *F.W.A.* denotes papers in *N.O.R* category that provide enough information to make clear following their experiments will be difficult. *Approx. A.* denotes papers in *N.O.R.* category that, upon a close reading, we believe can be approximated by experiments in Section 2.5. *F.R.* denotes papers in *N.O.R.* category that are fully replicable. *Approx. B.* denotes papers in *U.O.R.* category that, upon a close reading, we believe can be approximated by experiments in Section 2.5. *F.W.B.* denotes papers in *U.O.R* category that provide enough information to make clear replication will be difficult.

3.2.2.1 No Outside Resources

Of the 21 papers that were in the *KLEE Dependent* category and didn't have any *unofficial outside resources*, exactly two provided all the information necessary to clear up all of the possible ambiguities of their *KLEE Dependent* evaluations, meaning they were fully replicable (F.R. in Figure 3.2).

The first of these papers, *Directed Symbolic Execution* [103], used KLEE without any modifications. The paper evaluated easy to find test artifacts, provided full details about the size of the symbolic arguments, and directed readers to the original KLEE paper's setups to handle any ambiguities related to KLEE's configuration. The second paper, *Industrial Application of Concolic Testing Approach* [25] also used KLEE without any modifications. As with the previous paper, the authors provided a complete explanation of the command-line options that were used in their evaluation. In both these cases, the details provided made it very easy to understand how KLEE was being used. This information, in turn, made it easy to understand how our improvements would likely affect their results and conclusions.

Of the remaining 19 papers, 16 of them seem to have used KLEE without any dramatic alterations to its core functionalities (*Approx. A.* in Figure 3.2). Unfortunately, beyond this basic fact, the papers provided very little information about how to recreate their experiments. A common problem was that the basic behavior of KLEE was obvious (i.e. whether a comprehensive test suite was needed or whether coverage was more important), but it wasn't at all clear whether the authors included any extra command-line options to speed the execution. For example, while the paper *Automatic Detection of Floating-Point Exceptions* made it clear that their evaluation was not generating thousands of test cases, it provided almost no specific information on the types of command-line options used in their evaluation [100]. As was seen in Section 2.6, where the *Core-utils Coverage* setup was improved by over 10X with our improvements while the *Generic Coverage* setup was only improved by 2X, the lack of documentation leaves a lot of room for misinterpretation.

Finally, the three remaining papers (*F.W.A.* in Figure 3.2) made it clear that they altered KLEE's most basic behavior: full program exploration [60, 63, 105]. Each of the papers changed KLEE's search algorithm so that it would target a specific line or path. Due to these fundamental changes, combined with the fact that we could not locate any code associated with these studies, understanding how our improvements would affect these papers would be very difficult. Only by implementing the algorithms described in the papers would we be able to understand how their evaluations relied on Klee.

3.2.2.2 Official Outside Resources

Only one of the 26 papers in the *KLEE Dependent* category provided an *official outside resource*. In this paper, the authors present a tool called HAMPI [56], "an efficient and easy-to-use string solver". In one of their experiments, the authors pair HAMPI with KLEE to speed the analysis of programs "with structured input formats". The paper itself provided a lot of the necessary configuration details. In addition the authors directed readers to an online repository for further information. When we visited the repository, we found that it includes code, configuration details, artifacts, and the data that the authors drew their conclusions from. The only information it was missing was the version of KLEE used in the experiments. Therefore, the amount of information provided made it very easy to understand how Klee was being used in the experiment, enabling an understanding of how our improvements would likely effect their results.

3.2.2.3 Unofficial Outside Resources

As previously discussed, after a web-search, we were able to find *unofficial outside resources* for four of the papers in the *KLEE Dependent Category*. The first such repository [17] was associated with the *KleeNet* [58] paper. We knew this because there was a document in the repository directly referencing it. While the repository seemed to provide all of the code necessary to get KleeNet running, it was unclear whether the version of the code tested in the paper was the same as the version present the repository. In addition, we couldn't find any of the artifacts evaluated in the paper within the repository.

The next repository [18], was associated with the *GKLEE* [57] paper. As with the previous repository, we knew that the two were related because there was a document in the repository directing referencing the *GKLEE* paper. As with the previous repository, while all of the code required to get GKLEE running was there, it was unclear how the version of the code tested in the paper related to the code present the repository. Finally, this repository included some, but not all, of the artifacts evaluated in the original study.

The last *unofficial outside resource* we located was a website that provided access to source code associated with both the *BugRedux* [62] and *Reproducing field failures for programs with complex grammar-based input* [106] papers. Both of these studies evaluated the effectiveness of *BugRedux*, a tool that attempted to find specific faults by altering KLEE's full program exploration behavior [108]. The difficulty with this website was that the page devoted to the original *BugRedux* tool directed users to a second page on the site. On this second page users could download a tool called *F3*. This tool was described as a framework that extends BugRedux with "automated debugging capabilities". Here, readers were given instructions on how to download artifacts evaluated in the original *BugRedux* paper, but we couldn't find any reference to the artifacts evaluated in the *Reproducing field failures for programs with complex grammar-based input* paper.

These four instances show how difficult it is to find and use the information provided in *unofficial outside resources* to understand an experiment. The exact relationship of each of these code-bases to the original experiments was unclear. It was possible that they were the same version as the ones appearing in the papers, but it was also possible that there have been significant changes in the time since. In addition, while two of these repositories included all [62], or some [57] of the artifacts that were presented in the paper, the instructions, or lack thereof, made it clear that replicating the experiments with implementations that included our improvements would be very challenging.

3.2.2.4 Distilling the evidence

After all of this analysis and subdividing, two questions that remained to be answered: one theoretical and one practical. The theoretical question directly addresses RQ2. As previously stated, the optimal solution for understanding the effects of our improvements on these studies would be to replicate each of their experiments with out improvements included. In total, however, we found only two papers [25, 103] that would allow for this to happen. For the remaining 24 papers of the *KLEE Dependent* category, we had a much more difficult time understanding how they relied on KLEE.

We compiled the information we gathered from the analysis of the papers and the web-search. The results of our analysis can be seen in columns three through seven of Table 3.1. The third column shows whether all of the otherwise ambiguous configuration details for KLEE were provided. The fourth column shows whether the version of KLEE or the KLEE dependent tool was provided. The fifth column shows whether system information about the computer on which the evaluation was run was provided. The sixth column shows whether all of the code necessary to recreate the experiments involving KLEE was provided. The seventh column shows whether information about how to get the artifacts that were tested in the experiments involving KLEE was provided.

This table makes it clear that any attempt to replicate all of these experiments would be very difficult. The two configuration details that were most often reported were the computer setup (12) and how to reproduce or find the artifacts that were being evaluated (20). Many studies (19) didn't even provide the code that was used in the experiment. If there was one take-away from Chapter 2, its that the implementation details matter. In addition, only four studies provided version information of KLEE or the KLEE dependent tool they were testing. This information is interesting because it is likely that at some point all of our improvements will be integrated into KLEE. Therefore, any future attempt to replicate the experiments in the *KLEE Dependent* category will be choosing from versions that have up to a 11X difference in performance. Finally, it is worth noting that almost 60% of papers provided *CPU* information while less than 10% provided all of the necessary configuration details for KLEE. This shows that researchers are generally aware of how important it is to report their setups, but have a misunderstanding of what details are important to report.

The other question that needed to be addressed was how to handle each of the 26 papers in the *KLEE Dependent* category so that we could best understand the effects of our improvements. There are, in essence, three types of papers in our analysis. There are those papers (three) that made it very easy to closely follow their KLEE-based experiments (*O.O.R.* and *F.R.* in Figure 3.2). These papers can be mapped onto one of the three setups we describe in Section 2.5. There are those papers (five) that provided just enough information to make it clear that attempting to follow their experiments would be very difficult (*F.W.A.* and *F.W.B.* in Figure 3.2). These papers do not map onto any of these three setups and, therefore, we set them aside for future work to see how our improvements would affect their results. Finally, there are the 18 papers that left out critical information required to reliably replicate their experiments (*Approx. A.* and *Approx. B.* in Figure 3.2). Handling these papers was a little more difficult.

In the end, while we were a little uncomfortable guessing at the setups of many of these studies, the lack of documentation often made it a necessary step. Since so many of them had excluded critical information for replication, the only way to understand how they would be affected was through approximation. Therefore, we carefully examined each paper and conservatively estimated the KLEE setups they were most likely to have used. In order to do this, we adopted the strategy where we assumed that a paper used KLEE in the most effective way possible. This led us to divide the 18 papers into one of two camps. If it was clear that KLEE was being used to create thousands of tests, we assumed its setup was most similar to the *Generic Regression* setup. Otherwise, we assumed it was a part of the *Generic Coverage* setup discussed in Section 2.5. Adopting this strategy also meant that we were being conservative in terms of the effects of our improvements, since we never mapped a paper to the *Core-utils Coverage* setup, the setup that our improvements affected the most. For example, in the paper *Software Dataplane Verification* [96], it was unclear whether the authors were using KLEE to generate a lot of tests. Based on a close reading of the paper, however, it was clear that a large test suite was not required. Therefore, we assumed that the setup they used matched the setup that would best suite their purposes while minimizing the effects of our improvements, leading us to the *Generic Coverage* setup. The results of how we mapped out papers can be seen in the second column of Table 3.1.

3.2.3 RQ3: How do our improvements to KLEE affect the results and conclusions of these studies?

In order to answer this question, we focused on the 3 papers whose evaluation we could closely follow (*O.O.R* and *F.R.* in Figure 3.2) and the 18 papers that could be best understood through approximation (*Approx. A.* and *Approx. B.* in Figure 3.2) discussed in the previous section. In order to get an initial understanding of the different types of evaluations being conducted in these papers, we classified each of them using the *Time Dependence Rubric*. We classified four papers as *Time Reported*, five as *Timeout Used*, and 12 as *Time Compared*. In addition, we examined how the papers in each category would be affected by our improvements to KLEE. To do this, we evaluated each of the papers using the *Effects of Improvements Rubric*. The results of combining these two rubrics can be seen in Table 3.2.

When examining Table 3.2, we see that only one paper has results that we believe

–	Negatively Affected	Unaffected	Positively Affected	Total
Time Reported	0	0	4	4
Timeout Used	1	0	4	5
Time Compared	8	1	3	12

Table 3.2: *Time Dependence Rubric* x *Effects of Change Rubric*

would be unaffected by our improvements to KLEE. This particular paper presents a technique to detect buffer-overflows in C programs [102]. As part of the initialization process for their technique, the authors use Howard [109], a tool that builds on KLEE's code-base, to discover data structures in a binary. In this paper, the authors focused solely on the execution time of their overflow detection technique, ignoring the initialization time. The fact that such a small percentage of the papers (5%) were immune to the impact of our improvements to KLEE shows that the cost of SymExe is almost always a significant factor in determining the efficacy of a technique.

More generally, from this initial breakdown, it is clear that the papers that compare KLEE to some other technique are much more likely to have their results negatively affected (8 out of 12) by improvements to their implementations than the other evaluation setups. In the next sections we look at each category in the *Time Dependence Rubric* individually to see to what degree the study's results and the conclusions they lead to would be affected by the improvements.

3.2.3.1 Time Reported

When examining the results of applying the *Time Dependence Rubric*, there are four papers in the *Time Reported* category. These papers are "proofs of concept," using SymExe to solve new problems. For example, one paper in this category examines how long it takes for KLEE to explore a particular type of driver design [97]. Another looks at the time required to create useful models from a more complicated system [99]. Overall, papers in these category use time to show that a particular application of KLEE is *possible*.

To understand why each of the papers in the category would likely be positively affected by our improvements, we will use the paper *KleeNet* as an example [58]. In this paper, the authors present the KleeNet tool, an extension of the original Klee tool that is designed to handle “unmodified distributed sensor network applications. It considers symbolic input values from the environment and generates execution paths of participating nodes at high-coverage”. In their paper, the authors test their implementation on five different network related bugs. In the end, KleeNet only has trouble finding one of these five bugs. In this case, “KleeNet reached the configured 1GB memory cap after 22 hours of execution generating thousands of test cases”, none of which were useful. Only by “applying domain knowledge...[could the authors] efficiently reduce this execution complexity to a reasonable level”. Given the likely improvements for this particular setup (as seen in Section 2.6, the Generic Regression setup achieved a 10X improvement), it is possible that the problem could now be solved without “domain knowledge”, making the technique more effective.

When examined in the light of the improvements we made to KLEE, we believe the conclusions of the papers in this category are very robust. The reason for this is that none of the papers in this category make a value judgement: declaring one technique to be superior to another. Instead, each paper presents a new technique and argues that the time required to solve the problem is acceptable. The setup of these studies means that any improvements in the underlying infrastructure will, in turn, benefit the technique itself.

3.2.3.2 Timeout Used

The *Timeout Used* category has five papers. As with the *Time Reported* category, this category is also composed of papers that are “proofs of concept”. For example, one paper in this category builds a system on top of KLEE for finding flaws in floating point programs [100]. In their evaluation the authors set a timeout and, because of this, several executions are not completed. This paper shows that the difference between this category and the *Time Reported* category is that authors determine at what point the cost/benefit of a particular technique is no longer acceptable.

Of the five papers in the category, we believe that only one of them would have its results negatively affected. This particular paper presents a technique specifically designed to obfuscate code from SymExe analysis, using neural networks to make simple predicate statements more complicated [64]. To test their technique, the authors use KLEE to try to create a test case for both the *True* and *False* paths through an obfuscated *if* statement. If this study was re-run with the improvements made to KLEE, we would expect KLEE's performance to increase by approximately 2X. This is because the setup of their experiment is similar to the *Generic Coverage* setup we tested in Section 2.6. Despite these improvements, however, it seems unreasonable to think that this increase would alter the paper's conclusion that their technique can be used to effectively obfuscate full programs against KLEE and other SymExe analyses. This is because even if the improvements would allow KLEE to pass through a single *if* statement before the timeout was reached, the difficulty of doing so makes it highly unlikely that KLEE would be able to scale to even the smallest of programs obfuscated in this way.

For the four papers that could be positively affected, there was one in particular whose results seemed likely to drastically change. In this paper, the authors ran KLEE for 24 hours, generating thousands of tests to be used for a secondary analysis [26]. Were this study to be run again, we would anticipate approximately a 10X increase in the performance of KLEE. This is because the authors use of KLEE resembles the *Generic Regression* setup we tested in Section 2.6. Given the magnitude of KLEE's improvement, it seems likely that the technique would be able to finish much more quickly. Therefore, we find it likely that the conclusions reached in this paper could have been significantly strengthened. This illustrates an example where KLEE may have been an unnecessarily limiting factor on the results of the paper and therefore limited the impact of their findings.

For the remaining three papers that we believe would be positively affected, the likely importance of the improvements is less than the previous two examples. In one case the authors used exceptionally short timeouts, meaning the effects of the improvements would be limited [101]. In other cases, the results were already so persuasive (e.g. One tool "discovered inputs that generated 2091 floating-point exceptions" "across the 467 functions" [57]) that improving them further would seem to be of little note [61, 100]. Overall, the results of these three did not rely on KLEE in such a way that our improvements would alter their results enough as to significantly strengthen or weaken their conclusions.

As with the previous category, we believe that the conclusions of the papers in this category are robust. Of the possible four papers whose results would likely be positively affected by our improvements, we identified only one that may have have its conclusion significantly strengthened [26]. In the worst case, it is possible that the potential impact of this technique may have been reduced due the problems in KLEE. As for the one instance where a paper's results would likely be negatively affected by our improvements, we find it unlikely that the changes would be drastic enough to significantly weaken its conclusion [64]. Just like with the *Time Reported* category, these five papers all show that a technique is *possible*. It would require a fundamental flaw to refute these types of results.

3.2.3.3 Time Compared

The *Time Compared* category has 12 papers. After analyzing each of them, we believe that this category contains many more questions than the other categories in the *Time Dependence Rubric*. Whereas only one of nine papers (11%) in the previous categories

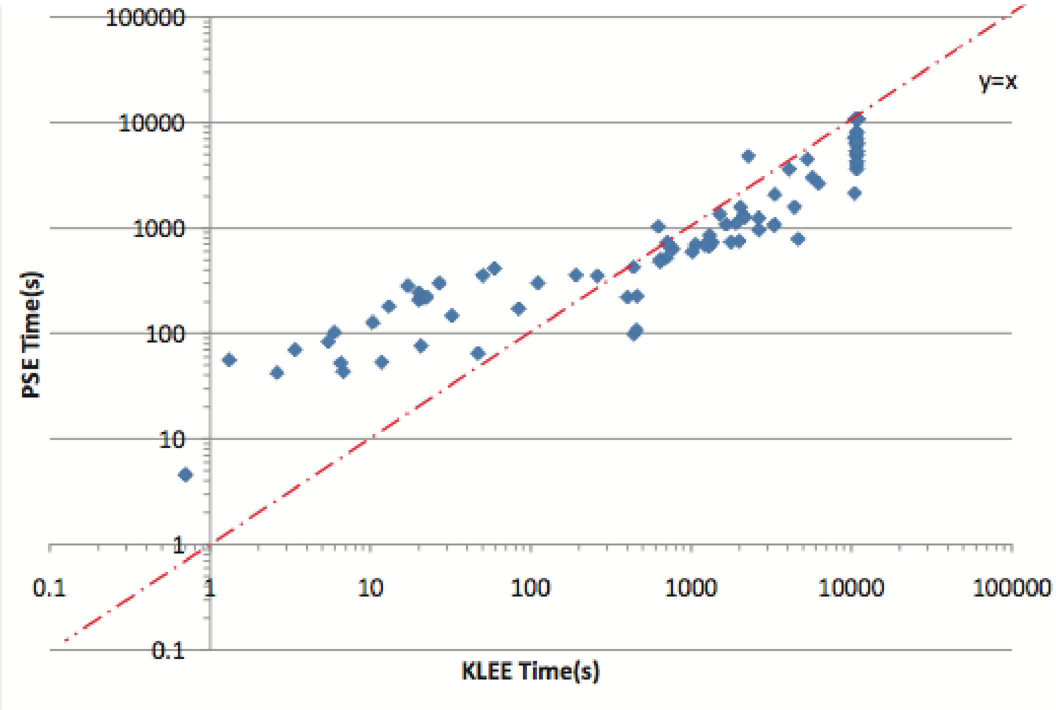


Figure 3.3: Reproduction of Figure 2 from the paper *Postconditioned Symbolic Execution* [22]. This figure compares the time required to run KLEE combined with the technique proposed in the paper (PSE) against the time to run base KLEE (KLEE).

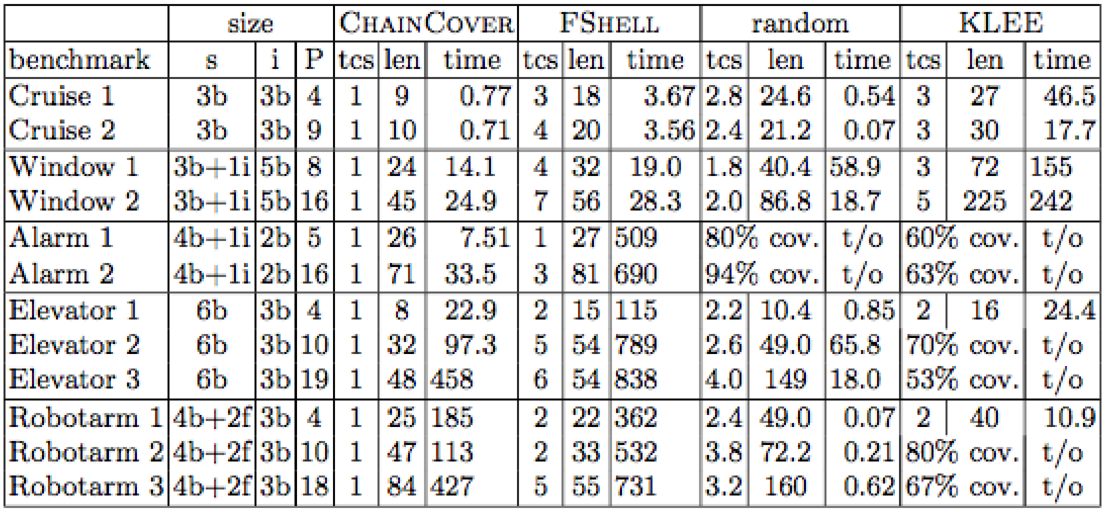
had results that may be significantly altered (positively), 50% of the conclusions of the papers in the *Time Compared* category could be significantly strengthened or weakened. The main reason is that the previous two categories were about the *possible* whereas this one is about *better*. Given the number of variables underlying many of these systems, *better* can be a very difficult thing to pin down.

When looking at the papers that would likely be negatively affected, there are four papers that, in some way, compensate for the test creation problems in KLEE. Each of these papers is similar to the *Generic Regression* setup. Given that KLEE exhibited a 10X improvement with this setup seen in Section 2.6, we believe the conclusions of these four papers are questionable.

One paper uses state matching "to identify and eliminate common path suffixes that are shared by multiple test runs" [22]. The authors report reduction in the number test cases of around 5X. In addition, they report that their technique "can have a significant speedup over state-of-the-art methods in KLEE". With this they imply that their technique is the best of both worlds: not only does it reduce the number of tests that have to be run but these tests are generated faster. Their technique does have significant overhead. They provide a graph, reproduced in Figure 3.3, that compares the time to run KLEE versus KLEE with their technique added. As can be seen, there is a tipping point when the costs of SymExe become high enough to warrant their technique. Upon applying the 10X improvement seen for this setup, it is possible that this tipping point is never reached. This in turn would call into question "the trade-offs between effective redundancy removal and the computational cost of detecting and eliminating such redundancy".

Another paper creates a test reduction technique by first identifying important "states under test from concrete executions ... and then symbolically" executing them [24]. They claim that their technique significantly reduces the time required to generate a test suite while only reducing test coverage a small amount. For example, in one of their experiments, their technique takes "30 minutes which includes 3.5 minutes for state selection and 26.5 minutes for test generation." When examining a larger set of "6000 states ... selected using the random strategy. It takes 1 day [after which] only two new test cases are generated". Our improvements call into question the cost benefit analysis presented in the paper. If the time required to produce tests can be reduced by 10X, then these missing test cases may end up being worth the extra time.

A third paper attempts to "discov[er] a test case chain - a single test case that covers a set of multiple test goals and minimises overall test execution time" [55]. In order to do this, they compare several techniques, including KLEE, against their own technique. The results of their experiment can be seen in Figure 3.4. This figure shows the five techniques they tested (ChainCover, FShell, Random, and KLEE) and the number of test cases in



	size			ChainCover			FShell			random			KLEE		
benchmark	s	i	P	tcs	len	time	tcs	len	time	tcs	len	time	tcs	len	time
Cruise 1	3b	3b	4	1	9	0.77	3	18	3.67	2.8	24.6	0.54	3	27	46.5
Cruise 2	3b	3b	9	1	10	0.71	4	20	3.56	2.4	21.2	0.07	3	30	17.7
Window 1	3b+1i	5b	8	1	24	14.1	4	32	19.0	1.8	40.4	58.9	3	72	155
Window 2	3b+1i	5b	16	1	45	24.9	7	56	28.3	2.0	86.8	18.7	5	225	242
Alarm 1	4b+1i	2b	5	1	26	7.51	1	27	509	80% cov.		t/o	60% cov.		t/o
Alarm 2	4b+1i	2b	16	1	71	33.5	3	81	690	94% cov.		t/o	63% cov.		t/o
Elevator 1	6b	3b	4	1	8	22.9	2	15	115	2.2	10.4	0.85	2	16	24.4
Elevator 2	6b	3b	10	1	32	97.3	5	54	789	2.6	49.0	65.8	70% cov.		t/o
Elevator 3	6b	3b	19	1	48	458	6	54	838	4.0	149	18.0	53% cov.		t/o
Robotarm 1	4b+2f	3b	4	1	25	185	2	22	362	2.4	49.0	0.07	2	40	10.9
Robotarm 2	4b+2f	3b	10	1	47	113	2	33	532	3.8	72.2	0.21	80% cov.		t/o
Robotarm 3	4b+2f	3b	18	1	84	427	5	55	731	3.2	160	0.62	67% cov.		t/o

Figure 3.4: Reproduction of Table 1 from the paper *Chaining Test Cases for Reactive System Testing* * [55]. The table shows KLEE's performance (KLEE) relative to other techniques at creating test case chains.

the chain (tcs), the length of the test case chain (len), and the time it took to find the chain. When the half hour timeout is reached, the maximum coverage achieved so far is output. While they were many complexities to how they used KLEE, in the end they say "exhaustive exploration (i.e. Klee) is not suitable for our problem". As proof of this they state that despite the fact "Klee found test case chains on a few of the benchmarks in very short time, [it] did not achieve full coverage within an hour on half of the benchmarks". We find it highly likely that a 10X faster KLEE would have a much greater chance of finishing many of these benchmarks. If this were the case, it might call into question the conclusion reached in their paper that ChainCover was the superior tool.

Finally, the fourth paper creates an entirely new Symbolic Execution tool, using its ability to output more tests and achieve greater coverage in a set amount of time as proof that their implementation is superior [25]. The authors test a single test artifact in two different ways, in the end observing that their tool is "10 to 28 times faster than KLEE in terms of test case generation speed". Considering that there were instances of programs

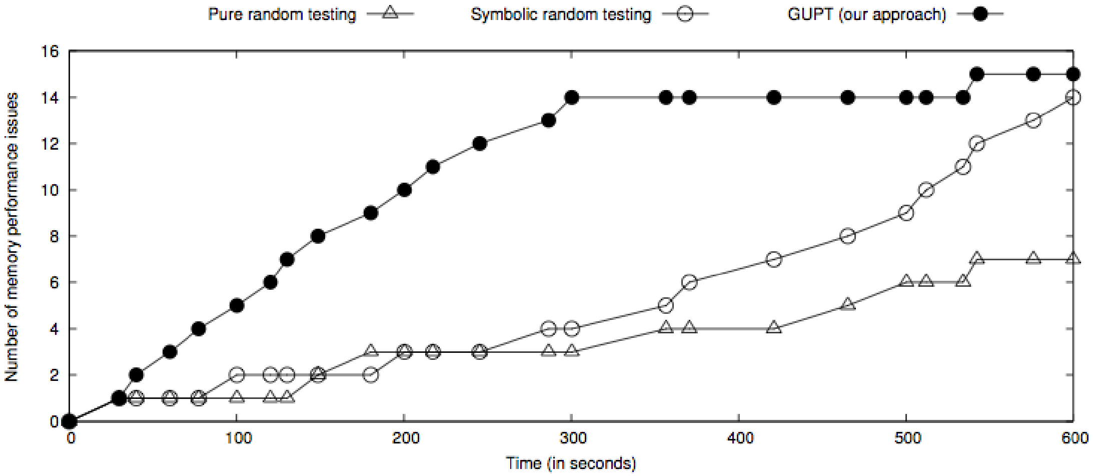


Figure 3.5: Reproduction of Figure 6 from the paper *Automated Software Testing of Memory Performance in Embedded GPUs* [23]. This figure tracks the ability of three different techniques to discover memory performance issues over time. *Pure Random Testing* is fuzzing the GPU with concrete values. *Symbolic Random Testing* is testing the GPU with GKLEE. *GUPT* is the authors proposed approach.

that took up to 68X less time to complete in our study, their assertion that “The speed of test case generation by CREST-BV was 28 times faster than that of KLEE” is now highly questionable.

Each of these four papers each attempted to circumvent KLEE’s original test creation process in different ways. In the process, they likely mistook a mis-application on KLEE’s part as an opportunity for discovery. By applying the *Complete Solution Re-computation Optimization* and the turning off *Default Enabled no-prefer-cex*, however, KLEE’s test creation difficulties are dramatically reduced. Given this reduction, we believe the conclusions of the four papers could be significantly weakened.

Another paper that we believe could have its conclusions significantly weakened presents a technique designed to “[detect] the inefficiency of...software developed for embedded GPUs” [23]. In order to expose these problems, the authors create their own tool, combining a static analyzer with GKLEE, and compare its performance with a base version of GKLEE. In the experiments, the authors note “the performance problems are detected early during our test generation” while “the efficacy of symbolic [execution] is slowly growing”. The authors provide a representative graph that can be seen in Figure 3.5. The figure compares the effects of running their approach (GUPT) versus GKLEE (Symbolic Random Testing) and Pure Random Testing. On the y-axis is the number of faults a technique is able to find and on the x-axis is the time, in seconds, required to find the faults. When examining this figure, it seems that the newly proposed technique is running about 2X as fast as base GKLEE. We believe that our improvements would enhance this base GKLEE by approximately 2X since it is similar to the *Generic Coverage* setup seen in Section 2.6. While it is likely that their version would also speed up, the overhead of the static analyzer would mean that the difference between the two techniques would be reduced. This reduced performance might mean that the additional complexity of their technique is no longer justified.

Several things are clear from these five papers whose conclusions we believe might be significantly weakened by our improvements. First, it is clear that KLEE is so complex that even when directly looking at and working with a mis-application, it is often difficult to identify it. This is obvious in the fact that four separate papers addressed KLEE’s test output problems without discovering the underlying cause. We ourselves admit to working with KLEE for over two years before locating these improvements. Second, it seems that new features (discovery) are favored over maintenance (application). While these potential improvements remain unaddressed, more and more techniques are piled up on top.

Of the papers that were positively affected by the improvements we made to KLEE, we believe that only one of them could have its conclusions significantly strengthened. This particular paper uses test suites generated by SymExe to find faults in file system checkers [59]. In their evaluation, the authors compare their work against a test suite that accompanied one of the checkers. The setup they describe is similar to the *Generic*

Regression setup we tested, that resulted in a $10X$ speed up. In their discussion, they observe that "applying [our technique] to the file system checker led to a lower code coverage than that obtained with the...test suite". The authors don't necessarily view this as a failure, saying the "results are nonetheless positive for [our technique]". We believe, however, given the magnitude of the likely improvements in KLEE, that either the coverage would dramatically increase or the time required to run the technique would be dramatically reduced. Therefore, if these improvements had been made sooner, it is possible that the technique would have produced a larger impact.

In total, only six of the 12 papers in the *Time Compared* category are unlikely to have their conclusions significantly affected by our improvements to KLEE. The reasons that these papers are able to do this varies. Some papers addressed the underlying theoretical problems of SymExe: running up against the limits of SMT Solvers [63] or tackling the path explosion problem [96]. In other cases, KLEE's performance does not affect the outcome of the research questions being investigated [102, 103]. For example, in *Directed Symbolic Execution* the authors are investigating different search strategies to reach particular lines in test artifacts [103]. They include KLEE's search strategy in their study, implementing it in their own SymExe tool. While the do test how well KLEE does on these test artifacts, they seem only to include KLEE as an implicit endorsement that their version was implemented correctly. Finally, in *A Synergistic Analysis Method for Explaining Failed Regression Tests* [104], the runtimes are short enough to limit the amount of improvement seen in the technique.

3.3 Discussion

After conducting this study, there are three points that we believe should be taken away. First, there seems to be a disconnect between research that is intended to benefit SymExe in general and the application of these techniques into SymExe tools. We saw this in Section 3.2.1.2 where over half of the papers in the *SymExe Dependent* category, whether explicitly stated or implied, believe that their techniques could benefit a number of these tools. In actuality, it is very rare for a technique to be implemented in more than the one tool that is required for evaluation and publication. Indeed, only three of the 27 papers in the *SymExe Dependent* category went so far as to incorporate and test their technique on multiple tools. Following the publication of a paper, there seems to be even less chance that a technique will be implemented across the set of SymExe tools: we couldn't find any of the techniques proposed by the papers within this category in KLEE's code-base. This disconnect leaves the community open to both duplication and confounding techniques.

Second, after examining the information provided by papers in the *KLEE Dependent* category, we believe it would be very difficult to replicate the results reported in many of them. In an ideal world it would be easy to identify the dependencies in a particular experiment so that it can be understood in the context of future work. This would in turn make it easier to understand the data-points presented in each paper and synthesize it into a complete picture of the field. Instead, we often had to infer, based on incomplete evidence, what was being asked of KLEE in many of these papers. When making these decisions we tried to be as conservative as possible, but the reality is that it often difficult to understand how important KLEE is to the overall system. This in turn makes it difficult to understand how improvements may affect the presented results.

Finally, when examining how the improvements made to KLEE would likely affect the 21 papers we examined, there was a clear dichotomy. As can be seen in Table 3.3, papers that address the *possible* are highly robust to improvements to their underlying parts. Short of finding a bug that makes the output incorrect or a drastic misunderstanding of the problem being addressed, few things can change the results of whether something is *possible* or not. In contrast, experiments that used time to make value judgements such

–	Weakened	Unaffected	Strengthened	Total
Time Reported	0	4	0	4
Timeout Used	0	4	1	5
Time Compared	5	6	1	12

Table 3.3: *Time Dependence Rubric* x Likely effects of improvements on conclusions. *Weakened* stands for papers whose conclusions we believe could be significantly weakened. *Strengthened* stands for papers whose conclusions we believe could be significantly strengthened. *Unaffected* stands for papers whose conclusions we believe are not likely to be significantly affected in either direction.

as *better* are much more susceptible to performance improvements affecting their results. The problem is that, as a community, we cannot simply rely on experiments that show what is *possible*. Experiments that show what is *better* push the performance of different parts of the field. They allow researchers to judge the relative merits of a technique, in turn enabling experiments that make more things *possible*. Speed matters because speed leads to possibility. Unfortunately, speed is also contingent on a whole host of factors. From the clock rate of the CPU all the way up to the configuration details of a tool, with many variables interacting in highly complicated ways. How best to achieve progress in such fluid and complicated environments is an exceptionally difficult question.

The lack of application of discovered techniques makes it difficult to be sure which problems have been solved and which problems remain. The lack of documentation makes it difficult to understand the dependencies within these systems, obscuring the effects of proposed techniques. Finally, the push for discovery multiplies these problems, moving research forward without enough opportunity to reflect on the state of the systems being tested. Only by taking deliberate steps can these difficulties be overcome. In the next chapter, we examine various ways that these problems can be addressed.

Chapter 4

Lessons Learned and Moving Forward

The previous three chapters have given evidence that there were opportunities for significant improvement in a foundational research tool that were the result of an under-emphasis of the application phase of scholarship in Software Engineering. In addition, we analyzed 100 papers to attempt to understand how the state of the art was affected by these improvements. From this analysis, we made three observations. First, there seems to be many newly proposed techniques that do not end up being implemented into the tool. Second, when examining papers that included the tool in their study, we often had difficulty understanding how it was being used. Finally, even when conservatively estimating how these studies use the tool, there were many whose conclusions we believe could be significantly strengthened or weakened due to the improvements.

In this chapter, we examine how our case-study fits into the context of two larger systematic questions. First, we look at what lessons can be drawn from our study about the communities coherence to the principles of replication. Second, we examine four different proposals to re-align academia's current incentive structure. For each of these proposals, we discuss how they would affect the continued development and improvement of KLEE. Finally, we present related work to our study and our plans for future work.

4.1 Replication

Replication has been presented as an evolution that changes along with the maturity of the topic being investigated [110]. In “immature experimental disciplines”, it is important to closely follow the original experiments [6]. This initial replication makes sure that the reported results aren’t due to randomness. This initial replication also allows researchers to isolate variables within the code-base in order to better understand how the tool works. Over time, as the important variables are identified and tuned, the code base will mature. It is at this point that the code-base can may be combined with other “less similar replications” in order to understand more general principles.

Our study provides a strong example of the positives that come from following the principles of replication. The improvements made to KLEE in Chapter 2 were an actualization of the replication process; we were able to closely follow the experiments presented in the original KLEE paper [13] and identify small changes to improve the tool. In addition, as the code base continues to mature, KLEE can increasingly be used to identify principles that extend beyond the tool itself. For example, a mature KLEE could be compared with other white-box testing tools to examine the limits of this class of analysis. This evolution is possible because the developers of KLEE enabled the replication of their experiments, providing the code, artifacts, and configuration documentation used in the original KLEE paper.

Our analysis of 100 papers that cite KLEE, on the other hand, shows the problems that come from ignoring the principles of replication. As we discussed in Section 3.2.2, only two of the 26 papers that use KLEE included all the necessary information to fully replicate their KLEE-base experiments experiments. Attempting to “closely follo[w] the baseline experiment" [6] for the remaining 24 papers would vary in difficulty. Some provide enough information so that, after a little bit of effort, the original experiment could likely be closely followed. Most, however, are missing critical pieces of information that make following the original experiment very difficult. The inability to understand how a technique works on all but the most theoretical level makes it very difficult to assess. On the one hand, it is very difficult to refute the study, since it is always possible that the replicating experiment missed some key detail. On the other hand, this lack of specificity limits the study's scientific value, preventing the variables within the technique from being isolated and understood, making it difficult for more general principles being uncovered.

Therefore, given the benefits that came from being able to replicate the experiments in the original KLEE paper, combined with the difficulty we had understanding papers whose studies, at best, could only be approximated, it is clear that the replication movement is vital to the progress of our field. Fortunately, there has been a lot of recent progress in this area. "[S]everal ACM SIGPLAN conferences (OOPSLA, PLDI, and POPL) and closely related conferences (SAS, ECOOP, and ESEC/FSE) have ... initiated an artifact evaluation process that allows authors of accepted papers to submit software as well as many kinds of non-software entities (such as data sets, test suites, and models) that might back up their results" [37].

There are five different ways that software can be submitted to these conferences in order to be considered "replicable", including binaries, virtual machines, and websites [111]. We believe, however, that the best way to live up to the principles of replication for systems based research is through the highest fidelity option possible. Given this, virtual machines should be preferred over the other current options. Just as we showed how changes to KLEE could affect a large proportion of the studies that relied on it, it is also possible that changes to the lower level-systems on which KLEE relies (such as LLVM, uClibc, Linux, or even memory configurations) could affect its results. A virtual machine creates a snap-shot of all of the libraries and dependencies that support the artifact being studied. This snap-shot best allows researchers to understand the interaction between the layers of complicated pieces of software, meaning the conditions within the entire system be understood and controlled as much as possible. This level of control will enable work to mature and evolve proposed techniques, in turn providing a greater understanding of the field.

Finally, it seems clear that enabling replication, by itself, is not enough [5]. Simply making experiments replicable does not imply that the tools underlying them will mature. This can be seen in the fact that several very important improvements went unnoticed since KLEE's release in 2009. This shows that there have to be incentives to go into the code-base and mature these tools and techniques. Only then can the full potential of the replication movement be reached.

4.2 Misaligned Incentives

Unfortunately, the current system rewards discovery much more than it rewards application in the form of maintaining and improving existing code-bases. Grants, students, publications, and jobs all flow from pioneering research, overlooking the fact that good engineering can have a significant impact on the effectiveness of a tool [2]. This mentality leads to three problems.

First, unnecessary complexity ends up being added to the tool. This can be seen in the fact that four separate papers addressed KLEE's test generation difficulties through new techniques when debugging and existing techniques would have led to similar or better outcomes. We ourselves almost fell into this trap as detailed in Section 2.8. At the bare minimum it leads to wasted effort. Unchecked, this has the potential to lead to layers of techniques that correct the mis-applications of the techniques below [112].

Another problem with the current system is that there is little incentive to add a newly proposed technique to more than one tool. As we discuss in Section 3.2.1.2, there is a common practice of asserting that a technique, once shown to be effective on one tool, will therefore be effective on all tools. Despite these assertions, the technique can remain un-implemented years later. This can be seen in the fact that two of our optimizations remained unimplemented in KLEE years after they were proposed. While the results we presented in a new paper might be strong, it is also possible that the reported effectiveness might be reduced or even eliminated by previously proposed, and similarly unimplemented, techniques.

Finally, the lack of implementation of existing techniques raises questions about the strength of experiments that use KLEE to argue that SymExe in general is ill-suited for a particular problem. The results we achieved by implementing just a few of these existing techniques were an integral part of the 2-11X improvements we saw in Section 2.6. Overall, to reject the possibility of Symbolic Execution, overlooking much of the progress that has been made in the field since 2008, creates a low bar for techniques to be considered "successful" [113].

4.2.1 Potential Solutions

There has been a lot of work detailing the mis-aligned incentives as well as investigating ways to rectify the problems they create. In the sections below, we investigate four of these proposals and examine how they might affect KLEE's development.

4.2.1.1 Cite Papers and Tools

The practice of citation is used as a means of "authentication and authority" and as a "provision of credit and acknowledgment" [29]. This "academic reputation economy" [7] is used in both hiring and tenure decisions [3]. Unfortunately, "activities that facilitate science ... are not currently rewarded or recognized" [29] having "detrimental effects on funding, future library development, and even scientific careers" [114].

To combat this effect, several proposals have been made to increase the visibility of work that maintains and improves software. The main idea of most of these is to prevent a static list of developers from becoming associated with a particular project, ignoring outside contributions. One system suggests citing the actual location of projects rather than simply citing the paper that accompanied the tool. Within the repository that stores the tool, a list of authors can then be consulted to see whose work enabled the project in its current form [112]. Another proposal goes even further, discussing a system of transitive credit, where the author of a tool rewards underlying systems and authors with a percentage of the credit [29]. This idea extends the programming language *R*'s practice of suggesting developers include a citation function that can then be used to create a list of citations of all packages used in an analysis [115].

It is not enough, however, to merely acknowledge software improvements. These suggestions are predicated on changes in the way the institution understands and values application. There have been several attempts to communicate the importance of these changes to the community at large and facilitate their adoption. For example, two papers [2, 3] discuss how hiring and tenure review committees should interpret different types of scholarship beyond discovery. In addition, *COIN-OR* is a non-profit foundation that was originally created to "create and disseminate knowledge related to all aspects of computational ... research" [28]. The foundation "classifie[s software] on a scale from

0 to 5 according the level of development of the project" [116]. The hope is that this classification will allow reviewers to quickly understand the importance and maturity of a particular project [2].

Applying the more modest of these proposals would likely require minimal changes on KLEE's part. While the GitHub [117] repository that stores KLEE does not currently have a list of authors, people would simply have to examine the commit history provided by GitHub.[1]Instead, the change would have to come from the community at large. We would have to change the way that we cite these tools and how we interpret and use these citations.

Adopting more aggressive proposals, such as transitive credit, would require effort from the KLEE community. While the main developers are quick to provide credit to outside help, it would likely require some intense discussion to quantify how much credit different projects and developers should receive.

4.2.1.2 Increase Institutional Support

Another suggestion to incentivize the application phase of scholarship is to pay people to act as stewards of the tool [7]. This approach would create a staff of dedicated developers whose job is to maintain and improve the code. While there are many current instances of this model [30, 31, 32], the fact that the improvements we made to KLEE weren't discovered sooner suggests that their roles would likely have to change in order to become an acceptable solution.

In the context of KLEE, it seems likely the bugs we found could have been discovered by a team of dedicated engineers. When it comes to effectively integrating proposed optimizations into the code-base, however, it seems likely that the people who discovered the technique should be involved. The reason for this is that the information provided by

[1] https://github.com/klee/klee/pull/205

researchers is often not enough to successfully replicate a technique [27]. Bringing together the people who understand the tool and the people who understand the technique would likely increase the chances of success.

4.2.1.3 Promote Competition

One potential solution is to rely on competitions, and the incentive of winning them, to realign the discovery and application phases of scholarship. This model was used in the verification community, "facilitat[ing] the proliferation of different...approaches, algorithms and implementations" [33]. At every competition, a winner is chosen and the lessons learned are quickly integrated into the community. The success of these competitions led to the creation of software testing and verification specific competitions [34, 35]. In *SVComp*, C-based SymExe tools are evaluated alongside model-checkers and other program analysis tools on a large corpus of C programs [34]. The *Unit Testing Tool Competition* uses java programs for their test artifacts [35]. As with *SVComp*, all types of testing tools are welcome.

Given the infrastructure currently in place, it would be possible to include KLEE in competitions such as these. At this point, however, it seems that competitions has very little traction within the SymExe community; so far only three SymExe tools have competed in *SVComp* and none have competed in the *Unit Testing Tool Competition* [35]. The reason for this lack of traction is that SymExe has been established as a powerful technique within the testing community. Therefore, there is relatively little to gain by competing in the competition and a lot to lose. What seems more likely to benefit the community is a competition among SE tools. This way the lessons drawn from a winning tool would be more easily transferred.

4.2.1.4 Create Maintenance Track

Another potential solution is adding a maintenance specific track to every major conference. Papers submitted to this category would be judged on three criteria: importance, magnitude, and scope. Importance is a measure of how important the tool is to the community at large: fixing a tool used centrally in hundreds of experiments deserves more recognition than fixing one that was used only a few times. Magnitude is the degree to which a program is improved with larger improvements being favored over smaller improvements. Finally, the scope of the improvements would relate to how well the authors explore the impacts of their fixes and what effect it has on the community's understanding of the state-of-the-art. Therefore, this track would not only reward work that improves important tools, it would also serve as a forum through which "artifactual" [6] techniques can be identified.

The reason for the creation of a separate track is that papers that claim something new are much more likely to be published [1, 5]. By creating a maintenance track, the community would be incentivizing application through the traditional reward mechanisms in Science: publication. These papers would be given the same visibility at a conference as traditional discovery papers, making it easy to fit into the current hiring and tenure practices [3].

In terms of KLEE, parts of this document is likely very similar to the types of papers that would be submitted to the application track. In the first half of the paper, the authors would provide the improvements they have made and discuss what the magnitude of these changes were (seen in Chapter 2). In the second half of the paper, work that depends on KLEE would be re-examined to find whether the improvements had any impact on their conclusions (seen in Chapter 3).

4.3 All Together Now

We believe that combining further efforts to implement the replication movement along with re-aligning the incentive structure to reward the scholarship of application has the potential to yield benefits greater than those that would be achieved by each individually. Since replicating experiments becomes easier, it would also be easier to re-run an experiment given a certain bug-fix or optimization The scope and impact of these tool improvements could therefore be quickly assessed, removing unnecessary techniques and strengthening the evidence of the efficacy of others. This would result in a continual process of vetting and improving both tools and the state-of-the-art. False starts would be more quickly identified and improvements would be more quickly integrated. In the end, this would allow the community to more effectively understand the state-of-the-art and what needs to be done to improve it.

4.4 Related Work

There has been a long history of papers suggesting structural changes to increase the veracity of scientific results. Perhaps the most recent example is *Why Most Published Research Findings are False* [113], in which the author argues that "most research findings are false for most research designs and for most fields". Software Engineering exhibits many of the practices that the author attributes to generating false results, such as small sample sizes, large numbers of untested variable relationships, flexible experimental setups, and conflicts of interest (financial or otherwise).

When focusing on Science and its dependence on software, there has been a lot of recent work on "sustainable software" [112, 36]. There is a particular focus on the misaligned incentives for improving software [29]. As an example, in one paper [7], the authors investigate how Blast [118], a bioinformatics tool, has been maintained and improved since its release in 1990. They conclude that those motivated by academic credit are less likely to integrate their improvements back into the original tool. While the software sustainability movement attempts to benefit a large number of fields, Software Engineering's relationship with computation is more complicated than most since we often evaluate the artifacts we create. This means that dealing with these misaligned incentives is particularly important for our community.

Finally, there has been a lot of recent work in understanding and arguing for greater coherence to the principles of replication [6, 36, 119, 110, 120]. Two papers in particular that highlight the problems are *Reproducible Research–What, Why and How* [27] and *Repeatability and Benefaction in Computer Systems Research* [38]. In the first paper, the authors detail their efforts to do a meta-analysis of code-reading techniques. Upon compiling 18 different studies, the authors were unable to determine which technique was the best. This revelation leads them to "strongly advocate the adoption of [replicable research] by software engineering and computer science researchers and data analysts."

In the second paper, Proebsting *et al.* compile a set of 601 papers "from ACM conferences and journals" [38]. After an extensive web-search, hundreds of emails, and a lot of work, Proebsting *et al.* determined that they could locate and build the code associated with 48.3% of these papers. While our paper is similar to this study because we examine the replicability of a set of papers, what we do with this information differs. Proebsting *et al.* focus on "weak repeatability" since they do not attempt to verify any of the results in the paper; they are focused on merely building the objects that were used in the studies. We, on the other hand, attempt to use the ability to replicate previous experiments in order to understand the effects of changes to their underlying implementations. In terms of "weak repeatability" we were able to find code associated with 26% of the papers in the *KLEE Dependent* category (see Figure 3.1). This compares favorably with the 27% of code that Proebsting *et al.* were able to find following links in a paper in an initial web search. We did not, however, go the extra step of attempting to contact authors that increased their build-rate from 27% to 48.3%. We instead relied on approximation to understand the effects of our changes.

In their communications with the authors of the papers in their study, Proebsting *et al.* encountered many of the same problems we did, including versioning issues and commercial tools whose code could not be released. At the end of their study, they say the reason the community "do[es] not produce solid ready-to-share artifacts or attempt to replicate the work of ... peers [is] because there is little professional glory to be gained from doing so". They conclude that "finding new reward structures that encourage us to produce solid artifacts, to share these artifacts, and to validate the conclusions drawn from the artifacts" is the only way to avoid these problems. We believe that this study forms a very solid base but doesn't go far enough. Proebsting *et al.* are concerned with raising the level of computer science research incrementally: requiring the release of code that can be built and examined is a first, necessary step. As noted in Section 4.1, however, merely providing replicable experiments does not incentivize researchers to examine the code or update the tools on which so much of our discipline depends. By addressing both problems, replication and application, the amount of wasted effort will be reduced and progress by the community at large will be increased.

4.5 Future Work

There are many things to do in order to better understand the scope and applicability of the observations made in this documents. First and foremost we plan on further investigating the seven papers whose conclusions we believe could be significantly strengthened or weakened. We were unable to obtain the configuration information, code, and documentation associated with six of these seven papers. Therefore, we plan on attempting to contact the authors of each of these six studies so that we can fully replicate their experiments and understand how our improvements affected them without any qualifications. When all of this information is assembled, we plan on following each study as closely as possible in order to definitively understand the effects of our improvements.

Second, we plan on investigating the papers discussed in Section 3.2.2.4 that provided enough information to make it clear that replication of their studies would be difficult. Understanding the effects of our improvements on these papers will generate extra data points, providing a much clearer picture about the robustness of papers in the *KLEE Dependent* category.

Finally, we plan on investigating other research tools in Software Engineering. As we have previously discussed, we believe that the KLEE community has done everything expected of an influential research tool. This would imply that other tools are likely to have similar opportunities for improvement. By investigating other tools, we can provide further insight into extent of the problems caused by the current incentive structure.

Chapter 5

Conclusion

In this thesis, we presented evidence that a foundational tool in Software Engineering has been under-developed in the years since it was first released. We believe, based on the fact that is KLEE is one of the most developed and tested SymExe tools available, that these problems are not unique to a single tool. Instead, the current incentive structure favors discovery over application. Since a significant portion of Software Engineering research involves testing systems that are written by humans, mis-application may be misidentified as discovery. Without more effort being put into correct implementation, papers that rely on these foundational tools are resting on shaky ground.

To address these problems, we examine several proposals designed to increase the emphasis on application. The goal of each of these proposals is to reward researchers for doing the necessary work of strengthening these tools. We believe that combining these proposals with further improvements in the replication movement will lead to significant gains for the community at large. It will become easier to test and verify existing techniques. The capability of tools will more accurately reflect their true potential. Finally, as code-bases mature, they can be used as sources from which generalized knowledge can be produced.

The truth is that the system in its current form is self-correcting. Eventually bad ideas will be discarded and good ideas will catch on. By recognizing and rewarding application as a form of scholarship, however, the speed of progress can be increased. There will be a much greater understanding of the problems that have been solved and the problems that remain. In the end making these changes will make foundations of research more robust, improving both past and future knowledge.

Bibliography

[1] E. L. Boyer, "Scholarship reconsidered: Priorities of the professoriate." 1990.

[2] L. Hafer and A. E. Kirkpatrick, "Assessing open source software as a scholarly contribution," *Communications of the ACM*, vol. 52, no. 12, pp. 126–129, 2009.

[3] C. R. Association *et al.*, "Evaluating computer scientists and engineers for promotion and tenure," 1999.

[4] R. D. Peng, "Reproducible research in computational science," *Science (New York, Ny)*, vol. 334, no. 6060, p. 1226, 2011.

[5] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering, ICSE 2015,(to appear)*, 2015.

[6] N. Juristo and O. S. Gómez, "Replication of software engineering experiments," in *Empirical software engineering and verification*. Springer, 2012, pp. 60–88.

[7] J. Howison and J. D. Herbsleb, "Incentives and integration in scientific software production," in *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 2013, pp. 459–470.

[8] F. Brooks, "Academic careers for experimental computer scientists and engineers," 1994.

[9] J. N. Hooker, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, vol. 1, no. 1, pp. 33–42, 1995.

[10] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *Software Engineering, IEEE Transactions on*, vol. 40, no. 6, pp. 603–616, 2014.

[11] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[12] "KLEE project," https://github.com/klee/klee/graphs/contributors, accessed: 2015-7-3.

[13] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[14] "Google scholar," http://scholar.google.com/, accessed: 2015-6-4.

[15] "S2E project," https://github.com/dslab-epfl/s2e/blob/master/S2E-AUTHORS, accessed: 2015-7-3.

[16] "Cloud9 project," https://github.com/dslab-epfl/cloud9/blob/master/CLOUD9-AUTHORS, accessed: 2015-7-3.

[17] "KleeNet project," http://www.comsys.rwth-aachen.de/research/projects/kleenet/, accessed: 2015-7-3.

[18] "GKLEE project," https://github.com/Geof23/Gklee/graphs/contributors, accessed: 2015-7-3.

[19] "KLEE symbolic virtual machine github repo," https://github.com/klee/klee, accessed: 2015-4-30.

[20] N. McDonald and S. Goggins, "Performance and participation in open source software on GitHub," in *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '13. New York, NY, USA: ACM, 2013, pp. 139–144. [Online]. Available: http://doi.acm.org/10.1145/2468356.2468382

[21] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[22] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Postconditioned symbolic execution," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 1–10.

[23] S. Chattopadhyay, P. Eles, and Z. Peng, "Automated software testing of memory performance in embedded GPUs," in *Proceedings of the 14th International Conference on Embedded Software*. ACM, 2014, p. 17.

[24] K. Cong, F. Xie, and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 303–310.

[25] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 1143–1152.

[26] C. Lucas, S. Elbaum, and D. S. Rosenblum, "Detecting problematic message sequences and frequencies in distributed systems," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 915–926, 2012.

[27] L. Madeyski and B. Kitchenham, "Reproducible research–what, why and how."

[28] "Computational Infrastructure for Operations Research (COIN-OR)," http://www.coin-or.org/, accessed: 2015-7-3.

[29] D. S. Katz, "Citation and attribution of digital products: Social and technological concerns," *WSSSPE (Working towards Sustainable Software for Science: Practice and Experiences) at Supercomputing 2013*, 2013.

[30] "Google summer of code," https://developers.google.com/open-source/gsoc/, accessed: 2015-7-3.

[31] "National science foundation," http://www.nsf.gov/, accessed: 2015-7-3.

[32] "Software sustainability institute," http://www.software.ac.uk/, accessed: 2015-7-3.

[33] C. Barrett, M. Deters, A. Oliveras, and A. Stump, "Design and results of the 4th annual satisfiability modulo theories competition (smt-comp 2008)," *To appear*, vol. 6, 2008.

[34] D. Beyer, "Status report on software verification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 373–388.

[35] S. Bauersfeld, T. E. Vos, and K. Lakhotia, "Unit testing tool competitions–lessons learned," in *Future Internet Testing*. Springer, 2014, pp. 75–94.

[36] J. Howison and J. Bullard, "How is software visible in the scientific literature?"

[37] S. Krishnamurthi and J. Vitek, “The real software crisis: repeatability as a core value,” *Communications of the ACM*, vol. 58, no. 3, pp. 34–36, 2015.

[38] T. Proebsting and A. M. Warren, “Repeatability and benefaction in computer systems research,” 2015.

[39] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: whitebox fuzzing for security testing,” *Queue*, vol. 10, no. 1, p. 20, 2012.

[40] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” in *Botnet Detection*. Springer, 2008, pp. 65–88.

[41] “SPF project,” http://babelfish.arc.nasa.gov/hg/jpf/jpf-symbc, accessed: 2015-7-3.

[42] “JCUTE project,” https://github.com/osl/jcute, accessed: 2015-7-3.

[43] N. Tillmann and J. De Halleux, “Pex–white box test generation for. net,” in *Tests and Proofs*. Springer, 2008, pp. 134–153.

[44] G. Li, I. Ghosh, and S. P. Rajan, “KLOVER: A symbolic execution and automatic test generation tool for c++ programs,” in *Computer Aided Verification*. Springer, 2011, pp. 609–615.

[45] K. Sen and G. Agha, “CUTE and JCUTE: Concolic unit testing and explicit path model-checking tools,” in *Computer Aided Verification*. Springer, 2006, pp. 419–423.

[46] “CREST project,” https://github.com/jburnim/crest/graphs/contributors, accessed: 2015-7-3.

[47] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 2008, pp. 443–446.

[48] "CIVL project," http://vsl.cis.udel.edu/civl/, accessed: 2015-7-3.

[49] "Fuzzball project," https://github.com/bitblaze-fuzzball/fuzzball/graphs/contributors, accessed: 2015-7-3.

[50] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur, "The yogi project: Software property checking via static analysis and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 178–181.

[51] "CAUT project," https://github.com/tingsu/caut-lib/graphs/contributors, accessed: 2015-7-3.

[52] O. Crameri, R. Bachwani, T. Brecht, R. Bianchini, D. Kostic, and W. Zwaenepoel, "Oasis: Concolic execution driven by test suites and code modifications," Technical Report LABOS-REPORT-2009-002, EPFL, Tech. Rep., 2009.

[53] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[54] "uClibc," http://www.uclibc.org/, accessed: 2015-5-27.

[55] P. Schrammel, T. Melham, and D. Kroening, "Chaining test cases for reactive system testing," in *Testing Software and Systems*. Springer, 2013, pp. 133–148.

[56] V. Ganesh, A. Kieżun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. Ernst, "HAMPI: A string solver for testing, analysis and vulnerability detection," in *Computer Aided Verification*. Springer, 2011, pp. 1–19.

[57] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic verification and test generation for gpus," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 215–224.

[58] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 186–196.

[59] J. C. M. Carreira, R. Rodrigues, G. Candea, and R. Majumdar, "Scalable testing of file system checkers," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 239–252.

[60] T. Kuchta, C. Cadar, M. Castro, and M. Costa, "Docovery: toward generic automatic document recovery," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 563–574.

[61] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 266–276.

[62] W. Jin and A. Orso, "BugRedux: reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 474–484.

[63] W. N. Sumner, T. Bao, and X. Zhang, "Selecting peers for execution comparison," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 309–319.

[64] H. Ma, X. Ma, W. Liu, Z. Huang, D. Gao, and C. Jia, "Control flow obfuscation using neural network to fight concolic testing." Proceedings of the 10th International ICST Conference on Security and Privacy in Communication Networks (SecureComm 2014), 2014.

[65] "STP project," https://github.com/stp/stp/blob/master/AUTHORS, accessed: 2015-7-3.

[66] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 58.

[67] J. Hoffmann and J. Koehler, "A new method to index and query sets," in *IJCAI*, vol. 99, 1999, pp. 462–467.

[68] "KLEE LLVM execution engine website," http://klee.github.io/, accessed: 2015-5-21.

[69] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and applications of satisfiability testing*. Springer, 2004, pp. 502–518.

[70] A. Simon and A. King, "Taming the wrapping of integer arithmetic," in *Static Analysis*. Springer, 2007, pp. 121–136.

[71] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering–a systematic literature review," *Information and software technology*, vol. 51, no. 1, pp. 7–15, 2009.

[72] C. Song, A. Porter, and J. S. Foster, "itree: efficiently discovering high-coverage configurations using interaction trees," *Software Engineering, IEEE Transactions on*, vol. 40, no. 3, pp. 251–265, 2014.

[73] R. Bachwani, O. Crameri, R. Bianchini, D. Kostic, and W. Zwaenepoel, "Sahara: Guiding the debugging of failed software upgrades," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 263–272.

[74] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang, "Golden implementation driven software debugging," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 177–186.

[75] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala, "Finding latent performance bugs in systems implementations," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 17–26.

[76] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles, "Instrumentation of annotated c programs for test generation," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 105–114.

[77] L. Luu, S. Shinde, P. Saxena, and B. Demsky, "A model counter for constraints over unbounded strings," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 57.

[78] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and repairing configuration inconsistencies in large-scale system software," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 531–551, 2012.

[79] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: a trustworthy in-kernel interpreter infrastructure," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 33–47.

[80] N. S. Evans, A. Benameur, and M. C. Elder, "Large-scale evaluation of a vulnerability analysis framework," in *Proceedings of the 7th USENIX conference on Cyber Security Experimentation and Test*. USENIX Association, 2014, pp. 3–3.

[81] C. Oh, M. Schaf, D. Schwartz-Narbonne, and T. Wies, "Concolic fault abstraction," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 135–144.

[82] H. Seo and S. Kim, "Predicting recurring crash stacks," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 180–189.

[83] D. Zhang, D. Liu, W. Wang, J. Lei, D. Kung, and C. Csallner, "Testing C programs for vulnerability using trace-based symbolic execution and satisfiability analysis," *Proceedings of the 40th Annual IEEE/IFIP InternationalConference on Dependable Systems and Networks (DSN'10). Piscataway, USA: IEEE Press*, pp. 321–338, 2010.

[84] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 622–631.

[85] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 166–176.

[86] S. Falke, F. Merz, and C. Sinz, “Extending the theory of arrays: memset, memcpy, and beyond,” in *Verified Software: Theories, Tools, Experiments*. Springer, 2014, pp. 108–128.

[87] P. Dinges and G. Agha, “Solving complex path conditions through heuristic search on induced polytopes,” in *SIGSOFT FSE*, vol. 14, 2014.

[88] S. Bucur, J. Kinder, and G. Candea, “Prototyping symbolic execution engines for interpreted languages,” in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 239–254.

[89] Y. Kim, M. Kim, and N. Dang, “Scalable distributed concolic testing: a case study on a flash storage platform,” in *Theoretical Aspects of Computing–ICTAC 2010*. Springer, 2010, pp. 199–213.

[90] H. Seo and S. Kim, “How we get there: a context-guided search strategy in concolic testing,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 413–424.

[91] M. Kim, Y. Kim, and G. Rothermel, “A scalable distributed concolic testing approach: An empirical evaluation,” in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 340–349.

[92] Y. Zhang, Z. Chen, J. Wang, W. Dong, and Z. Liu, “Regular property guided dynamic symbolic execution,” in *ICSE*, 2015.

[93] W. Le, “Segmented symbolic analysis,” in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 212–221.

[94] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *37th International Conference on Software Engineering, ICSE*, vol. 15, 2015.

[95] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 337–348, 2012.

[96] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA*, 2014.

[97] S. Ahn and S. Malik, "Modeling firmware as service functions and its application to test generation," in *Hardware and Software: Verification and Testing*. Springer, 2013, pp. 61–77.

[98] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic, "A soft way for openflow switch interoperability testing," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 265–276.

[99] C. Sturton, R. Sinha, T. H. Dang, S. Jain, M. McCoyd, W. Y. Tan, P. Maniatis, S. A. Seshia, and D. Wagner, "Symbolic software model validation," in *Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on*. IEEE, 2013, pp. 97–108.

[100] E. T. Barr, T. Vo, V. Le, and Z. Su, "Automatic detection of floating-point exceptions," in *ACM SIGPLAN Notices*, vol. 48, no. 1. ACM, 2013, pp. 549–560.

[101] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, “Static analysis driven cache performance testing,” in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 319–329.

[102] A. Slowinska, T. Stancescu, and H. Bos, “Body armor for binaries: Preventing buffer overflows without recompilation.” in *USENIX Annual Technical Conference*, 2012, pp. 125–137.

[103] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, “Directed symbolic execution,” in *Static Analysis*. Springer, 2011, pp. 95–111.

[104] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang, “A synergistic analysis method for explaining failed regression tests,” in *International Conference on Software Engineering*, 2015.

[105] S.-K. Huang, H.-L. Lu, W.-M. Leong, and H. Liu, “Craxweb: Automatic web application testing and attack generation,” in *Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 208–217.

[106] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, “Reproducing field failures for programs with complex grammar-based input,” in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 163–172.

[107] “Google,” https://www.google.com, accessed: 2015-7-5.

[108] “BugRedux project,” http://www.cc.gatech.edu/~{}wjin6/mypage/bugredux.html, accessed: 2015-7-5.

[109] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures.” in *NDSS*. Citeseer, 2011.

[110] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[111] "Guidelines for packaging AEC submissions," http://www.artifact-eval.org/guidelines.html, accessed: 2015-7-3.

[112] D. S. Katz, S.-C. T. Choi, H. Lapp, K. Maheshwari, F. Löffler, M. Turk, M. D. Hanwell, N. Wilkins-Diehr, J. Hetherington, J. Howison *et al.*, "Summary of the first workshop on sustainable software for science: Practice and experiences (WSSSPE1)," *arXiv preprint arXiv:1404.7414*, 2014.

[113] J. P. Ioannidis, "Why most published research findings are false," *Chance*, vol. 18, no. 4, pp. 40–47, 2005.

[114] M. G. Knepley, J. Brown, L. C. McInnes, and B. Smith, "Accurately citing software and algorithms used in publications," Technical Report 785731, figshare, 2013. http://dx. doi. org/10.6084/m9. figshare. 785731, Tech. Rep., 2013.

[115] R Core Team, *R: A Language and Environment for StatisticalComputing*, R Foundation for Statistical Computing, Vienna, Austria, 2015. [Online]. Available: http://www.R-project.org

[116] M. J. Saltzman, "Open-source software and the Computational Infrastructure for Operations Research (COIN-OR)," in *Encyclopedia of Operations Research and Management Science*. Springer, 2013, pp. 1070–1081.

[117] "Github," https://github.com/, accessed: 2015-7-31.

[118] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[119] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller, "Replication's role in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 365–379.

[120] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[121] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standardversion 2.5," 2010.

Appendix A

Factorization of Bit-Vectors

In this appendix, we discuss how factorization works in general, and discuss the particular details of adapting it to bit-vector constraints [121]. This information is used to supplement the discussion in Section 2.2. The information is used in Section 2.2 to explain how the first filter in KLEE's *Solver Chain* works.

In concept, factorization is fairly simple. It works by breaking down a constraint set into smaller independent subsets where each subset is only composed of constraints with variables that reference each other. If all of the subsets resulting from factorization are SAT, then the original constraint set is also SAT. If, instead, one of the subsets is UNSAT, then the original constraint set is also UNSAT. For example, given the constraint set $\{(a \neq 4) \wedge (a \neq 5) \wedge (x = 10) \wedge (x = y) \wedge (y = z)\}$, the constraint $(a \neq 4)$ would be put into the same subset as $(a \neq 5)$ since they deal with the variable a. In addition, the constraints $(x = 10)$, $(x = y)$, and $(y = z)$ would be put into a second subset since the constraint on x transitively affects the value of z. Since each independent subset is SAT, the original constraint set is SAT as well.

When factorizing bit-vector constraints, as long as the indices of the arrays being accessed are concrete, then everything is the same as the concept explained above. This is because we can separate the individual elements of arrays, since they can be identified as different variables. For example, the constraint set $\{(arr[1] < 6) \wedge (arr[2] > 8)\}$ could be broken down into two separate subsets because the value contained in $arr[1]$ could never affect the value in $arr[2]$.

When dealing with symbolic indices, however, things become more complicated. Consider what happens when we add the constraint $(arr[x] < 10)$ to the constraint set above. In this case, it is possible that $arr[x]$ is referring to $arr[1]$. It is also possible that $arr[x]$ is referring to $arr[2]$. Therefore, in order to be safe, SymExe tools will group these three accesses into a single constraint set. This illustrates how symbolic indices can cause otherwise independent subsets to be grouped together. Indeed, it is common for a single symbolic index to cause dozens of otherwise independent subsets to collapse into a single, much larger, constraint set. While this is unfortunate, since it makes it much more expensive for the SMT solver to produce an answer, it is the simplest way to guarantee correct results.

In order to formally present how bit-vector constraints with symbolical indices have traditionally been handled, we will expand upon the work presented in [66]. In this paper, the authors used a graph reachability approach to define how to factorize linear integer constraints [121]. We first need to define three functions.

Let *sym* be a function that takes a constraint as an input and returns all of the array accesses that involve symbolic indices in the constraint as a set of pairs $\langle x, y\rangle$ where x is the array that is accessed and y is the symbolic index that is accessed. For example, $sym\big((arr[x] = arr[7])\big)$ would return $\{\langle arr, x\rangle\}$

Let *access* be a function that takes a constraint as an input and returns all of the array accesses in the constraint as a set of pairs $\langle x, y\rangle$ where x is the array that is accessed and y is the index, symbolic or concrete, that is accessed. For example $access\big((arr[x] = arr[7])\big)$ would return $\{\langle arr, x\rangle, \langle arr, 7\rangle\}$.

$$\{(x,y)|x,y \in CS, access(x) \cap access(y) \neq \emptyset \vee f(sym(x)) \cap f(access(y)) \neq \emptyset\}$$

Figure A.1: Traditional Factorization Definitions

Finally, let f be a function that takes a set of pairs and returns a set consisting only of the first element. For example, $f\big(\langle 1,2\rangle\big)$ would return 1.

With these helper functions defined, we will create the function $traditionalBVFactor$. Let $traditionalBVFactor$ be a function that takes two inputs: a constraint C and a constraint set CS. The function returns all constraints in the constraint set CS that reference C (directly or transitively) and therefore should be in the same independent subset as C. We define the function as follows. Create a graph $G = (V, E)$ where the vertices V are the constraints in the constraint set CS. We define the set of edges E using the formula seen in Figure A.1. Return the set of constraints that are reachable in G from the input constraint C.

Appendix B

Further Factorization of Bit-Vectors

In this appendix, we present a more precise way to factorize bit-vector constraint sets. This information is used to supplement the discussion in Section 2.8. The information is used in Section 2.8 to explain how we used the output from KLEE to identify a way to reduce the number of times an SMT solver was required during a normal execution. This reduction in SMT invocations reduces the time required to use KLEE.

The main idea for this technique is to use constraints within the constraint set to prove that symbolic index of an array (of the form $arr[x]$) could not access other parts of an array. This increased precision allows us to further factorize constraint sets, meaning that the solver is required to handle simpler and fewer constraint sets.

For example, consider the constraint set $\{(arr[x] = 2) \wedge (x < 3) \wedge (arr[4] \leq 127)\}$. Given that $(x < 3)$, we know that x could not possibly be 4 and therefore $arr[x]$ could not possibly refer to $arr[4]$. Therefore, the constraint set can be broken down into 2 independent subsets ($\{(arr[x] = 2) \wedge (x < 3)\}$ and $\{(arr[4] \leq 127)\}$) rather than the single constraint set that would normally be caused by a symbolic index.

Formally, our approach is designed to create fewer edges then the naïve definition presented in Appendix A and therefore limit the number of constraints that can be

1. $$\{(x,y)|x,y \in CS, access(x) \cap access(y) \neq \emptyset\}$$

2. $$\{(x,y)|x,y \in CS, f(sym(x)) \cap f(access(y) \neq \emptyset \wedge SAT(s(sym(x)), s(access(y)), r(x,G_1), r(y,G_1))\}.$$

Figure B.1: Factorization Definitions

reached. In order to define how our process works, we need to define functions in addition to those defined in Appendix A.

First, let s be a function that takes a set of pairs and returns a set consisting only of the second element. For example, $s\big(\langle 1,2\rangle\big)$ would return 2.

Second, let $SAT\big(a,b,x,y\big)$ be a function that takes two values, a and b, and two constraint sets, x and y, and returns *true* if a could possibly be equal to b given the constraint in x and y and *false* otherwise. For example, $SAT\big(x,y,\{(x>7)\},\{(y<6)\}\big)$ would return *false* since x could never be equal to y.

Finally, let r be a function that takes a constraint and a graph of constraints as inputs and returns a constraint set of all constraints that are reachable within the graph from the input constraint.

With this minutia out of the way, we create the function *newBVFactor* to highlight our intuition. Let *newBVFactor* be a function that takes two inputs: a constraint C and a constraint set CS. The function returns all constraints in the constraint set CS that reference C (directly or transitively) and therefore should be in the same independent subset as C. We define the function as follows. Create a graph $G_1 = (V_1, E_1)$ where the vertices V are the constraints in the constraint set CS. We define the set of edges E_1 as seen in item one of Figure B.1. After this, we create a second graph $G_2 = (V_2, E_2)$ where the vertices V_2 are the constraints in the constraint set CS. We define the set of edges E_2 as seen in item two of Figure B.1. Finally, we create a graph $G_3 = (V_3, E_3)$ where the vertices V_3 are the constraints in the constraint set CS and $E_3 = E_1 \cup E_2$. Return the set of constraints that are reachable in G_3 from the input constraint C.

Our definition differs in the way we handle arrays that have symbolic indices. In the naïve algorithm (discussed in Appendix A), all constraints that contain an array that has a symbolic index are automatically grouped into a single constraint set. Here, instead, we check to see whether the constraints that are directly associated with a particular symbolic index might end up making it impossible for it to be equal to another value. If this is the case, then the two constraints can remain separate.

When testing this more accurate technique on the constraint sets sent by KLEE to the SMT solver[1], we found that there were many opportunities for further factorization.[2]After further separating these "naïve" subsets, we discovered that there was up to a 3X reduction in the number of times an SMT solver was required to solve all of the constrain sets. The reason for this extreme reduction is that the more accurate factorization led to previously indivisible, unique constraint sets being broken down into smaller, precomputed pieces.

Unfortunately, the overhead required to identify these independences is many times the cost of just sending the original constraint set to the SMT solver. When attempting to do this for every constraint set, the time required was an order of magnitude greater than the naïve approach. Given the costs associated with proving independence, it became clear that a more light weight technique was required.

[1] For these experiments, we used the *Generic Regression* setup. We tested our approach on the output of 13 bin-util programs

[2] Code and instructions on how to re-run these experiments available at https://github.com/holycrap872/green-solver

Appendix C

A Heuristic for Further Factorization of Bit-Vectors

In this appendix, we present the heuristic we developed in order to capitalize on the opportunities for further factorization discussed in Appendix B. This information is used to supplement the discussion in Section 2.8. The information is used in Section 2.8 to explain how we used the formal definition for further factorization (described in Appendix B) to come up with a heuristic that could be used to reduce the execution costs of KLEE.

In order to understand the heuristic, it is necessary to understand a few particular aspects of KLEE. As discussed in Section 2.2, KLEE builds the constraint sets associated with a particular path incrementally. Therefore, the newest constraint in a constraint set can be differentiated from the older constraints. Further, by removing the newest constraint from the original constraint set, what remains is the constraint set associated with the parent of the current state. As discussed in Section 2.4.2, the solution to this constraint set is already cached and can be quickly accessed.

Consider the example $\{(arr[x] = 2) \wedge (x < 3) \wedge (arr[4] \leq 127)\}$. From this constraint set we can deduce several things. First, based on KLEE's conventions, we know that $(arr[4] \leq 127)$ is the newest constraint added to the path, since it is the last element in the list of constraints. Second, we know that the constraint set $\{(arr[x] = 2) \wedge (x < 3)\}$ is the parent of the full constraint set and therefore is both SAT and has its solution stored somewhere in the *Solver Chain*. Finally, while we cannot be sure $\{(arr[4] \leq 127)\}$ is in the *Solver Chain*, its small size increases the chances of a cache hit. Even if there isn't a cache hit, the small size means it will be faster for the solver to handle.

Understanding these facts allowed us to come up with a heuristic that is both simple and effective. Whenever the most recent constraint in a constraint set is involved in a factor that has an array with a symbolic index, we assume that, given enough time, they could be proven to be independent. By assuming this, we can simply find the solutions to the two factors that would result from this assumption.[1]We then stitch these two solutions together, just like in the *Complete Solution Re-computation Optimization* in Section 2.4. Upon creating a single solution, we can then evaluate it on the entire constraint set. If our assumption that the factors were independent is correct, then this synthesized solution is guaranteed to evaluate to *True*. If, instead, the new solution evaluates to *False*, then we simply forward the constraint set to the SMT solver.

In order to illustrate how this works, we will again use the example above. We first assume that $\{arr[4] \leq 127\}$ is in a different factor from $\{(arr[x] = 2) \wedge (x < 3)\}$. Therefore, we attempt to solve each factor separately. For the factor $\{(arr[4] \leq 127)\}$ we could get the solution $\{0, 0, 0, 0, 127\}$. For the factor $\{(arr[x] = 2) \wedge (x < 3)\}$ we could get the solution $\{2, 0, 0, 0, 0\}$. Stitching these two solutions together produces the complete solution $\{2, 0, 0, 0, 127\}$. Finally, we check to see if this synthesized solution evaluates to *True* on the original constraint set. In this case it does, meaning we don't have to send the

[1] If a solution does not exist for either factor, than the entire constraint set *must* be UNSAT. In this case, we could skip the rest of the heuristic and simply return UNSAT.

original, full constraint set to the solver. Instead, there are two possible outcomes. In one case, if $(arr[4] \leq 127)$ is in a cache somewhere in the *Solver Chain*, then the solver doesn't need to be invoked at all. If, instead $(arr[4] \leq 127)$ is not stored anywhere in the *Solver Chain*, the fact that we have reduced the number of constraints associated with it means that the solver can return an answer more quickly.

Appendix D

Evaluation of a Heuristic for Further Factorization of Bit-Vectors

In this appendix, we evaluate how the heuristic explained in Appendix C affected the performance of KLEE. This information is used to supplement the discussion in Section 2.8. The information is used in Section 2.8 to explain how the heuristic we developed ended up being completely dependent on a particular command-line option in KLEE. In the end, it turns out that this particular option was erroneously enabled in the versions we were examining since was is very costly while providing little gain for the average user.

In our evaluation we tested the performance of a KLEE instance with our heuristic and compared it with a KLEE instance without our heuristic. As mentioned in 2.8, however, at the point we started to evaluate our technique, we had *just* discovered the existence of the *no-prefer-cex option*. Therefore, due to our reservations about the usefulness of the technique, we ran two experiments: one with the *no-prefer-cex* enable (the default behavior), and one with *no-prefer-cex* disabled. Other than this difference, both tests used the *Generic Regression* setup discussed in Section 2.5.

In addition, it should be noted that the we were not comparing our heuristic against a version of KLEE untouched by any of our other optimizations. We instead were comparing a version of KLEE that had the *Quick Cache Bug*, the *Complete Solution Re-computation Optimization*, and the *Array Factory Bug* (described in Section 2.4) against a version of KLEE that included all of these improvements *plus* our heuristic. The reason for this decision was two-fold. First, our heuristic is dependent on these three improvements. Second, these improvements are the ones that we had discovered prior to implementing our heuristic (we found the other improvements described in Section 2.4 later). Comparing our heuristic with all of these improvements against a version of KLEE without any of the improvements would give an unfair picture of the capabilities of our technique.

We set up our experiments in the same way as described in Section 2.5. The "finish line" for each experiment was determined by running the instance of KLEE that had our heuristic enabled for one hour configured using the *Generic Regression* setup. We recorded the number of forks that this experiment was able to reach and then used that as the "finish line" for the instances without our heuristic. For test subjects, we used the core-utils programs that were presented in the original KLEE paper [13].

D.1 Results

The results for our experiments can be seen in Table D.1. Both box plots show the relative performance increase of an instance of KLEE using our heuristic relative to an instance of KLEE not using the heuristic. The label *Enabled* shows the improvement when *no-prefer-cex* is enabled. The label *Disabled* shows the improvement when *no-prefer-cex* is disabled.

When *no-prefer-cex* is enabled, the average improvement is 2.2X and the median improvement is just over 2X. The only program that runs slower is *yes* which goes from 903 seconds to 3792 seconds. The greatest improvement is seen in *dircolors* which goes

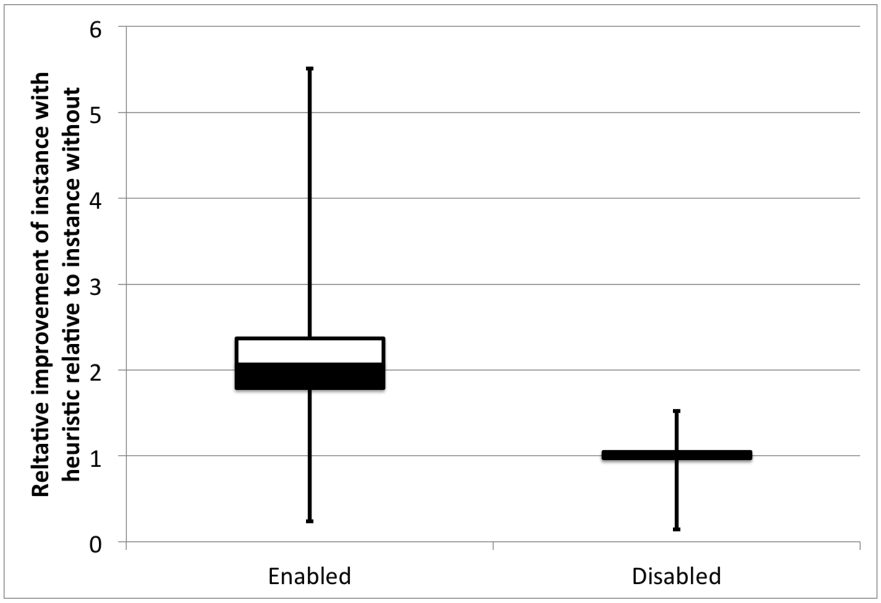


Figure D.1: Box-plots showing the relative improvement of an instance of KLEE that uses our heuristic over an instance of KLEE without our heuristic. The *Enabled* plot illustrates the relative improvement when *no-prefer-cex* is enable (its default position). The *Disabled* plot illustrates the relative improvement when *no-prefer-cex* is disabled

from 45660 seconds to 8281 seconds.

When *no-prefer-cex* is disabled, however, the average and median improvements are less than 1%. The worst degradation in performance is *yes*, which goes from 518 to 3603 seconds. The largest improvement was seen in *wc* which went from 3497 seconds to 2292 seconds.

D.2 Discussion

The results of our evaluation shows that our heuristic is completely dependent on the *no-prefer-cex* option being enabled. As previously discussed in Section 2.4, we determined that this option was unnecessary for the average user and it has since been switched to default disabled in the main KLEE distribution. Therefore, our heuristic, rather than providing a generalizable improvement to all SymExe engines, was instead dependent on

```
for each symbolic array in state
  for each byte in symbolic array
    constraint = $(0 \leq byte \leq 127)$
    if SAT(state.constraintSet() $\wedge$ constraint)
      state.addToConstraintSet(constraint)
return state
```

Figure D.2: *no-prefer-cex* Algorithm

an unnecessary default option in a particular SymExe tool.

In order to understand the reason for this drastic shift, it is important to understand what the logic controlled by the *no-prefer-cex* option does. As previously explained, the goal of the *no-prefer-cex* logic is to sanitize the inputs in every test case in order to make them as human-readable as possible. Figure D.2 shows the basic logic. In general, for each byte of a test case, KLEE attempts to bound it to be between the ranges of 0 and 127. If the character can be bound in this way with the overall test case remaining SAT, then the constraint on the character is retained. Otherwise, it is simply dropped.

Knowing this, we can examine the running example in Appendix C. We chose this example because it is representative of the type of constraint sets generated by KLEE when *no-prefer-cex* is on. First there is almost always a series of complicated constraints that contain arrays with symbolic indices, represented by the first two constraints, $(arr[x] = 2)$ and $(x < 3)$. This can be thought of as the path that the state took through the program. Second, there is an attempt to bound a character in the array to be human readable, as seen by the constraint $(arr[4] \leq 127)$. Finally, there are some elements in the array that were never referenced on the path the state took through the test program. This is represented by the fact that x could never equal 4. This means our optimistic assumption that the two factors were independent is correct and we are able to generate a satisfying solution from these two smaller pieces.

A realistic example of this process occurs when testing the core-utils program *ls* with a 10 element symbolic array as suggested by KLEE's website [68]. There are many command-line options to *ls* that do not require 10 characters. Therefore, the latter elements in the array are often never referenced. Attempts to constrain these latter elements of the array would, therefore, be independent from the computation done on the beginning of the array. Once the *no-prefer-cex* option is disabled, however, these opportunities vanish completely.