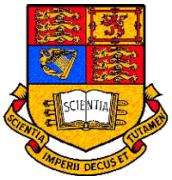


Imperial College of Science, Technology and Medicine

Final year project report

Testing symbolic execution engines

Timotej Kapus

supervised by

Dr. Cristian Cadar

June 18, 2017

Abstract

Symbolic execution has seen significant interest in the last few years, across a large number of computer science areas, such as software engineering, systems and security, among many others. As a result, the availability and correctness of symbolic execution tools is of critical importance for both researchers and practitioners. In this report, we present two testing technique targeted towards finding errors in both concrete and symbolic execution modes of symbolic executors. The first testing technique relies on a novel way to create program versions, which combined with existing program generation techniques and appropriate oracles, enable differential testing of symbolic execution engines. Second, the differential approach is extended to enable metamorphic testing, where we create two equivalent versions of a program, through semantics preserving transformations. Three innovative ways of comparing two multi-path symbolic executions are then presented, which provide an effective mechanism for catching bugs in symbolic execution engines. We have evaluated the techniques via case studies on the KLEE, Crest and FuzzBALL symbolic execution engines, where it has found more than 20 different bugs, including subtle errors involving structures, division, modulo, casting, vector instructions and more, as well as issues having to do with constraint solving and replaying test cases.

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Cristian Cadar, for his time and effort. His advice and guidance made this project truly an enjoyable experience. In addition I would like to thank the entire *Software reliability group* for listening to my problems and suggesting solutions every week. I would also like to thanks my second marker, Dr. Alastair F. Donaldson, for early advice on metamorphic testing. Finally I would like to thank my friends and housemates for their continuing support, especially Max and Eva, who took their time to read this report.

Contents

Introduction

Symbolic execution is a form of program analysis which systematically explores all the possible behaviors of the program it is analyzing. In simple terms, symbolic execution marks some program inputs or variables as "symbolic", which means they can take *any* value. Then it modifies all the interactions between the program and those symbolic variables. For example when the program branches on a symbolic variable, symbolic execution follows both branches, if they are feasible. This can be used as an automated software testing mechanism. Symbolic execution can be applied to automated software testing in several ways, from showing two implementations of the same specification have the same behavior, to automated test case generation. Consequently, symbolic execution has great potential to ease the burden on human developers when it comes to testing and writing test cases manually.

Despite establishing itself as an effective testing method with several companies reporting successful adoption [8], there are still many research groups improving it. Key to this progress has been the availability of symbolic execution tools, which allows industrialists to use and extend the technique, and allow researchers to experiment with new ideas. Notable examples include open-source tools Crest [16], KLEE [5], FuzzBALL [32] and Symbolic JPF [1], and the closed-source tools PEX [44] and SAGE [23].

The quality of symbolic execution tools is essential for continuous progress in this area; both for researchers, so they don't get bogged down with avoidable bugs and for industrialists who want to improve the testing of their software. The reliability and correctness of symbolic execution is especially important for the latter use case. As vividly illustrated by Cadar and Donaldson [4], bugs in program analyzers can go catastrophically wrong, when they give false confidence to their users. However, despite their importance to the software engineering field and beyond, there is currently no systematic effort to test symbolic execution engines.

There are several crucial features of modern symbolic execution engines for which this work aims to develop effective automated testing techniques. For example, mixed concrete-symbolic execution [22, 6] significantly improves the performance of symbolic executor by running non-symbolic parts of the program concretely or normally. A reliable tool has to correctly implement both concrete and symbolic execution modes. On the concrete side, symbolic execution engines either embed an interpreter for the language they analyze: Java bytecode in the case of Symbolic Java PathFinder, LLVM IR in the case of KLEE, and x86 code in the case of FuzzBALL and SAGE, or modify and instrument the code statically: e.g. both Crest and EXE first transform the program using CIL [37], and instrument it at that level. As shown in Chapter 2, the execution fidelity of the interpretation or instrumentation can be effectively tested by adapting program generation and differential and metamorphic testing techniques employed in compiler testing [47, 27].

On the symbolic side, the accuracy of the constraints gathered on each explored path is of critical importance if symbolic execution tools are to avoid exploring infeasible paths (which report spurious errors) and generate inputs, that when run natively, follow the same path as during symbolic execution. The approach presented tests the symbolic execution mode in two ways. First, the inputs are constrained to follow a single path. The key idea is simple, but effective: starting from an automatically-generated or real program, program versions are created which produce an output for one path only (say when x=10) and then symbolic execution is checked to correctly follow that path and output the expected results. Second, by running small programs symbolically, and devising clever techniques that verify equivalent symbolic runs.

In both the concrete and symbolic cases, there are two approaches presented based on differential

testing (§2) and metamorphic testing (§3). With differential testing the symbolic execution run is crosschecked against a native run of the same program. Metamorphic testing takes a different approach by crosschecking a symbolic run of a program with a symbolic run of a transformed version of the program that behaves equivalently to the original. There are several transformations presented that achieve this equivalent transformation.

In both cases an effective crosschecking between the two runs needs to employ effective and inexpensive oracles. That is, oracles that find important bugs and do not add a significant runtime cost. Oracles are small programs that monitor the executions and decide whether there might be a bug present. Broadly speaking there are five different oracles used in this report. First the symbolic execution tool is checked not to crash. Second the two runs are checked to produce identical outputs. The function call chain oracle ensures the two executions generate the same sequence of function calls, while coverage oracle establishes whether the two runs achieved the same coverage. Finally the runtime of the two executions is recorded and any unexpected performance slowdown is flagged by the performance oracle. In all the cases the approaches also take advantage of advances in automatic program generation [47], which allows quick creation of small deterministic programs without undefined and unspecified behavior, which is essential to find useful bugs and perform experiments on mass scale.

Whenever a generated program finds a bug in a symbolic execution tool, existing program source code reducing techniques [39] are used in combination with the oracles in order to obtain a small program (with fewer than 30 lines of code) that forms an easy-to-understand, reproducible bug report. The approach was applied to three symbolic execution engines: KLEE, Crest and FuzzBALL, where it has found several serious functionality bugs, most of which have already been fixed or confirmed by the developers.

In summary, the main contributions of this work are:

1. The first adaptation of differential and metamorphic testing techniques targeted towards finding errors in both the concrete and symbolic execution modes of symbolic execution engines.

2. A novel way to create program versions, in three different modes, which combined with existing program generation techniques and appropriate oracles, enables differential testing within a single symbolic execution engine.

3. A toolkit implementing all the approaches presented together with comprehensive case studies on three symbolic execution engines—KLEE, Crest and FuzzBALL—implementing different styles of symbolic execution (e.g. concolic variant vs. keeping all paths in memory, interpretation vs. instrumentation, etc.) and operating at different levels (source, LLVM bitcode and binary). The approach found over 20 important bugs in these engines.

This report is structured as follows. Chapter 1 gives the background on symbolic execution, differential and metamorphic testing as well as other related topics. Chapter 2 starts by giving the fundamentals of the approach presented with a focus on differential testing (§2.1), then moves on to describing the implementation details of the toolkit used to perform the experiments (§2.2). Finally Section 2.3 presents the case studies for the differential testing case studies on KLEE, Crest and FuzzBALL. Chapter 3 then extends the techniques presented by Chapter 2 to metamorphic testing. It starts of by describing additional challenges with program generation (§3.1), the equivalent behavior transformations (§3.2) and approaches to dealing with comparing two symbolic executions (§3.3). Finally Section 3.4 concludes with the evaluation of metamorphic testing techniques on the case studies of KLEE, Crest and FuzzBALL.

Chapter 1

Background

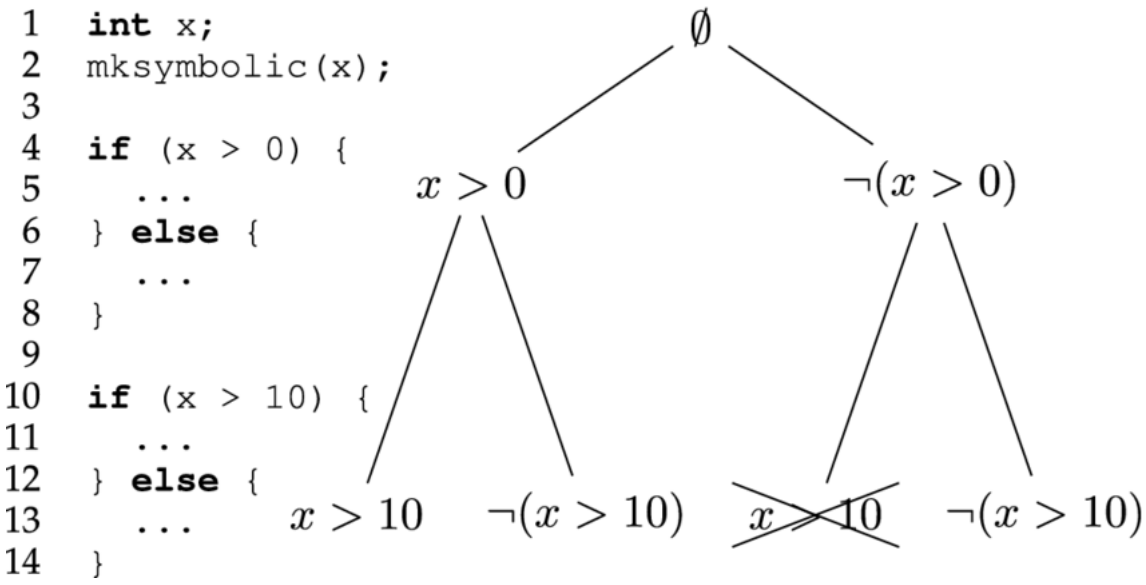
1.1 Symbolic execution

Symbolic execution is a form of dynamic program analysis. It starts by marking some of the program input (or some program variables) as "symbolic", which initially means it can take any value. It then runs the program with this symbolic input, replacing all operations depending on the symbolic data with symbolic expressions i.e. a program expression where program variables are replaced with symbolic ones. For example, an program expression $x + 1$, which adds one to x, would be replaced by a symbolic expression $\boldsymbol{x} + 1$, where $\boldsymbol{x}$ is a symbolic variable associated with a program variable x. The symbolic expression does not perform any operation on $\boldsymbol{x}$, it only represents a value that is one bigger than $\boldsymbol{x}$. If a program branches on a symbolic expression, symbolic execution explores both possible paths, adding the branch condition as a constraint to the symbolic data on each path appropriately [5, 7].

A simple example of executing a program symbolically can be seen in Figure 1.1. It considers a small program with one symbolic input ***int*** *x*. Symbolic execution keeps two data structures in memory: a map(σ) from the program variables to symbolic expressions and symbolic path constraints(PCs) over those expressions. Initially the map is empty and path constraints are set to *true* or empty set($\emptyset$) as denoted in the figure. As the symbolic executor runs its course, these data structures are updated accordingly. For example, the *mksymbolic(x)* call in Figure 1.1 adds $\sigma(x) = s$ to σ, where s is a fresh symbolic variable. When reaching a branch point at line 4 in Figure 1.1, the execution is split in two cases, one where x is greater than 0 and one where x is smaller or equal to 0. The path constrains for the two branches are updated accordingly as shown in Figure 1.1.[9] The execution is then split again at the next branch point at line 10 in a similar manner. Note that the crossed out path is not explored, as the variable x cannot be both smaller than 0 and greater than 10. In the case presented in figure the map doesn't get updated after line 2, but it is easy to imagine a case where it would. Consider that line 5 would update another variable as $v = x + 1$. In this case σ would be updated to include the symbolic expression for v as $\sigma(v) = \sigma(x) + 1$.

If left to run to completion, symbolic execution will explore all the paths in the program, with the ranges the symbolic data can have on those paths i.e. constraints. This can be used to check that the divisor can never be 0, program assertions cannot be broken or whether two different programs produce the same results for all inputs [5]. Unfortunately symbolic execution is computationally expensive and often does not terminate in feasible time even for small programs. The problem is two fold: there might be a huge number of paths to be explored if the program contains loops, and secondly, constraints have to be checked for satisfiability at each branch point, which is an NP-hard problem. Therefore most tools do not attempt to fully explore all the paths, but try to focus on paths that have the highest chance of containing bugs, for example by maximizing coverage [5] or focusing on testing code introduced by a patch [31]. It should be noted that symbolic execution is not just limited to software testing. More recently document recovery tools have been proposed based on symbolic execution [26].

Figure 1.1: Illustration of a program and the corresponding paths that are explored by a symbolic execution engine. The third path is crossed out, due to its constraints being infeasible. A variable cannot be both smaller than 0 and greater than 10, so this path is not explored.[15]



1.1.1 History

Symbolic execution was first proposed in the mid 1970s [3, 12, 25]. These papers introduced the concept of running programs symbolically. The systems were severely limited. For example EFFIGY [25] worked on its own specially designed language whose only data type were integers and whose operations didn't go beyond simple arithmetic. SELECT [3] was more advanced because it operated on a subset of LISP and attempted to aid with automatic test case generation. Despite this pioneering work, symbolic execution has only recently become practical by advances in satisfiability (SAT) solvers and mixing concrete and symbolic execution [9].

Mixed concrete-symbolic execution is the concept of running the instructions involving exclusively concrete operands normally, that is either directly on the machine hardware or using an interpreter. Concrete operands have only one value and can either be program variables that were never symbolized in the first place or symbolic expressions that resolve to one value and were concretized by the solver already. The approach was pioneered by EGT [6] and DART [22]. Concretization of symbolic variables is an important aspect of modern symbolic executors for the work in this report. For testing symbolic execution, it is important not to let the symbolic executor concretize the symbolic values as that would not test the symbolic aspects of the executor.

The ideas from EGT and DART were extended by EXE [7] and later KLEE [5], to model the symbolic variables with bit precision accuracy and purpose-built solvers, which are good at solving symbolic execution based queries. These have found many subtle, easy to miss bugs in popular programs, such as *coreutil* utilities, web servers and file systems. The tools were even applied to kernel code.

1.1.2 Fuzzing vs symbolic execution

In some respect symbolic execution could be seen as a type of "informed" fuzzing. Fuzzing is a type of automated software testing where programs are run on random inputs and checked for crashes. Even in this simplest form, fuzzing has managed to find a large number of crashes, and therefore bugs, in standard Linux utilities [34, 35]. Fuzzing approaches have proved to be quite successful, however they suffer a major downside.

Consider fuzzing the program show in Listing 1.1. If the fuzzer throws random numbers at it, it has an abysmal chance (1 in 4 billion) of getting past the if statement in line 3. This is where symbolic

Listing 1.1: A difficult to fuzz program

```
int x;
scanf("%d", &x);
if(x != MAGIC_NUMBER) exit(0);
//code we really want to test
```

execution truly shines. It would have no problem determining that there are two paths in the program, where one exits immediately. Therefore, it will focus solely on the path with interesting behavior. Symbolic execution breezes through a problem that would cause a naive fuzzer to get stuck indefinitely, with minimal impact on the runtime. There are solutions for getting around this problem within fuzzing [36], however they are not relevant to this report. Conversely symbolic execution is not a silver bullet, i.e. always better than fuzzing. It can also get stuck on different kind of programs. For example, a long loop where each iterations causes an expensive solver query could easily stop a symbolic executor dead in its tracks.

1.1.3 Concolic execution

There are several flavors of symbolic execution, most notably concolic execution. Instead of constructing symbolic expressions, it picks a random value for the symbolic variable and then executes normally, still gathering path constraints, by exploring one path. Then it backtracks to one of the previous branch points and negates a part of the gathered constrains, asking the solver for a new solution that satisfies these new constraints. It then sets the symbolic variable to this value and continues normal execution and repeats, until a termination condition is reached or all the paths have been explored. This flavor of symbolic execution is implemented in DART [22] and Crest [16].

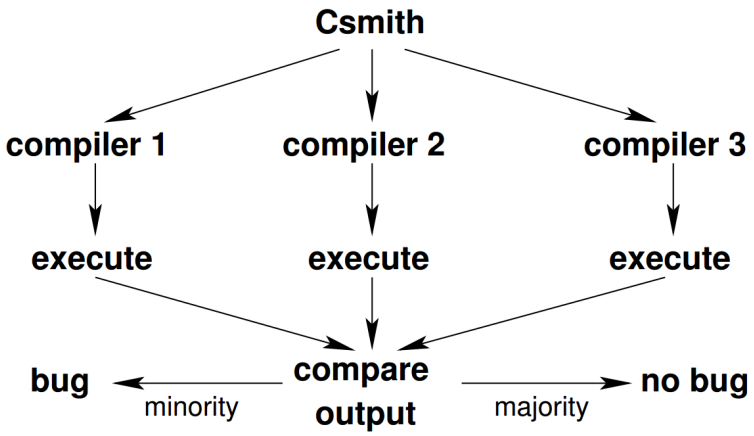
1.1.4 Available symbolic executors

The purpose of this project is to test symbolic execution engines. Therefore a symbolic execution engine is needed in the first place. Ideally the approaches should be shown to work on multiple symbolic execution engines. There are several symbolic execution engines readily available: PathFinder and Symbolic JPF [1] for Java, SymJS for JavaScript, Rubyx for ruby and PEX [44] for .NET. There are also symbolic executors for C and lower representations of C, that it compiles down to. Crest [16] and Otter [40] come with their own compilers and could therefore be considered to operate on C. KLEE [5] works on LLVM bitcode and tools like FuzzBALL [32] operate on machine code directly.

When choosing which symbolic executors to use for this project the most important factor to consider is: can the symbolic executor be run easily on interesting programs? In other words can it be run on programs that are not trivial and similar to programs used everyday. That means a language needs to be chosen for which the programs, that will become the test cases, can be generated and manipulated easily. This limits the pool of available symbolic executors to Java and C based ones. There are usually one maybe two symbolic execution tools for other programming languages, so there isn't much space for comparing the approaches over multiple symbolic executors. The choice between Java and C was made on the basis of how easy it is to automatically generate programs in the given language. In Csmith [47] (§1.2.2) and C-reduce [39] (§1.2.3), C has well established tools for automatically testing program analyzers or more specifically compilers. Similar tools for Java do not seem to exist. Therefore, we decided to focus on testing on symbolic execution engines targeting some form of C.

Three symbolic executors were chosen for this project: KLEE, Crest and FuzzBALL. They provide a wide variety of design decisions and operating principles between them. They all operate at different levels, from C directly with Crest, through LLVM IR with KLEE to machine code directly with FuzzBALL. They also implement different flavors of symbolic execution: KLEE and FuzzBALL perform symbolic execution, however KLEE explores multiple paths in parallel, while

Figure 1.2: Illustration of differential testing of 3 compilers, using a Csmith generated program as an input. [47]



FuzzBALL is more lightweight and executes a single path to completion before attempting a new one. Crest on the other hand implements concolic variant of the symbolic execution and is designed differently from the other two. It instruments the code to perform symbolic execution using CIL and then runs the program multiple times natively trapping the symbolic cases and handling them through a library. KLEE and FuzzBALL on the other hand interpret the code and handle the symbolic aspects within the interpreter.

1.2 Compiler testing

In general, little work has been done in testing program analysis tools [4], which includes symbolic execution engines. However, compilers correctness has attracted a great deal of attention. Due to the similarities between compilers and symbolic execution engines as presented in Section 1.2.4, some of the work done on testing compilers can be leveraged to test symbolic execution. Differential testing (§1.2.1) and metamorphic testing (§1.2.5) are especially prominent for testing symbolic executors. Other more formal techniques such as formally verifying compilers as done with CompCert[28] are less applicable. While there could be attempts at formally verifying a symbolic executor, the work would not benefit from the previous work based on compilers in significant manner.

1.2.1 Differential testing

The idea of differentially testing compilers was introduced by McKeeman [33]. It is presented as a form of random testing that somewhat resolves the problem of an oracle. One of the problems with random testing is designing an oracle. Given a random string or program, which is fed into a compiler, how does one know whether the compilation result was wrong? The simplest answers is to check for crashes, a crash is never the intended result. This is the approach fuzzers take (§1.1.2). However this is not where the nastiest compiler bugs exist. If a compiler crashes, the programmer will take notice and change either the program or the compiler. The worst possible compiler bug is mis-compilation, where the compiler silently produces a program that does not conform to the language standard and therefore has a different behavior than what the programmer intended.

Differential testing is a solution for building an oracle that determines whether a program was miscompiled. Given a program, differential testing compiles it with two or more different compilers. Then it runs the binaries on the same input and compares their outputs. If the outputs differ, a bug was found as illustrated in Figure 1.2. For example if out of five compilations, one differs from the others, the bug is probably in that one. The technique can be used with the same compiler,

but using different compilation options, such as optimization levels can make the same compiler act as two different compilers.

Unfortunately, there is a problem of undefined behavior when applying differential testing in practice [33]. As discussed in detail by Xi Wang et al. [45], undefined behavior gives the compiler the freedom to generate programs that behave arbitrarily in the cases where undefined behavior is exhibited. Therefore, if the program which is used for differential testing contains undefined behavior, differential testing becomes meaningless as the different compilers have the right to produce programs with different behavior. This problem is amplified by the fact that the programs should be interesting, i.e. exhibit interesting behavior that gives the opportunity to find interesting bugs.

1.2.2 Csmith

The two issues presented by McKeeman [33] have been address by the work of Regehr et al.[47] with a tool called Csmith. It is a tool for randomly generating C programs, which takes great care to produce interesting programs without undefined behavior. For example all arithmetic operations are wrapped in a "safe" library which defines some semantics for otherwise undefined behavior. The divisor is checked to be non zero and if it is, the result of division is the numerator and if over-shifting is about to occur the shiftee is returned instead. Unspecified order of evaluation of function arguments is tackled by only generating programs where order of evaluation does not change the result of execution. Similar tricks are performed for all possible undefined behaviors as defined by the C standard making Csmith an ideal tool for compiler testing. The technique proved extremely fruitful in testing compilers, finding over 200 bugs in LLVM alone [47].

Differential testing of symbolic execution has the same problem with undefined behaviors as compiler testing. Therefore Csmith was chosen as the program generation tool for this project, due the quality of generated programs and its flexibility. Further argumentation for the use of automatically generated programs, and Csmith in particular, can be found in Section 2.1.1.

Static program analyzers

Interestingly, Regehr et al.[17] have also applied differential testing with random programs to test static analyzers. They used Csmith to test a popular static analysis framework Frama-C [21]. They argue that despite the fact that static analyzers are either correct by construction or verified otherwise, they are still compiled by a compiler with possible bugs. Therefore the actual binary of the tool can still contain bugs. They have found over 40 bugs in Frama-C [21], a popular and mature static analyzer. This shows that using differential testing with Csmith can be fruitful even outside of compiler testing, which makes the prospect of applying it to symbolic execution even more exciting.

1.2.3 C-reduce

Another practical problem with differential testing is that once a bug was found, it is hard to report it. The randomly generated programs are usually a huge convoluted mess, that no one would want to debug. The bug is also likely to be exposed by only by a small portion of the program, which is why it needs to be reduced [33]. In other words the number of lines of code in the program should be reduced to a point, where the minimal number of lines is still present that exposes the bug.

A modern example of an automatic reduction tool is C-reduce. At a high-level, C-reduce tries various source-level transformations to reduce a C program (e.g. deleting a line) and then uses an oracle (in the form of a `bash` script) to decide if the transformation was successful and the transformed program still exposes the bug. If so, C-reduce keeps the reduced program and attempts to reduce it further, otherwise it rolls back the change and tries other transformations. Integrating C-reduce in the bug-finding process should be easy, as the same oracle can be used as the one used to find the bug in the first place. One challenge is that unlike Csmith, C-reduce can introduce

undefined behavior. Vu et al. [27] tackle this issue by leveraging Clang warnings and sanitizers as well as leveraging static analysis tools such as Frama-C [21]. In our experience, making the compiler reject programs that produce compiler warnings and running the programs with Clang's address and memory sanitizers is sufficient for C-reduce to not induce any undefined behavior. However, should a program arise that would keep reducing to undefined behavior, Frama-C could become a viable option for strengthening the oracles. Note that all warnings cannot be treated as errors, as Csmith programs already generate some warnings during compilation.

C-reduce builds on the idea of delta debugging first introduced by Zeller and Hildebrandt [48]. The concept was then further realized by McPeak and Wilkerson in what is referred to as Berkley Delta [20]. The basic idea is to delete various contiguous areas (ie. lines) of the program and then test them against an oracle (the oracle is a test script in the context of C-reduce). If the oracle is still interested in this reduced program, the process is repeated on the reduced program, otherwise the process backtracks and selects another area for deletion. C-reduce incorporates this approach as one of its main passes, but it can be seen as a framework that combines it with various other transformations. For example, it intelligently deletes arguments from functions or eliminates dead code. Using this technique and others authors report 25x smaller reduced programs than the ones reduced by Berkley Delta.

1.2.4 Symbolic execution engines vs compilers

Since the approaches presented in this report are based on compiler testing techniques, it is important to highlight the similarities and differences between symbolic execution engines and compilers. On one hand, both take programs as inputs. It will become convenient to later combine compiling a program and running it as one operation for Csmith generated programs. With this in mind we could also say that symbolic execution and compilers produce similar outputs. In other words they take a program and return the output of running the program. Note that Csmith programs don't take any input, so combining compiling and running a program as a combined concept makes sense. This is the reason we believed compiler testing techniques can be leveraged to find bugs in symbolic execution.

However, despite these similarities they perform very different tasks. While symbolic execution can leverage a compiler techniques, the two have very different goals and areas of difficulty. Compilation involves parsing and analyzing programs, spotting opportunities for optimization and then performing them. Whereas symbolic execution does not concern itself with optimizing the program, the main goal of symbolic execution is to explore as many interesting paths in the program as possible in the allocated time. Note that an interesting path might be one where there is higher likelihood of finding bugs in the context of software testing, for example a yet to be covered piece of code. There are two main areas of difficulty symbolic execution has to solve: efficient constraint solving and good path exploration heuristics [7, 5]. The testing approach for symbolic execution should therefore focus on testing these areas as opposed to testing parsing and optimization passes, which is what compiler testing focuses on [47].

A more trivial but equally important difference between symbolic executors and compilers is that compilers are a more mature and widespread technology. Compilers have been around for longer and their basic theory is quite well understood. Symbolic execution on the other hand is still a very active area of research with a number of elementary problems such as floating point computation [14] and array optimizations [38, 7] still being under development. A logical consequence of this is that there are many more mature compilers to test than symbolic execution engines.

1.2.5 Metamorphic testing

The work of Le et al. [27] demonstrates another approach to compiler validation. Instead of compiling the same program with two different compilers (§1.2.1), it compiles two different programs with the same compiler. The trick is that the two programs must have the same observable behavior. More formally they introduce a concept of equivalence modulo inputs(EMI): two programs P,Q are equivalent modulo inputs (EMI) w.r.t. an input set I common to P and Q iff $\forall i \in I.P(i) = Q(i)$,

where $P(i)$, denotes the output of running P with the input i [27]. Obviously, if we compile programs P and Q, run them and observe that they produce different outputs for the same input, a mis-compilation must have taken place. Le et al. [27] also proposes that EMIs can be adapted to test program analysis systems in general, but does not solve the particular difficulties in applying the technique to symbolic execution nor does it evaluate its effectives. Chapter 3 explores how this approach can be adapted to symbolic execution and how effective it is.

The proposed way of obtaining P and Q is to apply a semantic preserving transformation to P, to obtain Q. The authors have taken a so called "Profile and Mutate" Strategy for the transformations. For the "profile" part, they've taken programs that either take no inputs (from compiler test suites) and open source programs with test suites, run them, and collected coverage. The coverage shows the dead code in the programs when run on those inputs. The "mutation" then involves randomly selecting dead code and removing it. Different combinations of removed lines give different programs, therefore it is possible to generate a large number of variants from a single program using this approach. Le et al.have implemented the transformations using LLVM's LibTooling [30], which is a modern library for source to source transformations. They report that LibTooling has the capability to mutate the AST, which is why LibTooling was chosen to perform the transformation in this report.

Le et al.'s paper also shows that the EMI approach found more bugs faster than compiler differential testing presented by Yang et al.with Csmith [47]. They also argue that it is is more easily extensible to other languages, such as C++. In addition it is not a random fuzzing approach as they don't generate programs, they take existing programs—compiler test suites for example— and generate variants from them. That means the programs are more similar to those humans are likely to write as opposed to completely randomly generated programs.

Application to symbolic execution engines

In the context of symbolic execution, dead code is not as interesting as for compilers. A compiler has to look at the source code as a whole and cannot in general recognize dead code. In contrast a symbolic execution engine is in general not concerned with dead code as it simply doesn't execute it. For most intents and purposes one could say that dead code is invisible to a symbolic execution engine.

Therefore, other transformations need to be devised in order to apply metamorphic testing. An equivalent transformation to dead code injection would be "dead condition" injection. The idea is to add something logically equivalent to "&& true" to a conditional branch. Conditional branches lie at the heart of symbolic execution and therefore one would expect the described transformation could expose some bugs. This transformation is discussed in further detail in Section 3.2.1.

Another interesting but harder to implement transformation would be to exploit the commutative properties of logical operators and switch their order. However to truly preserve semantics of the program care must be taken to not change the order of execution when the condition short-circuits.

1.3 Other efforts for testing program analysis tools

To our knowledge, this is the first approach specifically targeted at testing symbolic execution tools, and the first effort to present the experience of adapting compiler testing techniques to check mature symbolic execution engines.

More generally however, the research community has started to invest effort into ensuring the correctness and reliability of program analysis tools. One example is represented by competitions among tools, which have the important side effect of finding bugs in participating tools: the annual SV-COMP competition for software verification tools is a prime example of such competitions [2]. This section presents other efforts in the community to test program analysis tools.

1.3.1 Many-Core Compiler Fuzzing

The general idea of applying both differential testing and metamorphic testing has also successfully been applied to compilers for GPU kernels written in OpenCL [29]. They too apply compiler testing based techniques in a new domain of GPU kernels. However, their contribution is very different from the one presented in this report. Their main area of difficulty is constructing a program generator for OpenCL kernels as well as devising a semantics preserving transformation that does not depend on dead code being present in the program.

Lidbury et al. [29] have built their own program generator called CLsmith, based on Csmith, that generates OpenCL kernels. They first present an approach for lifting Csmith generated programs to embarrassingly parallel kernels, by making each thread execute the Csmith generated program completely independently. Then they present several methods for making those threads communicate in a deterministic and safe manner. For testing symbolic execution this is obviously not needed as it happily operates on standard Csmith programs.

Their approach to testing OpenCL compilers with the EMI approach is more interesting. Practical kernels do not have dead code, hence the "Profile and Mutate" strategy (§1.2.5) from the original EMI paper [27] cannot be applied. Therefore, they use program snippets generated by CLsmith to generate dead code. These snippets are then wrapped by an if statement, whose condition can never evaluate to true so that the code is indeed dead. They also ensure that the condition is not trivial enough for a compiler to realize the code is dead and remove it. This approach is still not easily applicable to symbolic execution for the reasons outlined in Section 1.2.5.

The approach was shown to be successful with over 50 OpenCL compiler bugs found. This further strengthens the case that differential testing and metamorphic testing techniques are well suited for testing program analysis tools.

1.3.2 Testing of Clone Detection Tools

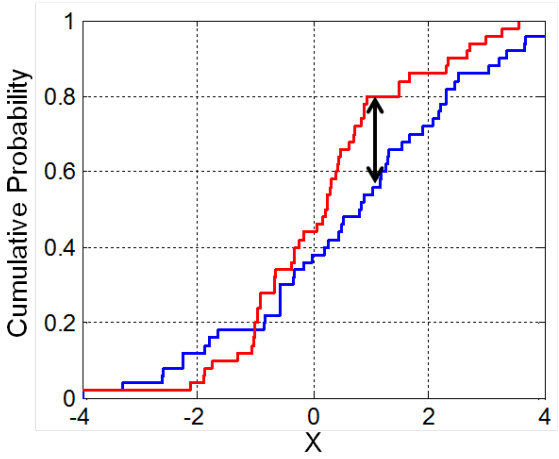
An example of testing program analysis tools by program generation is the work of Roy and Cordy work on evaluating clone detection tools [41]. Clone detection tools attempt to find snippets of code, called code clones, that are very similar to each other, within a larger program. Code clones are common in real code bases, as programmers tend copy snippets of code. Obviously if a copied snippet contains a bug, the bug is likely present in all the copies. Clone detection tools attempt to find and highlight these clones to the programmers. Roy and Cordy's approach starts from real programs, which are mutated to artificially contain code clones. Several clone detection tools are then tasked with finding these newly introduced clones and the performance of the tools is evaluated.

Although this work is similar at a high level to the approach of testing symbolic execution engines presented in this report, it is done in a different domain. Therefore, they have faced challenges unrelated to symbolic execution. Conversely, testing symbolic execution comes with its own set of challenges that are not resolved by their work. Their main focus is on testing the ability of clone detection tools to detect slightly modified cloned code fragments, that are likely to be found in real programs. Examples of mutation range from adding a little white space to replacing *for-loops* with equivalent *while-loops*.

1.3.3 Testing Refactoring Tools

A technique similar in spirit to the approach presented in this report although different in terms of application domain and techniques used, is Daniel et al.'s [18] work on testing refactoring engines. They combined program generation and differential testing between refactoring engines. Their program generation contribution was providing developers with a declarative way of constructing abstract syntax trees for Java programs using a bounded-exhaustive approach. The differential testing was performed using a crash oracle as well as several other oracles that take into account the semantics of the refactoring.

Figure 1.3: An illustration of the KS test, the red and blue lines represent empirical distribution functions of a random variable X, with the black arrows showing the KS statistic. (source: wikipedia)



1.3.4 Testing Alias Analysis

Wu et al.. [46] present a system for checking pointer alias analysis implementations. Pointer alias analysis attempts to find pointers in the program that can point to the same location. Listing 1.2 shows a snippet that could be analyzed by pointer alias analyzer, to conclude that pointers *p1*, *p2* and *p4* all alias each other, whereas *p3* does not alias any other pointer.

Listing 1.2: Snippet for demonstrating pointer aliasing.

```
int a,b;
int *p1 = &a;
int *p2 = &a;
int *p3 = &b;
int *p4 = p1;
```

Wu et al.'s approach validates the results of pointer alias analysis tools against the pointer values observed at runtime. This is a form of differential testing, between dynamic and static information. They present an alias analyzer agnostic tool, for testing alias analysis. The tool instruments a program to track the pointer addresses at runtime, the user then runs the program on a workload. The pointer alias information gathered during runtime is compared with the result of alias analysis. If a contradiction is found, i.e. two pointers alias during runtime, when the static analysis reported that they couldn't, a mismatch is reported.

1.4 Kolmogorov–Smirnov test

The Kolmogorov-Smirnov test or the KS test is a statistical test of the equality of two distributions. It can be used as a test of whether two samples of a distribution are taken from the same distribution. Its null-hypothesis is that the two samples are drawn from the same distribution. Briefly speaking the KS test works by looking at the greatest divergence between the cumulative empirical distributions of the two samples as seen in Figure 1.3. [19]

Further details of the KS test are not needed as the *scipy* implementation [24] of the test was used. Scipy's *stats.ks_2samp()* function implements this version of the test. It takes two lists of numbers, containing the samples from the two distributions we want to compare. It then returns the KS-statistic, which we can ignore and a p-value. Since the null hypothesis states that two distributions are the same, a low p-value indicates that we need to reject the null and therefore conclude that two distributions are different. This test was found useful in Section 3.1, where it is necessary to asses if two distributions are the same or not.

1.5 GNU parallel

GNU parallel [43] is a command line utility that calls a program with the arguments from its standard input. It can be used as a drop-in replacement for a more standard *xargs* utility. GNU parallel was leveraged heavily in this work to run large scale experiments in parallel. It was chosen over *xargs* for its superior parallelization capabilities. For example if *xargs* runs the jobs in parallel, their *stdout* could be jumbled together, whereas GNU parallel makes sure the outputs of different jobs are not intertwined. In addition, it provides the capability to run the jobs on multiple machines. Therefore it was easy to scale one of the computationally heavy tasks to multiple machines (§3.1), which made that approach feasible.

Chapter 2

Differential Testing of Symbolic Execution

In this chapter several techniques for comprehensively testing symbolic executors are presented, together with case studies that applies them to find bugs in several different symbolic execution engines KLEE, Crest and FuzzBALL. The techniques are an adaptation of differential testing of compilers for symbolic execution tools. As introduced in Section 1.2.1, differential testing has seen tremendous success in revealing important bugs in popular compilers [47]. This gives the techniques great potential for finding bugs in symbolic executors, as they are in some important aspects similar to compilers.

More precisely, the techniques are based on program generation and differential testing, adapted to exercise several key inter-related aspects of symbolic execution tools: execution fidelity and accuracy of constraint solving—that is, whether the symbolic execution tool correctly follows the paths it intends to follow, gathering precise constraints in the process—as well as correct forking behavior and replay—that is, whether the generated inputs execute the same paths as the ones followed during symbolic execution.

The method is effective for both dynamic symbolic execution tools which keep multiple path prefixes in memory, as in EXE [7], KLEE [5], Mayhem [10], Symbolic JPF [1] and S2E [11], as well as for those which implement the concolic variant of symbolic execution, in which paths are explored one at a time, as in DART [22], Crest [16] and CUTE [42].

The rest of this chapter is structured as follows. Section 2.1 gives an overview of our technique, showing how we generate random programs (§2.1.1) and create versions of these programs (§2.1.2) to be crosschecked using four different oracles (§2.1.3), and reduced to produce small bug reports (§2.1.4). Then we dive into the technical details of the implementation of the testing infrastructure in §2.2. Finally Section 2.3 presents our case studies on the KLEE, Crest and FuzzBALL systems, reporting the effectiveness and performance of the technique.

2.1 Testing Approach

Conceptually the main stages of our testing approach are shown in Figure 2.1. The input is a configuration of the experiment which includes: the symbolic execution engine to be tested, compiler flags and versions of all the programs involved, i.e. Csmith, constraint solver, symbolic executor, as well as all other configuration options presented in further sections: generated program parameters §2.1.1, different modes (§2.1.2), oracles (§2.1.3) and constrainers (Table 2.1).

In the first stage (*Generate programs* in the figure), random, deterministic programs are generated with the help of Csmith tool [47] (§1.2.2) and instrumented to support our oracles. In the second stage (*Create & run versions*), several different versions of a given generated program are created: a native version, designed to execute natively; single-path versions, designed to run a single path

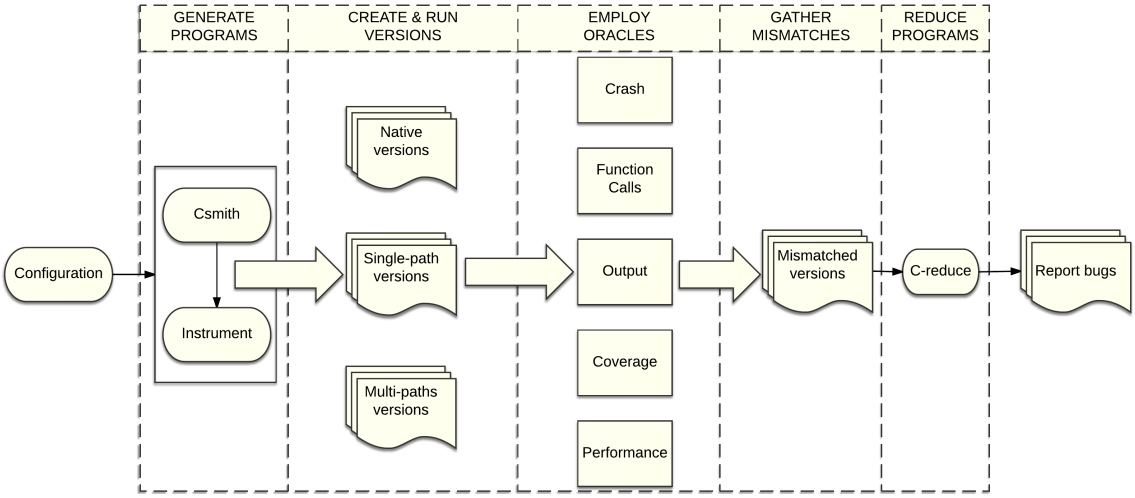


Figure 2.1: The main stages of the testing approach.

when executed symbolically; and multi-path versions, designed to run multiple paths when executed symbolically. These different versions are run and crosschecked using our four oracle types: crash detection, output, function calls and coverage comparison (*Employ oracles*). Any programs exposing mismatches (as flagged by the oracles) between the native and symbolic execution runs (*Gather mismatches*) are then reduced using the C-reduce tool [39] (§1.2.3) configured with the oracles (*Reduce programs*) and reported to developers.

While the testing approach is general, the infrastructure built is targeted toward testing symbolic execution engines for C code ecosystem.

2.1.1 Generating random programs

The first step of the approach is to generate small programs using the Csmith [47] tool used in compiler testing. Csmith is a tool that can generate non-trivial C programs that leverage many features of the C language and which has been used successfully to find many bugs in mature compilers [47, 29] (§1.2.2).

Csmith generates programs in a top-down fashion. It starts by creating a single function, which is called from *main*. Csmith then picks a structure from its grammar randomly and checks if it is appropriate for the current context. For example, *continue* can only appear in loops. Should the check fail, it makes a different choice until it succeeds. If the chosen structure needs a target (e.g. a variable to read or a function to call), it randomly chooses between using an existing construct and generating a new one. Care is taken not to generate constructs with undefined or unspecified behavior, e.g. by guarding every division operation to ensure the divisor is not zero. Types are handled in a similar manner.

If the selected structure is a non-terminal, the process repeats. Finally, several safety checks are performed to ensure there cannot be any undefined or unspecified behaviour. If that fails, the changes are rolled back and the process starts from the most recent successful stage.

When the process requires a new function to be created, the generation of the current function is suspended until the new function is generated completely. Thus Csmith terminates once the first function is completed. At this point *main* is generated, where the first function is called and after it returns, the checksum of all global variables is computed and printed out.

The generated programs take no input, perform some deterministic computation and output the checksum of all global variables, giving an indication of the state of the program upon termination. The length and complexity of the generated code is highly configurable. By default the Csmith programs are on average 1600 lines long, containing about 10 functions and 100 global variables. The global variables can have a wide range of types: signed and unsigned integers of standard widths, arrays, randomly generated structs and unions, pointers and nested pointers. The functions

take varying number of arguments of different types and return a randomly-chosen type. Function bodies declare several local variables and include *if* and *for* statements, which in turn contain assignments to both local and global variables. The expressions assigned are deep and nested, reading and writing to multiple global and local variables, performing pointer and arithmetic operations and calling other functions.

There are several reasons for using Csmith-generated programs as opposed to using real software.

1. Csmith programs are valid C programs without undefined or unspecified behavior. This is important because the compiler used to generate the native version of the program and the engine used to symbolically execute the program might take advantage of undefined or unspecified behavior in different ways, which might lead to spurious differences.
2. Csmith programs, by design, have a good coverage of C language features, which a limited collection of real programs might miss.
3. Most of the language features being used in Csmith programs can be enabled or disabled via command-line arguments. This is important because once the symbolic execution tool is found to mishandle a certain feature, we want to be able to continue testing without repeatedly hitting that same bug.
4. Csmith programs are deterministic and the input and output are easily identifiable: the input is represented by the set of global variables in the program, and the output consists of a checksum of its global variables which is printed at the end of the execution.
5. Unlike real programs, Csmith programs are relatively small (or more exactly, Csmith can be configured to generate small programs), which allows us to perform a large number of runs.

Disadvantages of Csmith programs (and automatically generated programs more generally) are that they are artificial, hard to read by humans, and not guaranteed to terminate. The second issue is addressed by automatically reducing the size of the program (§2.1.4), and the last issue by using timeouts, as recommended by the Csmith authors [47] (§2.3.1).

2.1.2 Creating and running versions

The bug-finding effort could be partitioned into three domains. These domains relate to the three modes of execution which we imposed on a symbolic executor: interpreter mode, single path symbolic mode and multi path symbolic mode. Any given Csmith generated program could be run in any of the three modes, depending on how it was instrumented.

For each generated program, we first create and run an unmodified native binary version of the program. Then, for each of our three testing modes, we create a modified version of the program to be run by the symbolic execution engine under test.

Mode A: Concrete mode

This mode is designed to test the concrete execution of the symbolic execution engine. For this mode, we run the program with the symbolic execution engine without marking any variable as symbolic. For example, the code would be compiled to LLVM bitcode and then ran with KLEE directly without any symbolic input.

The symbolic execution run is then validated against the native one, using the oracles (§2.1.3). For example, the function call chain oracle would check that the native and symbolic runs generate the same sequence of function calls.

Mode B: Single-path mode

The aim of this mode is to test the accuracy of the constraints gathered by symbolic execution and its ability to correctly solve them. Essentially, this mode is checking the symbolic execution of individual paths in the program. For this mode, the code is modified to mark all the integer global

Table 2.1: Four ways of constraining a variable x to a constant value v. d_i is a prime divisor of v.

Type	**Constraint**
$<$,$>$	$\neg(x < v) \wedge \neg(x > v)$
$\leq$, $\geq$	$x \leq v \wedge x \geq v$
range	$\neg(x \leq v - 2) \wedge \neg(x \geq v + 3) \wedge$ $\neg(x = v - 1) \wedge \neg(x = v + 1) \wedge \neg(x = v + 2)$
divisors	$\wedge_i \neg(x \mod d_i \neq 0) \wedge x > 1 \wedge x \leq v$

variables of the generated program as symbolic and constraining them to have the unique value assigned to them in the original program. This essentially forces the symbolic execution engine to follow the same execution path as in the native version, but also collect and solve constraints on the way.

Constraining a variable to have a unique value needs to be done in such a way that the symbolic execution engine does not infer it has a unique value (and reverts to concrete execution for that variable). In particular, assigning a symbolic variable to have a constant value (e.g. `x = 4`) or comparing it with a constant (e.g. `if (x == 5)`) would typically make the engine treat that variable as concrete on that path.

Four different ways of constraining a symbolic variable x to a given value v were used. They are listed in Table 2.1. For example, the second method adds the constraint that x is less than or equal to v and greater than or equal to v, while the fourth method adds the constraint that x is divisible by all the prime divisors of v, is greater than 1 and less than or equal to v. At the implementation level, for each integer global variable initialization such as **int** x = 5;, we add the following code at the start of `main`, should we for example, follow the first constraining method:

Listing 2.1: Snippet showing symbolizing and constraining a variable *x*.

```
make_symbolic(&x);
if (x < 5) silent_exit(0);
if (x > 5) silent_exit(0);
```

In this code, the `make_symbolic()` function is used to mark the given variable as symbolic, while the `silent_exit()` function terminates execution without generating a test input on that path.

Therefore, after executing the code fragment above, the symbolic execution engine will continue along a single path with the path condition $\neg(x < 5) \wedge \neg(x > 5)$, which effectively constraints x to value 5.

Once such a version of the program is constructed, its execution can be validated using the oracles, as for the previous mode A. Note that one oracle that is effective here, as shown in the evaluation, is to check that the symbolic execution engine executes a single path. However, an explicit oracle was not added for this, as other oracles, such as the function call oracle, would almost always catch such a bug.

Mode C: Multi-path mode

While the prior mode tested that the engine correctly performs symbolic execution of a given path, this final mode checks that it explores multiple paths and generates inputs that exercise exactly those paths.

For this mode, all integer global variables are simply marked as symbolic, without constraining them to any value, and let the symbolic execution engine explore multiple execution paths. As a result, not all oracles are well applicable to this mode. In particular the output oracle was not used for non-concolic execution engines, as the output could now be a function of some symbolic variables. This does not work well when comparing it with native execution.

Besides the crash oracle, the function call chain oracle was the only one used, which was the easiest to adapt for this scenario. The approach was to record the sequence of function calls on each path

explored during symbolic execution, and then, for each path, to run natively the generated test input and check whether it generates the same function call sequence.

2.1.3 Oracles

The five oracles that we used to validate the executions are discussed in detail below.

Crash oracle

The first basic oracle consists in detecting generic errors during symbolic execution runs, such as segmentation faults, assert violations and other abnormal terminations.

Output oracle

As discussed in Section 2.1.1, Csmith programs are designed to have no undefined or unspecified behavior and produce deterministic output. More exactly, the programs print at the end a single value, the checksum of all global variables. For mode A, the checksums printed out by the native and symbolic execution runs are simply compared.

For mode B, it was found that computing checksums for symbolic variables is very expensive, resulting in many time-consuming solver queries. The solution was to exclude the symbolic variables from the checksum computation, and instead simply print out their individual values. For non-concolic engines, the instrumentation first ask the constraint solver for a solution (which in this case is unique) before printing out the symbolic value.

Function call chain oracle

The function call chain oracle compares the sequence of function calls executed by the native and symbolic execution versions. This oracle provides the ability to catch some bugs where symbolic execution follows the incorrect path, but without having any influence on the output. For modes A and B, this oracle checks that the unique path followed by the symbolic execution engine produces the same sequence of function calls as the native execution. For mode C, this oracle checks that when natively replaying a generated input, the same function call sequence is produced as in the corresponding path explored during symbolic execution. Because some execution paths may not be fully explored by non-concolic tools in mode C (due to timeouts), the oracles actually checks that the function call chain generated during symbolic execution is a prefix of the corresponding native function call chain.

Note that the function call oracle could lead to false positives if the native and symbolic execution evaluate function call arguments in different order. We only encountered this in Crest, and we found this easy to filter out.

Coverage oracle

The coverage oracle was used in a similar way as the function call chain oracle, to ensure that the native and symbolic execution runs execute the same lines of code, the same number of times. While we could have used this oracle in mode C as well, to check whether the natively replayed execution covers the same lines of code as during the corresponding path explored during symbolic execution, it was more difficult to implement efficiently.

Performance oracle

The last oracle that we used was for finding performance anomalies. We flagged all programs for which symbolic execution took disproportionately longer to run compared to the corresponding native execution. More details about this oracle can be found in sections 2.3.1 and 2.2.4.

2.1.4 Reducing bug-inducing programs

As indicated before, Csmith generated programs are large and hard to read by human developers. The code consists of huge nested expressions without any high-level meaning, referring to mechanically-named variables. Debugging such programs would be highly difficult. Therefore, for each generated program that exposes a bug, the C-reduce tool [39] (§1.2.3) was used to reduce it to a manageable size.

2.2 Implementation details

This section focuses on presenting and discussing technical implementation details of the testing infrastructure used to conduct the experiments presented in this report. The design decisions are motivated by the following two goals the infrastructure aims to achieve:

- Implement the process described in Section 2.1 to be as automated as possible.
- Aid and benefit from manual experimentation. For example, once a users knows how to run a symbolic execution engine manually, it should be trivial and natural to use it within the infrastructure. Conversely, if an experiment is already automated, it should be easy to manually perform the individual steps of the experiment. This is important for reducing and reporting the programs with bugs.

We start by talking about the instrumentation stage in Section 2.2.1. It is the heart of the testing method, preparing the programs for symbolic execution and setting up the oracles. Then the collection of scripts that slowly builds up to automated experiments is presented in Section 2.2.3. Finally, generating programs and reducing them is presented in Sections 2.2.5 and 2.2.6 respectively.

2.2.1 Instrumentation

There are two main insights behind the design decisions for the instrumentation implementation. First, the instrumentation needs to be inserted at source code level to maintain generality. It is always possible to compile source code to lower representations such as LLVM IR or machine code, whereas going the other way is much harder and edging impossibility. Some symbolic executors, like Crest, have their own compilers and therefore cannot execute pre-compiled programs. Whereas for others like FuzzBALL it is trivial to compile down to machine code, on which FuzzBALL operates. Note that performing instrumentation on the source code level is the highest abstraction layer practically possible, since the program generator generates C code. If there existed an even more general program generator, it would make sense to perform instrumentation there.

Second, the instrumentation should be made as lean as possible. To be more precise, the instrumentation should merely provide necessary hooks into the program and not include any logic. The logic should be put into a library that uses those hooks and can be easily swapped depending on experiment configuration. In more practical terms, the instrumentation inserts function calls into a library at appropriate places. To achieve this, a symbolic executor agnostic interface for symbolizing variables (marking variables as symbolic) and printing symbolic variables was defined. The instrumentation process then inserts the calls to this interface appropriately. When the instrumented program is to be run, an appropriate implementation of the library would be linked in. For example, the implementation of the library in native runs for symbolizing variables would be a noop as we cannot perform symbolic execution in native runs. For KLEE it would delegate to *klee_make_symbolic()*. The library can also have different implementations for the same symbolic executor. Different constrainers are implemented this way.

This approach enables running the same instrumented program with different symbolic executors and even natively. This means programs don't need to be regenerated and re-instrumented for each different run paving the way for actually saving the generated and instrumented programs, thus improving efficiency and reproducibility of experiments.

Source level instrumentation

Instrumenting programs on the source level is a significantly less mature idea than instrumenting at lower abstraction levels, which have the full weight of compiler optimization behind them. This is perhaps best illustrated within clang, LLVM world. LLVM has a dedicated and well supported toolkit for instrumentation in LLVM passes. The way to perform source level instrumentation is with clang's LibTooling [30]. It is a comprehensive library for tapping into clang front end. The main use case for LibTooling are refactoring tools and therefore using it for instrumenting programs is less pleasant compared to an LLVM pass. On the other hand, it is much easier to write the instrumentation with LibTooling as one can inject C code directly. In contrast one needs to construct LLVM IR through its C++ API when using an LLVM pass.

LibTooling provides two basic facilities of interest to us that enable us to insert instrumentation: walking the clang AST and rewriting the source code. In short, we walk the AST for two purposes: to gather information about the program and mutate it as needed. For example, the AST needs to be traversed to find all the global variables and the values they are initialized to. Then the rewrite facility can be used to insert the calls to the symbolizing interface at the start of *main()*. There are two kinds of instrumentation inserted:

Function logging instrumentation is straightforward. It first inserts a global variable *instr_filename*, containing the name of the file being instrumented. This is used later on to determine what file to write the function calls to. Then it visits each function (FunctionDecl clang AST node) and inserts a call to a logging function, passing in the name of the function. So for function *foo()*, *logFunction("foo");* is inserted at the beginning of the body of *foo()*. *logFunction* is then implemented separately in the library and linked in. The *logFunction* implementation uses the *instr_filename* to open a file with that name and append the name of the function being passed in to the end.

Symbolizing global variables is more involved. Initially all global variable declaration are visited while remembering unsigned integers with initialized values. For each initialized unsigned integer, a call to the symbolizing interface is added at the start of *main()* passing in the pointer to the global variable and the bit width of the variable. The variable name is also passed in for debugging purposes. For each of these variables, a call to the print symbolic variable interface is also inserted at the end of *main()*. Finally, all the calls to *transparent_crc()* function in main that have one of the initialized unsigned integers passed in is commented out. *transparent_crc()* is a Csmith function that computes the checksum of global variables. If it is executed in symbolic context, the symbolic execution becomes unfeasibly slow. The solution is to remove all the calls to *transparent_crc()* involving symbolic variables. Instead, the symbolic variables are printed out to still check their values.

2.2.2 Implementing constrainers

Constrainers are implemented as a part of the *make_symbolic()* function in the library. The $<$,$>$; $\leq$, $\geq$ and range constrainers are implemented exactly in the spirit of Listing 2.1. Different constrainers obviously require different versions of the library, however the symbolizing part of the function should be shared between the implementations. Therefore, there is a single symbolizing source file for each symbolic executor. It uses a *CONSTRAIN()* macro to constrain the variables. The *CONSTRAIN()* macro is then defined in a header file. The symbolizing library source file is than compiled three times, once for each of the three constrainers, with different header files included to change the constrainers.

Divisor constrainer. The divisor constrainer deviates slightly from this pattern as it is harder to implement. At library compile time it uses the sieve method to generate the first 10000 primes and put them in a static array. Given N is the value the constrainer needs to constrain a variable to, it loops through the static array of primes. For each prime p that divides N, it finds the highest exponent e of p that still divides N. It then adds the constrain $N \% p^e == 0$ to the path condition. Finally it divides N by p^e and repeats. When N reaches 1 it stops and returns. If it runs out of primes, it adds the final remainder of N as a simple equality constraint. This approach was

taken as it enables control over the number of modulo constraints added to the paths, simply by changing the number of generated primes. This enables controlling the strain on the solver.

2.2.3 Testing infrastructure

The main driver behind the design choices for this implementation of testing infrastructure is the ability to aid and benefit from manual experimentation. To understand this, let's take a look at the process of setting up a new symbolic execution engine. Obviously the very first step after building the symbolic executor is getting it to execute a small program symbolically, for example the one shown in Listing 2.2. If the two different messages are printed, we've successfully managed to get the symbolic executor running. In essence the same sequence of bash command should compile and run symbolically any Csmith program as well. This sequence of command is then put in a script and called a *compileAndRun* script. It proved useful to have this script used for both automated and manual experimentations, as it avoided the need to reinvent compiling and running a program in a separate automated framework. The *compileAndRun* script is a elementary operation that is combined with symbolizing library (§2.2.1) to orchestrate an experiment. These experiments can then further be combined to run whole batches of experiments.

Listing 2.2: A simple program with two paths

```
int x;
make_symbolic(&x);
if(x < 0) printf("smaller than zero");
else printf("greater than zero");
```

An overview of the scripts, and how they combine together can be seen in Figure 2.2. Given a set of generated programs as further described in Section 2.2.5 and the setup scripts, there is a script that runs a whole set of experiments, described in Section 2.2.4. It uses a script that runs a single experiment. For a single experiment a test case is compiled and ran natively and symbolically, with appropriate instrumentation library, depending on the experiment setup.

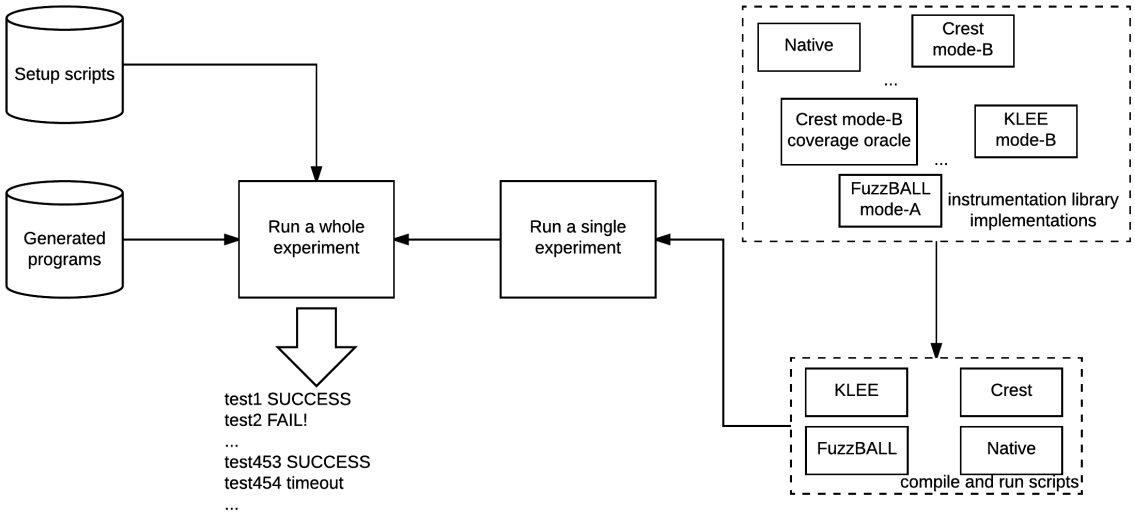


Figure 2.2: The collection of the important scripts that make the testing infrastructure and how they interact

Compile and Run scripts

A *compileAndRun* script can be seen as a pure function from a C source file of a program to its output. In most use cases the program will be compiled for every time it is run, therefore the benefit of compiling once and then running multiple times is non existent. In addition each symbolic executors wants their program to be compiled differently. KLEE requires LLVM bitcode,

Crest has its own compiler and FuzzBALL works on native bytecode. *compileAndRun* scripts provide an excellent generalization over the actually symbolic executor used. They hide away most of the complexity of compilation for a symbolic executor and running a program with it. There was no case for the whole extent of this work where a need for separating compiling and running has presented itself.

Listing 2.3: A sketch of a compile and run script

```
#!/bin/bash
compile -c -o tmp1.o $1 &&\
link -o tmp2.o tmp1.o $PATH_TO_SYMBOLIZING/library.o  &&\
symbolic-executor tmp2.o
rm tmp1.o tmp2.o
```

Listing 2.3 shows a sketch of how a *compileAndRun* script would be structured. The temporary files would be obtained with the *mktemp* utility. It is essential that this script has no visible side effects so that it can be ran alongside other *compileAndRun* scripts. In addition it shouldn't pollute the folder it runs in, so that no manual cleanup is needed. Note that the similar kind of script can be used for native execution by replacing line 4 with *./tmp2*.

Single experiment script

With the *compileAndRun* abstraction in hand it is easy to define a general single experiment we would perform for the majority of this report. It consists of two *compileAndRuns* and a transformation between them. The basic outline of the script is shown in Listing 2.4. For Chapter 2 we don't use the transformation, so we set it to an identity transformation. In other words, we set the *TRANSFORM* variable to *cp* command. It will become apparent later in Chapter 3 why the transformation is useful, but it is presented now, because the same script can be used there.

Listing 2.4: A sketch of general single experiment script

```
#!/bin/bash
$COMPILE_AND_RUN_1 $1 > first_run_output &&\
$TRANSFORM $1 temp0 &&\
$COMPILE_AND_RUN_2 temp0 > second_run_output &&\
diff first_run_output second_run_output
pretty_print_the_result
clean_up
```

To perform the experiment described in §2.1 the *COMPILE_AND_RUN_1* variable is set to point to a native compile and run script. The second compile and run script is then set to one of the compile and run scripts for the symbolic executor chosen in the experiment configuration.

Similarly to compile and run scripts, this script should also be side effect free and produce no garbage files after termination for the same reasons as before. It should be able to run with other instances of itself in parallel as well as keep the working directory clean. The temp files are therefore also obtained with *mktemp* utility and removed before the script terminates.

2.2.4 Oracles

There are no explicit oracle scripts as they are implemented implicitly by a combination of the instrumentation library, *compileAndRun* scripts and the single experiment script. The *compileAndRun* scripts should produce output that enables the oracles to identify mismatches. The *diff* utility in the single experiment script then performs the actual comparison. As shown in Listing 2.4, it takes the output of both runs and compares them. If no difference is found and the *compileAndRun* scripts follow their obligation, we can conclude that oracles have not found a mismatch. As a consequence, and perhaps a downside, each compile and run script is uniquely tied to a set of oracles and should only be used with other compile and run scripts that are tied to the same list of oracles.

To illustrate this, let's take the output and function call oracle. A compile and run script implementing only the output oracle should just pass the stdout of the program through, without adding anything else. Comparing the output of two compile and run scripts implementing only the output oracle with the diff utility will therefore report mismatches in the output of the two programs if any.

A compile and run script implementing only the function call oracle should discard the stdout of the program under test. Instead it should dump the function call chain to its stdout. In this case *diff-ing* the outputs of two compile and run scripts of this sort will report mismatches of function call chains. It is now obvious why compile and run scripts can only be run with compile and run scripts implementing the same list of oracles. Comparing function call chain with the output of a program is non-nonsensical.

It is also possible to have multiple oracles in the same compile and run script by concatenating the output. For example, having a compile and run script implementing both output and function call oracle is just the combination of the two above. First it should pipe the stdout of the program through and then output the function call chain to stdout. The order is obviously important.

Coverage oracle is of the same nature as the function call oracle and can be seen as a more precise version of it. However, an interesting challenge was encountered while implementing this oracle for KLEE. The performance overhead was extremely high even when gathering coverage information on a single execution path. The problem was that the coverage instrumentation would generate `select` instructions to index into an internal buffer used to track coverage, which would make the symbolic buffer symbolic, leading in turn to expensive constraint solving queries.

In particular, the coverage instrumentation framework used, GCov, uses an internal buffer to count the number of times an edge in the control flow graph (CFG) was followed. GCov instruments each edge in the CFG with code that updates this buffer depending on which edge was taken. Conceptually, the code looks like this, where `a` and `a+1` are constants used to name two edges:

Listing 2.5: A conceptual snippet of a portion of gcov instrumentation

```
edge_taken = condition ? a : a+1;
gcov_internal_buffer[edge_taken]++;
```

The first statement generates a `select` expression of the form `select(condition, a, a + 1)`. If the `condition` expression is symbolic, this symbolic `select` expression is used to index into the internal buffer. Once a buffer is indexed by a symbolic expression, symbolic execution must treat the buffer as symbolic, sending all its values to the constraint solver [7]. Since the internal coverage buffer is very large (it has one entry for each edge in the CFG), the resulting constraints become prohibitive to solve.

Once we diagnosed the issue, the solution was simple: the GCov instrumentation was modified to generate explicit branches instead of `select` instructions. This made a huge impact on performance, making this oracle usable. More generally, this is an issue that one has to be aware of during symbolic execution when instrumenting programs with coverage information.

Performance oracle The approach to the performance oracle is slightly different than the other oracles. In general it is hard to judge when performance oracle should report a mismatch so it is not included in on the fly reporting. Instead the run time information is stored for later statistical processing, where potentially interesting programs are found.

There is a problem however in passing the run time information around using the described architecture insofar. The run time information is available to the compile and run scripts which are able to measure it. However it only becomes useful when compared with another compile and run script in an experiment. Bash doesn't really provide a facility for passing information around. Therefore compile and run scripts must break their purity principle a bit by writing the runtime information in a file unique to the run. For example based on the input name, such as *$1.info*. These files can then be collected by the experiment script to report run time information. Other messages could potentially also be passed around the infrastructure using this method.

Running a batch of experiments

With the ability to run an experiment with a single program it becomes easy to run batches of experiments using standard linux utilities such as GNU parallel [43] or *xargs*. I have opted for GNU parallel over *xargs* for its better parallelization abilities which came in handy later. However *xargs* could be used as a drop in replacement.

As alluded to before, experiments within this testing infrastructure aim to be fully described by the values of certain environment variables: *COMPILE_AND_RUN_{1,2}* and *TRANSFORMER*, to name a couple. The values of these variable for a certain experiment can be put into a file and *sourced* before the experiment thus choosing what kind of experiments we want to run. Given that there is a folder with a set of programs suitable for this infrastructure, running a batch of experiments is just a *map* operation of a single experiment script over the files. Listing 2.6 shows an example of how the map operation can be implemented in bash. It first *sources* the experiment setup and then performs the *map* operation over the programs in the current directory, which are assumed to be compatible with the infrastructure. How these programs are obtained is described in Section 2.2.5.

Listing 2.6: Running a batch of experiments sketch

```
source path/to/experiment/setup/file
ls | parallel -L1 ./generalSingleExperiment.sh
```

In addition to what is shown in Listing 2.6, the actual script also performs some sorting beforehand and records information about machine state. In other words, it records the current git commits of this infrastructure, Csmith, KLEE, Crest, FuzzBALL to name a few. This is done in order to help keep a better record of the experiments run and their configuration to be able to answer any potential queries that arise in the future.

2.2.5 Generating programs

The idea behind generating programs is similar to the one for running the experiments described before. Generation of a single program is automated and then GNU parallel is used to generate a batch of programs. A naming convention for generated programs was adopted. They would be numbered with each one having the name of the form: *test{sequence number}.c*. Listing 2.7 shows a example of how 100 programs would be generated with a script that generates a single program.

Listing 2.7: Generating multiple programs

```
seq 1 100 | parallel -L1 ./generateProgram.sh
```

As described before in Section 2.1.1 Csmith was used to actually generate the programs, however Csmith generated programs are not guaranteed to terminate. Fortunately, it is easy to determine whether a program terminates by simply running it. Having to deal with programs timeouts due to Csmith introduces unnecessary noise in the experiments. Therefore it was decided that non-terminating programs should not be generated within this infrastructure. This was achieved by re-generating the program until it terminated within a small timeout. The timeout was chosen at 2 seconds, which should be plenty of time for natively executing Csmith programs, which are quite small by nature.

Finally the Csmith generated terminating program is also instrumented to produce the final result. The instrumentation providing hooks for symbolizing some global variables and function call logging is described in more detail in Section 2.2.1. There are several reasons for inserting the instrumentation at this point. First, the behavior can remain unchanged if the hooks are linked with *noop* functions, therefore nothing is lost by inserting the instrumentation here. Second, the instrumentation is used by most of experiments so it is convenient to insert it at this point. It can also be a time consuming operation so it was deemed best to pay the price once, upfront. Finally these scripts should also adhere to the side-effect, garbage free policy so that they can be ran in parallel with each other.

2.2.6 Reducing programs

The testing infrastructure can also aid in designing an oracle for C-reduce. We can provide a template implementation of the oracle revolving around the single experiment script that should require minimal programmer intervention to be able to be adapted for a given bug.

The general outline is to compile the program under strict error policies to weed out as much of possible undefined behavior introduced by C-reduce. Then run it natively to ensure it still terminates and pass it to the single experiment script to actually expose the bug. The output of the single experiment script is captured. The user then only needs to capture the essence of the bug in this output. That usually involves identifying a few snippets of text to be present in the output and enforcing them with *grep*. Note that this script must run in the same environment as the one where the mismatch is reported, so the single experiment is performed in the same configuration as before.

2.3 Case Studies

This Sectionpresents an account of applying the testing approach to find bugs in the KLEE, Crest and FuzzBALL symbolic execution engines.

2.3.1 KLEE

KLEE was chosen as the main case study, as it is a popular and well maintained symbolic execution engine. It is also developed in-house so the access to feedback to our report, was quick and easy. This enabled many iterations of the testing approach and thus was most successful.

The experimental setup is described first (§2.3.1) followed by the description of the methodology used (§2.3.1). Then an overview of the experiment runs (§2.3.1) is given as well as a summary of the bugs found, with a discussion of a few representative bugs (§2.3.1). Finally the experience of applying the approach to a real application (§2.3.1) is discussed.

Experimental Setup

KLEE commit `637e884bb` was used to start the experiments and then patched as needed to get around some bugs. KLEE was built using LLVM version 3.4.2 and STP commit `a74241d5`. Initially, version 1.6 of STP was used in a small number of our experiments. Programs were generated with Csmith version 2.3.0 and reduced C-reduce commit `49782e718`. The experiments were run on an 8-core 3.5GHz Intel Xeon E3-1280 machine with 16GB of memory. Clang-3.4 was used to compile the Csmith programs and *whole-program-llvm*[1] to compile *grep*.

To automate most of the experiments, the infrastructure described in Section 2.2 was used. This gave the ability to consistently repeat a certain configuration of the whole system while also giving great flexibility should configuration need to change.

Methodology

The experiments were conducted in *batches*, with essentially one batch for each bug found. In each batch, the following steps were performed:

1. Configure the experiment (what kind of programs to generate, mode to use options to pass to KLEE). Initially the default configuration of Csmith and KLEE were used.
2. Run the experiment (typically overnight).
3. Reduce the first program exposing a bug, and sometimes further manually simplify it slightly to make it more readable.

[1] https://github.com/travitch/whole-program-llvm

Table 2.2: Summary of runs in different modes.

Mode	# runs	Avg input size (LOC)	Avg time per run (s)	
			Native	KLEE
A	520,930	1,622	0.0500	0.808
B $<$,$>$	42,162	1,642	0.0581	1.69
B $\leq$, $\geq$	42,162	1642	0.0581	1.69
B range	42,162	1,642	0.0581	1.68
B divisors	42,162	1,642	0.0581	2.56
B $<$, $>$ & coverage	1,992	1,637	0.0726	91.8
C	6,625	1,640	15.82[1]	22.21

[1] Combined runtime of all replayed test cases.

4. Report the bug, attaching the reduced program.
5. Find a way to avoid the bug and reconfigure the experiment accordingly.

The reason for the last step is that certain bugs would reappear over and over again, making it difficult to identify new bugs. Therefore an iterative approach was adopted in which once a bug was identified, we worked on either fixing it (or incorporating the developers' fix if available in a timely manner), or more often reconfiguring our experiment to avoid it. In the latter case, some C features would be disabled in Csmith so that the bug would not be triggered (for example, once a bug involving incorrect passing of structures by value was found, passing structures as arguments was disabled) or changed the KLEE options so that the affected code would not run (for example, by disabling the *counterexample cache* [5] which was involved in one of the bugs).

At the end of our experiments the following options were used: *no-arg-structs*, *no-return-structs*, *no-arg-unions*, *no-divs* and *no-const* in Csmith, and *check-overshift=false* and *use-cex-cache=false* in KLEE. For mode C the *no-checksum* option was used in Csmith to disable the expensive checksum computation, since the output oracle was not used.

Optimization levels -O0 or -O1 were used to compile the generated programs, each with equal probability. Higher optimization levels were also attempted, however every Csmith-generated program compiled with optimization level -O2 or higher exposes the *vector instruction unhandled* bug in KLEE (§2.3.1), and therefore the higher optimization levels were used only for a small number of runs. The timeout for KLEE in modes B and C was set at 100s, as we accounted for constraint solving. In mode C, we also set the maximum number of forks (i.e. paths to be explored) to 200.

Summary of Runs

In total almost 700,000 programs were generated and tested. A summary of all the runs can be found in Table 2.2. They are divided by the different modes they used. Most runs, around 520,000, were performed in mode A, which as expected have the shortest average running time. In mode B around 168,000 runs were conducted, which on average took twice as long as those in mode A. Finally, 6,625 runs were performed in mode C, which were around 44 times more expensive than those in mode A. Note that the average runtime for native runs in mode C includes replaying all generated test cases. Overall, around 124 hours was spent in mode A, around 140 hours in mode B, and around 70 hours in mode C. The technique found the bugs within the first 5000 runs of each batch. This means that the batches could have been configured to run for only 2.5h, but it was convenient to do longer overnight runs.

For mode B, Table 2.2 also shows the number of runs performed with each way of constraining inputs to a single value. The runs involving inequalities and ranges took a similar amount of time, while those involving divisors took longer, as they involved more difficult constraints. The expensive coverage oracle was excluded in all runs, except 1,992 mode B runs with $<$, $>$ constraints, which took around 92 seconds per run on average. Using the coverage oracle was observed to involve about 10 times more instructions (as the instrumentation also uses code from `libc`), and a

Table 2.3: Summary of bugs found in KLEE, including the mode used, the oracle(s) that detected them, and the size of the reduced program used in the bug report. Issues in bold have been fixed. We omit links to the bug reports in this version of the paper, to preserve double-blind reviewing.

Bug description	Mode	Oracle	Reduced size (LOC)
Some unions not retaining values	A	output	11
Incorrect by value structure passing	A	output	18
Overshift error triggered by -01 optimisations[1]	A	output	5
Vector instructions unhandled, caused by -O2 optimisations	A	output	6
Floating point exception	A	crash	14
Incorrect handling of division by 1	B	function calls & output	17
Execution forks unexpectedly	B	function calls	14
Segmentation fault due to % operator	B	crash	12
Incorrect casting from signed to unsigned short[3]	B	output	27
Abnormal termination in STP solver	B, C	crash	10
Assertion failure in STP solver 1.6[2]	B, C	crash	–[2]
Replaying read-only variables not handled	C	crash	8
Unexpected interaction between file system model and replay library	C	function calls	9
Divergence b/w test generating path and test replay path[3]	C	function calls	21

[1] Remains unclear whether this is a KLEE or compiler optimization bug.
[2] Not explored further as the bug seems to have been fixed in the newest release of STP.
[3] Fixed prior to reporting as the side effect of what looks to be an unrelated patch.

significant number of extra I/O operations, all of which contribute to the significantly higher cost per run.

Finally, note that individual runs in modes B and C varied considerably, depending on the constraint solving queries generated in each run. For instance, mode B runs ranged between 0.01s to 99.6s (remember our mode B timeout was 100s). The reason some runs were very quick was that only a small part of the code was executed at runtime.

To reduce bug-inducing programs, C-reduce usually took a few hours. The most expensive reducing job was for the performance anomalies, where the reducing process took more than 24 hours. However, C-reduce only need to be run a few times, once for each bug found.

Bugs found

Table 2.3 summarizes the 14 bugs found using the approach. All bugs were reported to the developers, except one which had already been reported and another three which had already been fixed before we managed to report them. At the time of writing, the bugs in bold had already been fixed.

As can be seen from Table 2.3, a variety of bugs have been found, involving the handling of structures, division, modulo, casting, vector instructions and more, as well as issues having to do with constraint solving and replaying test cases. These bugs were revealed by different modes and oracles. 5 bugs were found in Mode A, 6 bugs in mode B and 5 bugs in mode C, with 2 bugs found in both modes B and C. In terms of oracles, the crash oracle found 5 bugs, the output oracle 6 and the function call chain oracle 4, with 1 bug found by both the output and the function call chain oracles.

Listing 2.8: Reduced program exposing a bug where union fields are not updated correctly. The native run correctly prints *f3 534*, while the KLEE run prints *f3 22*.

```
1  union U0 {
2    signed f3 :18;
3  };
4
5  static union U0 g_988 = { 0UL };
6
7  int main(int argc, char* argv[]) {
8    g_988.f3 = 534;
9    printf("f3␣%d␣\n", g_988.f3);
10   return 0;
11 }
```

Listing 2.9: Bug in which execution forks unexpectedly. *Should be printed once* gets printed twice in KLEE.

```
1  static int g_10 = 0x923607A9L;
2
3  int main(int argc, char* argv[]) {
4     klee_make_symbolic(&g_10, sizeof(g_10), "g_10");
5     if(g_10 < (int)0x923607A9L)
6         klee_silent_exit(0);
7     if(g_10 > (int)0x923607A9L)
8         klee_silent_exit(0);
9
10    int b = 2;
11    int i = g_10 % (1 % g_10);
12    i || b;
13    printf("Should␣be␣printed␣once\n");
14 }
```

As mentioned before, the size of the Csmith programs we generated is on average 1600 lines of code. The last column of Table 2.3 shows the size of the reduced programs. In all cases, C-reduce managed to reduce the bugs substantially, to fewer than 30 lines of code, with most bugs at 14 lines of code or fewer.

Below, some examples of the bugs found by the approach are given, including the reduced programs that were reported to developers.

Bug: Some unions not retaining values. Listing 2.8 shows an example of a bug found in Mode A. The program initializes a union containing a signed field of non standard length, and then writes 534 to that field and prints it. Running the program natively correctly prints out 534, while running it with KLEE prints out 22 (which represents the lower 9 bits of 534).

The root cause of this bug is an optimization in KLEE which uses faster functions for memory writes of size 1, 8, 16, 32 or 64 bits. The code contained a check which enabled the optimization only if the write in question was less than or equal to 64 bits. If this was not the case, the slower general approach was used.

Of course, this check was incomplete, which caused the program in Listing 2.8 (for which LLVM 3.4 generates a memory access of size 24) to incorrectly run the optimization path, and thus behave incorrectly. Interestingly this bug was only introduced when KLEE moved to LLVM version 3.4 as it is not present in LLVM version 2.9. The bug has now been fixed.

Bug: Execution forks unexpectedly. Listing 2.9 shows a bug found in mode B. It is an interesting bug, as it causes KLEE to fork on line 12 and explore two paths for no obvious reason. Unfortunately, it has not been debugged yet, so the root cause is still unknown. (Note that the text *Should be printed once* is added manually by us to the reduced program for clarity).

Listing 2.10: Program exposing division by 1 bug in KLEE.

```
1  #include <stdint.h>
2  static int32_t g_976;
3  int32_t func_46() {
4    printf("function call\n");
5    return 0;
6  }
7
8  void main() {
9    klee_make_symbolic(&g_976, sizeof g_976);
10   int32_t *l_1985 = &g_976;
11 lbl_2550:
12   func_46();
13   *l_1985 &= 2;
14   if ((3 ^ *l_1985) / 1)
15     goto lbl_2550;
16 }
```

Listing 2.11: Segmentation fault bug in KLEE due to incorrect handling of some modulo expressions.

```
1  int a, b;
2  safe_lshift_func_int16_t_s_u(short p1, p2) {
3      p1 < 0 || p1 ? p1 : p2;
4  }
5
6  main() {
7    klee_make_symbolic(&a, sizeof a);
8    if (a > (int)2453014441)
9      klee_silent_exit();
10   int i = a % (1 % a);
11   safe_lshift_func_int16_t_s_u(i, i || b);
12 }
```

The native version of the program omits lines 4–8, thus executing the program with `g_10 = 0x923607A9L`.[2] The extra instrumentation run by KLEE marks the global variable `g_10` as symbolic and constrains it to have the unique value `0x923607A9L`, using less than and greater than constraints, as discussed in Section 2.1.2.

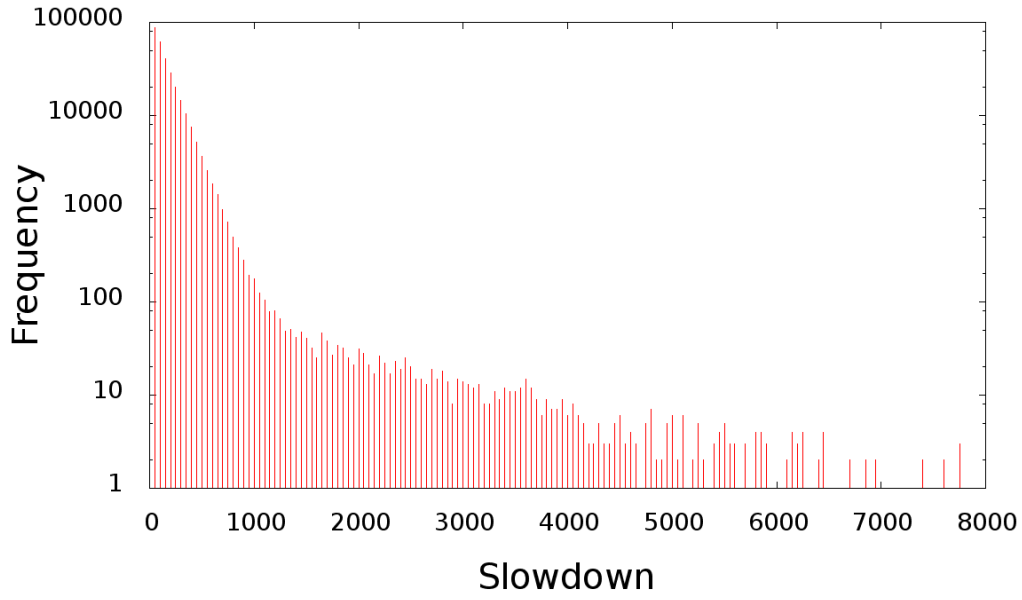
This bug is caught by the function call chain oracle, which shows additional function calls in the KLEE execution, which are caused by KLEE incorrectly exploring an additional infeasible path.

Bug: Incorrect handling of division by 1. When executed natively, the code in Listing 2.10 loops indefinitely. The `if` statement at line 14 keeps evaluating to true and therefore the execution jumps back to line 11. In KLEE, the `if` statement evaluates to false, so KLEE terminates after a single iteration. This bug was caught by both the output and function call chain oracles. The bug was found in mode B, but the reported program does not constrain the symbolic variable to have a single value, as we realized this is not needed to expose the bug (so the automatically reduced program was several lines longer). It should also be noted that prior to running C-reduce, the Csmith program exposed the bug without containing an infinite loop.

This bug was initially avoided bug by disabling division expression generation in Csmith, but the bug was later debugged and fixed by the developers. The problem was that division by a constant is optimized prior to invoking the solver using multiplication and shift operations. However, the optimization is incorrect for constants 1 and -1. The fix was to disable this optimization for these special cases.

[2] In practice, the native program is linked against a library that defines the `klee_*` functions to do nothing. This has the advantage of ensuring that exactly the same code is run both in native mode and with KLEE.

Figure 2.3: Distribution of the slowdown introduced by KLEE for the runs in mode A. Note that the y axis is logarithmic. There are 28 values larger than 8000 (with the highest one being 14,991) which are not shown for readability purposes.



Bug: Segmentation fault due to % operator. The code in Listing 2.11 causes a segmentation fault in KLEE. The bug was found in mode B and diagnosed by the developers to be caused by an incorrect semantics assigned to the % operator when negative numbers were used as divisors. The second part of the code that constrained variable `a` to have a single value was manually removed prior to reporting the bug, as it was not needed to expose this bug.

Performance analysis

For each random program generated and run, its native runtime and the time KLEE took to execute it were recorded. The ratio between the two runs (i.e. the slowdown added by KLEE) was then computed thus flagging any potential anomalies. The focus was on the runs performed in mode A, although in the future mode B runs could also be analyzed to identify difficult constraint solving queries.

Figure 2.3 shows the distribution of the slowdowns observed in mode A runs. The mean of the distribution is 120 with a standard deviation of 228. There are 28 values larger than 8000 (with the highest one being 14,991) which are not shown for readability purposes.

We believe such outliers could be considered performance bugs, or at least examples of features for which symbolic execution tools have a good reason to introduce a disproportionately high slowdown. Two such cases were reported to the KLEE developers, one which performs a large number of function calls, and the other in which a large number of memory objects are allocated on the stack.

The former program is shown in Listing 2.12. The program increases the value of g_647 in increments of 7 until it's equal to 37. Note that for the condition to be satisfied g_647 needs to overflow multiple times, meaning that the program has to run a total of 56,179 iterations. In this example, the addition is done using a function call which is performed at every iteration and is the likely cause of the performance anomaly.

Listing 2.12: Performance anomaly due to large number of function calls.

```
1  unsigned short safe_add(int ui1, int ui2) {
2    return ui1 + ui2;
3  }
4
5  static int g_647 = 0;
6  int main () {
7    for (; (g_647 != 37);
8          g_647 = safe_add(g_647, 7)) {}
9  }
```

Table 2.4: Function call divergence in grep.

KLEE	Native
epsclosure	epsclosure
state_index	state_index
dfamusts	dfamusts
kwsinit	kwsinit
kwsincr	kwsincr
dfamusts	dfamusts
kwsprep	kwsprep
grepfile	grepfile
fillbuf	fillbuf
fillbuf	fillbuf
grepbuf	
EGexecute	
kwsexec	
close_stdout	close_stdout
close_stream	close_stream

Grep case study

Applying the techniques presented in this chapter to real programs was also considered. For this purpose, a popular UNIX utility grep was used, which finds lines of text matching a certain string pattern.

We found several mismatches in mode C, which were caught by the function call chain oracle. An example of the difference between the function call chain executed on one path explored by KLEE and the corresponding native run can be seen in Table 2.4. This bug has not been reported yet, as it was found difficult to reduce (C-reduce works on a single C file) and debug (given the much larger size of grep). Overall, this experience has reinforced initial preference for using generated programs, which present the advantages discussed in §2.1.1. However, with more engineering work, the approach could be applicable to real programs too.

2.3.2 CREST and FUZZBALL

To show the generality of the technique, it has also been applied to two other symbolic execution engines. Crest [16] and FuzzBALL [32] were chosen, because they are different from KLEE in important ways: Crest is a concolic execution tool [22, 42], a variant of symbolic execution which differs significantly at the implementation level from the one used by KLEE, while FuzzBALL is a symbolic execution for binary code, which again results in significant differences in the way the tool is implemented.

At the implementation level, to apply the testing infrastructure to a new tool, one obviously has to be aware of the way the code is compiled and run with each new symbolic executor. As further described in §2.2.3 one needs to know the API the tool uses to mark inputs as symbolic. As described before the interface for creating and constraining variables is symbolic executor agnostic, which enables the use of multiple symbolic execution engines by simply changing the library linked appropriately.

Table 2.5: Summary of bugs found in Crest, including the mode used, the oracle(s) that detected them, and the size of the reduced program used in the bug report. Issues in bold have been fixed.

Bug description	Mode	Oracle	Reduced size (LOC)
Crest			
Return struct errror	A	crash	8
Big integer in expression	B	output	9
Exploring a branch twice	B	output	9
Non 32-bit wide bitfields	B	output	9
FuzzBALL			
STP div by zero failure[1]	B	crash	15
Strange term failure	B	crash	11
Wrong behaviour	B	output	12

[1] Fixed in the upstream version of STP.

Listing 2.13: Crest explores two branches in both of which a is smaller than 2 billion.

```
1 unsigned int a;
2 int main() {
3   __CrestUInt(&a);
4   printf("a: %d\n",a);
5   if( a < 2294967295) {
6     exit(0);
7   }
8 }
```

Crest

Crest implements the concolic form of symbolic execution [22, 42], in which the code is executed on concrete values and constraints are gathered on the side. To generate a new path, one constraint is negated, a new concrete input is generated and the process is repeated. Therefore, one important difference with KLEE is that paths are explored one at a time. A second important difference (but orthogonal to the first) is that Crest instruments programs for symbolic execution (using CIL) as opposed to interpreting them like KLEE.

There were several practical difficulties faced when applying this approach to Crest. First, Crest is less feature-complete than KLEE. For example, it does not support symbolic 64-bit integers and its solver does not support some arithmetic operations such as modulo. However, a workaround was devised to avoid this issues. We simply ignored the 64-bit integers in the symbolizing library and passed *no-modulo* flag to Csmith, to alleviate these issues.

Second and more importantly, Crest is not an actively developed project, and the tool does not seem to expose many options to enable or disable various sub-components, like KLEE does. Therefore it was difficult to find ways around the bugs we discovered, in order to find new bugs.

In spite of these difficulties, the approach found four bugs in Crest within 1000 runs or about 2

Listing 2.14: Crest explores the branch where $a = 1$ twice and then fails with *Prediction failed!* message.

```
1 unsigned int a, b;
2 int main() {
3     __CrestUInt(&a);
4     if( a < 1 || a > 1) {
5       exit(0);
6     }
7     (-1 > a ) || b;
8     printf("a: %d\n", a);
9 }
```

Listing 2.15: FuzzBALL crashes prematurely with *Strange term cast(cast(t2:reg32t)L:reg8t)U:reg32t ^ 0xbc84814c:reg32t in address* failure.

```
1  unsigned int g_54 = 0;
2  unsigned int g_56 = 3162800460;
3  //marked symbolic at runtime
4  unsigned int magic_symbols[1] = {0};
5
6  void main ( void ) {
7      g_54 = *magic_symbols;
8      if( g_54 < 0 || g_54 > 0)  exit(0);
9      g_56 ^= 0 < g_54;
10     printf("g_56: %u\n", *(&g_56));
11 }
```

Listing 2.16: FuzzBALL symbolic run prints out *1*, whereas native run with g_893 = 124 prints *0*.

```
1  unsigned int g_893 = 124;
2  int safe_sub(long long p1, int p2) {
3    return (p1 ^ ~9223372036854775807LL) - p2 < 0 ? 0 : p2;
4  }
5  static unsigned int magic_symbols[1] = {0};
6
7  int main() {
8    g_893 = *magic_symbols;
9    if(g_893 > 124) exit(0);
10   if(g_893 < 124) exit(0);
11   printf("%u\n", safe_sub(1UL ^ g_893, 1));
12 }
```

hours worth of computation time. The bugs were then reported to developers. A summary of the bugs found and reported is shown in Table 2.5.

The first bug is exposed by a program with functions that return structs or unions. Here the Crest compiler throws an error when given such programs as input. Interestingly, KLEE had similar problem with struts and function calls.

The other three bugs are exposed in mode B. For instance, the code in Listing 2.14 triggers a problem similar to the one triggered by the code in Listing 2.9 in KLEE. Here Crest explores the *else* side of the branch at line 4 twice, failing the second time with a "Prediction failed" error. This bug was not debugged, but it disappeared, once the fix for handling big unsigned integers was merged in.

Listing 2.13 shows a bug that looks like a trivial example, but it was still not behaving correctly in Crest. The bug was debugged by the developers to be the interaction between Crest and its solver. The values were passed to the solver as a C *int* type, which meant that unsigned integers bigger than 2 billion, as in the example, would overflow and become negative. This obviously caused the error. The developers proposed a fix in which the interaction with the solver would be done through other integer representations such as GMP[3] integer or a string. I implemented this fix which was later merged in.

FuzzBALL

FuzzBALL is similar to KLEE in that it implements the non-concolic style of symbolic execution, where execution starts with unconstrained symbolic variables. On the other hand, like Crest, FuzzBALL executes paths one at a time, keeping only a lightweight execution tree in memory. Finally, like KLEE, FuzzBALL interprets the code rather than instrument it for symbolic execution,

[3]https://gmplib.org

but does this at the binary rather than LLVM bitcode level. These design decisions make FuzzBALL an interesting complement to KLEE and Crest for the technique.

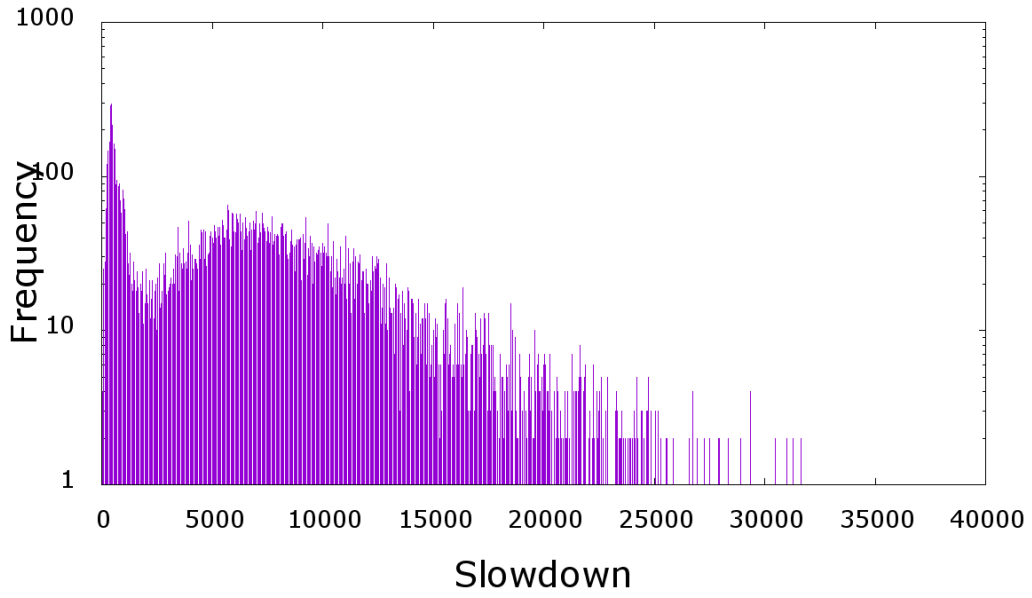
The only major engineering challenge that needed to be addressed to use the technique on FuzzBALL was related to the fact that unlike KLEE and Crest, FuzzBALL does not provide an API for marking variables as symbolic. Instead, one has to specify on the command line the address range(s) that the tool should mark as symbolic (e.g. 16 bytes starting with `0xdeadbeef`). Therefore, the library created for FuzzBALL defines a large static array which is marked as symbolic from the command line. At runtime, when a variable is supposed to be marked as symbolic, unused bytes from the static array are obtained and copied to the variable to emulate the behaviour of `make_symbolic` functions that higher level symbolic execution engines like Crest and KLEE provide.

Within 2000 runs or about a day worth of computation time the approach has found three bugs in FuzzBALL, all of which have been fixed. A summary of the bugs found is shown in Table 2.5. Like KLEE, FuzzBALL uses STP as its main constraint solver, and the technique managed to trigger the same STP bug while testing FuzzBALL. The other two bugs caused FuzzBALL either to crash or to compute the wrong results. The program triggering the latter is shown in Listings 2.16. The problem was debugged down to an incorrect simplification rule, which worked for small integers, but caused this problem when there was a signed overflow. The developers fixed this by removing the rule.

The bug caused by Listing 2.15 proved to be a lot harder to debug. As per developer response[4] it seems that the problem stems from the fact that FuzzBALL is unable to distinguish between pointers and integers well, due to the nature of machine code at which it operates. However it still finds this distinction useful for various reasons, therefore it employs some heuristics to classify words either into integers or pointers. This heuristics fails for this particular example, interestingly enough somewhere in the *printf* part of code. The fix was adding some more simplification rules and implement another option for FuzzBALL that helps better control the execution thus avoiding this issue.

Performance

Figure 2.4: Distribution of the slowdown introduced by FuzzBALL. Note that the y-axis is logarithmic



As another intriguing experiment the distribution of the slowdown incurred by FuzzBALL was also plotted in similar fashion than KLEE in Section 2.3.1. Note that this analysis is not interesting for Crest as Crest executes the code natively. The most interesting feature of this distribution is the hump, indicating a bi-modal distribution, a phenomenon not observed in KLEE's case. The mean

[4]https://github.com/bitblaze-fuzzball/fuzzball/issues/20

slowdown for FuzzBALL is 7000x with standard deviation of 5000x. The slowdown is significantly worse than in KLEE's case, which can be attributed to FuzzBALL executing more instructions symbolically. It symbolically interprets *libc* as well as the system calls, something KLEE does not do. In addition it starts the symbolic interpretation before main. For example, it interprets the dynamic linker, which accounts to a slow startup time. There are options to disable symbolic interpretation of code before start of *main*, but the experiments were run without them.

There were only 17 runs with slowdown larger than 30000x, with the highest one at 39700x slowdown. This could suggest that FuzzBALL has less large outliers than KLEE, however it is reasonable to assume that the tail end of the distribution was cut of by timeouts. In either case it would appear the performance oracle is less suited for flagging performance issues in FuzzBALL than KLEE, either because FuzzBALL has more consistent runtime slowdown or because it is too slow to catch programs with outlining slowdowns due to timeouts.

Chapter 3

Metamorphic Testing of Symbolic Executors

This chapter builds on the infrastructure presented in Chapter 2 and extends it to perform metamorphic testing. It borrows the idea of metamorphic testing (§1.2.5) and applies it to symbolic execution engines in a similar fashion Chapter 2 applied differential testing to symbolic execution. The infrastructure largely remains the same, as many of the concepts translate well between the two approaches. However metamorphic testing presents additional challenges and opportunities for testing symbolic execution.

More precisely, metamorphic testing enables easy comparison of symbolic execution with ... symbolic execution. Previously the testing technique was mostly focused on constraining the execution to a single path, however metamorphic testing gives the ability to explore the difference of execution over multiple paths. Therefore, this chapter focuses heavily on testing multi path execution (mode C in the terminology of Chapter 2), where metamorphic testing should provide the main benefit. Firstly the generation of programs in which the symbolic execution can exhaustively explore all the paths within a relatively short timeout is described in Section 3.1. This process turned out to be computationally intensive, therefore only a relatively small opus of thousand programs was generated on which the experiments were ran.

The main difficulty faced was the interaction of symbolic data with the outside world. In general, to achieve multi-path symbolic execution some symbolic variables must take more than one value. Therefore it becomes very likely that a certain symbolic variable will take more than one value within a single execution path. For example consider a simple two path program, which branches on a 32 bit *int* value. The range of possible values that integer could have on one of the branches is at least two billion! This becomes a problem once the output oracle is used as the solver needs to be consulted to get a concrete solution for that variable. Of course the solver is free to chose any value within the constrains. The output oracle requires the solution to be unique. Otherwise a spurious output mismatch could be detected for equivalent symbolic executions due to the solver choosing different values for logically equivalent sets of constraints.

Three different approaches are presented to solve this problem. Firstly clever tricks are used to force the solver to always give the highest possible value for the given set of constrains (§3.3.1). Then a crosschecking approach is presented in Section 3.3.2, where the two version are ran within a single execution. Then asserts are used to ensure the symbolic values are the same. In a sense the variables are compared symbolically instead of concretely. Finally, in Section 3.3.3, the output oracle is replaced by a more detailed control flow checking oracle to remove the issue of concertizing symbolic variables altogether. These approaches are then evaluated on KLEE, Crest and FuzzBALL accompanied with a detailed description of the bugs found in Section 3.4.

3.1 Generating multi-path programs

By picking a program generated by the infrastructure described in Section 2.2.5 at random, there is a high likelihood of getting a program with a single path. To be more precise, it is a program where symbolic data is not present on any branch point. Therefore a symbolic executor follows just a single path. As said in the chapter introduction it would be desirable to have multi path programs, where a symbolic executor can feasibly explore all the paths in the program.

Multi path programs present several advantages over single path programs when testing symbolic execution. They leverage a larger portion of symbolic executor's code, as they non trivially explore multiple paths. Depending on the type of symbolic executor used, they can test the fidelity of keeping multiple path prefixes in memory for tools like KLEE. Or rolling the state back to the previous branch point for concolic tools like Crest. Additionally they can stress the path exploration heuristics and algorithms as well as constraint caching, across multiple paths.

To generate multi-path programs, a simple but effective algorithm was devised. It builds upon the previous program generation tooling, by simply regenerating programs until it finds one meeting the requirements. Initially, the requirements were set so symbolic execution would terminate within the timeout while exploring more than one path. Note that symbolic execution terminating indicates that there is no possibility of further exploration as all the paths have been explored already. KLEE was chosen as the symbolic executor for this task as we deem it the most reliable of the ones used. The outline of the script for generating multi-path program is given in Listing 3.1:

Listing 3.1: Pseudo code for generating multiple paths programs, where symbolic execution terminates, exploring all paths. generateProgram is the process described in Section 2.2.5

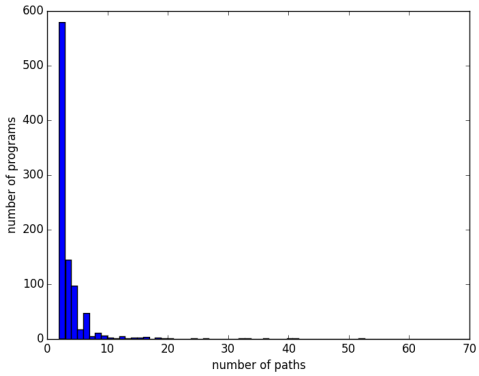
```
do {
        p = generateProgram
} while(symbolic execution times out || number of explored paths < 2)
```

At implementation level very little modifications were needed. There was no need to change the instrumentation of the programs. A dedicated version of the symbolizing library was written for this task. Instead of symbolizing and constraining the variables it just symbolized them. Additionally it kept a count of how many variables it had symbolized already and would stop symbolizing them after a certain threshold. In other words, after a threshold number of calls to the symbolizing function, the function would just return, without symbolizing anything. Controlling how many variables are symbolized was found important when exploring the relation between the amount of symbolic data and the number of paths in Csmith programs. Initially the threshold was set to 2.

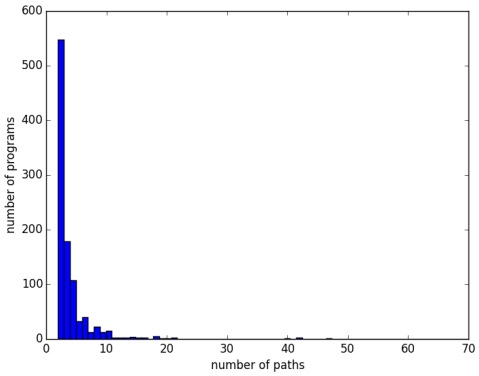
3.1.1 Csmith-generated programs path distribution

The algorithm above does successfully find Csmith programs with multiple paths. However, the number of paths in a typical resulting program would be 2, which is not ideal for testing symbolic execution. We theorized that increasing the number of symbolized variables and removing loops could be two contributing factors to the number of explored paths. Symbolizing more variables should make it more likely that more branches in the execution of the program depend on the symbolized variable. Whereas removing loops should prevent path explosion and therefore make it more likely that symbolic executor terminates.

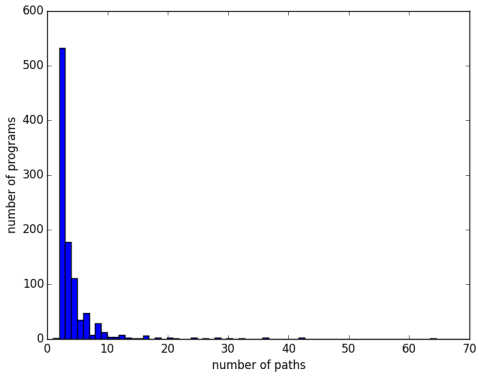
We decided to perform a more thorough experiment to explore the effects of symbolizing more variables and loops on the number of paths generated by the process. Several experiments with different parameters were ran. A summary can be found in Figure 3.1. For each experiment, a batch of 1000 programs were generated using the algorithm in Listing 3.1. Different numbers of symbolic variables and different configurations of Csmith with respect to loops were used for each experiment. The number of paths explored was then plotted against the number of programs with that number of paths to obtain histograms. Programs with one path were ignored in this experiment because preliminary experiments showed that they dominate and would therefore make



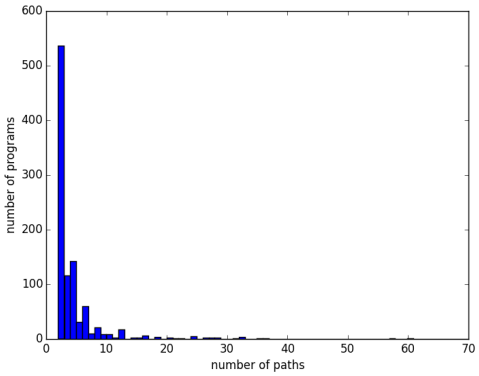
(a) Default



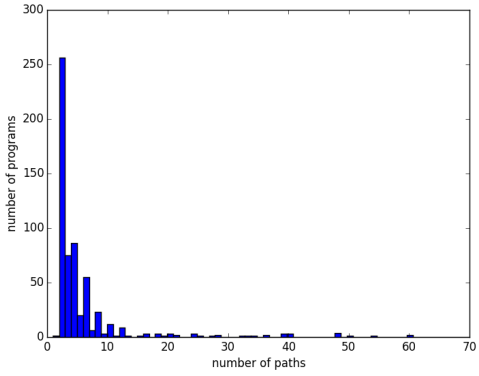
(b) No loops



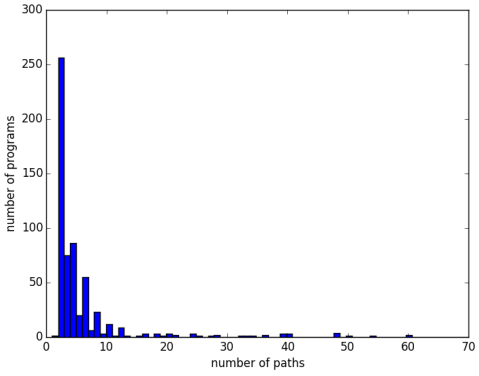
(c) No loops, no *goto*



(d) 8 symbolic variables



(e) No loops/*goto*, 8 variables



(f) No loops/*goto*, 100 variables

Figure 3.1: Number of completed paths distributions for various setups of generated Csmith programs. Unless otherwise noted default Csmith settings are used, with 2 variables being symbolized. The x-axis is limited to values between 2 and 70.

the result less clear. The batch sizes would need to be increased to get interesting results, which was found undesirable, due to resource limitations.

To get a good grasp of the behavior of number of paths in Csmith program with respect to loops and the amount of symbolic data, 6 experiments were run. A default Csmith configuration with 2 global variables symbolized showed in Figure 3.1a. Then Csmith probability of emitting loops was set to 0, and experiment repeated resulting in Figure 3.1b. For Figure 3.1c Csmith was also prevented from emitting *goto*-s, to further remove possibility of loops from the programs. The Csmith configuration was then reset back to default, but the number of symbolized variables increased from 2 to 8 resulting in Figure 3.1d. Csmith was then again prevented from emitting loops and *goto*-s and experiments were ran with 8 and 100 symbolic variables, resulting in Figures 3.1e and 3.1f respectively.

By manually inspecting the distributions in Figure 3.1, we concluded that the distribution seems to be exponential regardless of the experiment setup. This is not a particularly surprising result as programs with smaller number of paths are more likely to terminate within the timeout and are therefore more abundant. It can also be seen that removing loops seems to have little impact on the distribution, whereas increasing the number of symbolized variables does produce more favorable distributions, i.e. distributions where more programs produced have higher number of paths explored.

On the other hand, some graphs look very similar and it is impossible to answer some questions. For example is there a difference between keeping and removing loops? Or does increasing the number of symbolized variables from 8 to 100 make a difference? To answer these question KS-test was used. Its null hypothesis states that the two distributions under test are the same. Therefore a low p-value means that the distributions are different. A more detailed and precise description can be found in Section 1.4.

The KS-test p-value between the default and no loop distributions is 0.02. That means the null hypothesis would be accepted on some confidence levels and rejected on others. In other words it's inconclusive whether removing loops does anything useful in this context. The difference between the default and no loop, no *goto* configuration seems more significant with p-value of 0.003. Therefore, it can be concluded that completely removing loops does make some difference. The KS-test also confirms the difference between 2 and 8 symbolized variables in default Csmith configuration. There seems to be absolutely no statistically significant improvement between symbolizing 8 and 100 variables, since the p-value is 0.40 between the two distributions. Note that most Csmith programs do not have 100 variables to be symbolized, so symbolizing 100 variables means symbolizing all variables in most cases, which makes this result less surprising. Finally, the difference between loops and no loops does seem to be more pronounced in 8 symbolic variables case as opposed to two symbolic variables case with the KS-test giving a small p-value.

This information was used to decide how to configure the generation of the opus of tests that is used in the rest of this chapter. It was concluded that having 8 global variables symbolized is sufficient to generate programs with good amount of paths. It should also strain the symbolic execution less than having all of them symbolized, which is important to run the experiments as fast as possible. The loops were to be kept in as they don't seem to impact the distribution in a significant manner. Having Csmith as close as possible to the default is also desirable so keeping the loops in the programs seemed best. Default Csmith configuration is closer to real world programs and therefore more likely to find meaningful bugs. Finally it was decided that programs with small number of paths are not interesting enough to generate and they should have as many paths as possible. Unfortunately, due to exponential nature of the distribution, increasing the minimal acceptable number of paths significantly increases the amount of time it takes to generate a program. Therefore, the lowest number of acceptable paths was set at 30 for the final configuration, which was deemed a good trade-off between feasibility of generation and interestingness. That still increased the run time multiple times, which was solved by distributing program generation over a cluster of machines.

3.1.2 Distributed program generation

The problem of program generation is well suited for distributed computation. There is no need to distribute the inputs as the input is in essence a random seed and can be obtained locally. The result is just a single program of a couple kilobytes. The computation itself involves a significant amount of IO operations for generating programs, which importantly are not a part of the result and can therefore be done locally. It also includes symbolic execution where constraint solving pushes the CPU to its limits. This indicates that large gains can be made by distributing program generation and some might even call the problem embarrassingly parallel. Due to the design of the testing infrastructure it should be easy to make program generation distributed. The work involved can be broadly split into two tasks: getting the programs to run in distributed fashion and obtaining the machines to run it on.

The first problem of generating programs in distributed fashion has largely been solved by the design of the testing infrastructure. The existing infrastructure already leverages GNU parallel [43] heavily to execute multiple runs or generate programs in parallel. It is a map operator for the testing infrastructure's scripts, akin to Haskell's *map* function. For example, the program generation script can be seen as a function from a number to a program, where the number represents the name of the generated program. To generate a whole batch of programs, the script just needs to be *map*-ed over a list of numbers, which is exactly what GNU parallel (or *xargs* for that matter) does. Due to careful design of the scripts, so that they are effectively side effect free and can run alongside each other, GNU parallel's innate ability to distribute jobs across CPU cores can be used. In addition, it can also run the task across multiple machines by only flipping a switch, which is how distributed program generation was achieved. This does come with its own set of little challenges described in more detail further below.

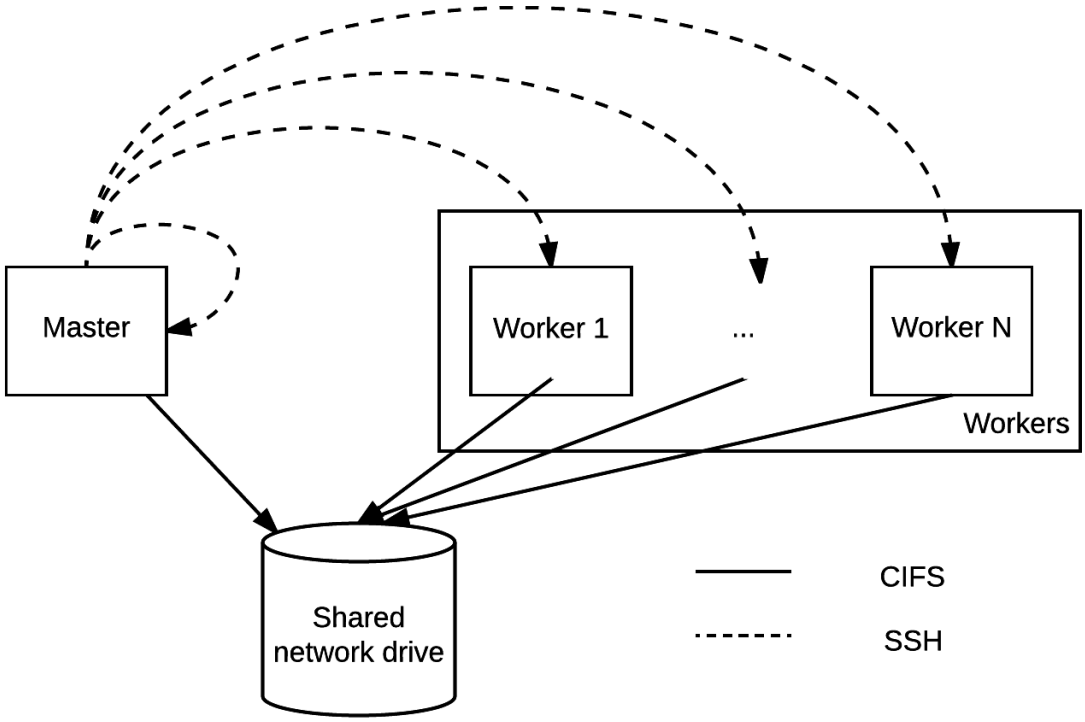
The second problem of obtaining machines to generate the programs on was solved by Microsoft's kind sponsorship of the department by giving us Azure credits to use on our individual projects. Conveniently, all of my development was also done in a virtual machine. So the idea was to upload the VM to Azure, spawn many copies of it and generate program in a distributive manner.

Going to the cloud

The migration from a VirtualBox VM, which is what I've been using, to Azure cloud is easy once you know all the tricks. It involves changing the format of the virtual disk and installing a couple of Azure related utilities to the guest OS. The biggest issue here were relatively large file sizes, which quickly consumed all the disk space on my machine and also caused the upload to take a couple of days. Once the disk was uploaded in the correct format, I was able to boot up an exact copy of my VM in the Azure cloud.

The migration turned out differently from what I imagined. What I thought I would get is a virtual machine image from which I would be able to spawn VMs, but what I got was a virtual disk image, which I could copy around and attach to VMs. The difference is subtle but important: the disk image retains everything. The best example perhaps is the name of the machine. By attaching different copies of the disk to different VMs, all of the VMs have the same name. If I were to have a VM image, Azure would be able to change them in accordance to what you see on the Azure dashboard. The reason for not going with the image approach was that my infrastructure was installed under *home/user* folder which would get deleted. The point of VM images is to have no user and user gets created once the machine is provisioned, so converting my VM to be compatible with VM image would mean deleting all users (so Azure could create new ones as needed), but that would mean losing my infrastructure and thus defeat the point of this process. In hindsight, I should have setup my infrastructure elsewhere, but the result I got was good enough for my purpose. I was able to spawn machines with relative ease, but it was still a manual process. Perhaps the other approach would automate the task of spawning machines even further and thus enhance the scalability of program generation.

Figure 3.2: Architecture of the distributed program generation



GNU parallel over multiple machines

Getting GNU parallel to split the task over multiple machines is as easy as providing it with a file of network locations of other workers. GNU parallel will then periodically read the file to see if any workers went down or new ones were added. Then it will SSH into a worker which has a slot for the task and run it. The catch is that GNU parallel needs to be able to SSH into the workers without a password. Obviously the command that performs that task must be able to be executed on any of the workers. Both of these problems were resolved by having copies of the exact same VM. They were set in such a way that they could SSH into themselves without a password and they were obviously able to execute the same commands, because they were the same.

The result collection, i.e. writing of the generated files was done on a shared network drive.The IO traffic to the shared drive is very low compared to the computation each node has to perform, so this mechanism of result collection scales well for this particular problem and reduces the complexity of distributing program generation.

Another problem faced were environmental variables. The infrastructure relies heavily on environment variables to carry the current configuration of the experiment or even more simply just the current directory (PWD), where the results are written. In this case PWD would be set to the location of the mounted network drive. These environmental variables obviously have to be shared among the nodes, so that each node writes to the same location. Otherwise all the workers would write to their *home* directory, making the results dispersed across the workers. GNU parallel does provide a facility to copy over the environment variables, but unfortunately that does not cover the current directory. This was solved by making the command run by GNU parallel spawn a sub-shell which *cd*-ed into the current directory as passed in by the master and then calling the script.

The other scripts in the infrastructure, such as actually running the experiments, could also be distributed using this method. However, the experiments were fast enough, only taking a couple of hours, that I was not motivated to do so. The approach would work out of the box, however the problem arises when all the scripts need to be synced between the workers. This is a very frequent operation when running the experiments as opposed to program generation. In the current setup it would involve shutting down all the machines (all the workers as well as the master), making

a snapshot of the master, and then spawn all the workers again. This process can easily take a better part of the hour, even when scripted, so it's not practical. A solution would be to move the infrastructure to the network drive, but this is a step I was not motivated enough to take, as the experiments were running fast enough.

3.2 Semantics-preserving source-code transformations

Semantics-preserving program transformations are the core of our approach. They provide the opportunities for bugs to present themselves, by changing the program in a meaningful way, while still keeping the behavior the same. It is vital that the behavior of the program does not change, since equivalent behavior is the basis for the oracles to find mismatches. Should the original and the transformed program not have the same behavior, comparing them would be meaningless.

The transformations should also exercise parts of the symbolic executor where one would expect to find bugs. For example, injecting dead code has proven to be very successful in compiler testing[27], since compilers cannot know what code is dead at least until inspecting it, at which point interesting bugs can already emerge. On the other hand, symbolic executors run the program, therefore detecting dead code is trivial, symbolic execution simply never reaches it. Consequently, dead code injection is not an interesting transformation for testing symbolic execution. Therefore two transformation were devised specifically targeting symbolic execution: dead condition injection (§3.2.1) and swapping branches (§3.2.2). In addition, off the shelf solutions for semantic preserving transformations were also used. In particular non trivial source code obfuscation tools are semantic preserving transformations. The application of one such tool—tigress [13]—is described in Section 3.2.3. Finally q non realized transformations is discussed in Section 3.2.4.

3.2.1 Transformation: dead condition injection

Dead condition injection is a simple but effective transformation. It is inspired by the dead code injection transformation put in the context of symbolic execution. The idea is to inject a logical tautology to the branching conditions. An example of this transformation can be seen in Listings 3.2 and 3.3, where a tautology about a global variable (g_2) is inserted. Note this transformation increases the number of paths symbolic execution explore, because the tautology might include symbolic variables. Looking at Listing 3.3, the variable g_2 is an integer variable, which are symbolized by the infrastructure. Therefore symbolic execution has to explore the *if* statement twice, corresponding to the *or* expression. At least one of the two paths should have the same behavior as the original program, the other path is irrelevant as it is a byproduct of the transformation. Therefore, a subset type of oracle (§3.3.1) needs to be used with this transformation.

Listing 3.2: Dead condition injection original snippet

```
if(g_345 < 34283) {
        //other code

}
```

Listing 3.3: Dead condition injection transformed snippet

```
if(g_345 < 34283
   && (g_2 < 3 || g_2 >= 3) ) {
        //other code
}
```

The rationale behind this transformation is to add additional constraints on the paths. In essence it is a dead code injection transformation, but on the level of gathered path constraints. It is designed to stress the solver further, both in accuracy of constraint solving and expression construction. In other words, can the symbolic executor construct expressions to represent new path constraints? In addition this transformation also affects query optimizations such as constraint independence and caching [5].

Implementation

The transformation operates on conditional branch points in the program, such as the one shown in listing 3.2. It shows a conditional statement on a global integer variable *g_345*. After finding such points in the program, the transformation injects an expression that should always evaluate to *true*, exploiting the fact that $a \wedge true \iff a$. The expression that should always evaluate to *true* was of the form $gv \leq c \vee gv > c \iff true$, where gv is another randomly picked global variable and c is a random constant.

At the implementation level this transformation uses clang's LibASTMatchers and Rewriter utilities. It finds all the *if* statements in the program using LibASTMatchers. It then gets an expression that always evaluates to true from a "FactGenerator" and inserts it at the end of the *if* condition. Currently there are only two FactGenerators implemented: a trivial one that simply returns *1 == 1* and the one that returns expressions seen in Listing 3.3.

Extensions

Extensions and variants of dead condition transformation could easily be devised. For example, other tautologies could be used such as $gv = c \vee gv \neq c$ or $gv > 2 \implies gv - 1 > 1$ to name a couple. Alternatively and perhaps more interestingly static analysis could be performed to learn interesting facts, which could then be used to create expressions that are always true for a particular program, but not necessarily always true. However, the lack of tools that perform meaningful static analysis of the source code makes this sort of extension out of scope of this work. Additionally, in my experience static analysis performs very poorly on Csmith programs, therefore the practical use of this approach is limited. Finally the form of conditions injection could exploit the fact $a \vee false \iff a$, which would extend the range of tautologies that can be used to the ones that always evaluate to false.

3.2.2 Transformation: swap branches

Swap branches is a simple but effective transformation that swaps the order of *if/else* statements as shown in Listings 3.4 and 3.5. Its main goal is to change the Control Flow Graph, however as can be seen for the Listings some of the path constrains get negated to preserve the original behavior.

Listing 3.4: Swap branches original snippet

```
if(condition) {
    //code A
} else {
    //code B
}
```

Listing 3.5: The transformed snippet

```
if(!condition) {
    //code B
} else {
    //code A
}
```

This transformation is meant as the tiniest meaningful transformation that can affect symbolic execution. It can change the order in which the paths are explored, by changing the CFG as well as change the path constraints slightly. Changing path constraints is important to test the accuracy of constraint solving as well as solver expression construction as outlined in Section 3.2.1. The path constraints modifications in swap branch transformation are different from the dead condition injection one in two ways. First, the constrains are changed significantly less aggressively and do not increase the load on the solver. This is important both for performance as well as debugging the bugs once they are found. Second, this transformation does not change the number of paths it explores. It is unique in this respect among the transformations that were used in this chapter. Having the number of explored paths the same for original and transformed version of the programs is convenient as it strengthens the output oracle. In addition, it also introduces the opportunity for *explored paths oracle*, which simply checks that the number of explored paths is the same for both versions of the program.

Implementation

Implementing this transformation with LibTooling proved tricky, with the final implementation limited to only swapping the top nested *if*/*else* statements. In principle, the task should be easy: find all the *if*/*else* statements, swap the two bodies and negate the condition. Finding the statements is indeed easy with LibASTMatchers as shown in §3.2.1 as well as negating the condition by simply bracketing it and inserting a *!* in-front. However swapping the two bodies proved difficult.

Changing the code with LibTooling does not change the AST, it only manipulates the textual representation of the source code. In other words it uses the AST as a read-only source of information, based on which the textual representation can be manipulated. Naturally, this means a nice recursive tree-based algorithms cannot be used to manipulate the code. Consequently LibTooling is of no help when dealing with changing nested structures. For example, when traversing the AST consider encountering the top level *if* statement and swapping it. If then a nested statement is encountered, its location will still point to pre-swap location, so swapping it will completely ruin the code. There are similar problems when traversing the AST in a different order. This problem was simply avoided by only swapping the top level statement. Of course, solutions could be engineered to get around this problem, however they fall outside the scope of this project. Alternately, CIL could be used, but this problem was encountered late in the project so the move was not deemed worth it, especially due to OCaml being unfamiliar to me.

3.2.3 Transformation: source obfuscation

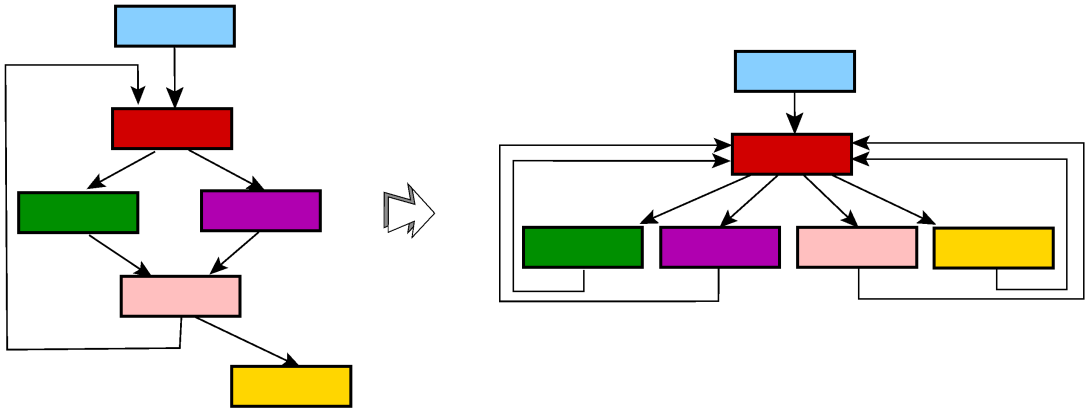
Source code obfuscation is a metamorphic transformation, which is performed off the shelf by certain tools. Some of them perform very basic and meaningless transformations, such as removing whitespace and renaming variables. They are of no interest to us, as these kind of changes do not trickle down to the execution of the program. More sophisticated tools however perform massive changes to the CFG, inject dead code and even come up with their own specially designed instruction set for a particular function, to name a few. Tigress is a freely available, highly configurable example of such tool, which is why it was used for this work.

Integrating *Tigress* into the infrastructure is as easy as writing a wrapper script that calls tigress with appropriate options. From a wide variety of transformations offered by tigress, two were used: flatten and virtualize transformations. There are several caveats to configuring tigress, to work with the symbolizing library. For example, the names of the function must not be obfuscated. In addition, the main function should be kept intact to ease debugging. Note that in Csmith programs, nothing interesting happens in the main function. Luckily, tigress provides a facility for excluding functions from being obfuscated, which we did for *main* and symbolizing library functions.

Flatten transformation

The flatten transformation removes most of the control flow from the program. It can be best summed by Figure 3.3. It flattens the CFG to one level, with a dispatch block on top. The dispatch blocks implements a state machine that captures the original control flow graph. There are several possible dispatch block offered by tigress. The default one is a switch statement, which is what was used. Tigress also offers other alternatives, such as *goto*s, jump tables or wrapping each basic block into a function.

Figure 3.3: Tigress flatten transformation outline (source: tigress website[1])



Virtualize transformation

Virtualize transformation is a heavyweight transformation that takes a function and turns it into an interpreter for a specially on-the fly randomly designed architecture, that has equivalent behavior to the original function. The details are not important for us, but this transformation changes the control flow beyond any recognition.

This, combined with the flatten transformation and the swap branches transformation, gives us a scale of various degrees of CFG changes. The swap branches transformation, performs the tiniest possible change. The flatten transformation transforms the program in quite a severe way, but there are still some resemblances to the original program. Finally the virtualize transformation goes completely crazy, removing any trace of the original from the CFG.

3.2.4 Other transformations

There was another transformation considered and implemented to an extent. However, it wasn't stable enough to be used for testing with Csmith programs in part due to problems relating to changing nested structures with LibTooling described in Section 3.2.2. The idea was to execute both conditional branches and store their result in temporary variables. Then the original variable would be assigned one of the temporaries depending on the condition. The simplest example of this transformation can be seen in Listings 3.6 and 3.7.

[1] http://tigress.cs.arizona.edu/transformPage/docs/flatten/index.html

Listing 3.6: Original snippet

```
a = func1();
if(condition)
  a = func2();
```

Listing 3.7: The transformed snippet

```
t1 = func1();
t2 = func2();
a = condition ? t2 : t1;
```

There is an even another bigger problem with this transformation. Functions *func1* and *func2*, might have side effects on global state. Therefore the *modset* of both functions needs to be captured into temporaries and written back depending on the condition. Unfortunately, Csmith programs are too convoluted and pointer based for static analysis tool available to be able to accurately determine the *modset* of the functions. This problem is exacerbated by the fact that the transformation is performed on the source level. The tooling for these sort of analysis usually operates on some compiler intermediate representation, which makes this an even less well supported problem. These are the reasons this transformation was not pursued further.

3.3 Multi path execution testing methods

As alluded to before there are additional challenges in using the oracles (§2.1.3) when applied to symbolic execution that explores multiple paths. This section presents three approaches for dealing with this problem. Firstly, an approach that directly builds upon the work from Chapter 2 is presented in Section 3.3.1. The main difficulty there is the nondeterminism of the constraint solver. Then the concrete output oracle is replaced by a "symbolic" output oracle in Section 3.3.2. The idea is based on the program crosschecking approach from the original KLEE paper [5]. Finally, the output oracle is abandoned altogether. Instead of the output oracle, Section 3.3.3 presents an approach where the values of symbolic values are recorded for each path on one version of the program and are then replayed on the other, checking that the same paths are followed.

3.3.1 Direct approach

This approach starts where Chapter 2 left off. In chapter 2 we compiled the program in Listing 3.8 and execute it both natively and symbolically, with x constrained to 5. The output in both cases should be *#1 x:5*. If symbolic executor prints anything else, a mismatch would be reported.

Listing 3.8: A minimal example of a program that can test symbolic execution.

```
unsigned x = 5;
void main() {
  make_symbolic(&x);
  if(x < 10) printf("#1 x:%u", x);
  else printf("#2 x:%u", x);
}
```

Due to the design of the testing infrastructure, it is easy to switch over to metamorphic testing. The program in Listing 3.8 is compiled and run symbolically as before. Then it is transformed using one of the transformers further described in Section 3.2. The transformed program is then compiled and run again symbolically in the same way as the original program was. The result of both executions should again be *#1 x:5*, since the runs are still constrained to 5. A mismatch either indicates a bug in symbolic executor or that the transformation isn't semantics preserving. Luckily, it is easy to see if there is a bug in the transformation by simply running the program natively instead of symbolically. If there is a mismatch found in the native run, the transformation contains a bug, otherwise a bug in the symbolic executor was found.

Within the infrastructure this is as easy as setting the two *COMPILE_AND_RUN* variables to the desired symbolic execution runs script and setting the *TRANSFORM* to one of the transformers from Section 3.2. Further details about this can be found in Section 2.2.3.

This setup replicates mode B (§2.1.2) from the differential testing chapter but in the context of metamorphic testing. Note that mode A can easily be achieved as well by linking in a symbolizing

library, which makes the call to *make_symbolic* a *noop*. I believe that this setup has a similar bug catching potential as the differential testing approach, which has an extensive evaluation in chapter 2. Therefore it wasn't used to run at large scale with the aim of catching bugs. Instead, the efforts were directed at making it applicable to multi path case.

Multi-path case

The advantage metamorphic testing gives us over differential testing is that both executions can be symbolic. That means both can explore multiple paths and enables the comparison between two full symbolic executions. At implementation level this is achieved by taking the mode B setup and using a symbolizing library that solely makes the variable symbolic and does not constrain them. The execution could now for example print *#1 x:5#2 x:15*. If we are lucky the transformed program might also print the same output. However, even assuming bug-free symbolic execution, it might print *#2 x:15#1 x:5* or *#2 x:345345#1 x:2* etc. as well, thus causing a report of spurious mismatch. Note there are two things that can differ in the output even when symbolic execu,tion is correct. Firstly, the order at which the statements are printed can differ and secondly the symbolic variables are nondeterministically concretized by the solver. Also note that we assume symbolic execution explored all the paths fully, which is why we only generate programs where full exploration of all paths happens within a short amount of time as described in Section 3.1.

Different order of output

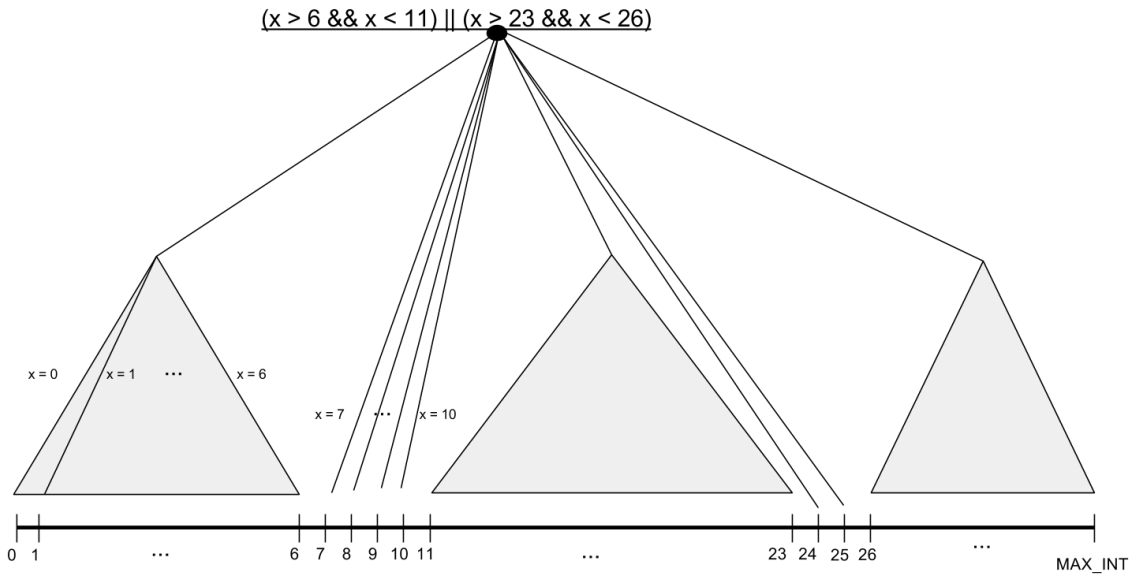
Different order of output has two remedies, depending on the transformation used. First, depth first search path exploration strategy is used. DFS path exploration is useful first and foremost because it is a deterministic strategy that is easy to understand and control. Therefore, all runs of the same program should have the same order of statements printed out. For transformation that do not change the CFG too much setting the DFS search strategy is even sufficient to remove spurious mismatches due to order. It is also the strategy that is present in all the symbolic executors used i.e. KLEE, Crest and FuzzBALL. Note that there is no guarantee order would be the same when two different symbolic executors are compared, as the compiled binaries might have differently ordered CFG due to different compilations. A downside of setting the search strategy to DFS is that the parts of symbolic executor dealing with more interesting and useful search strategies are not tested, but for this part of the work DFS is absolutely necessary.

If despite setting the path exploration strategy to DFS, the order of output is still not the same, it means the CFG is changed too much and an order sensitive output oracle must be dropped. However, by relaxing the order sensitivity, the output can still be used. In particular, the comparison of sorted output should still be able to detect mismatch. A downside of sorting is that the order information is lost, which gives more space for bugs to pass through undetected. On implementation level this approach is as easy as piping the output of the compile and run script through the Linux *sort* utility.

Subset output oracle

Another problem with the multi-path case is that some transformations—like dead-condition injection— might introduce new paths for symbolic execution to explore. These paths arise because additional conditions were added to the branching conditions. They force symbolic executor to explore more paths. However, by design of the transformation these new paths should behave exactly the same as the original paths. Therefore when comparing the transformed programs to the original one, the additional paths that are explored can be safely ignored. This was done simply by ignoring the diffs from the transformed side of the program, essentially making the output oracle check that the output of the original program is a subset of the transformed one.

Figure 3.4: Given a variable x with constrain $x > 6 \wedge x < 11 \vee x > 23 \wedge x < 26$ to printing function, the conceptual path tree the print symbolic function should make in the symbolic executor. The shaded triangles represent unfeasible paths.



Nondeterministic concretization

Even with the above modifications to the oracles, half of the programs under test were still reporting mismatches. Even worse, a test case would sometimes report a mismatch and sometimes pass without a hitch. This was due to the solver concretizing symbolic variables differently, within the range of its constraints. An innovative, symbolic executor independent solution was devised to get around this problem. The idea was to force the constraint solver to always pick the lowest value (or highest) in the range of possible values the symbolic variable could have. The catch was to do it inside the symbolizing library, without modifying the solver or the symbolic executor!

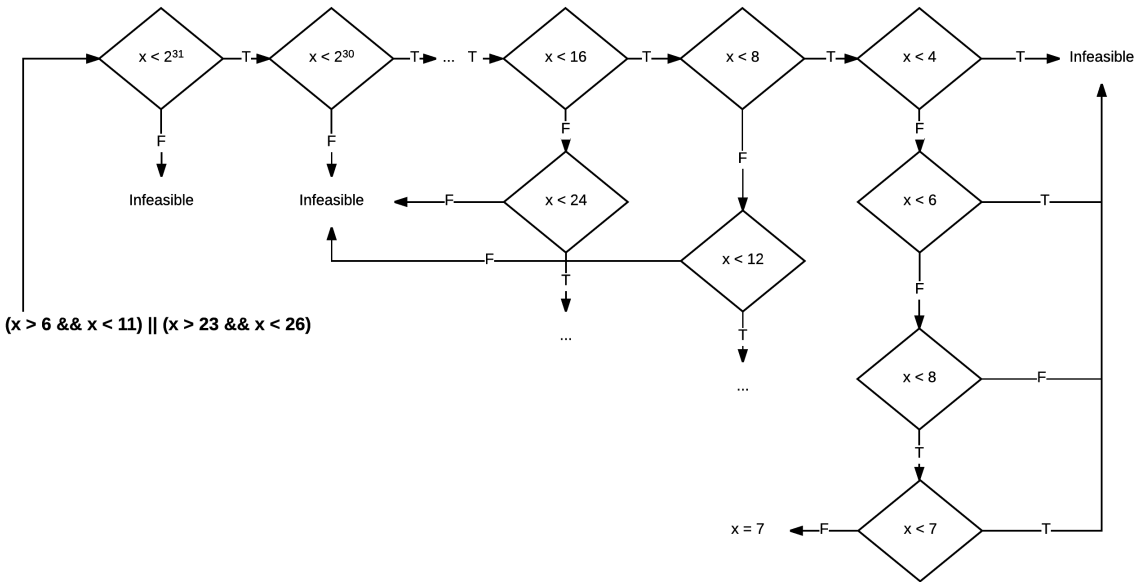
The basic idea is simple. Let's consider a variable x with constraints $(x > 6 \wedge x < 11) \vee (x > 23 \wedge x < 26)$ hitting the *print_symbolic* function. The conceptual idea is that inside the printing function we branch on x for each possible unsigned integer value like shown in Figure 3.4. It is important that the order of branching is exactly the same as in Figure 3.4. The unfeasible paths, represented by shaded triangles, are not explored. Therefore under depth first search the path where x equals 7 is explored first. The *print_symbolic* function can have the same behavior as before on this path and print the value of x. However all other feasible paths now need to be pruned, so not all possible values of x are output. Conceptually, *print_symbolic* needs to remember that it has already output the variable and silently exit if it has. This requires some shared memory between branches within the symbolic executor. As the solution needed to be symbolic executor agnostic, the filesystem was used as this shared memory. Before outputting the value, *print_symbolic* tries to write an identifier to a file. If that identifier is present before writing, it simply silently exits thus pruning the branch. Using this trick it is possible to have deterministic constraint containerization to the lowest value within the range, that works without modifying the symbolic executor or the solver. An example of implementing this is shown in Listing 3.9. Of course it is assumed that the *for* loop gets unfolded in such a fashion that lowest is are explored first.

It is obvious from Listing 3.9 that this naive approach is very slow. The symbolic execution needs to explore over 4 billion paths for each integer variable that needs to be printed. Note that for each of the branches the symbolic executor needs to determine if the path is feasible. If it is it needs to perform file system IO, thus making each path quite expensive. Luckily, this problem is easily avoidable by performing a binary search over the integer range, instead of a simple scan. This makes the implementation more complicated but brings the number of paths that need to be explored for each symbolic variable under 100, which is entirely feasible.

Listing 3.9: A naive implementation of printing symbolic variables with deterministic concretization. firstBranch() is a function that returns true if it hasn't been called with branchId on a different path.

```
void print_symbolic(unsigned h) {
  int branchId = rand();
  for(int i = 0; i < MAX_INT; i++) {
        if(h == i && firstBranch(branchId)) printf("h:%u", h);
    else exit(0);
  }
}
```

Figure 3.5: The path tree that symbolic execution would explore when printing a variable x with constrain $x > 6 \wedge x < 11 \vee x > 23 \wedge x < 26$, based on the binary search based algorithm. Note that exploring the graph in a DFS fashion by taking the *true* (T) branches first, the state where $x = 7$ is reached first, which achieves the goal of reaching the state with the lowest value in the constraint range.



Listing 3.10: A binary search based implementation of printing symbolic variables with deterministic concretization. *h* is the symbolic variable that needs to be printed out. firstBranch() is a function that returns true if it hasn't been called with branchId on a different path.

```
lb = 0;
up = MAX_INT;
prev = ub;
lbForUb = lb;
while((prev - ub > 0) && is_first_branch(branchId)){
    if(h < ub){
        prev = ub;
        ub = ub - (ub - lbForUb) / 2;
    }
    //at this point ub is smaller than h can be,
    //and prev is ub that can still shrink
    else {
        lbForUb = ub;
        ub = prev;
        prev = ub + 2;
    }
}
```

The binary search implementation is demonstrated by Figure 3.5 and Listing 3.10. Consider a variable x, constrained to $(x > 6 \wedge x < 11)||(x > 23 \wedge x < 26)$, hitting the *print_symbolic* function, where it needs to be deterministically concretized. The path exploration tree we would want to insert is shown in Figure 3.5. Note that assuming depth first search path exploration strategy, which takes the *true* (T) branch first, the state where $x = 7$ is reached first. At which point the value of x is output and the other branches get pruned, as described before.

At implementation level shown in Listing 3.10, the *while* loop tries to reduce the value of *ub* while it is still larger than a possible value of *h*. When *ub* goes lower than *h* can go, it rolls back and tries a smaller reduction. This is repeated until *ub* precisely narrows down to the lowest value of *h*. The *is_first_branch* function is called every iteration of the loop to prune the branches where the lowest value has be found already early.

Using this implementation of the symbolizing library the number of mismatches reported on the 1000 program benchmark fell drastically, to manageable levels. Unfortunately, this *print_symbolic* implementation makes reducing the programs and reporting bugs difficult. Despite numerous attempts we were unable to get a good reduced program out of the mismatches. The difficulties revolve around inlining this *print_symbolic* function in the programs with mismatches and reducing them. Since the function makes a significant impact on the symbolic execution, reducing without it was shown to be meaningless. Therefore the practical usability of this approach remains to be determined as part of possible future work.

3.3.2 Crosschecking approach

The second "crosschecking" approach is based on checking tool equivalence in the original KLEE paper [5]. The idea is to have the original and the transformed programs in the same binary. Then the input of each of the programs is made symbolic, the two programs are run. The process concludes by asserting that the results of the two programs are the same. A tiny example of this can be seen in Listing 3.11.

Csmith generated programs are well suited for this sort of transformations. Prefixing of identifiers can easily and reliably be done using simple text replacement. A standard Linux utility *sed* was used in this case to perform the replacement. Csmith programs also do not use the main function for any of the computation, so the main function can safely be removed. They have a different entry point which is always a void function that takes no arguments called *func_1*. So the process

Listing 3.11: Crosschecking outline, *func_1* represents the original program, *prefix_func_1()* represents the transformed program, where all the identifiers have been prefixed with "prefix_" to avoid name clashes. The *main* function is a newly generated program that serves as the oracle.

```
int g, prefix_g;
void func_1() {
    g = g + 1;
}
void prefix_func_1() {
    prefix_g = (prefix_g * 2 + 2)/2;
}
void main() {
    make_symbolic(&g);
    make_symbolic(&prefix_g);
    if(g != prefix_g) exit(0);
    func_1();
    prefix_func_1();
    assert(g == prefix_g);
}
```

of combining two Csmith programs is as easy as prefixing one of them, removing the main function and concatenating the text of the two files together, for example using the *cat* utility.

Removing the main function can also be performed with *sed* to a limited extent, by simply discarding anything that comes after the pattern matching the start of *main*. This was the initial implementation of removing main, since in original Csmith programs, swap branches and dead condition injected programs *main* always comes as the last function. However, tigress transformed programs don't have *main* as the last function, so a more robust method was needed. Removing whole functions with LibTooling is easy so a small transformation that removes the main function was written instead of the *sed* script.

Finally the *main* function of the combined program needs to be generated. This was done by running the original program natively, with a specially designed symbolizing library. This implementation of the symbolizing library would generate the main function on the fly as the calls to it are being made. This newly generated *main* would then be added to the concatenation of the original and the transformed program. Resulting in a program, similar in structure to the one seen in Listing 3.11.

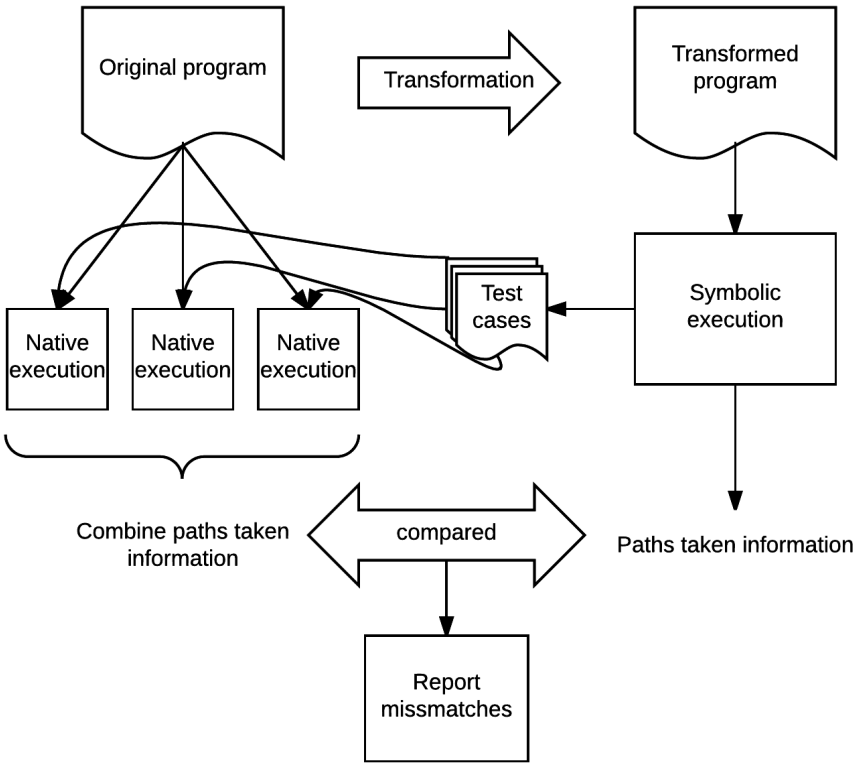
To sum the approach up, a new program is created that combines the original and the transformer program. The combined program ensures that given the two symbolic inputs to the two versions of the program are the same, their results must also be the same. Symbolic execution is then tasked with the job of finding an example where that is true. If such a case is found, there is either a bug in the symbolic executor or the transformation. In a sense this approach checks symbolic execution with symbolic execution.

3.3.3 Path approach

An overview of the path approach is given in Figure 3.6. As shown in the figure, the original program is first transformed. The transformed program is then run with a symbolic execution engine, which records a test case for each explored path. The test cases are then replayed on the original program natively. The original program (and consequently the transformed one) are instrumented to track the path the execution has taken. Therefore both the symbolic execution and the replayed test cases output a traces of the paths explored, which are compared and mismatches reported. This approach is similar in spirit to mode C in Chapter 2, but with the added benefit of the possibility of finding bugs in the transformation.

The output oracle cannot be applied in this case, as the symbolic executor is free to choose which values it will pick for the test case. Instead, the path oracle ensures that a path followed by the

Figure 3.6: An overview of the path approach



symbolic executor when generating a particular test case, is also followed when replaying that test case. The path oracle is similar to function call chain and coverage oracles, but sits somewhere in between them when it comes to strength of the oracle and cost. They all compare information on the path of execution, but at various levels of detail.

At the implementation level, the path oracle piggybacked on the infrastructure for the function call oracle. A call to *logFunction* was added at the start of every branch, with an unique identifier for that branch. In particular the call was inserted after every *for*, *if* and *else*, since those are the only branching structures present in the Csmith generated programs we used. Note that ternary operator is also present but was ignored, because it usually results in a *select* instruction and not a branch. Ignoring it also made the implementation of the path oracle far easier. In addition the calls were also wrapped in *ifdef*s so the path oracle could be toggled on and off as needed.

While test case recording and replaying infrastructure is present in some symbolic executors, like KLEE, it is not present in others like FuzzBALL and Crest. Therefore it was decided to design a symbolic executor agnostic version of path recording/replaying infrastructure. The implementation was based on symbolizing library by re-implementing *print_symbolic* on the record side, *make_symbolic* on the replay side and assuming a path exploration strategy that does not intertwine the paths for example depth first search. The *print_symbolic* function can then write the bytes of the integer it is printing to a file, resulting in a file where all integers across all paths are written in order. Given a program with 2 symbolic variables a, b and 3 paths where $a = 1$, $b = 2$ on the first one, $a = 1$, $b = 3$ on the second one and $a = 1$, $b = 4$ on the last one, t he file containing the recording of the symbolic execution would look like *121314* in decimal notation, assuming both symbolic variables have the same size.

The replaying is then performed by running the transformed program natively with a special implementation of the *make_symbolic* function. Both *make_symbolic* and *print_symbolic* have the size of the integer they are operating on passed to them as a parameter. In addition, they are called in the same order with respect to the variables they are operating on. Therefore *make_symbolic* simply reads however many bytes it is told to from the file containing the recording (which is passed in via environment variables) and assigns them the variable that was meant to be made symbolic. This essentially replays the first path recorded in the file, which is half way to replaying all the

Table 3.1: Summary of bugs found in Crest, KLEE and FuzzBALL using the methods described in this chapter. The table contains a short name for the bug, the approach (§3.3) where the bug was found, the oracle that found it and reduced size of the program.

Bug description	Approach	Transformation	Reduced size (LOC)
KLEE			
CreateZeroConst assertion failure	crosscheck	swap branch, dead condition	17
Output mismatch	direct	dead condition	n/a[1]
Independent solver fail	direct	dead condition, tigress flatten	n/a[2]
Crest			
Prediction failed	all	all	11
FuzzBALL			
STP division by zero	crosscheck	dead condition	20
Unable to concretize	direct, path	all	14
Tigress			
>= operation dropping	crosscheck	flatten	10
Assigning to read only variable	crosscheck	virtualize	n/a

[1] Failed to reduce and therefore not reported
[2] Found in an older version of KLEE (f5cc1a11f8), fixed prior to reporting

paths. To replay any further paths, the program only needs to restart itself with the file offset from the previous execution. This is achieved by forking on the very first call to *make_symbolic*, the child then proceeds as before, while the parent waits for it. When the child finishes, the parent checks if there are further bytes in the file and if so, forks again thus repeating the process. Because forked process share the file descriptors, the read offset is preserved between them, thus achieving a restart of the process with the file offset preserved.

3.4 Evaluation

The evaluation of the methods presented in this chapter was performed on the opus of 1000 multi path programs as presented in Section 3.1. Table 3.2 shows the number of mismatches the infrastructure reported for each approach for each symbolic executor. Unless otherwise noted, all the runs were performed with KLEE git commit: 82778651702 built with STP version 2.1.2. Some initial runs were performed with an older version of KLEE, so bugs found in that version are marked. The version was updated as we wanted to find relevant bugs in the most recent version, whereas some development was done using older versions of KLEE, which is when some of the bugs were found with these approaches.

Table 3.1 is a summary of the bugs that were found with metamorphic testing. The bugs found are discussed in further detail below. By comparing Tables 3.1 and 2.3, 2.5 it might be reasonable to conclude that less meaningful bugs were found with the approaches presented in this chapter. However several things should be considered before making that conclusion.

First and foremost the aims of the work were different. When working on approaches in Chapter 2, my goal was to find bugs, whereas for the work in this chapter I focused on developing and comparing the different approaches. Consequently, the bugs found in this chapter are a result of only one iteration of the whole process. In other words, the bugs presented in Table 3.1 have not been fixed/worked around and the process was then not repeated to find new bugs.

Secondly there were far bigger show stoppers encountered in the less mature symbolic executors with these approaches as they stress the symbolic executor more. For example, Crest failed to run any of the Csmith generated programs with unconstrained symbolic variables due to the prediction failed bug (§3.4.4). Similarly as discussed in Section 3.4.6, FuzzBALL does not provide facilities to concretize unconstrained symbolic, thus disabling the use of any output based approaches.

Finally most of the work in this chapter was done on version where all the previously found bugs

Approach:	Direct	Crosscheck	Path
KLEE	43	1	23
Crest	all	all	n/a
FuzzBALL	n/a	61	n/a

(a) Swap branches

Approach:	Direct	Crosscheck	Path
KLEE	34	18	13
Crest	all	all	n/a
FuzzBALL	n/a	61	n/a

(b) Dead condition injection

Approach:	Direct	Crosscheck	Path
KLEE	270	10	53
Crest	all	all	n/a
FuzzBALL	n/a	49	n/a

(c) Tigress Flatten

Approach:	Direct	Crosscheck	Path
KLEE	496	333	212
Crest	all	all	n/a
FuzzBALL	n/a	362	n/a

(d) Tigress Virtualize

Table 3.2: Number of mismatches reporting by our infrastructure, with respect to the transformation used, the symbolic executor and the approach used. All Experiments were ran on the opus of 1000 programs.

were fixed with the goal of finding bugs that developers would want to fix. Therefore it cannot be concluded that these approaches are unable to find the bugs found in Chapter 2.

3.4.1 KLEE bug: CreateZeroConst assertion failure

The manifestation of this bug is an assertion failure somewhere in the STP code and can be reproduced by running program shown in Listing 3.12 with KLEE. The bug was found both in swap branch and dead condition transformations when the crosschecking approach was used. For the swap branches transformation it was also the only mismatch reported, which indicates that swap branch has very little noise.

Listing 3.12: Causes KLEE to crash due to an assertion failure in STP.

```
1  char  a, g;
2  int b = 7;
3  long c, e, f;
4  long *d = &c;
5  short h;
6  void main() {
7    klee_make_symbolic(&b, sizeof b);
8    klee_make_symbolic(&c, sizeof c);
9    unsigned i = b;
10   f = --*d;
11   g = f + (4 != 0);
12   e = g % i;
13   if (e)
14     i;
15   h = i * 8;
16   (h != 2) + *d || a >> i;
17 }
```

Unfortunately, it is hard to pin down whether this is a KLEE or STP bug. The assertion failure is clearly in the STP code, therefore KLEE developers are reluctant to fix it. The bug can't easily be reported to the STP developers either, as the bug is not present when running the offending query through STP's command line interface.

3.4.2 KLEE: Output mismatch

This is a category of mismatches that are found with the direct approach(§3.3.1). It prints out different values for two symbolic variables on what appears to be the same branch. Unfortu-

nately more details on the bug itself are hard to determine since a reduced version could not be obtained.

The problem with the reduction is the fragility of the direct approach. It is possible to reduce a test case to manageable sizes. This reduction is performed with symbolizing library being linked in outside the reduction. In other words, a second reduction needs to be performed after the initial one. In this reduction step the code for symbolizing library needs to be included in the same file as the program. For all the attempted initial reductions, the mismatches disappear once the symbolizing library code is copied in. Therefore the question arises whether the implementation of the symbolizing library for the direct approach might be incorrect. Unfortunately, as of writing of this report this bug has not been found, so the usefulness of the direct approach remains inconclusive.

3.4.3 Tigress bug: >= operation dropping

Using the metamorphic testing approaches of this chapter, bugs in the transformations can also be found. Listings 3.13 and 3.14 show an example of such bug. The bug was reported to the tigress developers, who confirmed they will look into it. However as of writing of this report, they haven't come back with an explanation or a fix.

Listing 3.13: Original program that is transformed using the Tigress flatten transformation. Note that *a = 0.

```
uint32_t g_65, b, g_158;
uint32_t g_100, g_82;
uint32_t func_1() {
  uint32_t *a = &g_158;
  *a = 1LL >= 184467440701615;
  return 0;
}
int main() {
  func_1();
}
```

Listing 3.14: The transformed snippet, showing the functional bit of code. Note that the whole transformed program is over 100 lines long and includes a lot of unimportant boilerplate. Note that *a = 1.

```
...
switch (_1_func_1_next) {
    case 1:
    *a = (uint32_t )1;
    _1_func_1_next = 0; break;
    case 0: ;
    return ((uint32_t )0);
...
```

The problem seems to arise when transforming the program in Listing 3.13, using tigress flatten transformation. The transformed program is quite long as tigress preprocesses all the headers, defines initialization functions, etc. , so it is not shown here in its entirety. The code snippet that implements *func_1* in the original program is shown in Listing 3.14. The important thing to note is the assignment to **a* in both programs. In the transformed program it gets assigned simply a constant 1. However the original program assigns *1LL >= 184467440701615* to **a*, which should evaluate to 0. It appears as if tigress dropped the >= or evaluated the constant expression to the wrong value.

Listing 3.14 also shows quite well how flatten transformation with switch statement works. It builds a state machine out of the basic blocks of the function and puts each one as a separate case in the switch statement. The current state is tracked with the *_1_func_1_next* variable. Finally the switch statement is wrapped in an infinite loop. In this sample case there is no control flow in the function so there are just two blocks, the main one and an exit block.

3.4.4 Crest bug: prediction failed

A Crest prediction failed bug is a large class of errors reported by Crest. The high level cause for it stems from the inner workings of concolic execution. Once Crest finishes exploring a branch, it chooses a branch point and negates it, asking the constraint solver for a new concrete solution. It then runs the program with the symbolic value set to that solution. Based on what branch was negated, it predicts which branches will be taken. If the actual execution does not follow those branches, it prints a *Prediction failed!* message.

Listing 3.15: A program which makes crest explore 4 branches and causes prediction failure on two of them.

```
unsigned short a = 0;
unsigned char b = 1;

void main() {
  __CrestUShort(&a);
  __CrestUChar(&b);
  int ak = --b;
  a && 0;
  ak >= 2 || 0;
}
```

The underlying cause for a prediction failure can be very varied. For example, one was found in 2.3.2, where the cause was a mishandling of big *int* values. That bug was fixed and is not present in the Crest runs in this chapter. However, every single Csmith program I have run with Crest, where the symbolic values were not constrained printed a prediction failed message. One of them was reduced to a program shown in Listing 3.15 and reported to developers. Unfortunately, that meant that no further bugs could be found in Crest as every other potential bug was overshadowed by the prediction failed error. Note that the approaches were applied to simple toy programs and run with Crest, where it didn't report any prediction failures i.e. behaved as expected. Therefore I don't believe there is an underlying issue in the approach that prevents it from being applied to a version of Crest that is able to handle unconstrained symbolic execution of Csmith programs.

3.4.5 FuzzBALL bug: division by zero

This is a bug where FuzzBALL terminates due to a division where the divisor can be zero. As can be seen from Table 2.5, this bug has been found before in Chapter 2. Developers shifted the problem to an old version of STP, which was indeed unable to handle division by zero, upgrading the version did fix that bug. However the bug reappeared again when running the crosscheck experiments with FuzzBALL even with the newest version of STP, which handles division by zero well.

I believe the problem is that FuzzBALL gives up on division by zero too quickly. Listing 3.16 shows an example of two functions being checked to have the same behavior using the crosscheck approach. They both obviously have a division by zero problem, but are still equivalent, because they both fail for the same input of x. Unfortunately FuzzBALL still reports a division by zero error in this case, whereas KLEE for example works fine for this case. The bug or perhaps a feature request was reported to the developers.

3.4.6 FuzzBALL: Unable to concretize

An interesting aspect of FuzzBALL is that it symbolic executes all the code, including *libc* and system calls. An unfortunate consequence of this which was realized too late in the course of this work, is that it is impossible to concretize a symbolic variable to a single value.

Listing 3.17 shows a program with two branches. In the exact form shown, it does what one would expect, printing the value of *g_56* twice, once for each branch. If the printing of *g_54* is included, it explores and prints 256 branches, one for each possible symbolic value of *g_54*. Unfortunately it is still unclear to me if this is a bug or not. Printing of each possible value of *g_54* can be explained by symbolic execution of *printf* and *write* system call, which I assume branch on each bit of the value being printed at some point. However why this does not happen when only *g_56* is printed is unclear and awaiting developer response.

Listing 3.16: A program which illustrates the division by zero problem in FuzzBALL. Note that *magic_symbols* are set to be symbolic at runtime.

```
#include <assert.h>
int g_279 = 1;
int  a = 0;
int g_34 = 1;
static int magic_symbols[4] = {1,0,0,4};

int func(int x) {
    return 100 / x;
}

int func1(int x) {
    return 100 / (x  + 2 - 2);
}

int main() {
  g_279 = *magic_symbols;
  g_34 = *(magic_symbols +1);
  if(g_34 != g_279) exit(0);
  assert(func(g_279) == func(g_34));
  return 0;
}
```

Listing 3.17: A program which illustrates the concretization problem in FuzzBALL. Note that *magic_symbols* are set to be symbolic at runtime.

```
uint32_t g_54 = 0;
uint32_t g_56 = 3162800460;

static uint32_t magic_symbols[4] = {0,0,0,3};

void main ( void ) {
   g_54 = *(uint8_t*)magic_symbols;

   g_56 ^= 1 < g_54;
   printf("g_56: %X\n", *(&g_56));
//    printf("g_54: %X\n", *(&g_54));
}
```

Chapter 4

Conclusion

Symbolic execution has seen significant interest in the last few years across a large number of computer science areas, such as software engineering, systems and security, among many others. As a result, the availability and correctness of symbolic execution tools is of critical importance for both researchers and practitioners. This report has presented two compiler testing based techniques for testing symbolic execution engines. Firstly, the concept of differential testing was applied to three symbolic execution engines: KLEE, Crest and FuzzBALL. In the process, a toolkit for testing symbolic executors was developed, capable of performing both differential and metamorphic testing of symbolic executors. The toolkit was developed with KLEE, Crest and FuzzBALL in mind, but due to their differences could be extended to any other symbolic executors that run C code with minimal additions. Applying compiler based differential testing to symbolic executors required a novel way of creating program versions, where symbolic execution is constrained to a single path. In addition, an attempt was made to extend the differential testing approach to multi path symbolic execution. It was limited by test generation and replay capabilities of KLEE and therefore not extended to other symbolic executors. A comprehensive evaluation of these approaches was performed with the case studies on the three symbolic execution engines. It has found 21 bugs over the three symbolic executors, 12 of which have been fixed by the developers already. This shows the capability of differential testing to find real and important bugs in symbolic executors.

Differential testing has shown a lack of capability to effectively test multi path symbolic execution. Therefore a metamorphic testing approach was developed to tackle the challenge of testing symbolic execution engines. Four transformations were used for metamorphic testing, two specifically developed for this purpose and two based on a C obfuscation tool. The design of the toolkit enabled easy transition from differential testing to metamorphic testing in both single and multi path mode. The evaluation of metamorphic testing for single path mode did not go beyond demonstrating that it works. We believed that metamorphic testing has a similar bug catching potential to differential testing and therefore not explore it further. However, future work could include a comprehensive study of relative strengths and weaknesses of differential and metamorphic testing for symbolic execution.

For testing multi path execution with metamorphic testing, it proved essential to generate an opus of Csmith programs where symbolic execution can completely explore multiple paths within a short timeout. These programs were significantly more predictable when executed symbolically, because the number of paths explored was fixed. This combined with depth first search path exploration strategy enabled development of three approaches for comparing multi path symbolic execution. The first "direct" approach relied on concretization to be deterministic, which added huge complexity to the process. This complexity prevented producing good and meaningful bug reports thus limiting the usability of this technique. However with further work, it might be possible for this approach to mature and start producing better bug reports. The crosschecking approach is arguably the best of the three. It is easily applicable to multiple symbolic executors and has managed to highlight several bugs. Finally the path approach also successfully managed to flag potential bugs.

The experience in testing symbolic executors also varied greatly between KLEE, Crest and FuzzBALL.

KLEE was the easiest to work with, being the most reliable and feature complete of the three, therefore all of the approaches presented were applicable to it. The debugged bugs in KLEE were various edge cases such as incorrect optimization for division by 1. FuzzBALL has also been a reliable symbolic executor albeit less easy to work with than KLEE. FuzzBALL's symbolizing interface is less pleasant to work with as it requires disassembling of binaries and looking up addresses. But more importantly most pain points come from the fact that it symbolically executes both *libc* and Linux system calls. That meant the direct and path approach could not be applied to it, as they require concretization of the output. In addition, it lead to some hard to debug bugs, where the bug is in the implementation of *printf* as opposed to the code calling the *printf* function. Finally, in our experience Crest has been the most fragile of the three. The bugs found in Crest included simple features like not being able to handle large unsigned integers and lack of support for division. While by limiting the features Csmith generates experiments could be run in the single path case, Crest failed in the multi path case. It failed to correctly execute any of the programs in the opus with all three approaches.

In addition to minor future work already mentioned there are still other possibilities of extending this work. A glaring feature of symbolic executors that is not tested by any of this work are path exploration strategies of symbolic executors. Other approaches need to be developed to effectively test those. Additionally, generation of programs that test symbolic execution well could also be improved. The time it takes to generate a small path Csmith program severely limits the applicability of the multi path approach. It took 2 days on a cluster of 8 computers to generate the opus used. However, it takes only about half a day to run a typical experiment on the opus on a single computer. The generation is therefore slower than running the experiments, which limits the ability to do them at mass scale. A symbolic execution targeted program generator could do better at trying to generate program that test symbolic execution well and are generated quickly.

Chapter 5

Bibliography

[1] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Mar.-Apr. 2007.

[2] D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, Apr. 2015.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

[4] C. Cadar and A. F. Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 765–768, New York, NY, USA, 2016. ACM.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[6] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, Aug. 2005.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[8] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice—Preliminary Assessment. In *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

[9] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*, May 2012.

[11] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems (HotDep'09)*, June 2009.

[12] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering (TSE)*, 2(3):215–222, 1976.

[13] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

[14] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*, Apr. 2011.

[15] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering (TSE)*, 40(7):710–737, 2014.

[16] CREST: Automatic Test Generation Tool for C. http://code.google.com/p/crest/.

[17] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In *Proceedings of the 4th International Conference on NASA Formal Methods*, NFM'12, pages 120–125, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, Sept. 2007.

[19] W. Daniel. *Applied nonparametric statistics*, page 319–330. The Duxbury advanced series in statistics and decision sciences. PWS-Kent Publ., 1990.

[20] Delta. http://delta.tigris.org/.

[21] Frama-c. http://frama-c.com/index.html.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[23] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[24] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[25] J. C. King. Symbolic execution and program testing. *Communications of the Association for Computing Machinery (CACM)*, 19(7):385–394, 1976.

[26] T. Kuchta, C. Cadar, M. Castro, and M. Costa. Docovery: toward generic automatic document recovery. In *Proc. of the 29th IEEE International Conference on Automated Software Engineering (ASE'14)*, Sept. 2014.

[27] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*, June 2014.

[28] X. Leroy. Formal verification of a realistic compiler. *Communications of the Association for Computing Machinery (CACM)*, 52(7):107–115, 2009.

[29] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)*, June 2015.

[30] Llvm 3.9 documentation: Libtooling. http://releases.llvm.org/3.9.0/tools/clang/docs/LibTooling.html.

[31] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[32] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 337–348, New York, NY, USA, 2012. ACM.

[33] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.

[34] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin???Madison, 1995.

[35] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery (CACM)*, 33(12):32–44, 1990.

[36] C. Miller and Z. N. Peterson. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.

[37] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the 11th International Conference on Compiler Construction (CC'02)*, Mar. 2002.

[38] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar. Accelerating array constraints in symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA 2017)*, 7 2017.

[39] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, June 2012.

[40] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 445–454.

[41] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proc. of the 4th International Workshop on Mutation Analysis (Mutation'09)*.

[42] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[43] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[44] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.

[45] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.

[46] J. Wu, G. Hu, Y. Tang, and J. Yang. Effective dynamic detection of alias analysis errors. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[47] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*, June 2011.

[48] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering (TSE)*, 28(2):183–200, 2002.