Imperial College London
Department of Computing

Symbolic execution of verification languages and floating-point code

Daniel Simon Liew

April 2018

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration of Originality

The work presented in this thesis is my own original work except where acknowledged otherwise.

Copyright Declaration

Abstract

The focus of this thesis is a program analysis technique named symbolic execution. We present three main contributions to this field.

First, an investigation into comparing several state-of-the-art program analysis tools at the level of an intermediate verification language over a large set of benchmarks, and improvements to the state-of-the-art of symbolic execution for this language. This is explored via a new tool, Symbooglix, that operates on the Boogie intermediate verification language.

Second, an investigation into performing symbolic execution of floating-point programs via a standardised theory of floating-point arithmetic that is supported by several existing constraint solvers. This is investigated via two independent extensions of the KLEE symbolic execution engine to support reasoning about floating-point operations (with one tool developed by the thesis author).

Third, an investigation into the use of coverage-guided fuzzing as a means for solving constraints over finite data types, inspired by the difficulties associated with solving floating-point constraints. The associated prototype tool, JFS, which builds on the LibFuzzer project, can at present be applied to a wide range of SMT queries over bit-vector and floating-point variables, and shows promise on floating-point constraints.

Acknowledgements

I have been very fortunate to have worked with many people over the course of my PhD.

First, I would like to thank my two supervisors, Cristian Cadar, and Alastair F. Donaldson (Ally). They have both been instrumental in providing guidance and support during my PhD. Cristian and Ally are both excellent researchers. They taught me to ask good research questions, and showed me how to write papers well.

Finding the right research problem to work on can be hard. I started my PhD not knowing exactly what to work on, and after a year of not finding a research direction I was happy with, I very nearly quit pursing a PhD. Ally set me on the right track. He convinced me not to quit and to develop a symbolic execution tool for Boogie programs (Symbooglix). This work eventually led to my first (first author) paper, and a best paper award at ICST 2016. Thank you Ally!

The work on Symbooglix greatly benefited from the assistance of other researchers. I'd like to thank Nadia Polikarpova for her assistance using her Boogaloo tool. I'd like to thank Akash Lal for his assistance in using the Corral tool. I'd like to thank Zvonimir Rakamaric for providing translated SV-COMP benchmarks. Finally I'd like to thank Petr Hosek for providing me with guidance on using the C# programming language.

In between my Symbooglix work in 2015 I did an internship at Google. I'd like to thank Petr Hosek for helping me prepare for the Google interviews and also being an *unofficial* mentor to me while at Google. While at Google, I worked with floating-point arithmetic. I would like to thank Abe Stephens (my manager at Google) for introducing me to the weirdness of floating-point arithmetic. This weirdness inspired me to work on extending the KLEE tool to support floating-point arithmetic when I returned from my internship. During this internship, a fellow contributor to the STP project (Ryan Govostes) noticed that I was in the Bay area and invited me to visit Apple. At this meeting, I met Ryan, and the program analysis team within Apple. This meeting led to me applying for an internship at Apple for which I was subsequently accepted. Ryan, thank you so much for this introduction.

The work extending the KLEE tool to support floating-point arithmetic finally culminated in a paper that was accepted at ASE 2017. I'd like to thank Daniel Schemmel and Rafael Zähl for their hard work on this project. During this project I frequently interacted with the maintainers of the Z3 project (Christoph M. Wintersteiger and Nikolaj Bjorner) and I'd like to thank them for their assistance and for reviewing my patches.

In between my work extending KLEE in 2016, I did an internship at Apple. During this internship I worked on the LibFuzzer project. This work was part of the inspiration for the JFS tool, which forms an important chapter of my thesis. I'd like to thank Anna Zaks, Devin Coughlin, and Dmitri Gribenko for their support and guidance during this internship. I'd also like to thank Kostya Serebryany for reviewing my patches to LibFuzzer during my internship.

During my PhD, I worked on many tangentially related projects (e.g. KLEE, LLVM, STP, and Z3). I'd like to thank my supervisors for allowing me to work on these, despite no obvious benefit to my PhD at the time. In hindsight I can now see the positive benefit all this work has had on my PhD and future career.

During my PhD I have had the pleasure of being part of two excellent research groups. I'd like to thank all members—past and present— of these groups for providing friendship, and advise.

I'd also like to thank Rosie English for proofreading large portions of my thesis.

My PhD would have been impossible without funding. I'd like to thank the EPSRC and ARM for jointly funding my PhD.

And finally I'd like to thank my wonderful wife (Anna) for providing so much emotional support during my PhD, and for proofreading my thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Bugs in software are a huge problem. Aside from causing user frustration software bugs can lead to critical security vulnerabilities. For example the heavily publicised "heartbleed" [88] and "shellshock" [59] vulnerabilities are the result of latent bugs in heavily used open-source software. Software bugs have also lead to serious financial loss [127] and even death [118]. It has been observed [170] that fixing bugs late in the software development cycle is financially more expensive than fixing them earlier. Thus, it is desirable to detect and fix software bugs as early as possible during the software development process.

There exists a variety of techniques to detect bugs. It is well known that a common approach taken by developers is to manually write test cases to check that the software behaves as expected when run in a particular scenario (i.e. *manual testing*). This approach is far from ideal: not only is it very time consuming, it also easily allows latent bugs to remain hidden due to developers not considering edge cases.

Researchers have expended considerable effort investigating different approaches to improve on or complement manual testing through automatic and semi-automatic techniques. One such technique —*symbolic execution*— is the focus of this thesis.

Symbolic execution [102] is an automated technique to enumerate and explore the feasible paths of a program. The feasibility of each path is checked using a constraint solver. For each executed path, a test case can be generated that can be used to replay execution along that path. These automatically generated tests can then be used to create a high coverage test

suite. During execution it is also possible to automatically check for failing assertions[1] and low-level bugs (e.g. use of undefined language behaviour). Test cases for these bugs can also be generated which can be very valuable to a developer.

While symbolic execution has shown potential [80], it has not seen wide spread adoption due to the limited domains in which it can be effectively applied. For example symbolic execution does not scale well to large programs due to the number of program paths growing exponentially with the number of branches [41].

In this thesis we investigate applying symbolic execution to domains that previously had not seen much investigation: *intermediate verification languages* (IVLs) and floating-point arithmetic.

In Chapter 2 we discuss the necessary background and related work to understand our work.

In Chapter 3 we discuss applying symbolic execution to the domain of IVLs. These languages are designed to aid the development of *static verification* tools. Static verification, stemming from the seminal work by Hoare [89] and Floyd [74] is another technique that can be used to find bugs in programs. However static verification is different from symbolic execution in that it tries to prove the absence of bugs on all program paths at once rather than iteratively checking each program path. This makes static verification well suited to checking an entire program but less helpful when a bug is reported. Symbolic execution on the other hand is typically more useful for bug finding due to (usually) not having abstractions that introduce false positives and result in hard to read error traces; and is able to generate test cases for each bug found.

Intermediate verification languages are designed to provide a *separation of concerns*. A well designed IVL should be a simple programming language that abstracts away the details of a particular programming language (e.g. C [95]), and implementation (i.e. compiler and hardware architecture) so that a verifier ("back-end") does not need to know about these details. Examples of language specific details for C include: unsigned integer overflow rules (6.2.5/9 in [95]), and signed integer overflow having undefined behaviour (6.5/5 in [95]). An example of an implementation specific detail in C is object alignment (6.2.8 in [95]).

[1] Typically provided by the developer

At the same time a well designed IVL should not prescribe any particular verifier implementation so that the translator from a programming language to the IVL ("front-end") does not need to know precisely how the verifier is implemented. In particular there is no reason why the "back-end" couldn't use symbolic execution. However few attempts at doing this have been reported [151, 110].

The research problem we address is as follows. Although symbolic execution has known advantages over other verification techniques, no direct comparison between these techniques has been performed at the IVL level. Thus we perform a comparison of different tools (and by proxy different techniques) that all operate on the same IVL with the goal of testing two hypotheses:

1. Symbolic execution of an IVL is competitive with other techniques in terms of bug finding and verification.
2. The state-of-the-art for symbolic execution of IVLs can be improved.

To answer the first hypothesis we perform a comparison of several existing tools that target the Boogie IVL [116] —a popular IVL with several existing front and back ends— over a large collection of diverse benchmarks.

To answer both hypotheses we implement our own tool named *Symbooglix* (**Sym**bo**l**ic **Boog**ie e**x**ecutor) for the Boogie IVL (itself a contribution). We incrementally and systematically improve the tool's performance and then include it in the comparison with other tools. The comparison shows that Symbooglix significantly outperforms an existing symbolic execution tool for the Boogie IVL (supporting our second hypothesis) and is highly competitive with a verifier optimized for verifying code written for the CUDA and OpenCL programming languages. While Symbooglix is generally less effective than two other state-of-the-art verifiers it is complementary to these tools. This partially supports our first hypothesis.

The tool comparison uses a variety of different benchmark programs to compare the tools. However it contains very few programs that make use of floating-point arithmetic. One possible reason for this is that (at the time of performing experiments) the Boogie IVL has no support for floating-point types. In order to verify floating-point programs, the Boogie front-

ends resort to crude approximations that can result in back-ends reporting false positives and negatives. However floating-point programs should not be ignored which leads us to the next chapter.

In Chapter 4 we discuss applying symbolic execution to the domain of programs that use floating-point arithmetic. As previously mentioned symbolic execution relies on a constraint solver to determine the feasibility of paths. Most symbolic execution tools do not support directly reasoning over constraints using floating-point arithmetic due to a lack of support from the underlying constraint solver. This is the research problem we tackle.

Recent advances in solver technology have led to support for floating-point reasoning being added to several constraint solvers. This has been accompanied by an effort to provide a standardised theory of floating-point arithmetic for solvers to implement [160].

Inspired by these advances in floating-point support in constraint solvers we hypothesised that the standardised theory of floating-point arithmetic can be used by symbolic execution tools to reason about path constraints. Unfortunately the Boogie IVL's lack of support for floating-point types meant that Symbooglix was not a suitable tool to test this hypothesis.

Instead we turn our attention to the KLEE [38] symbolic execution tool which executes the LLVM intermediate representation (IR) which *does* support floating-point types. The tool already supported floating-point operations with concrete operands and thus is a good starting point for investigating our hypothesis. We extend the KLEE tool to use the Z3 [140] constraint solver to handle floating-point constraints.

During this work we became aware of another group of researchers building a very similar tool. We decided to collaborate by developing a floating-point program benchmark suite and performing a comparison of each group's tool on these benchmarks to gain insights into the different design decisions made by each research group. Several insights were gathered from this work including answering our hypothesis. While it is indeed possible to use the theory of floating-point arithmetic supported by constraint solvers for symbolic execution of floating-point programs, such an approach does not scale well due to poor constraint solver performance on floating-point constraints which dominates the execution time of the tools. This research problem directly lead to our work in Chapter 5.

In Chapter 5 we discuss a technique to solve floating-point constraints. The key idea is to construct a program where the reachability of a particular statement in the program is equivalent to finding a satisfying assignment to the constraints being solved. To find a satisfying assignment another bug finding technique —*fuzzing* [136]— is employed. Our hypothesis is that this approach will in some cases solve floating-point constraints faster than existing approaches.

To test this hypothesis we develop a research prototype named JFS (JIT Fuzzing Solver) that implements this idea and perform an evaluation of its performance compared to several existing constraint solvers on three benchmark suites. Our evaluation shows that JFS is very competitive with existing solvers on floating-point benchmarks, both in terms of execution time and in the number of benchmarks solved. The results also show that JFS is not competitive with existing solvers on bit-vector benchmarks. However, JFS is able to complement existing solvers on a small subset of these benchmarks by being able to solve them faster than existing solvers. These results support our hypothesis.

Finally in Chapter 6 we discuss future directions for our work and conclude.

1.1 Publications

The text of this thesis is heavily based on the work and writing of several papers co-authored by the thesis author.

- Daniel Liew, Cristian Cadar, and Alastair Donaldson. "Symbooglix: A Symbolic Execution Engine for Boogie Programs". In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)*. Chicago, IL, USA, Apr. 2016, pp. 45–56
- Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael Zähl, and Klaus Wehlre. "Floating-Point Symbolic Execution: A Case Study in N-version Programming". In: *Proceedings of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*. Urbana-Champaign, IL, USA, Oct. 2017

For the first paper all implementation and experimental work was performed by the thesis author, but the paper was jointly written by the thesis author, Cristian Cadar, and Alastair Donaldson.

For the second paper:

- All "Imperial" benchmarks were mostly written by Cristian Cadar and Alastair Donaldson. However the "real world" benchmarks were based on existing sample code. Bug fixes for these benchmarks came from the thesis author, Daniel Schemmel, and Rafael Zähl.
- All "Aachen" benchmarks were mostly written by Daniel Schemmel and Rafael Zähl. However some of the "real world" benchmarks were based on existing sample code. Bug fixes for these benchmarks came from the thesis author or Daniel Schemmel.
- The "Imperial" extension to the KLEE tool was written by the thesis author.
- The "Aachen" extension to the KLEE tool was primarily written by Daniel Schemmel and Rafael Zähl. However features donated to this tool from "Imperial" implementation were written by the thesis author.
- The running of experiments of both tools was performed by the thesis author.
- The analysis of experiment data was mostly performed by the thesis author apart from the analysis of benchmark branch coverage which was performed by Daniel Schemmel.
- The paper was jointly written by the thesis author, Cristian Cadar, Alastair Donaldson, and Daniel Schemmel.

In addition to the above papers the thesis author is also an author on the following paper:

Ethel Bardsley et al. "Engineering a Static Verification Tool for GPU Kernels". In: *Proceedings of the 26th International Conference on Computer-Aided Verification (CAV'14)*. Vienna, Austria, July 2014, pp. 226–242

Although the text from this paper is not used in this thesis the benchmarks presented are used in the Symbooglix paper and thus in Chapter 3.

1.2 Formal acknowledgements

I would like to thank the authors of the GPUVerify tool [14] and the benchmarks used to evaluate it, which were used in Chapter 3.

I would like to thank Zvonimir Rakamaric for providing the SMACK [156] tool and the benchmarks generated from it. I am also indebted to the SV-COMP community for collecting such a large and useful set of benchmarks which are used in Chapter 3.

I would like to thank the SMT and SMT-COMP communities for developing benchmarks which are used in Chapter 5.

Finally, I would like to thank the EPSRC and ARM for funding the work detailed in this thesis.

Chapter 2

Background and related work

We now review the necessary background and related work for subsequent chapters.

The primary focus of this thesis is symbolic execution, however before we discuss this, we first discuss the broader field of program analysis in §2.1 to which symbolic execution belongs. We then discuss symbolic execution in §2.2 which will be used in chapters 3 and 4. We then discuss the field of static verification in §2.3, which includes a discussion of the Boogie IVL and related tools which will be used in Chapter 3. We then discuss fuzzing in §2.4 which will be used in Chapter 5. We then discuss floating-point arithmetic in §2.5 which will be important to chapters 4 and 5. This includes a discussion of existing work to analyse floating-point programs relavant to Chapter 4. Finally in §2.6 we discuss constraint solvers which are relevant to all three bodies of work (chapters 3, 4, and 5).

2.1 Program Analysis

Program Analysis is a broad term that describes any technique that attempts to compute one or more properties of a computer program. Example properties might include "all pointer accesses are preceded by a check for NULL", "no assertions fail", and "the program terminates".

Program analyses have a number of uses but their primary uses are to aid program optimization in compilers and to check varying degrees of *program correctness*. Program optimization is not the focus of this thesis so we shall not consider this further. The keen reader may want to read any good text book on compiler construction (e.g. [4]) for more on program analysis for compilers.

Program correctness can be defined as some set of properties always holding during the execution of a program. For example we could construct a simple program analysis that checks for syntactic occurrences of division by zero. Such an analysis is effectively checking the property that "the program is free from trivial divisions by zero". While such an analysis is easy to implement and could be used to issue helpful compiler warnings to developers, it does not provide very strong guarantees to a developer. For example what about freedom from non trivial divisions by zero, or even stronger properties such as freedom from memory leaks or crashes? Thus to aid developers we should aim to check properties that the developer would like to hold during program execution. If a property can be shown to not always hold during program execution then this is a *property violation*, also known as a *bug*. Hence forth we will use these terms interchangeably.

If all the properties that we wish to hold for a program do hold then we refer to the program as being *correct*, otherwise it is *incorrect*. We can place program properties into one of three disjoint categories; *Functional*, *Performance*, and *General*. *Functional* properties describe the high-level behaviour of a program (e.g. a function takes a list of integers and returns a sorted list). *Performance* properties describe the runtime performance of a program (e.g. a function sorts a list of integers in $\mathcal{O}(n \log n)$ time). *General* properties describe low-level program behaviour and typically describe freedom from low-level language errors such as NULL pointer dereferences and data races. *General* properties are often a target for program analysis tools because their description is dependent only on the programming language used by a program and not on the program itself. This means these properties need only be described once and then re-used many times. The other properties are dependent on the particular program being analysed and so must be provided for each program, usually by the developer.

An issue worth considering is how precise a program analysis is. The previous example of syntactically checking for occurrences of division by zero is **not precise**. It assumes that all program statements are reachable which is an over-approximation. This over-approximation

means that bugs may be reported, even though they can never occur when running the program. Incorrectly reporting a bug that cannot occur is referred to as a *false positive* (F^+). Table 2.1 shows terminology commonly used to describe classification of property violations.

Property actually \ Analysis reports property	holds	does not hold
holds	T^-	F^+
does not hold	F^-	T^+

Table 2.1: Terminology used to describe the classification of property violations (bugs) in programs by program analyses.

A *true negative* (T^-) is where the analysis determines the property holds (i.e. no bug is reported) and this property does hold during program execution. A *false negative* (F^-) is where the analysis determines the property holds but this property **does not hold** during all possible program executions. Finally a *true positive* (T^+) is where the analysis determines the property does not hold (i.e. a bug is reported) and this property does not hold during program execution.

An analysis that never produces false negatives is referred to as *sound,* that is to say the analysis never incorrectly reports a property to hold. An analysis that never produces false positives is referred to as *complete,* that is to say the analysis always reports that a property holds for all programs where that property actually holds.

An example of a trivial sound analysis is an analysis that reports, for all programs that the property always fails. This is of course not very useful. An example of a trivial complete analysis is an analysis that reports, for all programs that the property always holds. Again this isn't very useful.

It is actually impossible to have an analysis that is sound *and* complete for *all* programs. Such an analysis would only report *true negatives* and *true positives*. This would therefore require the analysis to be able to solve the undecidable [172] *halting problem* to determine whether or not a fragment of code known to violate a property was reachable.

Is all hope lost? Well, no. First there exists a subset of programs where the halting problem is decidable and so on that subset of programs it is possible for a program analysis to be both *sound* and *complete*. Second a program analysis can assume that a program terminates and

add the caveat (that program termination is assumed) to the property being analysed. Finally a program analysis can be designed such that it allows some *incompleteness* in order to increase *soundness* or vice versa.

Thus far we have said nothing of how program analyses are implemented. A dichotomy between *static analysis* and *dynamic analysis* is often used when classifying program analyses. A *static analysis* is an analysis that computes properties of programs without executing it, whereas a *dynamic analysis* does so by executing it.

Static analysis covers a broad range of techniques such as linting [97], data-flow analysis [101], model checking [48, 155], abstract interpretation [58], and weakest pre-condition generation [65]. Static verification is a subset of static analysis where an analysis tries to statically show that one or more properties relating to program correctness hold. Static verification is highly relevant to the work in Chapter 3 and so is discussed further in section 2.3.

Dynamic analysis covers a range of techniques. Most techniques are used for bug finding. Some techniques instrument programs with additional code to detect and report bugs at runtime. Well known examples include memory error detectors such as AddressSanitizer [163], and Valgrind [142]; and ThreadSanitizer [164], a race detector. A dynamic analysis does not have to detect bugs, one such analysis is Daikon [69] which detects likely program invariants at runtime, which can then be used by other analyses. Symbolic execution and fuzzing are two dynamic analysis techniques highly relevant to this thesis and so are discussed further in sections 2.2 and 2.4 respectively. Symbolic execution is used by chapters 3, and 4; and fuzzing is used by Chapter 5.

Dynamic analyses are typically more precise than static analyses because they have access to information only available at run time. However, because the analysis is performed at runtime rarely executed but buggy paths might be missed. In contrast static analyses can consider the paths that a dynamic analysis might miss but may need to resort to approximations (introducing imprecision) to improve scalability or avoid requiring information only known at run time.

2.2 Symbolic execution

Symbolic execution is a program analysis technique that provides the ability to automatically enumerate the feasible paths through a program. The technique will be used in chapters 3 and 4. We first provide a brief introduction to symbolic execution. This is then followed by a discussion on its scalability (§2.2.1), concolic execution (§2.2.2), and the program representations used by symbolic execution (§2.2.3).

The technique was first introduced by King [102] in 1976 but has seen renewed interest [41] due in part to improvements to the constraint solvers on which the technique relies, and the increased performance in computers since the technique's inception.

It has been implemented in many different tools [37, 162, 79, 38, 6] and applied to many different areas, such as software engineering, systems and security [41].

Instead of running the program on concrete input, where a particular input component might e.g. take the value 3, symbolic execution runs the program on *symbolic* input, where each input component is represented by a placeholder for an initially unconstrained value. As the program runs, symbolic execution keeps track of how program variables depend on the symbolic input. When a branch point (e.g. an `if` statement) dependent on symbolic input is encountered a constraint solver is used to check which branches are feasible. Each feasible path is then executed, making a copy (known as *forking*) of the current program state if necessary. On each feasible path (at least one must be feasible) the constraint that the branch imposed on the symbolic input is added to a per path set called the *path condition* (PC). The PC records all the symbolic branch conditions (branches that depend on symbolic input) that were traversed by the path. The PC has two purposes. First, it is used to check feasibility when encountering later branch points on that path. Second, the constraints in the PC can be solved and a *satisfying assignment* to the symbolic input can be found. This *satisfying assignment* can be used to replay execution of the program along the same path that (assuming that the program is deterministic) was symbolically executed and is effectively a test case. These automatically generated test cases can be useful to developers because they can be added to an existing test suite to increase code coverage or be used replay bugs.

To illustrate these ideas we now walk through a simple example of symbolic execution for the program shown in Listing 2.1.

Listing 2.1: A simple C program to illustrate symbolic execution.

```
1  int main() {
2    uint8_t x = symbolic();
3    uint8_t y = x + 1;
4    assert(y > x);
5    if (x % 2 == 0) {
6      return 0;
7    } else {
8      if (y % 2 == 0) {
9        return 1;
10     } else {
11       return 2; // Unreachable
12     }
13   }
14 }
```

The listing shows a simple program written in C [95] that can be symbolically executed. Figure 2.1 shows the paths that symbolic execution will enumerate for this program and the constraints gathered. Execution starts with a single path as shown in the figure.

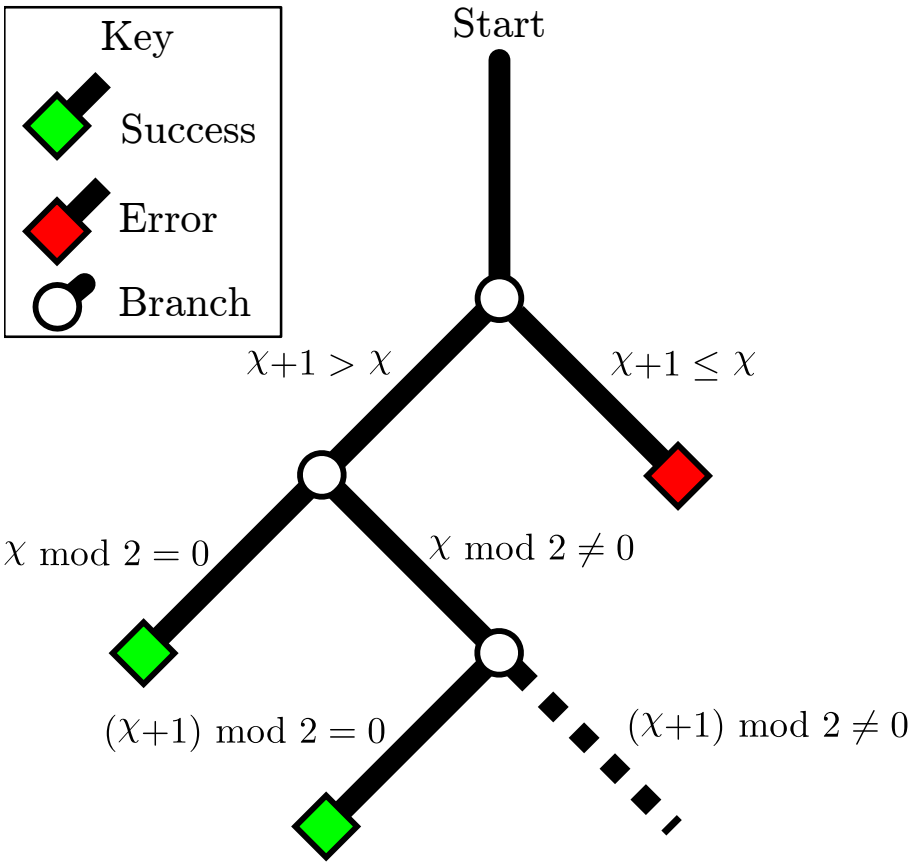


Figure 2.1: Paths for program shown in Listing 2.1. Edges are labelled with the constraint added on that branch.

First on line 2 the function `symbolic()` is called and it returns an unconstrained symbolic value (we'll use χ to represent this) that is written to the variable `x`. Next on line 3 the expression `x + 1` is written to the variable `y`. The value of `x` is currently symbolic so `y` becomes symbolic too, and stores the symbolic expression $\chi + 1$.

Next on line 4 an assertion is executed. An assertion is a branch point, if the condition `(y > x)` is `true` execution continues, otherwise execution terminates with an error. In this case the branch condition uses symbolic data and so both paths might be feasible. On the first path $\chi + 1 > \chi$ is true and on the second path $\chi + 1 \leq \chi$ is true. This branch and the constraint for each path are shown in Figure 2.1. A constraint solver can show that both conditions are satisfiable. Therefore execution must fork (i.e. both paths must be followed). On the true path $\chi + 1 > \chi$ is added to its PC and on the false path $\chi + 1 \leq \chi$ is added to its PC. Execution down the false path is an assertion failure so execution terminates. At this point a satisfying assignment (i.e. a test case) to $\chi + 1 \leq \chi$ can be requested which is the input that will cause the assertion to fail. In this case $\chi = 255$ will cause the assertion to fail because in C an unsigned integer addition wraps around so $255 + 1$ will result in 0.

Next execution continues to line 5 where an `if` statement that depends on symbolic input is executed. For execution to continue the feasibility of the branches need to be checked. The PC to check is the current PC conjuncted with the branch condition. The PC for the true branch is $(\chi + 1 \leq \chi) \wedge (\chi \mod 2 = 0)$ and the PC for the false branch is $(\chi + 1 \leq \chi) \wedge (\chi \mod 2 \neq 0)$. In this case both branches are feasible. For the true branch the `return 0` command is executed on line 6 and execution terminates. No errors occurred along this path and a test case can be generated (e.g. $\chi = 0$) for it. This successful termination is illustrated in Figure 2.1 as a leaf node in the graph that is coloured green.

For the path taking the false branch another branch is then encountered on line 8. The PC for the true path leaving this branch point is $(\chi + 1 \leq \chi) \wedge (\chi \mod 2 \neq 0) \wedge ((\chi + 1) \mod 2 = 0)$ and the PC for the false path leaving this branch is $(\chi + 1 \leq \chi) \wedge (\chi \mod 2 \neq 0) \wedge ((\chi + 1) \mod 2 \neq 0)$. The PC for the true path is satisfiable so execution can continue and then execute `return 1` and then successfully terminate generating another test case (e.g. $\chi = 1$). The PC for the false path is not satisfiable so execution does not continue down this path. This is illustrated by the dashed edge in Figure 2.1.

Now all feasible paths have been enumerated. Three feasible paths were found, one that resulted in an assertion failure and the others completed without error. For each path a test case was automatically generated.

2.2.1 Scalability

Despite the advantages symbolic execution seems to offer over other forms of testing it has seen little adoption in practice. The main reason for this is that symbolic execution does not scale well to large and/or complicated programs. There are two problems with symbolic execution that limit its scalability. First the number of paths to consider grows exponentially with the number of branches in the program. This is often referred to as the "path explosion" problem. Second the approach is limited by the performance of the constraint solver which often performs worse as the size/complexity of the PC grows.

The path explosion problem limits the upper bound on the number of paths that can be considered given a fixed time and space budget. Executing the exponentially growing number of paths eats into the time budget and storing the program state for each path eats into the space budget. To make the best use of the time budget, search heuristics are often employed [12] to prioritize certain paths. For example priority could be given to execute paths that will cover previously uncovered code. To make best use of the space budget two approaches can be taken. One approach is to compress the representation of the program state to reduce wasted space (e.g. by using copy-on-write data structures). Another approach is to use a technique called *concolic execution* which by design does not store the program state for multiple paths. This technique is discussed in detail later in §2.2.2.

The constraint solver performance problem lowers the upper bound on the number of branches that can be considered given a fixed time budget. Various approaches have been taken to reduce the severity of this problem. One approach is to simply call the solver less, KLEE [38] caches previous queries to avoid calling the solver when possible. Concolic execution by design only calls the solver once per path rather than for every symbolic branch on that path. Another approach is to simplify constraints given to a solver. For example KLEE splits constraints into independent sets and solves them separately. Another example is performing equisatisfiable transformations to solver queries where the transformation reduces the size of the query [150]. Another approach is to adapt a solver's configuration to be more performant on the kind of constraints generated by a particular symbolic execution tool. This could be as simple as tweaking solver options, or as sophisticated as exercising fine-grained control over the solver's reasoning steps using a tactic interface [141] that some solvers like Z3 [140] provide.

It should be noted that none of these techniques are a panacea to symbolic execution's scalability problems, instead they just reduce the severity.

We have mentioned the concolic execution technique several times which by design addresses some of the scalability issues of symbolic execution.

2.2.2 Concolic execution

Concolic execution is a portmanteau of **Conc**rete and symb**olic** execution and is a variant of symbolic execution. It was first pioneered by the DART [79] and the CUTE [162] tools. In concolic execution the program is executed concretely (i.e. uses a concrete input) but with certain inputs marked as being symbolic from a constraint collection perspective. During execution these symbolic inputs have a concrete value but at branches involving these symbolic inputs (and values that derive from the symbolic input) constraints are gathered as if the inputs marked as symbolic were symbolic. At the end of program execution the PC contains all the constraints that would have been gathered by conventional symbolic execution for that path but without forking or calling the constraint solver (i.e. fewer resources are used). The next step in concolic execution is to negate one of the branch conditions (and possibly drop some of the conditions) in the PC and give this modified PC to a constraint solver. If the constraint solver finds a satisfying assignment it can be used as a new input for the program that will direct the program down a path, different to the previously executed path. If the constraint solver proves the modified PC is unsatisfiable then the path represented by this modified PC is infeasible and so a different branch condition in the PC must be negated. By repeatedly modifying the PC, solving it, and then re-running the program with the new input all the feasible paths of a program can in theory be enumerated. This approach scales better than conventional symbolic execution because it calls the solver less frequently (once per path compared to once per branch) and consumes less space because it does not fork the execution state. It does have some disadvantages though. Techniques to perform state merging [105] are not possible and some path search heuristics (e.g. breadth first search) are not possible.

2.2.3 Program representation

Although the symbolic execution idea is universal to any imperative programming language there is some degree of flexibility in how this is implemented.

For languages that are compiled to native (i.e. binary) code (e.g. C, C++, Go) it is possible to concolically execute the binary. Tools such as SAGE [81], angr [165], S2E [45], pysymemu [70], Mayhem [42], and FuzzBall [128] do this. This approach has the advantage that it works on unmodified binaries and is agnostic to the original source language. However it has the disadvantage of being tightly coupled to a particular platform (e.g. x86_64 Linux), making it harder to support other platforms. This is partially mitigated by raising the binary code to an intermediate representation and then symbolically executing that. For example the angr tool uses the VEX intermediate representation (IR) that is used by the Valgrind binary analysis framework.

Symbolically executing an IR (somewhere between the original source code and final binary) is a popular approach taken by many tools. The approach is popular because using an intermediate representation often avoids some of the complexities of the original source language but also avoids some of the complexities of the platform being targeted. The degree to which these complexities are avoided is dependent on the design of the intermediate representation.

For managed languages such as Java and C#, the byte code emitted by those languages' compilers is a form of intermediate representation that is intended to be executed by a virtual machine. This representation is decoupled from the platform details of the machine it is being executed on and so it can be symbolically executed by implementing a virtual machine for the byte code. Java-SPF [148] does this for Java and Pex [171] does this for C#.

Some tools also work with a compiler's intermediate representation. For example the KLEE tool uses the LLVM IR used by the LLVM compiler infrastructure. LLVM IR primarily exists as a representation for performing platform independent optimisations. Being a compiler IR removes some of the complexities of dealing with source language features (e.g. templates from C++) but adds some additional complexity because some details only known at runtime are not available (e.g. which C library and other run time libraries will be used). KLEE partially mitigates this by providing its own runtime libraries, however these libraries might not accurately reflect what will be used at runtime by the program. Given that the LLVM IR format is not stable because it changes with every compiler release (unlike the Java and C# byte code) this requires the source code of the program to be available, unlike tools that work at the binary level.

Another IR that will become important later in this thesis is an IR for verification tools, an intermediate verification language (IVL). In particular the Boogie IVL will be used in Chapter 3 and is discussed more in §2.3.2. In Chapter 3 we develop a symbolic execution tool named Symbooglix for the Boogie IVL and also compare it against an existing symbolic execution tool for the Boogie IVL named Boogaloo [151].

Finally, some tools work at the program source level such as CREST [37] and EXE [40]. These tools add instrumentation at the source code level and then the program is compiled using the compiler normally used to compile the program. This has several disadvantages. First it requires the source code to be available (which is not always the case) and it also means that the implementation is tightly coupled with the programming language used by the tool and thus it must handle all the complexities that exist there.

2.3 Static Verification

First we give an overview of static verification and then discuss *weakest pre-condition generation* in §2.3.1. Next in §2.3.2 we discuss an intermediate representation for verification tools, the Boogie intermediate verification language (IVL). This IVL is the common representation used by the tools (most of which are based on some form of static verification) we compare in Chapter 3 and is used by the Symbooglix tool we develop in that chapter. After discussing the Boogie IVL, in §2.3.3 we discuss the available Boogie front-ends, some of which are used in Chapter 3 to generate benchmarks used in the comparison performed in that chapter. Finally in §2.3.4 we discuss the available Boogie back-ends all of which are used in the comparison we perform in Chapter 3.

Static verification is a branch of static analysis that tries to formally prove properties of programs. It has its roots in the seminal work from Hoare [89] and Floyd [74]. Floyd introduced a method for proving program properties using *flow graphs* (*control flow graphs* in modern parlance) from which Hoare derived a deductive system (i.e. a set of inference rules and axioms). This system could be used to derive a proof or disprove properties of programs. In particular these properties are expressed in what is now known as a Hoare triple. A Hoare triple, written as $\mathbf{P}\{\mathbf{Q}\}\mathbf{R}$ states that for a program $\mathbf{Q}$ if the pre-condition $\mathbf{P}$ holds then the post-condition $\mathbf{R}$ always holds.

2.3.1 Weakest pre-condition generation

Dijkstra [65] extended Hoare's work by introducing the concept of *weakest pre-condition* (wp) which is a predicate that relates the pre and post condition of a program such that:

$$\forall v_0, \ldots, v_n. \mathbf{P} \rightarrow wp(\mathbf{Q}, \mathbf{R}) \iff \forall v_0, \ldots, v_n. \mathbf{P}\{\mathbf{Q}\}\mathbf{R} \tag{2.1}$$

That is to say if it can be proved that the pre-condition **P** implies the *weakest pre-condition* $wp(\mathbf{Q}, \mathbf{R})$ for all possible values of the predicate variables $(v_0, \ldots, v_n)$ (i.e. $\mathbf{P} \rightarrow wp(\mathbf{Q}, \mathbf{R})$ is valid) then the Hoare triple $\mathbf{P}\{\mathbf{Q}\}\mathbf{R}$ always holds. The predicate variables are usually variables used in the program **Q** that are needed to express the post-condition. The left hand side of equation 2.1 is known as the *verification condition* (VC) and is shown in equation 2.2.

$$\forall v_0, \ldots, v_n. \mathbf{P} \rightarrow wp(\mathbf{Q}, \mathbf{R}) \tag{2.2}$$

The VC can be solved manually, using an interactive theorem prover, or a constraint solver (automated theorem prover). Modern static verification tools typically use constraint solvers to check generated VCs. These solvers are discussed in section 2.6.

Dijkstra gave a procedure for computing the *weakest-precondition* of a program. Unfortunately this procedure does not work well on real world programs. First, real world programs have loops which require special treatment. Second, analysing entire real world programs can create large and potentially complex VCs that are intractable for currently available constraint solvers.

Loops and procedure calls require special treatment that often involves approximating their behaviour. Over-approximation adds additional program behaviours and is therefore *sound* but may introduce false positives. Under-approximation removes existing program behaviours and is unsound because it may introduce false negatives. Some approximations are neither an over-approximation nor an under-approximation. For example approximating machine integers with mathematical integers adds new behaviours (integers are now unbounded) and removes some behaviours (integer overflow no longer occurs).

One approach to approximating loops is to unroll the loop a bounded number of times and then insert a statement to prevent further execution of the loop. This is an under-approximation if execution of the loop can continue after the unrolled iterations. Otherwise, it is precise. This under-approximation can be turned into an over-approximation by replacing the statement that prevents further execution with an over-approximating summary of the loop's behaviour. The most crude approximation is to allow all variables that a loop might modify to take any value. This approximation is likely to cause false positives to be reported. This approximation can be made more precise with the use of a loop invariant. A loop invariant is a Boolean expression that states a condition that holds every time the loop condition is evaluated (i.e. before and after every iteration of the loop body). A loop invariant can be used to summarise a portion of the behaviour of a loop body.

An analogous approach can be used to approximate procedure calls. A procedure call can be approximated by inlining up to a fixed recursion bound and then inserting a statement to prevent further recursive calls. This is an under-approximation if execution of the procedure call can occur past the recursion bound. Otherwise, it is precise. This approximation can be turned into an over-approximation by replacing the statement that prevents further recursive calls by a summary of the procedure's behaviour.

To handle real world programs the size and complexity of the VC may need to be reduced to become tractable. Dijkstra's procedure creates VCs that grow exponentially with program size. However Flanagan and Saxe [73] developed an approach for VC generation where the size of the VC grows quadratically in program size in the worst case. Although this approach to reduce the size of the VC is helpful it might not be enough to make generated VCs tractable. The only available solution to this is to introduce some approximations at the expense of soundness and/or completeness. One way to apply this is at the function level, which allows verification to be performed in a modular fashion. Each function must be annotated with a summary of its behaviour (i.e. what global variables it might modify and what its pre and post conditions are). Then each function can be analysed separately and calls to other functions can be approximated with their summaries. If all functions are proven correct (i.e. the implementation conforms to the summary) then the entire program is correct. Typically these pre and post conditions cannot be automatically inferred and so must be provided by a developer. There are many other approaches to approximation that are out of scope for this discussion.

So far we have discussed static verification through the lens of *weakest pre-condition* generation. However it is worth noting that other important static verification techniques such as *model checking* and *abstract interpretation* exist.

Model checking explores the states of the program and checks that after some number of execution steps one or more properties hold. Bounded model checking is a variant of model checking that employs the loop and procedure approximation methods previously to discussed to provide a guarantees of correctness up to a finite bound.

Abstract interpretation [58] formalises the notion of executing a program with approximations. Further discussion of these techniques is out of scope, but D'Silva et al. provide an excellent survey of model checking and other verification techniques [60].

2.3.2 The Boogie IVL

An intermediate verification language (IVL) simplifies the task of building a program analysis tool, by decoupling the handling of the semantics of a real world programming language from the method that is used to assess the correctness of programs. Much like a compiler, a program analysis tool can have a front-end and a back-end, linked by a common intermediate representation of the input program. The front-end translates the input program from a programming language (e.g. C) into the IVL; the back-end then analyses the IVL program. A single IVL back-end can thus act as an analyser for multiple high-level languages if the front-ends are available. Example IVLs include The Boogie IVL [116] (subsequently abbreviated to Boogie when it is clear what is meant from the context), WhyML [71] and the IVL proposed by Le et al. [110].

Boogie is a small and simple IVL with a clearly-defined semantics, in contrast to the semantics of real-world programming languages that are usually prone to a degree of ambiguity. A front-end targeting Boogie commits to a specific encoding of the source language, after which program analysis is performed with respect to the precise semantics of Boogie. Resolution of source language ambiguity is controlled explicitly by the front-end, and does not taint the underlying program analysis techniques. Boogie has several front-end and back-ends which are discussed in§2.3.3 and §2.3.4 respectively.

Boogie has traditionally been used for program verification, via the Boogie verifier back-end [15], but recently the IVL has also been used for bug-finding (e.g. Boogaloo [151] and Corral [108]). Corral recently replaced the SLAM tool [13] as the engine that powers Microsoft's Static Driver Verifier [107].

We illustrate some of the core features of Boogie using an example C program in Listing 2.2.

Listing 2.2: An example C program performing checked division.

```
1  bool err;
2
3  int checked_div(int a, int b) {
4    // POST: (err && b == 0) || (!err && \result == a/b)
5    err = false;
6    if (b == 0) {
7      err = true;
8      return a;
9    }
10   return a / b;
11 }
12
13 void test_div(int a, int b) {
14   // PRE:  a != 0 && b != 0
15   assert(checked_div(a * b, a * b) == 1);
16   assert(!err);
17 }
```

In Listing 2.2 the function `checked_div` takes two `int`s `a` and `b` and returns the result of dividing `a` by `b` unless `b` is zero, in which case `a` is returned and a global `err` flag is set. The function contains a post-condition (as comments) on line 4 that summarises the behaviour of the function. The function `test_div` checks that dividing `a*b` by `a*b` (by calling `checked_div` on line 15) yields the value 1 and on the next line it is asserted that the error flag is not set. The function contains a pre-condition (as comments) on line 14 that the function should be assumed to only be called with non-zero arguments.

Listing 2.3: An example Boogie program performing checked division that is a potential translation of the C program in Listing 2.2.

```
1  var err: bool;
2  function MUL(int, int): int;
3  axiom (forall a, b: int :: (a != 0 && b != 0 ==> MUL(a, b) != 0));
4  procedure checked_div(a: int, b: int) returns (r:int)
5    ensures (err && b == 0) || (!err && r == a div b); {
6    err := false;
7    goto l_then, l_end;
8  l_then:
9    assume b == 0;
10   err := true; r := a; return;
11 l_end:
12   assume b != 0;
13   r := a div b;
14 }
15 procedure test_div(a: int, b: int) returns (r:int)
16   requires a != 0 && b != 0; {
17   call r := checked_div(MUL(a, b), MUL(a, b));
18   assert r == 1;
19   assert !err;
20 }
```

In Listing 2.3 a possible translation of the C program to Boogie is shown. The `checked_div` and `test_div` procedures correspond to the C functions of the same name in the corresponding C program.

Instead of employing integer multiplication directly, multiplication is modelled using an *uninterpreted function* `MUL` : (`int` $\times$ `int`) $\rightarrow$ `int` (line 2). An *axiom* constrains `MUL` to satisfy the *integral domain* property of integer multiplication: if a and b are non-zero then $a \cdot b$ is non-zero (line 3). This abstraction captures exactly the property of multiplication needed to verify this example, avoiding potentially expensive reasoning about non-linear multiplication.

Procedure `checked_div` stores its result in an explicit return variable, `r` (line 4), and the post-condition (`ensures`) and pre-condition (`requires`) for `checked_div` and `test_div` are formalised (lines 5 and 16, respectively). The body of `checked_div` uses a non-deterministic `goto` statement (line 7) and two `assume` statements (lines 9 and 12) to model an `if` statement. Control may transfer non-deterministically to either one of the `l_then` and `l_end` labels targeted by the `goto`. An `assume` e statement blocks further program execution (in a non-erroneous manner) if the guard e does not hold. Thus for any concrete value of `b`, exactly one of the assume statements at lines 9 and 12 will cause execution to block.

The example illustrates semantic choice when translating from a language such as C into an IVL. The Boogie `int` data type represents the infinite set of mathematical integers, while the C `int` data type represents a finite set of values. In the translation of Listing 2.2 to Listing 2.3 we have ignored the possibility of overflow, pretending that the C `int` represents mathematical integers. With this encoding, the Boogie program of Listing 2.3 is deemed correct by the Boogie verifier [15]: the post-condition of `checked_div` holds for all inputs, and the assertions of `test_div` hold under the assumption of the precondition and the axiom constraining the behaviour of `MUL`. In contrast, the C program of Listing 2.2 is not correct for all inputs, due to arithmetic overflow (which is undefined for signed integers in C).

Listing 2.4: An example Boogie program performing checked division that is a potential translation of the C program in Listing 2.2. It is similar to Listing 2.3 but models the C `int` type as a 32 bit wide bit-vector rather than using mathematical integers.

```
1  var err: bool;
2  function {:bvbuiltin "bvmul"} MUL(bv32, bv32): bv32;
3  function {:bvbuiltin "bvsdiv"} SDIV(bv32, bv32): bv32;
4  procedure checked_div(a: bv32, b: bv32) returns (r:bv32)
5    ensures (err && b == 0bv32) || (!err && r == SDIV(a, b)); {
6    err := false;
7    goto l_then, l_end;
8  l_then:
9    assume b == 0bv32;
10   err := true; r := a; return;
11 l_end:
12   assume b != 0bv32;
13   r := SDIV(a, b);
14 }
15 procedure test_div(a: bv32, b: bv32) returns (r:bv32)
16   requires a != 0bv32 && b != 0bv32; {
17   call r := checked_div(MUL(a, b), MUL(a, b));
18   assert r == 1bv32;
19   assert !err;
20 }
```

In Listing 2.4 an alternative translation of Listing 2.2 is shown. In this translation the `int` C data type is modelled using Boogie's 32-bit bit-vector type (`bv32`). In this program multiplication is modelled using a function declared on line 2. The function declaration has a `bvbuiltin` attribute which means the function is not uninterpreted and is instead given the interpretation specified in the attribute. Here `MUL` is interpreted as the `bvmul` function defined by SMT-LIBv2 (see §2.6). Similarly, the `SDIV` function is declared on line 3 to model division with signed two's complement operands by using the `sdiv` function defined by SMT-LIBv2.

The Boogie program in Listing 2.4 is not correct. The `assert` on line 18 can fail. If `MUL(a, b)` is equal to 0 then `checked_div` returns 0, but the `assert` asserts that it returns 1. `MUL(a, b)` can return 0 is overflow occurs, for example if `a` and `b` are both `0x80000000`. Given that the original C program was incorrect due to overflow this shows that Listing 2.4 is a more precise translation.

We now discuss some further Boogie features not illustrated by the above examples that we shall refer to later in Chapter 3.

The havoc command accepts a sequence of variable names and sets each variable to a non-deterministic value. This allows abstract modelling of side effects, e.g. reading a value from the network with no knowledge of what that value might be.

Specification-only procedures abstractly describe the behaviours of procedures for which no implementation is available (i.e. no body), via a *contract*: a pre- and post-condition, and a *modifies* set specifying which global variables might be updated by the procedure. Calling a specification-only procedure involves *asserting* the pre-condition, *havocking* the variables in the modifies set, and *assuming* the post-condition. The modified variables thus take arbitrary values satisfying the post-condition of the contract.

Maps allow modelling of array and heap data. For types S and T, the type $[S]T$ represents a total map from S to T. If type S has an infinite number of distinct values (e.g. if $S = \mathbb{Z}$) then a map of type $[S]T$ has an infinite number of keys.

Global constants can be declared, and axioms used to restrict their values. The `unique` qualifier specifies that a global constant of type T should have a distinct value from any other `unique`-qualified global constant of type T.

Functions are mathematical functions and so are stateless. This makes them distinct from procedures which can have state. If the function has no body then it is treated as an uninterpreted function (i.e. nothing is assumed about the function other than that it must always give the same output when given the same input arguments). If the function has a body it is a single expression describing the result computed by the function in terms of its arguments.

Old expressions are expressions in Boogie that contain use of the built-in `old()` function. It represents the value of an expression at the entry point of a procedure. This can be used to compare the old and new value of expressions in the `ensures` clause when leaving the procedure. The value of these expressions can be different because the expression can refer to global variables which might be modified by the procedure.

For more discussion of Boogie see [116] and `https://boogie-docs.readthedocs.io/en/latest/`.

2.3.3 Boogie front-ends

Several front-ends for various programming languages have been developed including C (SMACK [156] and VCC [51]), Java (Joogie [7]), C# (BoogieBCT [23]), Dafny [115] and OpenCL/CUDA (GPUVerify [26]).

The SMACK and GPUVerify front-ends are of particular interest because they are used to create the benchmark suite used to compare various Boogie back-ends in Chapter 3.

Both tools are built on top of the LLVM compiler infrastructure, more specifically they use Clang [47] to compile existing source code to LLVM IR which is then converted to Boogie. Despite this similarity the tools are very different to each other. SMACK targets C programs (single threaded or concurrent) and (at the time the work in Chapter 3 was performed) models integers as mathematical integers . SMACK is designed to check general properties of C programs such as no assertion failures and freedom from signed integer overflow.

The GPUVerify front-end on the other hand, targets GPU kernels and models integers as bit-vectors. It is designed primarily to verify freedom from two classes of defects in GPU programming: *data races* and *barrier divergence*. A parallel kernel is translated into a sequential Boogie program, instrumented with assertions that check whether it is possible for two arbitrary but distinct threads to race or diverge on a barrier. It can also check user provided assertions.

2.3.4 Boogie back-ends

In Chapter 3 we perform an evaluation of various Boogie back-ends. We now briefly survey these back-ends. All these back-ends use Z3 as their constraint solver.

The Boogie verifier [15] applies weakest precondition generation methods as discussed in §2.3.1 to transform each procedure in a Boogie program into a VC to be checked by a constraint solver. The Boogie verifier uses loop cutting [16] to over-approximate loops in a sound manner. Procedure calls are soundly approximated by using pre and post conditions on procedures. Neither loop invariants nor pre and post conditions are inferred automatically. The lack of these often lead to false positives being reported, which makes the tool difficult to be used practically for bug finding, however being a sound tool, it will never report false negatives. The Boogie verifier supports all features of the Boogie IVL.

Boogaloo [151] is a symbolic execution tool (see §2.2) for Boogie programs that aims to provide a way of debugging failed verification attempts. Boogaloo incorporates several interesting features.

Boogaloo treats maps as having a finite domain, even if in Boogie's semantics they have an infinite domain. Boogaloo does this by representing every map access by a fresh variable. This approach is sound and is necessary because a map of infinite size cannot be instantiated on any real computer because would require an infinite amount of storage.

Boogaloo has a feature which the authors refer to as "concretization". This feature tries to workaround the problem of the set of path constraints growing too large. It does this by periodically asking its underlying constraint solver to pick concrete values for all symbolic variables (including maps), replaces the values of the variables in the program state with the concrete values, and then makes the path constraint set the empty set. This is unsound (i.e. may report false negatives), but will not introduce false positives. This is because "concretization" may cause feasible paths to be missed, but will not add additional paths. In this sense it is an under-approximation. This feature can be partially disabled by the user if needed, so that it only happens when execution of a path terminates. The user can also request that multiple solutions at the time of "concretization" are tried.

Boogaloo also tries to avoid giving quantified constraints to its underlying constraint solver. It does this because constraint solvers often exhibit poor performance on quantified constraints. When encountering a quantified constraint Boogaloo forks execution, one path assuming the quantified constraint be true, and on the other path assuming it to be false (i.e. the negated form of the quantified expression). This forking is necessary to decide what form of the constraint to add to the path constraints. Before adding the quantified constraint to the set of path constraints it is "Skolemized" which eliminates existential quantifiers by adding fresh variables. This approach eliminates all existential quantifiers (leaving only universal quantifiers in constraints) and is sound.

Boogaloo also has an unsound feature called "finitization" which eliminates any remaining universally quantified constraints. It does this during prior to "concretization", by instantiating the universally quantified expressions, and removing the universally quantified expression from the constraint set. For example the constraint $\forall a : f(a)$, where a is of Boolean type can be instantiated soundly as $f(\text{true}) \wedge f(\text{false})$. However Boogaloo doesn't instantiate quantifiers like this. Its strategy is to ignore all quantifiers that don't perform assignments to maps. For universally quantified constraints that do assign to maps they are instantiated for all map locations previously accessed. This unsound feature is based on the authors' observation that universally quantified constraints are frequently used to to write axioms over maps.

Finally Boogaloo has a "minimization" feature that tries to simplify generated test cases by performing a binary search to try to find small values for symbolic variables. The rationale behind this is that the authors believe test cases involving small numbers are more comprehensible.

Boogaloo does not provide complete coverage of the Boogie language because it lacks support for bit-vector types. The tool does not (modulo implementation bugs) report false positives.

Corral [108] is a whole-program analyser built on top of the Boogie verifier. Corral first transforms every loop into a tail-recursive procedure call, then uses Houdini [72] to infer (possibly trivial) pre/post specifications for all procedures. Starting at a given program entry point, Corral employs a technique called *stratified inlining*. Procedures are inlined up to a specific depth, after which two verification conditions (see §2.3.1), A and B are checked: in A, non-inlined procedures are over-approximated using pre and post conditions on procedures. In B, non-inlined procedures are under-approximated via an `assume false` command. Unsatisfiability of condition A implies that the program is correct, while satisfiability of condition B exposes a bug that can manifest within the inlining depth. Otherwise, analysis of the solution to A identifies a non-inlined procedure whose effect *may* trigger a bug; this procedure is inlined and the process repeats. Analysis bails out when a maximum inlining depth is reached. The depth-bounded nature of the procedure is reminiscent of bounded model checking techniques. This nature also means that the tool can only report a program as correct when the program can no longer be inlined further and the depth bound has not been reached. Corral does not (modulo implementation bugs) report false positives. Corral supports all of the Boogie language due to it being built on top of the Boogie verifier which has full support.

Duality [130] employs Craig interpolation to compute inductive invariants at program points, generalising the *Impact* algorithm [129]. The goal is to use these invariants to prove software correctness, though bugs may be identified during invariant search. As with Corral, loops are first transformed into tail-recursive calls. An iterative process is then used to search for bugs up to a given inlining depth. Each time a recursion point is reached, if no bugs are detected then Duality extracts an *interpolant* from the unsatisfiability proof of the current verification condition, generalising the program state on entry to the recursive procedure. If the states represented by this interpolant are contained inside the states represented by previously computed interpolants for the procedure, Duality concludes that no bugs can arise via further ex-

ploration of the procedure's execution. Duality is sound and does not (modulo implementation bugs) report false positives. Duality is actually built on top of components of the Corral tool, however its current implementation does not support bit-vectors.

GPUVerify [26] has already been discussed in terms of its front-end but it also has a sophisticated back-end that is built on top of the Boogie verifier. The GPUVerify back-end is only designed to verify Boogie programs generated by its front-end and thus is not a general purpose Boogie back-end. Most GPU kernels contain loops but the Boogie verifier provides virtually no built-in facility for automatically inferring loop invariants. GPUVerify attempts to automatically infer invariants by using GPU kernel specific heuristics to guess *candidate* invariants, and then by applying the Houdini algorithm [72] to compute the largest subset of guessed candidates that actually forms an inductive invariant. These inductive loop invariants are then added to the Boogie program and it is given to the Boogie verifier. The need for loop invariants means that GPUVerify may report false positives, however the back-end is sound. The GPUVerify back-end supports bit-vectors.

2.4 Fuzzing

Fuzzing is a simple but powerful technique for testing software. The basic idea is to generate random inputs and give them to a program in the hope of eliciting some kind of property violation (e.g. crash).

The origin of the term *fuzzing* can be attributed to Miller [135] who observed that noise on his dial-up connection was causing random input to be sent to his remote terminal and triggering crashes in some of the programs he was using. He wasn't expecting well used utilities to crash so easily, so he set his students a project to fuzz well known UNIX utilities. The project successfully found inputs that crashed 25-33% of the UNIX utilities they examined.

This class of fuzzing is *random fuzzing*. It is also referred to as *blackbox fuzzing* because inputs are generated without using any knowledge about the program under test. This form of fuzzing is very convenient to use but it is unlikely to find deep bugs. The program shown in Listing 2.5 is shown to illustrate this.

Listing 2.5: A simple C program to illustrate the limitations of *random fuzzing*.

```
1  int main() {
2    char* x = input(/*size=*/6);
3    if (x[0] == 'H') {
4      if (x[1] == 'E') {
5        if (x[2] == 'L') {
6          if (x[3] == 'L') {
7            if (x[4] == 'O') {
8              if (x[5] == '\0') {
9                abort();
10             }
11           }
12         }
13       }
14     }
15   }
16   return 0;
17 }
```

On line 2 a random input of 6 bytes is requested. This is then followed by a series of nested `if` statements. If all the conditions are true then `abort()` is called on line 9. A random fuzzer is very unlikely to guess that the input required to reach line 9 is the NULL terminated `"HELLO"` string because there are 2^{48} different possible bit patterns. Assuming the fuzzer uses a uniform probability distribution the probability of choosing at random the abort-triggering input is $\frac{1}{2^{48}}$. Given this, it can be shown (§A.1) that the expected number of tries on average (i.e. the expectation value) required to guess the abort-triggering input is 2^{48}.

For a fuzzer to make more progress on a program like this it can't treat the program under test as a black box. Instead it must use information about the program to guide the search for interesting inputs.

One such class of fuzzers are *mutation-based fuzzers*. Mutation-based fuzzers start with a set of existing inputs (know as a *corpus*) and randomly mutate these inputs to try to find new inputs that cause interesting program behaviour. An example mutation is performing "crossover", where multiple inputs are combined into one. By providing the fuzzer with a set of known valid inputs we are giving it information on what kind of inputs the program under test expects.

We can give the fuzzer even more information by providing it with information on what code in the program was covered by a particular input, thus indirectly communicating the structure of the program under test. This is known as *coverage-guided fuzzing*. Whenever a new input covers a new piece of code (e.g. a statement or branch), this can be used as an indicator that the input is an interesting input to be added to the corpus and mutated further. This approach is much more likely to find the right input to make the program in Listing 2.5 reach line 9. In this example the fuzzer only needs to guess that first byte needs to be `'H'` (it does not matter what

the value of the other bytes are) to receive feedback that a new branch was covered. When this happens the input that starts with `'H'` is added to the corpus. Now when the fuzzer mutates this new input it only needs to guess that the first byte should be left untouched and that the second byte needs to be `'E'` (the value of subsequent bytes does not matter). This process will repeat until the abort-triggering input is found to reach line 9. Assuming only one byte mutation is applied for each mutation, the probability of performing a mutation that covers a new branch in this example is $\frac{1}{6} \times \frac{1}{256}$. Given this, it can be shown (§A.2) that the expected number of mutations (i.e. the expectation value) required until the abort-triggering input is found is 9216. This is significantly less than the 2^{48} expected number of attempts required by random fuzzing.

This approach is essentially an *evolutionary algorithm* [64] where an input is considered *fit* if it covers new code. An evolutionary algorithm used in this context is part of a broader research area known as *search-based test case generation* [5].

Two well known coverage-guided mutation-based fuzzers are AFL [1] and LibFuzzer [119]. Both tools have found bugs in real world software [2, 120].

It is worth noting that fuzzers and concolic execution tools have similarities. Both repeatedly re-run a program with different inputs. A mutation-based fuzzer randomly mutates inputs (with guidance in the case of coverage-guided fuzzers) whereas concolic execution mutates gathered constraints and relies on a constraint solver to generate a new input from the mutated constraints. This similarity has not gone unnoticed and several tools [125, 168, 146, 42, 145] have been developed that are a hybrid of fuzzing and concolic execution.

Another class of fuzzers are *grammar-based fuzzers*. If the developer's goal is to test parsing of the input then applying random mutations is a good approach because it will generate lots of invalid inputs which the parser must recognise. However if the goal is to test components of a program that handle valid input (i.e. after parsing) then using random mutations is a bad approach because most inputs will not be valid and so execution will rarely make it through to the parser which is the code that we wish to test. This is where *grammar based fuzzers* come in. Grammar based fuzzers are aware of the structured nature of the inputs and restrict themselves to randomly generating "valid" inputs. CSmith [175] is a well known grammar-based fuzzer that randomly generates syntactically valid and semantically defined (i.e. no undefined behaviour is used) C99 programs. It has been used to find compiler crashes and correctness

bugs. Another notable example is LangFuzz [90] which is distinct from CSmith in that it aims to be language agnostic. It takes a grammar and is able to by-pass the need for language specific semantic rules by learning from a set of existing programs and mutating them.

2.5 Floating-point arithmetic

In this section we discuss floating-point arithmetic. This is relevant to Chapter 4, where we extend a symbolic execution tool to reason about floating-point programs, and Chapter 5, where we build a floating-point constraint solver.

First we give an overview a floating-point arithmetic. Then we discuss the IEEE-754 standard for floating-point arithmetic (§2.5.1). This is followed by a discussion of rounding modes and floating-point exceptions (§2.5.2). Finally we discuss notable work on the analysis of floating-point programs (§2.5.3).

For various problem domains (e.g. scientific computing) it is necessary to manipulate real valued quantities in programs. The set of real numbers is infinite and some values (e.g. irrational numbers) cannot be represented exactly with finite precision. Given that computers have a finite amount of storage space it is necessary to use a representation that approximates real numbers. This representation can either be fixed point or floating-point. In a fixed point representation the radix point[1] is at a fixed position which means the available precision (number of digits after the radix) is constant. In a floating-point representation the radix point is not at a fixed position and so the available precision depends on the magnitude of the real number being represented.

A *fixed-point* representation is useful if a fixed amount of precision is desired and the range required is small (e.g. $[-1.0, 1.0]$). However this representation is not suitable if the range required is large and so a floating-point representation must be used in this scenario.

In general, floating-point number representations provide finite approximations to the real numbers, trading range and precision for storage space. They utilize scientific notation of the form $(-1)^s \times m \times b^e$ where s is either 1 or 0 and controls the sign, m is a real number called the significand (typically in the range $[0, b)$), b is an integer base (typically 10 or 2), and e is an

[1] in base 10 this is called the decimal point

integer exponent. For example -0.75 can be written as -1.5×2^{-1}. By using a fixed number of digits for the exponent and significand, floating-point number representations restrict the range and precision, respectively, of representable numbers.

There have been several different floating-point representations implemented over the years but the most popular representation is the one described by the IEEE-754 2008 standard [91] (also known as IEC 60559:2011 [93]).

In this thesis when considering the correctness of programs that use floating-point arithmetic, we will be concerned with C programs. If an implementation of C respects Annex F of the C language specification [95], then most of its floating-point types and operations are IEEE-754 compliant (some exceptions to this are detailed below).

2.5.1 IEEE-754 floating-point on x86_64.

We provide details of IEEE floating-point representation implemented by the x86_64 family of processors.

Table 2.2: x86_64 floating-point types.

Name	Size	p	e_{max}	leading bit m explicit?
fp16	16 bit	11	15	No
fp32	32 bit	24	127	No
fp64	64 bit	53	1023	No
x86_fp80	80 bit	64	16383	Yes

Four primitive floating-point types are available on this target: 16-bit wide *half precision*, 32-bit wide *single precision* (IEEE-754 binary32), 64-bit wide *double precision* (IEEE-754 binary64), and 80-bit wide *double extended precision* (not an IEEE-754 basic format). We refer to these types as fp16, fp32, fp64, and x86_fp80 respectively. For the fp16 type only conversion operations are supported on x86_64.

Each type has a precision p (number of bits representing the significand), and maximum exponent value e_{max}. A full list can be found in Table 2.2. The last column ("leading bit m explicit?") states whether the binary encoding of the type contains the leading bit (i.e. the integer portion of) m, or if it is inferred from the remaining bits.

In an Annex F-compliant implementation of C on x86_64, fp32 and fp64 are the `float` and `double` types respectively. The C standard with Annex F only weakly specifies how the `long double` type should be implemented. The type must be able to represent all values that `double` can represent. All C implementations that target x86_64 that we are aware of treat `long double` as x86_fp80.

The IEEE-754 binary format contains several classes of data: normal, denormal, zero, infinity and NaN. Most floating-point numbers belong to the *normal* class which provides a unique encoding for representable numbers where the leading bit of the significand is always `1` and the exponent is in the range $[-e_{max}+1, e_{max}]$. The *denormal* class represents numbers close to zero and exists to provide a smoother transition from the smallest positive *normal* (largest negative *normal*) number to positive (negative) zero. *Denormal* numbers always have the exponent and leading bit of the significand set to $-e_{max}$ and `0` respectively. The *zero* class contains two values, positive and negative zero. The *infinity* class contains positive and negative infinity. These four classes allow every real number to be approximated.

The *NaN* class represents "not a number". NaN values arise from invalid computations, such as `0.0/0.0`. NaNs are not comparable: with the exception of the `!=` operator, comparing NaN with any value yields *false*. There are many different binary encodings for NaN but IEEE-754 only distinguishes between two types: quiet and signaling. The difference between these only matter when considering IEEE-754 exceptions which are discussed later.

In the binary encoding of the above classes the exponent is encoded in excess-e_{max} encoding (e.g. excess-127 for fp32). If the encoded exponent are all zeros the number is either the *zero* class (significand is zero) or the *subnormal* class. If the encoded exponent are all ones the number is either infinity (fractional significand bits are all zero) or NaN. Otherwise the number is *normal*.

The binary encoding used by IEEE-754 types is that the integer portion of the significand (1-bit) can be inferred for the *normal* and *subnormal* classes. Accordingly, this bit is not stored in the binary encoding and its value is implicit. For example, for the fp32 type, $p = 24$ but the significand only occupies 23 bits in memory.

The x86_fp80 type is not an IEEE-754 binary format. It consists of 1-bit sign, 15-bit exponent and 64-bit significand. The binary encoding is similar to that of the IEEE-754 binary format except that the integer portion of the significand is stored explicitly. This additional bit per-

mits extra classes known as *pseudo-NaN*, *pseudo-infinity*, *unnormal* and *pseudo-denormal*. Modern Intel® processors consider all these classes apart from pseudo-denormals as invalid operands [92]. Pseudo-denormals are treated as denormals so that legacy software can be supported.

2.5.2 Rounding modes and exceptions.

When evaluating operations, rounding may need to be performed due to the finite precision available. In IEEE-754, addition, subtraction, multiplication, division and `sqrt()` are *correctly* rounded (i.e. the result is as if the result was computed with infinite precision and an unbounded exponent and then rounded). IEEE-754 provides five different rounding modes: *round toward positive (RTP)*, *round toward negative (RTN)*, *round toward zero (RTZ)*, *round to nearest ties to even (RNE)*, and *round to nearest ties away from zero (RNA)*. The default rounding mode is usually RNE. In C the rounding mode is part of a program's thread-local state and can be modified via a standard library call. The C standard library provides the `<fenv.h>` header than can be used to get and set all of the above rounding modes apart from RNA. IEEE-754 also defines several different exceptions (invalid operation, division by zero, overflow, underflow and inexact) that can be raised when operations are performed. The default handling of these is to set one or more status flags and then continue execution. Most operations when given a signaling NaN as an operand will raise an invalid operation exception but will not raise that exception if all NaN operands are quiet NaNs. The C library provides the `<fenv.h>` header which contains functions for checking and setting the status flags.

2.5.3 Analysis of floating-point programs

The analysis of floating-point programs has received a significant amount of attention from the research community. This work will become relevant in Chapter 4, where we extend a symbolic execution tool to support floating-point programs. Although floating-point constraint solvers are a common component in the work we discuss here, we don't discuss solving floating-point constraints here and instead defer that discussion to §2.6.2.

Test case generation for floating-point programs has been investigated using a wide variety of techniques. One of the earliest works is that of Miller et al. [174] in 1976. In this work they propose collecting path constraints from a program, transforming those path constraints

into a mathematical optimisation problem and then solving it. If a solution is found then that is a test case for the program, if a solution is not found then the feasibility of the path corresponding to the path constraints is inconclusive. Recently Fu et al. [76] developed a symbolic execution tool that used this approach of reformulating floating-point constraints as a mathematical optimisation problem via their constraint solver XSat [77]. The symbolic execution tools, Pex [171] (via FloPSy [106]) and SPF [148] (via Coral [167, 31]) have taken a similar approach by using a solver that reformulates floating-point constraints as a search problem that is solved using meta-heuristic search methods. The FPSE [32, 11] symbolic execution tool uses an interval solver over real arithmetic combined with project functions to model floating-point arithmetic. Quan et al. [154] take a different approach to symbolically executing floating-point programs. They symbolically execute a software implementation of IEEE-754 floating-point arithmetic as part of the program under analysis. This avoids requiring a floating-point constraint solver because all constraints are over bit-vectors.

Earl et al. [18] take a very different approach to test case generation while still using symbolic execution. Their tool Ariadne is designed to generate test cases that trigger floating-point exceptions. During symbolic execution they approximate floating-point arithmetic using real arithmetic (using Z3 to check the constraints) and inject checks for floating-point exceptions after every floating-point operation. If they find a real number that triggers an exception, they then perform a local search for floating-point numbers in the neighbourhood of the real number and then natively replay (i.e. using IEEE-754 floating-point arithmetic) these candidate test cases to try to find a genuine test case. This step of performing native replay is a refinement step to filter out false positives.

Another test case generator worth mentioning is FPGen [3]. This test case generator was used to test an FPU (floating-point arithmetic unit). The generator isn't really designed to test full programs and instead is used to test the execution of one or more floating-point operations along with a set of a constraints. This tool generates random inputs using a portfolio of different constraint solvers.

Bounded model checking has also been applied to test case generation. Collavizza et al. [52] use bounded model checking combined with their FPCS [133] solver (an interval solver which appears to be the same as the one used by FPSE) to look for program inputs that can cause program outputs to exceed a user defined error tolerance. The bounded model checker CBMC [50] supports test case generation for ANSI C programs. The tool solves floating-point constraints

by converting floating-point operations into operations over bit-vectors (i.e. modelling the operation of an FPU) and then uses bit-blasting [104] to convert the constraints into a SAT problem, which is then solved using a SAT solver [35]. Note however due the size of the SAT formula that would be generated by the above approach, a mixture of over and approximations are introduced during the translation. Satisfying assignments or proofs of unsatisfiability are then used to refine the approximations if necessary. This is essentially a form of counterexample-guided abstraction refinement (CEGAR) [49].

Symbolic execution for the purposes of cross-checking floating-point programs has also been investigated. Collingbourne et al. [54] check the output equivalence of two floating-point programs by symbolically executing (speculatively executing paths) both programs and checking using syntactic rewrite rules that the expressions representing the program outputs are equivalent. The authors argue that floating-point expressions are only reliably equivalent if they consist of the same floating-point operations, and use this to justify their approach. This approach completely bypasses the need for a floating-point constraint solver, but is prone to false positives and cannot generate test cases.

Static verification of floating-point programs has also been investigated. Boldo et al. [28, 29] implement a tool that generates verification conditions from annotated C programs which are then solved using a formalisation [61] of the IEEE-754 standard in the Coq proof assistant [57]. The static analyser Astrée [27] based on abstract interpretation, uses an abstract domain over floating-point intervals to verify C programs. Putot et al. [153] also use abstract interpretation to study the propagation of rounding errors.

Some work has looked at *branch instability*. This is where the condition of a branch that depends on floating-point data is susceptible to change when small changes to the floating-point data, or compiler optimisations are applied. Gu et al. [85] define a procedure that, given a program, determines an input that may lead to different branch conditions depending on how the program was compiled. Lee et al. [111] propose a dynamic analysis to detect branch instability at run time. The approach instruments a program by replacing floating-point operations with vectorized floating-point operations where the lanes of the vector represent estimated upper and lower bounds of the result. As computation proceeds, these bounds are updated, and when a branch that uses the floating-point vector is reached, additional instrumentation checks if the error bounds allow for a branch divergence.

Compiler optimisations can have a huge impact on the behaviour of floating-point programs, especially if they are wrong! Lopes et al. [124] implemented a tool (Alive) that formally verifies peephole optimisations in the LLVM compiler. Both Menendez et al. [131] and Nötzli et al. [144], independently extended this tool to support verifying floating-point peephole optimisations. Both research groups found bugs in the way LLVM's floating-point peephole optimisations work, which were subsequently fixed.

Researchers have also looked at the problem of checking a floating-point program's accuracy. Ramachandran et al. [157] use symbolic execution via the SPF tool and their floating-point solver, REALIZER [113], to check the accuracy of programs by symbolically checking that a program's output is within the required error bounds. Panchekha et al. [147] describe their tool (Herbie) that performs a heuristic-based search to find replacements for floating-point expressions in programs that have improved accuracy.

Finally, researchers have also looked at the problem of precision tuning. Precision tuning is the process of increasing or decreasing the precision of floating-point operations (e.g. replace use of `double` with `float` in a C program) in a program to increase performance but at the same time maintain the required level of precision. González et al. [158, 159] demonstrate tools that modify programs at the LLVM IR level to use different floating-point precisions and then perform a guided search over the space of possible programs. Chiang et al. [44] present their tool (FPTuner) which generates mixed precision floating-point expressions from real valued expressions using an error threshold. This is useful for programmers that do not have floating-point expertise that wish to approximate some algorithm that assumes real arithmetic.

2.6 Constraint solvers

Constraint solvers are at the heart of many program analysis techniques mentioned earlier (e.g. symbolic execution and static verification). The main goal of a constraint solver is to prove whether or not a set of constraints are satisfiable.

2.6.1 SAT and SMT solvers

The simplest type of constraints that can be solved are Boolean constraints. Boolean constraints consist of one or more Boolean free variables joined by Boolean operators (e.g. $\wedge$, $\vee$).

Proving satisfiability or unsatisfiability of Boolean constraints (also known as *SAT*) is a well known problem and was the first problem to be shown as NP-complete [56]. Solving this problem has a wide array of applications (e.g. hardware verification) and so a significant effort has been invested in finding efficient ways to solve it. The algorithm underlying most SAT solvers is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [62] which stemmed from the earlier work of Davis and Putnam [63]. Many further advancements (e.g. conflict driven clause learning [166]) coupled with annually held competitions [2] have lead to the creation of high performance SAT solvers capable of handling millions of free variables. Example solvers include minisat [66], Chaff [139].

Of course Boolean constraints are not the only type of constraints that are useful to solve. In program analysis it can be useful to reason over additional types such as machine integers (bit-vectors), floating-point types, mathematical integers and real numbers. It can also be useful to reason using higher order concepts like *uninterpreted functions* and *quantifiers*. Constraint solvers tackling these kinds of constraints are known as *satisfiability modulo theories* (SMT) solvers. These solvers check satisfiability with respect to one or more theories.

A theory is a collection of types[3]; functions over those types; and axioms over those functions and types. A theory essentially describes the language and semantics for constraints over types. SMT-LIBv2.5 [19] is a standard that describes a constraint language for SMT solvers; a set of theories and logics (a collection of one or more, theories); and benchmarks (i.e. example constraint sets). Example theories include *Core* (Boolean), *FixedSizeBitVectors* (machine integers), *FloatingPoint*, and *ArraysEx* (arrays with extensionality). These theories are combined together to form logics. For example the `QF_ABV` logic is for constraints that are quantifier free and use bit-vectors and arrays of bit-vectors. In our work (chapters 4 and 5) we are concerned in particular with reasoning using the *FloatingPoint* theory which is a very recent addition to the SMT-LIB standard [34].

[2] `http://www.satcompetition.org/`
[3] also known as sorts

Listing 2.6: An example set of constraints written in the SMT-LIBv2.5 format in the QF_FPBV logic.

```
1  (set-logic QF_FPBV)
2  (declare-const x (_ BitVec 32))
3  (declare-const y (_ BitVec 32))
4  (assert (= x y))
5  (assert
6    (not
7      (fp.eq
8        ((_ to_fp 8 24) x)
9        ((_ to_fp 8 24) y)
10     )
11   )
12 )
13 (check-sat)
```

In Listing 2.6 an example set of constraints (often called a query) in the SMT-LIBv2.5 language is shown. On line 1 the query declares that is uses the QF_FPBV logic. This is the logic that combines the *FloatingPoint* theory and *FixedSizeBitVectors* theory without using quantifiers. On lines 2 and 3 the free variables x and y are declared. They are both 32-bit wide bit-vectors. Next on line 4 the first constraint is asserted. The constraint states that both x and y are equal to each other (i.e. have the same bit pattern). Then starting at line 5 and finishing at line 12 the next constraint is declared. This constraint converts both x and y to a 32-bit floating point value and then asserts they are not equal to each other. Here the conversion is specified by `(_ to_fp 8 24)`. The number 8 states the number of bits in the exponent and number 24 states the number of bits in the significand (including the implicit bit). Note the = and `fp.eq` functions are different from each other. `fp.eq` is IEEE-754 equality and = is SMT-LIB equality. Finally on line 13 the `(check-sat)` command tells an SMT solver to check the satisfiability of the previous listed constraints. This particular query is satisfiable if either x and y have the same bit pattern that corresponds to an IEEE-754 NaN.

Most SMT solvers use SAT solvers internally to determine satisfiability and thus have benefited tremendously from improvements to SAT solvers. To interface with a SAT solver two different approaches are often used – an eager and a lazy approach. In the eager approach the constraints are transformed into an equisatisfiable set of Boolean constraints. For example bit-vector constraints can be transformed into Boolean constraints using a technique called *bit blasting*. The eager approach is not always possible because some theories cannot be converted into Boolean constraints in all cases (e.g. the *Reals* theory). The lazy approach uses separate solvers for each theory that are combined by communicating via abstractions of the query being solved using only Boolean constraints. This approach is known as DPLL(T) [143].

Not all SMT solvers are based on SAT solvers. Some solvers use a random or guided search-based approach [75, 167, 77] which is incomplete because it can only show satisfiability and cannot prove unsatisfiability. Our work in chapter 5 falls into this category of search-based SMT solvers.

A yearly competition known as SMT-COMP[4] is run where SMT solver implementations compete on a subset of the SMT-LIB benchmarks. The competition aims to encourage further development of SMT solvers and benchmarks.

2.6.2 Floating-point constraint solvers

We now discuss constraint solving for floating-point constraints. This will become relevant in Chapter 5, where we develop our own floating-point constraint solver. Several program analysis tools that analyse floating-point programs rely on floating-point constraint solvers and are discussed in §2.5.3. Some of the solvers we mention here will become relevant in Chapter 5 because we compare against them in the chapter.

There are multiple approaches to solving floating-point constraints. Given that floating-point arithmetic is meant to model real arithmetic, it is tempting to treat floating-point constraints as real constraints and use a solver for real arithmetic. However this won't work well because the semantics of real and floating-point arithmetic are very different. For example floating-point arithmetic performs rounding at every operation, whereas real arithmetic is exact.

One approach is to convert floating-point constraints to bit-vector constraints (essentially replacing each floating-point operation with a circuit implemented using bit-vector operations) which are then bit blasted to a SAT problem. This is then solved using a SAT solver. This is the approach taken by Z3 [140] and SONOLAR [149]. The problem with this approach is that it can produce very large SAT formulas which are difficult to solve [35, 33]. It is also not possible to perform any high level reasoning (e.g. using knowledge of floating-point properties to aid the search for a satisfying assignment) once the problem has been converted to bit-vectors or a SAT problem.

[4] `http://www.smtcomp.org/`

Another approach is to use the previously described DPLL(T) [143] strategy. A problem with this approach is that it separates boolean reasoning from theory-specific reasoning, which can lead to poor performance [33]. Z3 also supports this approach and will use it when asked to solve constraints that combine the `FloatingPoint` theory with some other theories. For example, it will switch to this strategy when asked to solve constraints that use a combination of the `FloatingPoint`, and `ArraysEx` (arrays with extensionality) theories. When asked to solve constraints that use a combination of the `FloatingPoint` and `BitVector` theories it will use its bit blasting strategy. This behaviour of Z3 will become important later in Chapter 4 because it required us to implement the *array ackermanization* optimisation that we describe in the chapter.

Brain et al. [33] describe a different approach, which they implement in the MathSAT5 [46] solver (note this is not the default behaviour). In their approach all reasoning is performed in the floating-point theory itself. They use a combination of an interval solver combined with abstract conflict driven clause learning (ACDCL). The authors claim that it is frequently faster than the bit blasting strategy.

Zeljić et al. [176] describe a different approach, which they implement in the Z3 solver (note at the time of writing this is only available in a fork of Z3). They define a framework for applying approximations (that need not be under- nor over- approximations) to constraints. The approximated constraints are then solved using existing techniques, previously described. If the approximation of the constraints are deemed satisfiable, they then use the approximate model to construct a precise model. If the precise model also satisfies the constraints, satisfiability has been proved. If the precise model does not satisfy the constraints, then the precise model is used to guide refinement of the approximation and the solving process starts again. If the approximation of the constraints are deemed unsatisfiable then the proof of unsatisfiability is used to refine the approximation and the solving process starts again. If it is no longer possible to refine the approximation (i.e. the currently used approximation is precise) then the constraints are unsatisfiable. The approximation used in their implementation uses floating-point arithmetic with less precision than the original constraints. Approximation refinement consists of increasing the precision towards the precision used in the original constraints. The authors show that their approach is competitive with the ACDCL strategy used in MathSAT5.

This work is very similar to CBMC's strategy for handling floating-point constraints [35]. Both of these works are essentially a form of counterexample-guided abstraction refinement (CEGAR) [49].

Leeser et al. [113] describe another approach that they use in their solver, REALIZER. This solver uses real arithmetic to precisely model the effect of floating-point rounding. The transformed constraints are then given to an existing solver for the theory of real arithmetic. This approach is very suitable when floating-point constraints need to be combined with constraints that use reals. The authors' use case is to check that the deviation of floating-point arithmetic from its real counter part is within a certain error tolerance.

The FPCS constraint solver uses an interval solver over reals combined with projection functions to solve floating-point constraints [133, 134]. The COLIBRI [36], solver also takes a similar approach, however their representation of intervals allows for multiple domains (e.g. integer, known bits) and also supports the `FixedSizeBitVectors` theory. COLIBRI also distinguishes itself by supporting the SMT-LIBv2.5 input format and has competed in the 2017 SMT-COMP[5].

Some solvers take a search-based approach to solving floating-point constraints. These solvers are especially relevant to Chapter 5 because the solver we implement belongs to this category of floating-point solvers. These solvers typically generate random seeds (i.e. a candidate satisfying assignment) as a starting point for the search. These solvers are incomplete because they only search for satisfying assignments to floating-point constraints. They cannot prove constraints are unsatisfiable in most cases because doing so would require exhaustive enumeration which is usually not practical.

Both the XSat [77] and goSAT [25] reformulate finding satisfying assignments as a mathematical optimisation problem, and then apply existing mathematical optimisation techniques to solve it. Both solvers support a subset of the `FloatingPoint` SMT-LIB theory.

The FloPSy [106] and CORAL [167] solvers use meta-heuristic search [83] methods (e.g. alternating variable method) to search the space of potential satisfying assignments. A distinguishing feature of Coral is that it optionally support using an interval solver. When this feature is enabled Coral asks the interval solver to attempt to generate intervals that might contain solutions for each symbolic variable. Values from these intervals are then used to seed the search. FLoPSy, although open source, is tightly integrated with the Pex symbolic execution tool and

[5] `http://smtcomp.sourceforge.net/2017/`

so can't be in other contexts. Coral can be used as a stand-alone constraint solver, however it has its own constraint language that only partially intersects with the `FloatingPoint` and `Core` SMT-LIB theories.

Chapter 3

Symbolic execution of Boogie programs

3.1 Introduction

Symbolic execution (see §2.2) as a program analysis technique offers advantages over existing static analysis techniques such as weakest pre-condition generation (see §2.3.1) by providing precise error traces, and test cases without false positives. All of this is provided without requiring the user to provide program invariants.

Recently there has been growing interest in the use of symbolic execution for testing code written in production languages such as C [38, 81] and Java [6]. In contrast, few attempts to apply symbolic execution in the context of intermediate verification languages (IVLs, see §2.3.2) have been reported [151, 110], and to our knowledge no attempts have been made to compare symbolic execution and static analysis tools that work at the level of IVLs. This is the research problem we tackle in this chapter. To tackle this problem we perform a large scale evaluation of different symbolic execution and static verification tools that all operate on the same IVL. The work seeks to test two hypotheses:

1. Symbolic execution of an IVL is competitive with other techniques in terms of bug finding and verification.

2. The state-of-the-art for symbolic execution of IVLs can be improved.

In this work we use the Boogie IVL (see §2.3.2), a popular IVL with several existing front-ends for different languages, and several back-ends, including a symbolic execution execution tool. Having several front-ends potentially provides a large source of benchmarks for our evaluation and having several back-ends provides us with a significant number of tools to compare.

Performing a comparison of existing Boogie IVL back-ends that includes a symbolic execution tool allows us to answer our first hypothesis. The existing symbolic execution tool, Boogaloo [151] (see §2.3.4) could be used for this purpose. However performing this comparison does not answer our second hypothesis. To address both hypotheses we implement our own symbolic execution tool named Symbooglix, which we empirically optimise and then include in our comparison of Boogie IVL back-ends. The other Boogie IVL back-ends we compare against are the Boogie verifier [15], Corral [108], Duality [130], and GPUVerify [26] (see §2.3.4).

We evaluate the tools on two large benchmark suites. The first suite consists of 3749 C programs taken from the benchmark suite of the International Competition on Software Verification (SV-COMP) [55], translated to Boogie using the SMACK front-end [156] (see §2.3.3). The translated programs use mathematical integers to represent bit-vectors in the original C programs and have a large number of branch points. The second suite consists of 579 GPU kernels written in OpenCL and CUDA, translated to Boogie using the GPUVerify front-end [26] (see §2.3.3). These programs use bit-vector types and exhibit loops with large bounds.

Boogaloo does not support bit-vectors, so cannot handle the GPU benchmarks, and Symbooglix significantly outperforms Boogaloo on the SV-COMP suite. This supports our second hypothesis. Our results show that Symbooglix finds more bugs than GPUVerify in the GPU benchmarks, despite GPUVerify being highly optimised for this domain (albeit for verification not bug-finding). On the SV-COMP benchmarks, Symbooglix is generally less effective than Corral and Duality, but is complementary to them in terms of bug-finding ability (i.e. Symbooglix finds bugs that neither Corral nor Duality find and vice versa). This partially supports our first hypothesis.

Symbolic execution has already been implemented at the level of intermediate languages—notably KLEE [38], which operates at the level of the LLVM IR [109]. An industry-strength compiler IR such as LLVM's presents advantages and disadvantages for symbolic execution, stemming from the high precision with which language features and the target platform are taken into account (e.g. undefined behaviour, calling conventions, etc.). This precision en-

ables a symbolic execution engine to find low-level bugs, but also adds complexity, in terms of run time overhead, feature addition and code maintenance. In contrast, the syntactic and semantic simplicity of Boogie makes it an ideal platform on which to initially study the combination of symbolic execution with other analyses, and the design of new optimisations for more effective symbolic execution. In §3.4.6 we discuss problems that a make an apples-to-apples comparison between a symbolic execution execution tool for the Boogie IVL and KLEE difficult and then present a best effort comparison between Symbooglix and KLEE using the SV-COMP benchmarks. Because it is highly tuned towards C, KLEE is more efficient than Symbooglix and finds bugs in more programs, but Symbooglix is able to verify more programs than KLEE can, and finds a significant number of distinct bugs.

This chapter is structured as follows. First, we discuss the design and implementation of Symbooglix (§3.2). We then discuss optimisation of Symbooglix (§3.3). We took an *empirically-driven* approach to optimising Symbooglix: we selected a small training set from two large benchmark suites and used only this set to drive optimisation of the tool. This prevented us from over-fitting Symbooglix to our benchmarks, which might unfairly bias comparison with other tools and misrepresent how well Symbooglix would perform on further benchmarks. We then discuss our evaluation of Symbooglix, and five state-of-the-art Boogie IVL back-ends: the Boogie verifier, based on weakest pre-condition generation; Boogaloo, an existing symbolic execution tool; Corral, based on stratified inlining; Duality based on interpolation; and GPUVerify, a verifier for GPU kernels based on weakest pre-condition generation (§3.4). We then present a brief comparison of the Symbooglix and KLEE symbolic execution tools (§3.4.6). After this we note related work (§3.5) and conclude (§3.6).

3.2 Design and Implementation

We designed Symbooglix to be a reusable framework rather than a stand-alone tool, which enables it to be used as a program analysis method in existing (or yet to be created) Boogie-based projects. When designing Symbooglix we wanted to avoid "reinventing the wheel" as much as possible, so we reuse existing components (e.g. the parser and expression language) of the existing Boogie project from Microsoft Research [30]. As a consequence of using the existing Boogie project's libraries Symbooglix is written in C#.

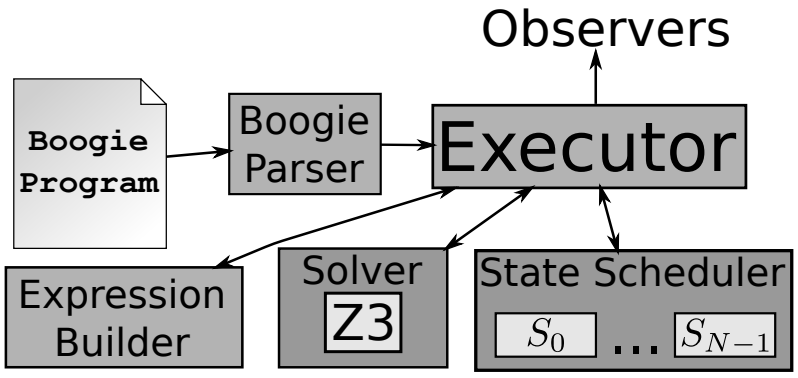


Figure 3.1: Architecture of Symbooglix.

Figure 3.1 shows a simplified view of the architecture of Symbooglix which illustrates several components.

The *Boogie Parser* is the parser of the Boogie project and is not part of Symbooglix. The figure illustrates a Boogie program being passed to the parser which generates the in memory representation of the Boogie program which is passed to the *Executor*.

The Executor executes the Boogie program and communicates with various components during execution. It asks for an *execution state* from the *state scheduler* and then executes the next command waiting to be executed in that "execution state". This process repeats until there are no more "execution state"s left to execute or a resource limit is reached.

The *Expression Builder* provides a clean interface to Boogie's expression language and optionally supports expression simplification (see §3.3) and constant caching. This is used to construct constraints that are added to an execution state's path constraint set.

The *Solver* component is used by the Executor to check the satisfiability of constraints. This component communicates with a constraint solver via the SMT-LIBv2 [20] text interface. Our current implementation supports the Z3 [140] and CVC4 [22] SMT solvers.

The *State Scheduler* stores execution states and decides how to schedule them.

Observers register themselves with the Executor in order to receive call backs when interesting events happens. For example a class that logs messages to the console to could subscribe to the state termination event so it can report to the user whenever a state is terminated.

We have implemented a tool that uses these components (named `sbx.exe`). For convenience we refer to this tool as Symbooglix.

3.2.1 Symbolic execution of the Boogie IVL

Symbooglix implements in-memory symbolic execution, where explored paths are stored explicitly as *execution states*. That is, for each explored path, Symbooglix keeps track of the execution state (program counter, stack and global variables) so that execution can be resumed at a later stage. As in previous execution systems [38], efficient sharing between states is achieved through a copy-on-write strategy (see *Efficient execution state cloning* in §3.3).

Symbooglix's operates on the unstructured representation of a Boogie program (i.e. a control flow graph). Execution proceeds as follows. First the executor runs several analysis passes over the input Boogie program to gather information useful to the executor. Next several programs transformation passes are run over the input Boogie program to simplify it (e.g. inline all functions, because of this Symbooglix does not support recursive functions) and put it in a canonical form. The initial execution state is then created based on a particular entry point (i.e. the implementation to start execution in).

The non-deterministic commands `goto` and `havoc`, not present in many languages, make Boogie a good match for symbolic execution. A `goto` command takes multiple target basic blocks and non-deterministically picks a target to which control transfers. This corresponds to the concept of forking new paths at a branch point in symbolic execution. However, there are two differences in the way forking new paths is implemented. First, in conventional languages, forking usually occurs at `if` statements, so that execution is forked into two paths, following the *then* and *else* sides of the branch. Even `switch` statements are typically compiled down into binary conditionals. In contrast, Symbooglix implements n-way forking to match the semantics of `goto` in Boogie. Second, branching at a `goto` is *unconditional* in Boogie; Boogie programs can exploit unconditional branching to model program behaviours abstractly. Traditional branching can be simulated via `assume` statements with mutually exclusive conditions (see lines 9 and 12 of figure 2.3), and Symbooglix is optimised for this case (see *Goto-assume look-ahead* in §3.3).

Recall that `havoc` is used to assign non-deterministic values to a set of variables. In the context of symbolic execution, this involves giving each variable a fresh symbolic value. Two other core Boogie commands that are central to symbolic execution are `assume` and `assert`. When Symbooglix interprets a command of the form `assume` e, it first asks its solver to check whether expression e is satisfiable in the current state. If so, e is conjoined to the current path condition

and execution continues. Otherwise, the path is terminated. To interpret an `assert` e command, Symbooglix checks both whether e and $\neg e$ are satisfiable in the current state (note it is possible for both to be satisfiable). If $\neg e$ is satisfiable, Symbooglix records that the assertion can fail and thus that the program under analysis is erroneous. Regardless of this, if e is satisfiable then execution continues with e conjoined to the path condition, so that analysis continues with respect to inputs that do not cause the assertion to fail.

The `requires` clause on program entry has the purpose of constraining the initial program state, thus it is treated like an `assume`. All other `requires` clauses are treated as assertions on procedure entry. Similarly, `ensures` clauses are treated as assertions on procedure exit. A specification-only procedure (which has no body) is executed by asserting its `requires` clause, havocking all variables in its `modifies` set (specifying which global variables might be updated), and assuming its `ensures` clause.

To handle *old expressions* (see §2.3.2) Symbooglix records for each stack frame the value of every used old expression in the `requires` and `ensures` clauses for the procedure corresponding to the stack frame.

3.2.2 Path exploration

An important aspect of symbolic execution is the order in which feasible paths are explored. This is typically controlled by search heuristics, in Symbooglix's case this is handled by the "state scheduler" component. In Symbooglix we use a variant of the depth-first search (DFS) strategy that aims to prevent the search getting stuck in loops by always preferring to follow the path leaving a loop if it is feasible. This variant behaves like a normal depth first search in all other aspects.

3.2.3 Constraint solving

Symbooglix's symbolic expressions are constructed using Boogie's expression building API. To support our work, we have contributed several related changes to the upstream Boogie project (including many bug fixes). The most important changes were the ability to make expressions

immutable and allow efficient structural equality testing between them. The former allows safe sharing of expressions across execution states, and the later is used by many optimisations in Symbooglix.

To answer satisfiability queries during exploration and obtain concrete solutions to the collected constraint sets, constraints are printed in the standard SMT-LIBv2 format [20], and then passed to an SMT solver. Our current implementation uses the Z3 constraint solver [140], due to its support for all the different features required by Symbooglix, such as integers, bit-vectors, quantifiers, maps and uninterpreted functions. Any other solver implementing the necessary features could be easily used if available. Using a text-based format is simple and portable between solvers, but substantially slower than directly interacting with a solver through an API due to text printing and parsing overhead; we plan to consider use of the Z3 API in future work.

3.2.4 Inconsistent assumptions

Boogie differs from conventional languages in that the entire program execution is subject to a set of initial constraints, specified via axioms, the `unique` qualifier on global constants, and `requires` clauses associated with the procedure from which execution commences (see §2.3.2). If the initial constraints are *inconsistent* (i.e. they are equivalent to `false`), the program is *trivially* correct. In our experience, inconsistency of initial constraints is often unintentional, and indicative of a problem with the Boogie program under consideration. To guard against this, Symbooglix supports an optional mode that checks the consistency of initial constraints before execution starts. If this mode is disabled then Symbooglix requires that the assumptions are consistent, otherwise its behaviour is undefined. To determine which assumptions are inconsistent, we used an optional feature of Symbooglix which checks if the assumptions are satisfiable using an SMT solver. If the assumptions are not satisfiable, the *unsat-core* feature of the SMT solver is used to determine exactly which assumptions are in conflict.

3.3 Optimisations

Our initial design of Symbooglix included only a few basic optimisations which we implemented without benchmarking the tool. We then optimised Symbooglix in an empirically-driven manner, guided by performance on a set of benchmarks. Because we wanted to com-

pare Symbooglix with other Boogie analysis tools and wished to understand the extent to which Symbooglix's optimisations would be generally applicable, we were cautious not to overfit Symbooglix's optimisations to the benchmarks used in our experimental evaluation. To avoid this, we randomly selected a *training set* consisting of 10% of our benchmarks, and benchmarked Symbooglix exclusively with respect to this training set during optimisation (see §3.4.4). Below, we summarise the main optimisations we implemented as a direct result of training.

Unique global constants constraint representation. This optimisation was motivated by performance problems we observed when running on the SV-COMP portion of the training set, which declares global variables with the `unique` qualifier (see §2.3.2). Our initial approach to handling `unique`, by emitting a quadratic number of constraints to assert pairwise disjointness, did not scale well. To improve performance, we took advantage of the SMT-LIBv2 `distinct` function, which returns true iff all its arguments are pairwise distinct and is efficiently handled by Z3.

Global dead declaration elimination. We observed that benchmarks in the SV-COMP training set often declare many global variables, functions and axioms that are not used by the program. These *dead declarations* are emitted by SMACK for every Boogie program it generates, e.g. for SMACK's general-purpose floating point representation and memory model. We implemented an analysis that initially marks a global declaration as *necessary* if it is used syntactically by a procedure in the program, and then iteratively marks further declarations as *necessary* if they are referred to by a declaration already marked as *necessary*. Once a fixed point is reached, all declarations not marked as *necessary* are removed.

Goto-assume look-ahead. We developed this optimisation based on intuition related to symbolic execution of Boogie programs. Recall from the example of §2.3.2 that conditional control flow is realised in a Boogie program through a combination of `goto` and `assume` commands. The initial implementation of Symbooglix would always fork at a `goto` command. However, the current path constraints often mean that one of the `assume` statements targeted by the `goto` will have a failing guard. (This is a known issue in symbolic execution: prior work has reported that often fewer than 20% of symbolic branches encountered during execution have both sides feasible [40].) As a result, Symbooglix would often spawn a new execution state, only to kill it at the next `assume` instruction. The *goto-assume look ahead* optimisation changes how the `goto`

command is handled by looking ahead at the next instruction of each target basic block. If the next instruction is an `assume`, we check whether it is satisfiable, and if not, we do not fork a new state for that path.

In effect, one can think of this as Symbooglix looking for a `goto` followed by an `assume` and treating it like a branch instruction from a conventional non-deterministic language.

A small complication identified by the SV-COMP training set was that SMACK often adds `assume true` statements at various locations to communicate debug information (as attributes on the instruction) for the original C program, which could interfere with our optimisation. We wrote a simple transformation to remove these trivial `assume`s.

The remaining optimisations are similar to the ones implemented by existing symbolic execution engines:

Expression simplification. These optimisations simplify expressions as they are constructed, by folding constants (e.g., `5+4 = 9`) and rewriting certain expression patterns (e.g., `1+x+2 = 3+x`). The patterns we simplify are based on potential simplifications we observed when running Symbooglix on the training set, and on some of the patterns used by KLEE. To help ensure preservation of the exact semantics of Boogie's operators during simplification, we used the Z3 SMT solver to verify correctness of many of the rewriting patterns. For bit-vector types, we checked the simplification with respect to a single, representative bit width only. One example of subtle semantics is that the `div` and `mod` operators use Euclidean division. Because our implementation language, C#, uses truncated division we had to implement Euclidean division in terms of truncated division (following [114]) to provide constant folding for `div` and `mod`.

Constraint independence. This constraint solving optimisation, inspired by EXE [39], eliminates those constraints which are not necessary to determine the satisfiability of a given query. The optimisation transitively computes dependent constraints by considering the set of used variables and uninterpreted functions until a fixed point is reached.

Map updates at concrete indices. This optimisation was inspired by the way in which KLEE handles array accesses at concrete indices. In Symbooglix the value of a map variable was originally represented as an expression tree. Initially the expression is just the symbolic variable representing the map (e.g. `m`). As the map is populated, map updates are added to this expression. To illustrate this consider the Boogie program shown in Listing 3.1.

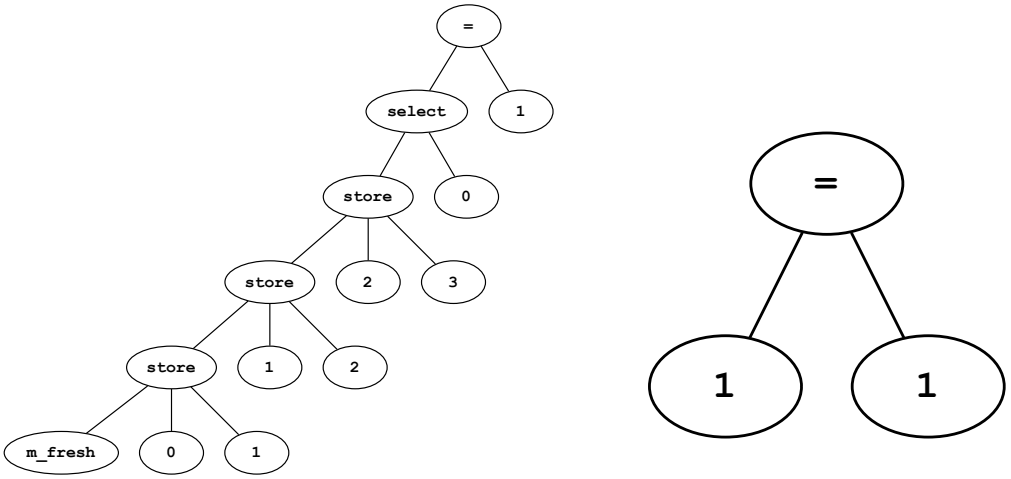
Listing 3.1: An example Boogie program illustrating performing several maps stores and then a read at concrete indices.

```
1 procedure main() {
2   var m:[int]int;
3   m[0] := 1;
4   m[1] := 2;
5   m[2] := 3;
6   assert m[0] == 1;
7 }
```

First on line 2 a map named `m` is declared. At this point the program the expression representing the map is just a fresh symbolic variable which we'll call `m_fresh`. Next on line 3 a map store of value `1` to index `0` is performed. At this point in the program the expression representing `m` is a `store` expression writing `1` at index `0` to `m_fresh`. Several more map stores at concrete indices occur and then an `assert` is performed. In the assert a `select` operation is performed which reads from index `0` on the expression currently representing the map `m`. The expression that the constraint solver would be asked check satisfiability for is shown in Figure 3.2a. This expression is not optimal. Notice that there is a chain of `store` operations at concrete indices, followed by a `select` operation at a concrete index that was stored lower down the expression tree. By visual inspection we can see that performing `select` on index `0` on the child expression should return `1`. Given this information a much smaller expression could be sent to the constraint solver which is shown in Figure 3.2b. This expression would normally be constant folded, but for clarity we don't show this. However our initial implementation of Symbooglix didn't try to optimise this case and so would pass the unoptimised version of the constraint to the underlying constraint solver. For this small example it is not a problem but if a large number of map stores were performed to the map (in a loop for example) then the constraint sent to the underlying constraint solver would be very large which would cause Symbooglix to exhibit very poor performance.

To address this, we optimised for the case where only concrete indices are used to index a map. In this case, rather than updating the expression tree representing the map with new `store` nodes, we store in a separate data structure a set of pairs mapping map stores at concrete indices to their corresponding value. When reading from a location at a concrete index that

was previously written, the corresponding value is returned directly. In the case of a read from a concrete index not contained in the set, a map select expression that reads from the current version of the expression tree (without the current set of concrete map stores) at that concrete index is returned. In the case of a map select/store at a symbolic index, Symbooglix switches to using an expression tree to represent the map by flushing all the concrete map stores to the expression tree.



(a) Unoptimised (b) Optimised

Figure 3.2: Constraint sent to constraint solver for `assert` command in Listing 3.1.

Map updates at symbolic non-aliasing indices. This optimisation was motivated by several benchmarks where symbolic indices were used to index maps, but such that the associated indices could not alias. In the examples we investigated, the indices were always of the form `C + s`, with `C` a constant mathematical integer, distinct for each index, and `s` a symbolic mathematical integer variable, common among the indices. Clearly for constants $C_0 \neq C_1$, we are guaranteed to have $(C_0 + s) \neq (C_1 + s)$.

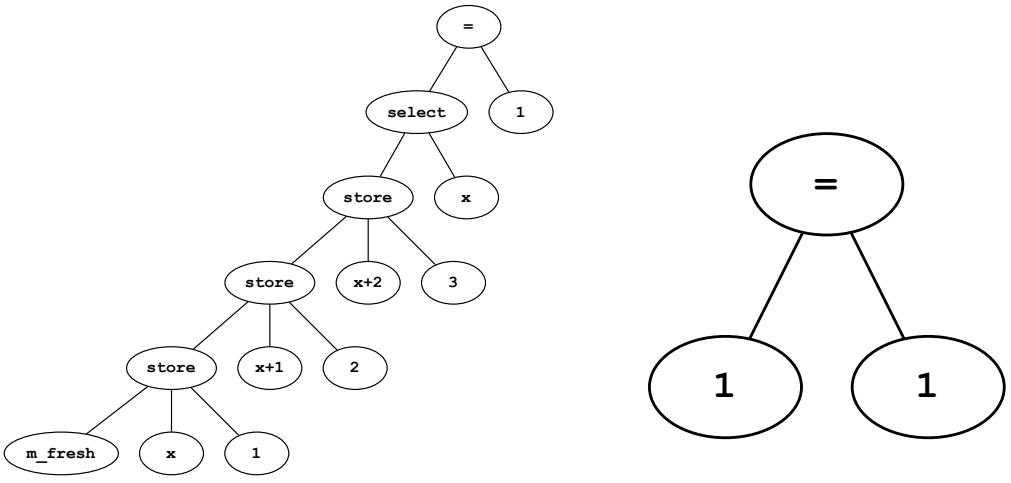
To illustrate this consider the Boogie program in Listing 3.2.

Listing 3.2: An example Boogie program illustrating performing several maps writes and then a read at concrete indices.

```
1 procedure main() {
2   var m:[int]int;
3   var i:int;
4   m[i] := 1;
5   m[i+1] := 2;
6   m[i+2] := 3;
7   assert m[i] == 1;
8 }
```

On line 2 a map `m` is declared and then on line 3 a symbolic integer `i` is declared. Several stores are then performed at symbolic indices with `i` used as a base for the index. Note that the indices are guaranteed to not alias. Then on line 7 an `assert` is performed that performs a map select. Figure 3.3a shows the expression that would be sent to the constraint solver. This expression is not optimal. Notice that there is a chain of `store` operations at symbolic indices, none of which alias. This means that the `select` operation at index `x` would return `1`. Given this information a much smaller expression could be sent to the constraint solver which is shown in Figure 3.3b.

This lead us to generalise the *map updates at concrete indices* optimisation to store a mapping of non-aliasing (concrete or symbolic) indices to expressions in a separate data structure (rather than constant indices to expressions). Expression aliasing is determined by simple syntactic patterns; we currently recognise the case of distinct constant literals (capturing the initial optimisation), and the set of expressions matching the pattern `C + s`, where C is distinct in each expression and where the types of `C` and `s` are mathematical integers. This optimisation is currently implemented only for integers, and not bit-vectors.



(a) Unoptimised (b) Optimised

Figure 3.3: Constraint sent to constraint solver for `assert` command in Listing 3.2.

Efficient execution state cloning. This optimisation was inspired by how KLEE handles cloning of execution states, and was motivated by high memory usage in Symbooglix on the training set. The number of execution states can grow quickly during symbolic execution, so efficient state cloning, both in terms of memory used and time taken, is important. Symbooglix originally cloned states in a simple, non-optimal manner. Because expressions are immutable in Symbooglix they never need to be cloned, but the data structures that a state uses to refer to

them (e.g. a dictionary mapping global variables to expressions) do. The initial implementation simply made new copies of these data structures. However, profiling several memory-intensive training set benchmarks revealed that a lot of memory was being used by these data structures. To overcome this, we implemented more efficient execution state cloning using C# immutable data structures from the C# `System.Collections.Immutable` library. For our internal representation of maps, which are not immutable, we added a simple copy-on-write mechanism similar to that which KLEE uses to represent memory objects.

3.4 Evaluation

We now present in detail our method for evaluating Symbooglix and a selection of other Boogie-based tools. For reproducibility, the tools and all non-commercially sensitive benchmark programs are made available online at `http://symbooglix.github.io`.

We first discuss the benchmark suites used for the evaluation in §3.4.1 and then in §3.4.2 the steps that were performed to prepare them. Next in §3.4.3 we discuss the tools that Symbooglix is compared against and how they were configured. In §3.4.4 we discuss the methodical manner in which Symbooglix was optimised in preparation for evaluating it against other tools. We then discuss this evaluation in §3.4.5. Finally in §3.4.6 we perform a brief comparison of Symbooglix against another symbolic execution engine – KLEE.

3.4.1 Benchmark suites

We consider two benchmark suites containing Boogie programs from two distinct problem domains, and originating from two different languages.

The *SV-COMP benchmark suite* (abbreviated to SV-COMP) consists of programs generated from the C benchmarks used in the 2015 "International Competition on Software Verification" (SV-COMP 2015), translated into Boogie using SMACK [156]. We used the SMACK-translated programs made available online[1] by the SMACK authors. The repository contains 3760 benchmarks, of which we use 3749: four benchmarks exhibit inconsistent assumptions (see §3.2) and seven are empty[2].

[1] `https://github.com/smackers/sbb`
[2] `https://github.com/smackers/sbb/issues/1`

The *GPU benchmark suite* (abbreviated to GPU) consists of Boogie programs generated from a set of 579 GPU kernels written in OpenCL and CUDA, which have been collected to evaluate the GPUVerify tool [26, 14]. The original kernels are drawn from a number of open-source GPU benchmark suites and one commercial suite. Among these kernels, at least 32 exhibit data race errors (which manifest as failing assertions in the Boogie programs generated by GPUVerify): 5 are genuine bugs previously found by GPUVerify, and 27 are artificial bugs injected in a previous evaluation of GPUVerify [26].

We regard a benchmark as buggy if the Boogie program can exhibit an assertion failure (includes `requires` and `ensures` failures). Typically such an assertion failure corresponds to a bug in the original C program or GPU kernel, but this is not always the case, e.g. due to the use of over-approximating abstractions during the translation into Boogie, or due to bugs in the translation tools. Focusing on finding assertion failures in the Boogie programs, regardless of how those programs were generated, provides a fair basis on which to compare the Boogie analysers that we evaluate.

The SV-COMP and GPU suites provide a comprehensive and challenging set of evaluation benchmarks, covering correct and buggy examples. The SV-COMP suite utilises mathematical integers while the GPU suite utilises bit-vector operations.

Running program analysis tools on these benchmarks allowed us to examine and compare several important performance characteristics of these tools. These characteristics are the number of true negatives, true positives, false negatives, false positives, and inconclusive results (along with the reason); and the tool's run time.

3.4.2 Benchmark preparation

Checking for inconsistent assumptions. As discussed in §3.2, Symbooglix expects the initial set of assumptions of a Boogie program to be consistent if its optional checking mode is disabled. In our evaluation we decided not to use the checking mode because no other tool performs this check and thus Symbooglix would be unfairly penalised in terms of tool run time.

We were not expecting to find programs with inconsistent assumptions, but we found four cases in each suite. In SV-COMP we discovered global constants marked as `unique`, requiring their values to be distinct, together with axioms constraining the constants to have the same value. We removed these benchmarks and reported the issue[3] to the SMACK authors.

In GPU we discovered that three of the programs had inconsistent `requires` clauses which had gone unnoticed for several years; we applied obvious fixes to these clauses and retained the benchmarks. The other example in GPU, arose from a single-threaded kernel on which race checking was performed. GPUVerify's front-end generates inconsistent axioms for this example by design, because a single-threaded kernel is vacuously race-free.

Labelling benchmarks. To compare Symbooglix with competing tools in terms of bug-finding ability and capability to perform exhaustive verification, we tagged each benchmark with one of the following labels:

1. *Correct:* The benchmark is free from bugs
2. *Incorrect:* The benchmark contains at least one bug
3. *Unknown:* The benchmark may or may not contain bugs

To infer as many *correct* and *incorrect* labels as possible, we devised the following experiment. For each benchmark, we ran each compatible Boogie tool introduced in §2.3.4 (in multiple configurations, as detailed in §3.4.3), with a timeout of 900 seconds and a memory limit of 10 GiB. We did not run Symbooglix at this point, because initially we wanted to use the labels to select a training set for Symbooglix and evaluate its progress over time (see §3.4.4). If one tool classified a program as *correct* and another tool classified the program as *incorrect*, we investigated the reasons for this, knowing that one tool must be wrong in its analysis. Otherwise, if at least one tool classified a program as *correct* we labelled the program *correct*, while if at least one tool classified a program as *incorrect* we labelled the program *incorrect*. Because Boogie and GPUVerify can produce false positive bug reports, we ignored cases where these tools classified a program as *incorrect* during the labelling process. We labelled a benchmark *unknown* if no tool could reliably classify the program as *correct* or *incorrect*, except in cases where an existing label indicating the program's status was provided by the benchmark suite.

[3] `https://github.com/smackers/smack/issues/71`

We found eight benchmarks for which Boogaloo reported a bug and at least one other tool reported successful verification. We traced this to a bug in Boogaloo: the tool did not support builtin function attributes, treating functions equipped with these attributes as uninterpreted functions rather than mapping them directly to specific SMT-LIB operations. We reported this to the Boogaloo developers and they provided a fix which we used for all future experiments.

During the labelling process, we checked for generic failures and crashes in the tools. This revealed one bug[4] shared by Boogie and Corral, four bugs in Corral (including a case where Corral would report a false positive), and a crash bug[5] in Duality. We reported these bugs and in some cases provided our own fixes. The Corral bugs were promptly fixed.

The SV-COMP suite already provides labels for its constituent benchmarks. However, we found 76 discrepancies between these existing labels and the ones we inferred. Because the labels are for the original C programs in SV-COMP, the mismatch could be caused by an incorrect translation from C to Boogie by SMACK, or it could be a genuine mislabelling of the original benchmark. We did not examine all 76 cases, but for the ones we did examine, the discrepancy was caused by the translation process. For example, two of the benchmarks check whether a self-equality comparison on an arbitrary floating point variable can fail. The assertion in the original C program could fail because `NaN == NaN` is false[6]. However, the corresponding assertion in the Boogie program could not fail because `NaN` was not represented in SMACK's model of floating point numbers. We reported this issue[7] to the SMACK developers.

For the GPU suite, we labelled the benchmarks that contained deliberately injected bugs as *incorrect*. We surprisingly found mismatches between two kernels that were supposed to have injected bugs and the results reported by the tools applied to those kernels. In one of the kernels it turned out that the injected code did not actually induce a bug so this kernel was removed from our benchmark suite. The other kernel had a mistake (introduced by an error in the injection process) causing the injected bug to be unreachable. We fixed this kernel in our benchmark suite so that the injected bug was reachable.

The *Initial* columns in table 3.1 summarise our labelling without Symbooglix. At the end of our study, having optimised Symbooglix, we were able to re-label the benchmarks based on additional accuracy from Symbooglix's results. The *Final* columns in table 3.1 reflect this labelling.

[4] `http://boogie.codeplex.com/workitem/10246`
[5] `http://corral.codeplex.com/workitem/1`
[6] NaN stands for "Not a Number". Floating-point comparisons always return false if one of the operands is NaN
[7] `https://github.com/smackers/smack/issues/66`

Note that the number of *Correct* labels decreases in the *Final* labelling because the labels provided with some SV-COMP programs were incorrect with respect to the Boogie programs (as discussed earlier).

Table 3.1: Initial and final benchmark labellings.

	SV-COMP			GPU		
	Initial	**Final**	**Training**	**Initial**	**Final**	**Training**
Correct	2705	2704	270	479	491	45
Incorrect	1044	1045	104	37	38	3
Unknown	0	0	0	63	50	9
Overall	3749	3749	374	579	579	57

3.4.3 Tools evaluated

We compare Symbooglix with all actively maintained open-source tools that analyse Boogie programs: the Boogie verifier, Boogaloo, Corral, Duality, and GPUVerify (see §2.3.4).

All these tools use Z3 for constraint solving. In our experiments we used Z3 v4.3.1 with Symbooglix and Boogaloo, and Z3 v4.3.2 with the remaining tools. This discrepancy is due to Boogaloo's dependence, at time of running experiments, on Z3 v4.3.1, and due to time constraints preventing us from re-running Symbooglix using Z3 v4.3.2.

We did not run Boogaloo and Duality on the GPU suite because they do not support the bit-vector types and operations generated by the front end of GPUVerify. We could not apply GPUVerify to SV-COMP, because GPUVerify only supports analysis of Boogie programs generated by its own front-end. The configuration used for each tool was as follows:

Boogaloo was run twice on SV-COMP: once with a loop bound of 8 and once without a bound. Concretisation was disabled to avoid false negatives and counter-example minimisation was disabled to increase performance.

Boogie was run with the `-errorLimit:1` option, so that at most one error is generated.

Corral was run twice on each suite: once with a recursion bound of 8 and once with a very large bound ($\sim 2^{30}$) on SV-COMP; and once with a bound of 64 and once with a very large bound ($\sim 2^{30}$) on GPU. We picked a bound of 8 for SV-COMP because this was used by the authors of SMACK for their SMACK+Corral SV-COMP 2015 submission [87]. The larger bound of 64 for GPU was chosen because our experience with these benchmarks indicated that loops

with large iteration counts are common (a consequence of the throughput-oriented nature of GPU applications). The very large bound ($\sim 2^{30}$) is approximately half the largest integer that Corral supports for specifying the bound; the Corral authors advised against using the largest integer due to potential overflow bugs in Corral.

Duality was run using the same large bound as used for Corral ($\sim 2^{30}$); we used the same bound since Duality is built on top of Corral. We did not consider a smaller bound because the interpolation-based analysis used by Duality depends upon the ability to unwind a program to a significant depth.

GPUVerify was run on the GPU suite with automatic invariant inference enabled. We disabled extra invariants with which the benchmarks had been manually annotated, so that GPUVerify ran in an unassisted manner.

Symbooglix was run using its default settings, except that the checking of inconsistent assumptions was disabled (see §3.4.2) and the timeout per solver query was set to 30 seconds. Having a solver timeout prevents Symbooglix getting stuck checking the feasibility of a particular path but may prevent full exploration of the benchmark.

Each tool was allowed a maximum execution time of 900 seconds per benchmark (which is the time used at the last edition of SV-COMP, except that we use wall clock time instead of CPU time). A run of a tool on a single benchmark consists of two pieces of information, the result type and the execution time. The former is the answer the tool gives—*bug found, verified* (i.e. no bug found, no tool crash, and no bound, memory limit, solver or global timeout reached), or *unknown* (i.e. no bug found and not *verified*). In the case of Symbooglix and Boogaloo *verified* is equivalent to exploring all feasible paths and finding no bugs.

Each tool was executed three times on the same benchmark, and these runs were combined using the following approach. For results types, if at least one run reports *verified* or *bug found,* we take that result (we initially observed conflicting result types due to tool bugs, but these disappeared once the bugs were fixed). Otherwise, if all runs result in *unknown,* the overall result is *unknown*. The repeat runs of tool is to handle non-determinism. There are several sources of non-determinism. First, for the large tool comparison, the nodes used on the compute cluster were not guaranteed to have identical hardware. Second, there could be

some non-determinism in the tools themselves (although we did not observe this). Finally, if a tool reports a result very close to the timeout, in repeat runs this may fluctuate between a result and a timeout due to small fluctuations in timing due to the operating system scheduler.

To combine the execution times, we treat any of the three results that were of type *unknown* as having taken the maximum allowable time (i.e. 900 seconds). We then compute the arithmetic mean of these times. The rationale here is to penalise tools that terminate abnormally (e.g. crash) after a short amount of time.

All tools except Boogaloo are written in C#. To run them on our Linux machine, we used Mono `3.12.1`, with a minor patch[8] to fix crashes we were experiencing.

3.4.4 Evaluation of Empirically-Driven Optimisations

As discussed in §3.3, we took an empirically-driven approach to optimising Symbooglix, incrementally optimising the tool guided by a *training* set. Doing this work is necessary because, in order to answer hypothesis 2 (see §3.1) we need to make Symbooglix as performant as possible, if we are to improve on the state-of-the-art. The comparison of the Boogie tools in §3.4.5 uses the version of Symbooglix that results from this optimisation work.

The training set was obtained by taking the prepared benchmarks (see §3.4.2) and randomly selecting 10% of each label for both benchmark suites. The number of benchmarks used for our training set broken down by label can be seen in the *Training* columns of Table 3.1, totalling 374 benchmarks from the SV-COMP suite and 57 benchmarks from the GPU suite. The size of the GPU training set is not exactly 10% of the initial benchmark labelling because the training set was selected based on results from an early run of the tools in which the tools were not run optimally. This led to fewer benchmarks being labelled *Correct* and more benchmarks being labelled *Unknown*.

At various intervals during Symbooglix's optimisation we stopped development and ran that version of Symbooglix on the training set. We refer to these versions of Symbooglix as *snapshots*. We monitored our progress by comparing the performance of the latest snapshot to

[8] `https://github.com/mono/mono/pull/1649`

previous snapshots. The following list details the eleven snapshots, giving each a short name and indicating the order in which the optimisations of §3.3 were added. Due to the nature of our development, the optimisations are applied cumulatively.

1. **Baseline:** the starting point for our optimisation work; incorporates *unique global constants constraint representation*
2. **GlobalDDE:** adds *global dead declaration elimination*.
3. **GotoAssumeLA:** adds *goto-assume look-ahead*.
4. **ExprSimpl:** adds *expression simplification*.
5. **ConstrIndep:** adds *constraint independence*.
6. **RemSomeRecur:** improves an algorithm that searches expressions for symbolic variables and uninterpreted functions by making it iterative (rather than recursive) and caching results; adds further expression simplification rules; adapts stack size to avoid overflow errors.
7. **RemSomeDbg:** removes a data structure used for debugging that was accidentally left behind. We discovered this after profiling the memory usage of Symbooglix.
8. **MapConstIdx:** adds *map updates at concrete indices*.
9. **MapSymIdx:** adds *map updates at symbolic non-aliasing indices*.
10. **EffcntClone:** adds *Efficient execution state cloning*.
11. **SmplSolv:** optimises the solver interface to assess whether the expression to be checked for satisfiability is constant or already in the constraint set.

Experimental setup. To assess the progress of our optimisation effort, we ran each snapshot on the training set on a single machine with an eight core Intel® Xeon CPU (3.3GHz) with 48GiB of RAM running Linux. We used the process described in §3.4.3 to run Symbooglix, enforcing a 5GiB memory limit per benchmark.

To visualise the progress of Symbooglix over time we use *quantile function* plots as used in SV-COMP [55]. In Figure 3.4 the top and bottom plots show results for the eleven snapshots for the SV-COMP and GPU training sets, respectively. Each curve represents a run of Symbooglix on a particular snapshot. We compute a score for each snapshot by adding one point for each benchmark that the snapshot accurately classifies as *correct* or *incorrect* and subtracting one point for inaccurate classifications. For classification we used the *initial* labelling of §3.4.2, but updated this labelling as Symbooglix managed to classify additional benchmarks. Each point denotes a benchmark that was correctly classified as *correct* or *incorrect*. The y-coordinate is the time taken to analyse the benchmark, and for each curve, benchmarks are sorted based on time. The x coordinate represents the accumulated score for the snapshot. Thus a point

(x, y) shows that analysis takes y seconds or fewer for the previous x benchmarks plotted. The y-axis uses a linear scale between 0 and 1, and a log scale thereafter. This prevents times close to 0 from making the range of the y-axis excessively large. The ordering of the rightmost data point on the axis ranks the snapshots in terms of classification ability, the width of the curve along the x-axis is the number of correct classifications, and the area under a plot represents the total execution time for the correctly-classified benchmarks in that snapshot.

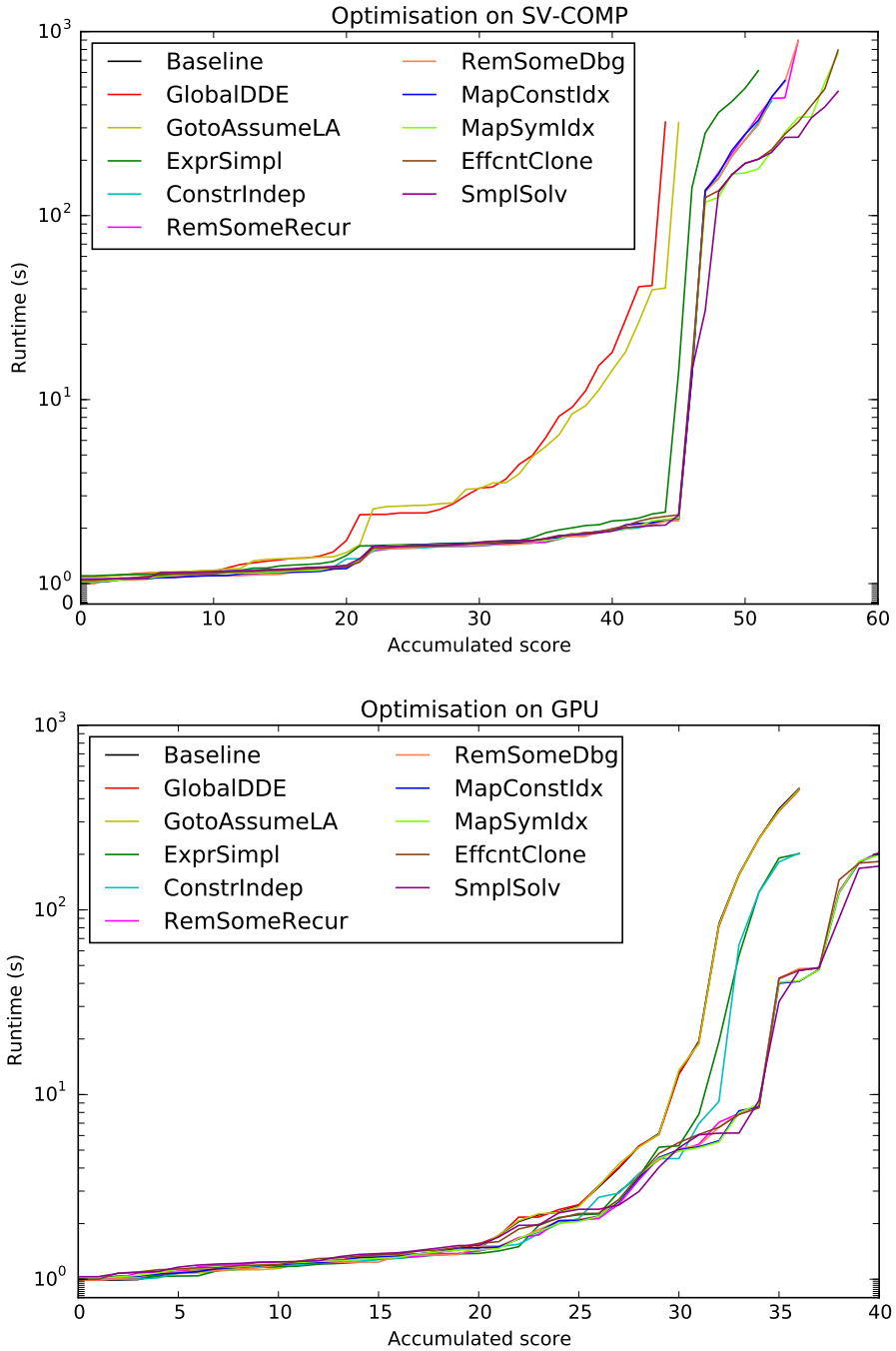


Figure 3.4: Quantile function plot for Symbooglix running on the training sets for SV-COMP (top) and GPU (bottom) at different snapshots. Due to the large number of snapshots, the plot is designed to be viewed in colour. The maximum possible accumulative scores are 374 (SV-COMP) and 57 (GPU).

Snapshot results for the SV-COMP training set. The top plot in Figure 3.4 shows the quantile plot for SV-COMP. Most snapshots bring improvements to either the number of correctly classified benchmarks or the overall run time (or both). The most substantial impact is made by the *GlobalDDE* and *ExprSimpl* optimisations. Note that the *Baseline* is not visible on the plot because the SV-COMP benchmarks contain many unused universally-quantified axioms over uninterpreted functions. In *Baseline* these are all passed to the constraint solver, causing all benchmarks to time out. The *GlobalDDE* snapshot eliminates these unused axioms, allowing Symbooglix to make progress. The other high-impact optimisation is *ExprSimpl*, which allows six additional benchmarks to be correctly classified, and also brings a significant improvement in running time. Finally the *MapSymIdx* optimisation allows four additional benchmarks to be classified.

Snapshot results for the GPU training set. The bottom plot in Figure 3.4 shows the quantile plot for the GPU suite. *Baseline* is present in this plot because the GPU suite does not use quantified axioms, allowing Symbooglix to make progress from the beginning. As in SV-COMP, *ExprSimpl* improves performance (though here does not classify additional benchmarks). *RemSomeRecur* leads to a significant performance gain and four additional correctly-classified benchmarks. Several optimisations do not make any difference on the GPU suite: *GotoAssumeLA* (due to the very limited amount of forking that occurs in the GPU benchmarks), *MapConstIdx* (GPUVerify's front-end employs a symbolic representation of thread ids, meaning that maps are rarely indexed concretely), and *MapSymIdx* (because the optimisation does not currently support bit-vectors).

Snapshot results for the entire SV-COMP and GPU suites are included on the project website `https://symbooglix.github.io`.

3.4.5 Comparison of Boogie back-ends

In this section we compare the tools discussed in §3.4.3 on the benchmarks discussed in §3.4.1. The goal of this work is to answer hypotheses 1 and 2 as discussed in §3.1.

Experimental setup. We ran the comparison of the tools on a large general purpose computing cluster[9] with 20-core Intel® Ivybridge CPU nodes, each with 128 GiB RAM running Linux. We used the approach discussed in §3.4.3 to run each tool, and enforced a memory limit of 10 GiB per benchmark. Our timing results are prone to fluctuations due to hardware differences between nodes; we in part account for this by reporting averages over three independent runs.

Results table. Table 3.2 shows the extent to which the tool configurations we compare were able to verify or find bugs in the SV-COMP and GPU suites. For Boogaloo and Corral, the 8, 64 and NB suffixes indicate whether the tools were invoked with a bound of 8, 64, or with no bound (for Corral, NB actually means the huge bound of $\sim 2^{30}$). The *Verified* and *Bug found* columns indicate, for each tool, the number of benchmarks labelled *correct* and *incorrect*, respectively, that the tool could accurately classify as such. *False alarms* identifies cases where a tool reports a *correct* benchmark as *incorrect*. *Unknown* captures timeouts, memory exhaustion and crashes. As expected, only the Boogie verifier and GPUVerify report false alarms, and no tool reported a false negative (classifying an *incorrect* benchmark as *correct*).

Table 3.2: Results for Boogie analysis tools applied to the SV-COMP and GPU suites, using final classification labels.

SV-COMP suite				
Tool	**Verified**	**Bug found**	**False alarm**	**Unknown**
Boogie	0	1021	2668	60
Boogaloo-8	43	122	0	3597
Boogaloo-NB	64	122	0	3563
Corral-8	1348	541	0	1860
Corral-NB	1365	553	0	1831
Duality	1856	426	0	1467
Symbooglix	236	395	0	3118
GPU suite				
Tool	**Verified**	**Bug found**	**False alarm**	**Unknown**
Boogie	260	35	165	119
Corral-64	298	28	0	253
Corral-NB	297	28	0	254
GPUVerify	403	34	76	66
Symbooglix	303	35	0	241

Comparison of Symbooglix and Boogaloo. Table 3.2 shows that for the SV-COMP benchmarks, Boogaloo-NB is more effective than Boogaloo-8. Symbooglix verifies more benchmarks than Boogaloo-NB: 236 vs. 64. The tools verify 58 common benchmarks, with Symbooglix verifying 178 benchmarks for which Boogaloo-NB reports *unknown*, and Boogaloo-NB verifying 6

[9] `http://www.imperial.ac.uk/admin-services/ict/self-service/research-support/hpc/hpc-service-support/service/`

benchmarks for which Symbooglix reports *unknown*. Symbooglix was also able to find more bugs than Boogaloo-NB: 395 vs. 122. The tools find bugs in 107 common benchmarks, with Symbooglix finding bugs in 288 benchmarks for which Boogaloo-NB reported *unknown*, and Boogaloo-NB finding bugs in 15 benchmarks for which Symbooglix reports *unknown*. Recall that Boogaloo cannot be applied to the GPU suite because it does not support bit-vectors. This shows that Symbooglix has improved on the state-of-the-art of symbolic execution for the Boogie IVL, given that Boogaloo is the only other symbolic execution tool for the Boogie IVL. These results support hypothesis 2 (see §3.1).

Comparison over the SV-COMP suite. Comparing all the tools applied to the SV-COMP suite, Table 3.2 shows that Corral and Duality are the clear winners, with Corral-NB performing best in terms of bug-finding ability, and Duality proving most capable at verifying benchmarks. It is not surprising that Symbooglix is less effective at verification than these tools, since symbolic execution is primarily geared towards finding bugs, and suffers from path explosion on bug-free programs. In terms of bug-finding ability, Symbooglix is some way behind Corral-NB, finding bugs in 395 vs. 553 benchmarks.

To assess whether Symbooglix and Corral-NB have *complementary* bug-finding capabilities, we compared times taken for these tools to find bugs for all SV-COMP benchmarks labelled *incorrect*. The comparison is visualised by the scatter plot of Figure 3.5. A point at (x, y) indicates that for a given *incorrect* benchmark, Corral-NB and Symbooglix took x and y seconds, respectively, to find the bug. Cases where the tools reported *unknown* are treated as reaching the 900 second timeout limit. Points above the diagonal indicate that Corral-NB outperformed Symbooglix (295 cases), points below the diagonal indicate that Symbooglix outperformed Corral (309 cases). Both tools reported *unknown* for 441 benchmarks (these points lie at the top right corner of the plot). The shape of the plot clearly shows that the tools have complementary abilities when it comes to bug-finding: the tools find bugs in 344 common benchmarks, but in 51 cases Symbooglix finds a bug where Corral-NB does not (the points lying on the far right vertical) and in 209 cases Corral-NB finds a bug where Symbooglix does not (the points lying on the top horizontal). The large number of points lying close to the x-axis indicate cases where Symbooglix finds a bug within a matter of seconds, but where the time taken by Corral varies dramatically. For 70 benchmarks where Corral-NB takes more than 100s to find a bug,

Symbooglix finds a bug within 10s, and there are no benchmarks for which the reverse is true. An analogous plot comparing Symbooglix with Duality, presented on our project website[10], shows a very similar picture.

Note that our choice of a timeout of 900 seconds does not affect our conclusions. If a smaller timeout had been chosen that some points along the axes would move to the top right corner but the same trend of complementarity would be observed. If the timeout was increased the trend of complementarity would continue. All points that are currently not timeouts would remain where they are and the others points would either remain as timeouts as the timeout was increased or would move to be positioned along one of the axes.

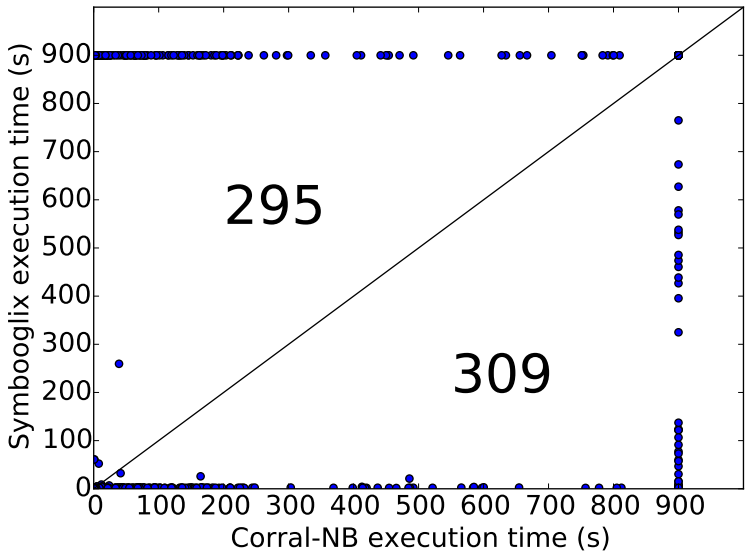


Figure 3.5: Comparison of bug-finding times on the SV-COMP suite for Symbooglix and Corral-NB.

Comparison over the GPU suite. Table 3.2 shows that Symbooglix finds the most bugs in GPU among the tools that do not report false alarms: 35 bugs compared with 28 bugs found by both Corral-NB and Corral-64 (the same bugs are identified by each Corral configuration). Furthermore, Symbooglix finds a superset of the 28 bugs found by Corral. The Boogie verifier also find 35 bugs, but with a high false alarm rate (165 alarms); GPUVerify finds 34 bugs, with a lower false alarm rate (76 alarms). Comparing GPUVerify and Symbooglix further, both tools find bugs in 31 GPU benchmarks, with Symbooglix finding 4 bugs not found by GPUVerify, and GPUVerify finding 3 not found by Symbooglix.

[10] `https://symbooglix.github.io`

Symbooglix is able to verify slightly more benchmarks than Corral-64: 303 vs. 298, and the tools are highly complementary at verification: each tool managed to verify 225 common benchmarks, with Symbooglix verifying 78 benchmarks where Corral-64 reports *unknown*, and Corral-64 verifying 73 benchmarks where Symbooglix reports *unknown*. As expected, since it was designed for this purpose, GPUVerify is able to verify the largest number of GPU benchmarks: 403. Symbooglix and GPUVerify can verify 258 common benchmarks, with Symbooglix able to verify 45 benchmarks where GPUVerify reports *unknown*, and GPUVerify able to verify 145 benchmarks where Symbooglix reports *unknown*.

Our results show a stronger performance from Symbooglix on the GPU suite compared with the SV-COMP suite. We attribute this to the fact that the benchmark suite makes no use of quantifiers (in translating OpenCL and CUDA kernels to Boogie, the GPUVerify front end uses domain-specific strategies for avoiding quantifiers [14]), and to a process of *predication* applied by the GPUVerify front-end, whereby conditional code is largely flattened, reducing the number of paths in the resulting Boogie program [26]. This predication is similar to a *phi node folding* optimisation incorporated in the KLEE-CL tool [53], suggesting that this may be a useful optimisation to incorporate in Symbooglix more generally.

These results partially support hypothesis 1 (see §3.1), that symbolic execution of an IVL is competitive with other techniques. On the GPU benchmarks Symbooglix is highly competitive given that it finds the most bugs without reporting false positives, and manages to verify a significant number of benchmarks. Symbooglix is less competitive than the other tools on the SV-COMP benchmarks but does show bug finding capabilities that complement the existing tools.

3.4.6 Comparison with KLEE

Because we also apply Symbooglix to benchmarks that arise from C programs, it seems natural to compare the tool with KLEE, a state-of-the-art symbolic execution tool targeted towards C code. Various issues make this comparison less straightforward than it might appear: due to various engineering issues, KLEE cannot be applied out-of-the-box to the SV-COMP examples, and an apples-to-apples comparison is not possible because the Boogie benchmarks that

Symbooglix analyses have already been translated by the SMACK front-end, which may have changed the semantics (and shape) of the benchmarks. However, we believe that a brief comparison is still useful in highlighting some differences between the tools.

We took the 374 SV-COMP benchmarks used during Symbooglix's training phase and removed 7 floating point benchmarks which KLEE cannot handle (Symbooglix can handle them because SMACK provides an abstraction for floating point operations). We then removed the benchmarks where the original SV-COMP labels (i.e. those for the C programs) did not match the labels inferred for the corresponding Boogie programs. Label mismatches between a C program and the corresponding Boogie program generated by SMACK indicate that SMACK has changed the semantics of the program in a manner that affects its correctness status, making a comparison between KLEE and Symbooglix on that program meaningless or that the C program was mislabelled. After this filtering, we were left with 361 benchmarks[11], which we call the *reduced training set*.

We modified[12] KLEE to support the built-in verifier functions used by the SV-COMP benchmarks and ran it on the reduced training set. This revealed several engineering issues. The SV-COMP benchmarks are a mix of 32-bit and 64-bit C benchmarks, and KLEE only works correctly when it is compiled for a target that matches the compilation target for the benchmarks. This required us to build a 32-bit and 64-bit build of KLEE to run on the 32-bit and 64-bit benchmarks respectively. We also found that KLEE cannot run the majority of the 64-bit benchmarks, which are based on code from the Linux kernel and use `extern` globals that are not initialized. Symbooglix does not have these issue because the Boogie IVL is architecture independent, and SMACK's translation does handle `extern` globals. These issues illustrate a trade-off between the levels at which the two tools operate: KLEE runs LLVM bitcode, which precisely models system implementation details, while Symbooglix runs Boogie programs, where such details are left abstract. The former approach is better at finding subtle implementation-level bugs, but is more time-consuming to apply (as illustrated by the issues above). While the latter can miss such bugs, avoiding precise system implementation details can simplify looking for bugs that are independent from these details.

[11] `https://github.com/symbooglix/sv-benchmarks/tree/klee_svcomp15_smack`
[12] `https://github.com/symbooglix/klee/tree/svcomp`

Table 3.3: Results for KLEE and Symbooglix on the reduced training set.

Tool	Verified	Bug found	False alarm	Unknown
Symbooglix	17	31	0	313
KLEE	10	54	1 (see text)	296

With these issues in mind: Table 3.3 shows how Symbooglix and KLEE compare in terms of benchmark classification. Symbooglix was able to verify more benchmarks than KLEE. The tools verified 9 common benchmarks (KLEE was faster 2/3 of the time), with Symbooglix verifying 8 benchmarks that KLEE did not, and KLEE verifying 1 benchmark that Symbooglix did not. KLEE was able to find more bugs than Symbooglix. The tools found bugs in 18 common benchmarks (in all cases KLEE found the bug faster), with Symbooglix finding bugs in 13 benchmarks that KLEE did not, and KLEE finding bugs in 37 benchmarks that Symbooglix did not. The single false alarm reported by KLEE is, in fact, not really a false alarm: the associated benchmark is labelled as correct, but KLEE reports an out-of-bounds memory access. The benchmark is from an SV-COMP category in which memory-safety checking is not required. SMACK omits array bounds checks when translating benchmarks in this category to Boogie, but KLEE always checks array bounds and thus raises this (genuine) error. If similar issues apply in the application of KLEE to other SV-COMP benchmarks, the number of bugs found by KLEE in Table 3.3 might be higher than it would be if we could disable KLEE's automatic checks when appropriate; however, KLEE does not support disabling of these checks.

Finally, note that a useful feature of KLEE is that, on detecting a bug, KLEE can generate a concrete input to trigger it. With engineering effort, we could extend Symbooglix to query the SMT solver in order to generate conditions that would cause a buggy Boogie program to fail. If the Boogie program was generated by a front-end (e.g. SMACK or GPUVerify), extra effort, tailored to the nature of the translation into Boogie, would be required to map the failure conditions for the Boogie program to a bug-triggering input in the original program.

3.5 Related Work

There is a large body of existing work that seeks to compare program analysis techniques either by empirically testing tools [126, 103, 8, 68, 67] that implement them or by discussing fundamental techniques [9, 84, 60]. In addition to the numerous publications there is also the yearly software verification competition (SV-COMP)[13] where various program analysis tools compete on a common set of benchmarks on their performance and precision.

Kassios et al. [100] perform work closely related to our comparison. They empirically compare symbolic execution and weakest pre-condition generation on an experimental language for verifying concurrent programs called Chalice [117]. They compare the chalice verifier (based on weakest pre-condition generation) with a symbolic execution tool named Syxc [96]. They observe that over their 29 test cases symbolic execution is approximately twice as fast weakest pre-condition generation. This is not a trend we generally observe in our work but this likely because our set of benchmarks is substantially larger and more diverse.

As far as we are aware, ours is the first work to compare several different program analysis techniques on the same IVL.

Symbooglix and Boogaloo are not the only symbolic execution tools applied to an IVL. Le et al. [110] create their own IVL for verifying SystemC modelling programs and create a symbolic execution tool for that IVL.

All the Boogie back-ends we compare against are related work and are mainly discussed in §2.3.4. We now discuss them in relation to our Symbooglix tool.

The Boogaloo back-end is the most similar to Symbooglix given that they both implement symbolic execution. Boogaloo has superior support for quantifiers compared to Symbooglix which handles them naively by passing them unmodified to the underlying constraint solver. However Boogaloo's approach is unsound whereas Symbooglix's approach, although naive is sound. Symbooglix could potentially adopt Boogaloo's approach, however given our preference for being sound we would likely look for a different approach. Boogaloo has superior test case generation support compared to Symbooglix. It supports test case minimisation and reports test cases to users automatically. Symbooglix on the other hand doesn't support minimising test cases and doesn't provide an easy way to obtain test cases. There is no technical reason for

[13] `https://sv-comp.sosy-lab.org/`

this limitation of Symbooglix. Symbooglix supports more of the Boogie IVL than Boogaloo because it supports the bit-vector family of types and all operations on those types. Symbooglix is also more optimised (for our benchmarks at least) than Boogaloo because many of the optimisations discussed in §3.3 are not implemented by Boogaloo. One exception to this is the map updates at concrete indices optimisation which Boogaloo handles in a some what similar manner. However as far as we know Boogaloo doesn't try to handle symbolic map indices that don't alias. Boogaloo could adopt the optimisation ideas used by Symbooglix. In terms of re-usability, Symbooglix is superior to Boogaloo because it built on top of the existing infrastructure that all other Boogie back-ends use which means its components (e.g. its logic for constant folding expressions) could potentially be re-used in all those back-ends. Boogaloo on the other hand is written from scratch in Haskell and so it's components cannot be re-used in existing back-ends.

Symbooglix is not similar to the other back-ends because they use fundamentally different techniques. As we have seen in §3.4.5 their performance is considerably different and in the case of Corral and Duality complementary. This hints that some hybrid combination of the techniques may be beneficial and may be a potential avenue for future work [82, 24, 86]. Some of the optimisations we implement for Symbooglix are not applicable to the other back-ends because they are symbolic execution specific. However unique global constants constraint representation, global dead declaration elimination, and expression simplification could be used.

3.6 Conclusion

We have presented a large scale evaluation of several program analysis tools including our own symbolic execution tool, Symbooglix, with the goal of addressing the research problem stated in §3.1 that no comparison of program analysis tools had been performed on an IVL.

We have also presented Symbooglix, a new symbolic execution engine for the Boogie intermediate verification language, and described an empirically-driven approach to optimising the tool. Through a large experimental evaluation on two diverse benchmark suites, we find that Symbooglix significantly outperforms Boogaloo, an existing symbolic execution tool, in terms of applicability and analysis coverage. This supports our second hypothesis which is that the state-of-the-art for symbolic execution of an IVL can be improved.

The evaluation also shows that Symbooglix is competitive with GPUVerify and out-performs other state-of-the-art Boogie analysers on a suite of benchmarks for which GPUVerify is highly optimised. On a suite of Boogie programs derived from the SV-COMP 2015 benchmark suite, the overall analysis capabilities of Symbooglix are lower than those of the Corral and Duality tools, but Symbooglix is highly complementary to these tools in terms of bug-finding ability. This partially supports our first hypothesis which is that symbolic execution of an IVL is competitive with other techniques in terms of bug finding and verification.

Chapter 4

Symbolic execution of programs using floating-point arithmetic

4.1 Introduction

A key component underpinning any symbolic execution tool is a constraint solver, often a *satisfiability modulo theories* (SMT) solver, which does the heavy lifting associated with determining whether execution paths are feasible at runtime, and whether there exist values of the symbolic inputs that cause correctness conditions to fail.

Due to the challenges associated with constraint solving for floating-point arithmetic, most symbolic execution tools do not directly support symbolic floating-point reasoning, instead either using approximations [17], using structural equivalence of expressions as a proxy for equality [54], or rejecting programs that use floating point as out of scope [38]. This is not ideal because these limitations limit the applicability of symbolic execution on floating-point programs. This is the research problem we tackle in this chapter.

Recent advances in solver technology have led to several SMT solvers adding support for floating-point reasoning along with an effort to provide a standardised theory of floating-point arithmetic [160]. Our hypothesis is that this standardised theory of floating-point arithmetic can be used by symbolic execution tools to analyse floating-point programs.

In the previous chapter we developed a symbolic execution tool (Symbooglix) that operates on the Boogie IVL. Adding support for reasoning about floating-point arithmetic to this tool might seem like a natural path to follow. However, this is not practical because the Boogie IVL does not support floating-point types and operations. As a consequence supporting symbolic execution for floating-point programs via Symbooglix would require not only modifications to Symbooglix but also to the Boogie IVL, and to one or more existing Boogie front-ends. This is a prohibitively large amount of engineering work because it requires non-trivial changes at all three levels of the stack. So, instead we turn our attention to the KLEE symbolic execution tool [38].

KLEE reasons about symbolic constraints with bit-level accuracy, and supports the entire C language with a few exceptions, the most notable of which is symbolic floating-point arithmetic. The original reason for the lack of floating-point support was the absence of a suitable solver. However, LLVM IR (the intermediate representation that KLEE executes) and Clang (the C to LLVM IR front-end) both support floating-point arithmetic. This means that supporting symbolic execution of C programs that use floating-point arithmetic via KLEE only requires changes to KLEE; the other layers of the stack require no changes. This is substantially less engineering work, and so this is the path we pursue in order to answer our hypothesis in this chapter.

The open-source version of KLEE on which we base our work has very limited support of floating-point arithmetic. This version of KLEE forces all arguments to floating-point operations to be concrete during execution, by asking the underlying constraint solver to pick satisfying assignments to the arguments. This effectively means that KLEE will reason about a single set of floating-point values on each explored path, rather than the set of all possible values. This is an under-approximation that can cause bugs to be missed. To illustrate this consider the C program in Listing 4.1.

Listing 4.1: A simple C program to illustrate the limitations of open-source KLEE.

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  float f = 0.0f;
  klee_make_symbolic(&f, sizeof(f), "f");
  if (f > 0.0) {
    printf("f is +ve\n");
  } else {
    assert(f <= 0.0f);
    printf("f is -ve or zero\n");
  }
  return 0;
}
```

On line 7 a floating-point variable `f` is declared and on the following line it is marked as a symbolic variable. Next on line 9 the program branches on the value of `f`. On the true branch the message `"f is +ve"` is printed (line 10). On the false branch an assertion is made (line 12) and then the message `"f is -ve or zero"` is printed (line 13). What happens when open-source KLEE is run on this example? It successfully makes `f` symbolic but as soon as it encounters the branch on line 9 it is forced to pick a concrete assignment to `f`. In this case it picks `f` to be $+0.0f$. Now execution continues down the false path and the `assert` on line 12 is executed. `+0.0f <= 0.0f` is true, so the assert passes, then a message is printed and the program exits. So KLEE reports only one path and no bugs in this program. This is incorrect, the program actually has three paths. In addition to the path that open-source KLEE found, there is a path that takes the true side of the branch and there is a path on which the `assert` on line 12 fails (i.e. a bug). When we apply our modified version of KLEE that we develop in this chapter to this example it finds all three paths, generating a test case for each path, and reports the failed assertion as a bug . For the path where the `assert` fails our version of KLEE generates a test case where `f` is a NaN. This is correct because `(NaN <= 0.0f)` is false. This example demonstrates how open-source KLEE's approach to handling symbolic floating-point arithmetic is inadequate, and that our modified version of KLEE addresses this inadequacy.

Early on in our work extending KLEE to support reasoning over floating-point constraints we became aware of another research group at RWTH Aachen who were undertaking an identical task, namely adding support for symbolic floating-point constraints to KLEE. When we became aware of each other's activities (via communication on the KLEE mailing list), we set up a meeting to understand the status and maturity of each implementation, aiming to avoid duplication of effort. It became clear that we were too late: both teams had already invested significant effort and created mostly complete implementations.

This coincidence gave us a rare and valuable opportunity to empirically compare, in a very direct manner, two distinct and independent implementations of the same functional specification in the same framework, via a case study in N-version programming [10, 43]. This process also allows us to answer our hypothesis, and in-fact strengthens the validity of our answer because we have two independent implementations that show the same result. Our results show that while it is indeed possible to use support for the theory of floating-point arithmetic in constraint solvers to reason about floating-point programs, several unsolved problems remain which limit the applicability of the approach. This supports our hypothesis, but with some caveats (see §4.5.3).

This chapter is structured as follows. First, we describe the methodology for our experimental comparison (§4.2). We then detail the floating-point benchmarks we developed (§4.3). Next we give a brief over of the design and implementation of KLEE (the starting point for both extensions) and then discuss the changes made in each of the of extensions to KLEE (§4.4). We then discuss the results of our experimental comparison (§4.5) in the context of our hypothesis. Our experimental comparison focuses on correctness and performance issues raised by cross-checking the implementations, and what the results say about the validity of our hypothesis. We then discuss related work (§4.6) and finally conclude (§4.7).

4.2 Methodology

On first point of contact, both teams had relatively feature-complete floating-point extensions to KLEE that had undergone preliminary correctness testing and performance benchmarking. We structured our controlled N-version programming experiment around three phases: benchmark preparation (§4.2.1), benchmark and tool improvement (§4.2.2), and in-depth comparison (§4.2.3).

4.2.1 Phase I: Benchmark preparation

During a period of approximately one month, each team devoted resources to *independently* preparing benchmark programs to be used for evaluation. Each team prepared 43 benchmarks, divided into 28 synthetic and 15 "real-world" benchmarks. The real-world benchmarks

were adapted from existing open-source applications. The synthetic benchmarks were written from scratch, with some designed to test particular aspects of floating-point semantics, and others encoding simple algorithms. The benchmarks are described in §4.3.

Both sets of benchmarks were prepared with symbolic analysis in mind: the teams ensured that most benchmarks had at least some inputs marked as symbolic, though a few fully concrete benchmarks were included to thoroughly test concrete interpretation. Due to the known limited scalability of solvers with respect to floating-point reasoning, both teams tried to restrict the amount of symbolic data to find a "sweet spot" where symbolic execution would issue interesting, yet not intractably hard, floating-point queries. Importantly, this fine-tuning was performed by each team in isolation with respect to their own tool.

Each benchmark includes a specification stating how the benchmark should be compiled and whether the benchmark is expected to be correct. An incorrect benchmark's specification states a number of expected property violations, e.g. that an assert should fail, or a division by zero or invalid memory dereference should occur. In each case a set of allowed error locations (source file and line number) are provided. The schema for this specification can be found online[1]. Having a specification for each benchmark is important because it allowed both teams to unambiguously communicate how the benchmark should be compiled and what property violations it contains (if any).

During this phase, the teams were free to improve their tool with respect to their own benchmarks. At the end of phase I, all benchmarks were pushed to a common repository.

Phase I resulted in a set of floating-point benchmarks suitable for evaluation of symbolic execution tools, with one half known to be somewhat tractable for Aachen's tool, and the other half for Imperial's tool, but importantly with no single benchmark having been prepared knowing the capabilities of both tools.

4.2.2 Phase II: Benchmark and tool improvement

The full set of benchmarks allowed each team to assess the correctness and performance of their independently-developed tool, through semantic problems and optimization opportunities raised by the other team's benchmarks. Each team spent one month fixing and optimiz-

[1] `https://github.com/delcypher/symex-fp-bench/blob/master/svcb/svcb/schema.yml`

ing their tools. Notable tool changes arising from benchmark exchange are discussed in §4.4. During this phase the teams communicated any benchmark problems not already identified during phase I, but did not exchange tool implementation details. These benchmark problems are discussed in §4.5.1.

At the end of phase II, the teams froze development of their tools and exchanged source code, enabling each team to subsequently (a) understand the design decisions of the other team via source code inspection, and (b) compare experimentally with the other team's tool.

4.2.3 Phase III: In-depth comparison

The teams now set about comparing the tools on the finalized benchmarks. Since both teams' tools leveraged Z3 and LLVM, it was agreed that both should share the same Z3 version (`4c664f1c`) and LLVM version (3.4.2) so as to restrict behavioural differences to design decisions in the KLEE extensions themselves, rather than in their dependencies.

Tool changes based on preliminary experiments. Our intent had been to conduct our in-depth comparison using exactly the tool versions frozen at the end of phase II. However, preliminary experiments revealed a number of remaining tool bugs that were easy to fix, and whose fixes were not influenced by implementation details of the opposite team's tool. We also realised that the tools were forked from different versions of KLEE, leading to potential behavioural differences unrelated to the innovations of each team. Finally, a *dynamic solver timeout* feature (§4.4.2), implemented by Imperial and orthogonal to floating-point support, allowed KLEE to terminate in a more reliable manner that influenced tool comparison. Based on this experience, we upgraded both tools by rebasing to use a common KLEE revision (`2852ef63`), donating the dynamic solver timeout feature to Aachen, and applying a number of bug-fixes following an inter-team review to confirm that fixes were not inspired by details of the opposite tool.

One fix that is worth discussing is a change to how the Z3 solver was instantiated (`Z3_mk_solver()` vs `Z3_mk_simple_solver()`). It was discovered that the array ackermannization optimisation was not causing the intended effect when `Z3_mk_simple_solver()` was being used, leading to sub-optimal performance. However when the other solver API call was used the optimization worked as intended. The reason for this is that `Z3_mk_simple_solver()`) bypasses all of Z3's logic-specific strategies and uses the DPLL(T) [143] method (lazy translation to SAT) of solving constraints. This is frequently slower than using Z3's logic-specific strategies. In particular

the array ackermannization optimization turns queries in the logic of quantifier free arrays of bit-vectors (`QF_ABV`) into the logic of quantifier free bit-vectors (`QF_BV`). When `Z3_mk_solver()` is used and the logic in the query is `QF_BV` Z3 applies a strategy that uses eager bit-blasting to SAT [104]—in place of the frequently slower DPLL(T)-based approach[2].

Unfortunately, this issue was discovered after phase II and Aachen did not agree with integrating this change into the tools. As a result the paper version [123] of this work discusses the version of the tools that uses the sub-optimal Z3 API call. However in this chapter our end goal is to answer our hypothesis, and we should try to answer the hypothesis using the most performant versions of the tools. Therefore in this chapter we will use the versions of the tools that use the solver API that leads to better performance.

Running the tools. We ran the finalised versions of both teams' tools on the finalised benchmark suite, on a machine with two Intel® Xeon® E5-2643 v4 CPUs (6 physical cores each) with 256GiB of RAM running Arch Linux. Hyper-threading and turbo boost were disabled. Each tool was run 20 times per benchmark. Each tool was executed in parallel over the set of benchmarks, running on at most 8 benchmarks in parallel. Each KLEE process was pinned to a single CPU core and the CPU's nearest NUMA node. The CPU cores used for pinning were isolated from the kernel scheduler using the `isolcpus` kernel parameter. The pstate CPU governor was set to "performance" requesting the same min/max frequency (3.5GHz) and a `0` performance bias. We enforced a 100 GiB memory limit per KLEE process, enforcing that the swap file was not used. Each tool was executed in a Docker [132] container to keep the processes isolated. Address space layout randomisation (ASLR) was disabled.

KLEE has two distinct execution phases: a path exploration phase followed by a final test case generation phase. We enforce a time limit of 1 hour for each phase.

Each team's tool was configured to use the same path exploration strategy (non uniform random search prioritizing coverage, with a fixed random seed).

Comparing the tools. In order to compare the tools we extracted the following information from each run.

The validity of a reported bug, i.e. whether it is a true or false positive. A test case that detects a particular issue at a certain benchmark source location is considered to detect a true positive if and only if the benchmark's specification indicates that an issue of this type is expected at

[2] `https://github.com/Z3Prover/z3/issues/1251`

that location. Checking validity was achieved by replaying KLEE-generated cases natively for reported bugs and verifying that the bug type, source file and line number match what KLEE reported. Checking for expected assertion failures and calls to `abort()` required no special instrumentation. To check for out-of-bound memory accesses and division by zero—undefined behaviours in C that are not guaranteed to raise a runtime exception—we instrumented the benchmarks using AddressSanitizer [163] and UndefinedBehaviorSantizer [173] respectively.

Branch coverage achieved on each benchmark. This was measured by replaying all KLEE-generated tests natively on a coverage-instrumented (via gcov [78]) build of the benchmarks. Coverage excludes the C library to avoid bias; e.g. a team's tool might interpret a C library function (may lead to additional test cases) rather than modelling it in Z3 (no additional test cases), possibly leading to greater coverage of the C library which upon replay could inflate the team's coverage artificially.

Although KLEE has its own internal counters to track the achieved coverage it was decided to not use this information because this information could be wrong (intentionally or unintentionally). Using the coverage achieved by replaying KLEE generated test cases avoids this problem.

Execution time and crashes. We recorded execution time for each run of a tool on a benchmark, and noted cases where a tool crashed due to an internal error or running out of memory.

Memory and time limits of 10 GiB and 5 minutes respectively were used when replaying test cases.

Due to the insufficient coverage support in Clang 3.4.2, GCC 6.2.1 (optimisations enabled) was used to compile benchmarks for replay.

We merged the repeated runs of the same tool configuration as follows. The true and false positives for a tool with respect to a benchmark were identified by replaying the test cases generated across *all* runs, crediting a tool for finding a true positive during *any* run, but similarly penalising for any instances of false positives. We computed branch coverage and execution time for a tool with respect to a benchmark as the arithmetic mean across all runs. We counted the number of crashes for a tool with respect to a benchmark as the total number of crashes observed across all runs.

We then ranked the tools on a per-benchmark basis using the following rules, applied in order:

(A) A tool wins if it reports no **false positives** and the other tool reports at least one **false positive**.

(B) Most **true positives** wins.

(C) Highest mean **branch coverage** wins.

(D) If at least one tool **crashed**, smallest crash count wins.

(E) Otherwise, smallest mean **total execution time** wins.

The tools draw if they are not distinguished by these rules. The rationale for ranking is: a symbolic execution tool should never exhibit false positives (A), because its primary goal is to accurately find bugs (B), with a secondary goal of producing a high-coverage test suite (C). Thereafter, we prefer a tool that does not crash (D), and a fast tool when neither crashes (E). It is hard to meaningfully compare false positives (two distinct false positives might not be equally serious), so in (A) we do not rank tools by number or nature of false positives.

When comparing mean branch coverage and execution time we use 95% and 99.9% confidence intervals, respectively, regarding results as indistinguishable if intervals overlap. Mean execution time differences of less than one second are also treated as indistinguishable.

4.3 Benchmark suite

As mentioned in §4.2.1 both teams independently contributed 43 benchmarks (86 in total), written in C99 [94] or C11 [95]. Each team aimed to choose examples that would be challenging yet not intractable for symbolic execution, being free to choose benchmarks that played to the strengths of their tool, with benchmarks prepared by the other team posing an unknown challenge. The suite contains 52 benchmarks expected to be correct and 34 expected to be incorrect. The suite uses KLEE-specific functions (e.g. to introduce symbolic data) for our convenience, however it would be easy to port the benchmarks to a more popular interface (e.g. the SV-COMP interface [55]) so that the benchmarks are applicable to other analysis tools. Branch counts reported below are the number of static branches in the compiled LLVM IR.

The infrastructure for building the benchmarks can be found at `https://github.com/delcypher/symex-fp-bench`.

4.3.1 Aachen's benchmarks

These benchmarks can be found at `https://github.com/danielschemmel/fp-benchmarks-aachen`.

Synthetic (17 correct, 11 incorrect): These focus on checking a wide range of floating-point semantic features, split up so that each benchmark tests as few individual features as possible, allowing floating-point symbolic execution errors to be easily pinned to underlying causes. Some check properties that are uncommon in real-world programs, to ensure that unusual and erroneous uses of floating-point numbers are handled accurately. These benchmarks have between 1 and 56 (median 3) branches and request between 0 and 32 (median 8) symbolic bytes.

Real world (13 correct, 2 incorrect): These were drawn from multiple publicly available sources, with care taken to include both large and well-tested software, reflected by benchmarks built upon the GNU Multiple Precision Arithmetic Library[3], and sample programs not intended for production use, e.g. numerical code taken from [152]. These benchmarks have between 6 and 2903 (median 220) branches and request between 1 and 20 (median 8) symbolic bytes.

4.3.2 Imperial's benchmarks

These benchmarks can be found at `https://github.com/delcypher/fp-benchmarks-imperial`.

Synthetic (15 correct, 13 incorrect): These comprise simple algorithms (e.g. binary search), and tests for fundamental properties of floating point (e.g. commutativity and non-associativity of addition). We include a port of William Kahan's *paranoia* benchmark [99, 98]. These benchmarks have between 1 and 301 (median 8) branches and request between 0 and 128 (median 8) symbolic bytes.

Real world (7 correct, 8 incorrect): These were written against the GNU Scientific Library[4] (libGSL), adapted from tutorial examples included with the library source code. Creating these benchmarks involved iteratively increasing the presence of symbolic input, stopping just before the tipping point beyond which Imperial's tool could not make reasonable progress. Some of the libGSL benchmarks used long doubles, a feature that Imperial's early tool did not sup-

[3] `http://gmplib.org/`
[4] `https://www.gnu.org/software/gsl/`

port; we modified the associated benchmarks to avoid long doubles. These benchmarks have between 6 and 254 (median 67) branches and request between 4 and 48 (median 16) symbolic bytes.

4.4 Design Details

We first discuss the high-level architecture of KLEE which is the same in both tools and then discuss notable similarities (§4.4.1) and differences (§4.4.2) in design decisions made by both teams as determined by source-code examination. The source code of the Imperial and Aachen tools are available at `https://github.com/srg-imperial/klee-float` and `https://github.com/COMSYS/klee-float` respectively.

Figure 4.1: Architecture of KLEE.

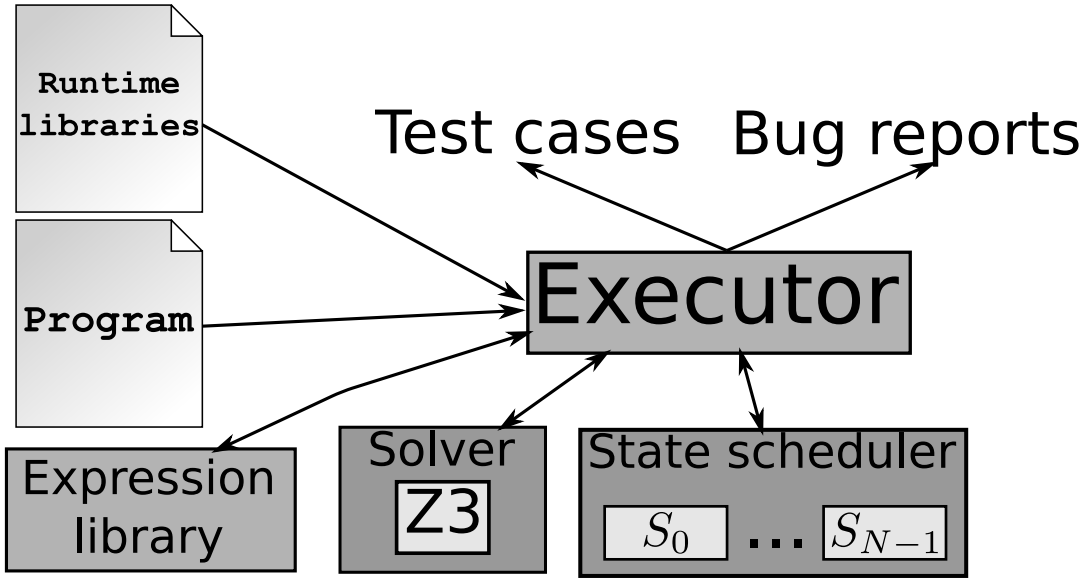


Figure 4.1 illustrates the high-level architecture of KLEE. A program (in our case a C program, possibly linking with several libraries) is compiled to LLVM IR bitcode. This is then parsed into LLVM IR and then given to the *executor* to execute. The executor is the core of KLEE, it interprets the LLVM IR and coordinates all actions that result from this interpretation. Before it is able execute the program it links the program against several runtime libraries (e.g. C library). The executor then sets up the initial execution state (e.g. heap, global, and stack memory; and path constraints) and then gives that state to the *state scheduler*. The state scheduler stores each execution state (each corresponds to a feasible path in the program) and decides which execution state the executor should run next. As the executor runs it will use the *expression li-*

brary to construct expressions to represent the state of memory and constraints. When the executor needs to check the feasibility of a path it invokes the *Solver* to check a set of constraints that are represented using expressions that were created using the expression library. KLEE supports multiple constraint solvers but in our work we use the Z3 constraint solver which is why it is shown inside the solver component. If a path terminates the executor generates a *test case* for it and if the path terminates due to a bug it also emits a *bug report*. This description of KLEE is grossly over simplified but is sufficient to understand the work in this chapter because all the important components that were modified by the two teams are shown in the figure.

At the end of phase II (§4.2.2) we exchanged tool source code, allowing comparison of tool designs that had previously been unknown across teams.

Exchanging benchmarks at the end of phase I clearly led to improvements in the design of both tools. We consider this a positive outcome, illustrating how a shared set of independently-gathered benchmarks can drive tool development.

4.4.1 Notable similar design decisions

We discuss design issues where both teams took similar approaches, highlighting noteworthy differences in the details.

Constraint solver. Both teams used Z3 [140] as the constraint solver; this was a natural choice as Z3 supports floating point and was already integrated into open-source KLEE.

Floating-point types, operations and functions. KLEE is primarily designed to execute programs written in C, but actually executes LLVM intermediate representation (IR). Both teams assumed the x86_64 target and thus that the `float`, `double`, and `x86_fp80` LLVM types map to the IEEE-754 fp32, fp64, and x86_fp80 types. Assumption of IEEE-754 semantics was key, as they are used by the SMT-LIB floating-point theory [34] that Z3 implements.

Both teams assumed that the primitive arithmetic LLVM IR instructions (e.g. `fadd`) map to corresponding IEEE-754 operations, except for `frem`, whose semantics are not the same as the IEEE-754 remainder function [131]. Both teams assumed that operations on LLVM types consistently use the same precision and range, so that excess precision/range are not used

during computation. This assumption holds if during native code generation the compiler uses SSE instructions (rather than the legacy x87 FPU) to do floating-point computations on fp32 and fp64 types [137].

Vector instruction support. LLVM IR provides vector types derived from the basic floating-point types; enabling compiler optimizations can cause vector instructions to be emitted. To process these, both teams adapted LLVM's `Scalarizer` pass to scalarize as much as possible prior to symbolic execution, so that KLEE can assume that most instructions (e.g. `fadd`) only take scalar operands. A few instructions—`InsertElement`, `ExtractElement`, and `ShuffleVector`—required special handling; both teams added varying levels of support, sufficient to run the benchmarks (§4.3).

It is worth noting that at the end of phase I, Aachen and Imperial's support for vector instructions differed greatly. Imperial compiled their benchmarks with optimizations enabled, necessitating vector support; Aachen compiled their benchmarks without optimizations, thus did not need vector support. During phase II it became necessary for Aachen to add vector support in order to handle Imperial's benchmarks.

IEEE-754 rounding modes. Both teams implemented support for all IEEE-754 rounding modes available from the C standard library interface, by having a per-execution state flag that stores the current concrete rounding mode and ensuring that this is used when constructing constraints (e.g. floating-point addition is affected by the rounding mode, making it a ternary operator). KLEE has the ability to call *external functions*—functions missing from the program under analysis that can nevertheless be executed by the running KLEE process on behalf of the program under analysis. Both teams ensured that when calling external functions the rounding mode of the KLEE process is changed to that of the calling execution state and reverted back on return. One slight difference in Imperial's implementation is that the rounding mode is allowed to be symbolic, whereas Aachen's implementation concretizes a symbolic mode. Imperial's implementation achieves this by forking on symbolic rounding modes (i.e. one path per rounding mode plus an extra path for an invalid rounding mode).

Standard math functions. The teams initially used different approaches for handling C standard library math functions. Aachen represented these functions where possible as operations in KLEE's constraint language (e.g. `fabs`, `sqrt`) and interpreted UClibc's implementations for other functions. However, Imperial simply interpreted the implementation for all math func-

tions in UClibc's C library. Imperial's approach suffers from path explosion when operands to the functions are symbolic. Upon exchanging benchmarks at the end of phase I, Imperial discovered their tool performed poorly on several of Aachen's benchmarks due to path explosion and so switched to handling `fabs` and `sqrt` as operations in KLEE's expression language. Both teams had to add UClibc's math library to the list of KLEE's runtime libraries because open source KLEE does not link with it.

IEEE-754 exceptions and flags. Neither team implemented this portion of the IEEE-754 specification because it would significantly complicate symbolic execution: one would need to maintain the symbolic value of the flags and check if any exceptions could be raised by every floating-point instruction executed. A consequence of this (and of the SMT-LIB floating-point theory) is that symbolically neither tool can distinguish between quiet and signaling NaNs.

4.4.2 Notable differences

Extending KLEE's expression language. Both teams extended KLEE's expression language to incorporate floating-point expressions in similar ways, but with some notable differences related to how comparison operations and constants are handled. Aachen added operations that correspond directly to the opcodes of the LLVM `FCmp` instruction. The instruction has *ordered* and *unordered* variants, which return false and true, respectively, if either argument is a NaN. Instead, Imperial only added operations that correspond to the *ordered* comparison opcodes because the *unordered* operations can be expressed in terms of the *ordered* comparison operations and the `IsNan` predicate. We consider Imperial's approach to be a better choice because it is a simpler extension to KLEE's expression language. Aachen represented floating-point constants as a separate expression type whereas Imperial represented floating-point constants as implicitly bitcasted integer constants.

Representation of types. KLEE's expression language is based solely on bit-vectors, which was problematic when introducing floating-point operations. Imperial handled this by making conversion between floating-point and bit-vector types implicit, so that e.g. the bit-vector operands to a floating-point `add` instruction are first converted to floating-point types. Aachen instead made this explicit by introducing type conversion operations. We consider Aachen's

choice to be superior here because implicit conversion has ambiguities. In particular if an *if-then-else* expression has a mixture of bit-vector and floating-point types for its *then* and *else* expressions, the type of the *if-then-else* is ambiguous.

Array ackermannization. KLEE represents all symbolic data, including primitive symbolic variables, as arrays of 8-bit bit-vectors. Imperial observed that in a floating-point context, if all arrays of bit-vectors are replaced with simple bit-vector variables and given to Z3 in such a way that the new query is equisatisfiable with the original query, then performance usually improves. We refer to this transformation as *array ackermannization*. The Z3 developers confirmed this, suggesting that Z3 is not currently well-optimized for queries mixing the theory of quantifier-free bit-vector arrays with the theory of quantifier-free floating point[5].

Imperial's tool performs array ackermannization in the case where all array reads are at concrete indices and no writes have been performed to the array. This is a fairly common case because symbolic variables in the original C program being analysed are typically represented as a concatenation of byte reads at successive concrete indices of a symbolic array. Aachen's tool does not implement array ackermannization.

x86_fp80 support. At the end of phase I, only Aachen added support for x86_fp80 and benchmarks to exercise it. Thus during phase II it became necessary for Imperial to implement support for symbolic reasoning over this type too.

Our designs differed due to several characteristics of the x86_fp80 type. First, it is a padded type: its 80 bits are padded to 128 bits on x86_64, and KLEE does not handle such padding properly. Both teams handled this issue in the same way, making sure KLEE allocates the necessary padding during the allocation of x86_fp80 stack and global variables.

Second, because x86_fp80 is not an IEEE-754 binary type, constant folding expressions of this type required careful consideration, and expressions of this type could not be directly modelled in the SMT-LIB floating-point theory.

KLEE already had support for constant folding the x86_fp80 type via LLVM's `APFloat` class, which performs arbitrary precision floating-pointing arithmetic in a hardware independent manner. However, one of Aachen's benchmarks caused both teams to discover that `APFloat` has a bug[6] where *unnormal* operands are not handled correctly. Imperial solved this issue by

[5] `https://github.com/Z3Prover/z3/issues/577`
[6] `https://bugs.llvm.org/show_bug.cgi?id=31294`

evaluating all x86_fp80 operations natively within KLEE itself. Aachen solved this by storing a flag in every expression which is set to true iff the expression represents a member of one of the IEEE-754 classes. When the value is accessed through a x86_fp80 operation, this flag is examined and if it is false for any operands a NaN is returned, which mirrors how unnormal operands are treated on x86_64. A fix for the bug in `APFloat` would avoid the need for these workarounds.

To handle evaluating symbolic expressions over x86_fp80 with Z3, both teams used the `(_ FloatingPoint 15 64)` type (abbreviated as z3_fp79) which has a 15-bit exponent, a 64-bit significand, and an implicit integer significand bit. It has exactly the same range and precision as x86_fp80, but uses a different 79-bit binary encoding due to the SMT-LIB floating-point theory only being able to represent IEEE-754 classes. The different binary encoding requires special handling of conversions between a bit-vector that represents x86_fp80 data and z3_fp79.

Imperial chose to only allow the IEEE-754 classes of the x86_fp80 type. When converting to z3_fp79 from a bit-vector the explicit leading significand bit is removed and an additional constraint is added that asserts that the bit has the correct value for the bit-vector to represent an IEEE-754 value. When converting a z3_fp79 to a bit-vector, the explicit leading significand bit is added back and its value is inferred from the other bits to be an IEEE-754 value.

Aachen chose to allow the non IEEE-754 classes of the x86_fp80 type in addition to the IEEE-754 classes. This was achieved by representing expressions of the x86_fp80 type as a tuple. The first value in the tuple is of type z3_fp79. The second value is a boolean flag that is true iff the value represented by the tuple belongs to one of the IEEE-754 classes. These tuples are then handled by each floating-point operation. If one of the tuple operands does not represent a value from one of the IEEE-754 classes it returns the tuple (NaN, `true`).

Imperial's implementation results in simpler constraints being given to Z3 but is incomplete. Aachen's implementation is complete but the constraints are more complicated and also contradict the goals of array ackermannization because the tuple is represented as a two element array.

Dynamic solver timeout. Although KLEE can limit the time allowed for path exploration before switching to test case generation, KLEE does not try to interrupt the solver if the path exploration timeout is reached. For long-running solver queries—especially frequent when us-

ing floating-point constraints—this can cause the path exploration timeout to fire much later than intended. This leaves less time for test case generation (see §4.2.3), and may cause test cases to be lost.

KLEE supports setting a fixed (i.e. static) solver timeout, but this does not interact well with the path exploration timeout. A small solver timeout may leave paths unexplored despite there being available time for further path exploration, while too large a timeout may cause the path exploration timeout to be missed as discussed above.

Imperial implemented a *dynamic* approach to solve this problem. Every time the solver is invoked, the per-query solver timeout is set based on KLEE's current state. If KLEE is doing path exploration, the solver timeout is set to be the remaining path exploration time. When KLEE switches to test case generation, the solver timeout is set to the total allowed time for test case generation divided by the number of test cases to generate. This means that each test case is given an equal share of solver time.

While Aachen did not originally implement such a feature, as noted in §4.2.3, to ensure compatibility this feature was donated to Aachen's implementation.

NaN representation. Neither tool can distinguish between quiet and signaling NaNs. IEEE-754 does not precisely specify the binary encoding for either of these NaNs, which means there are many different encodings that represent the same type of NaN. Therefore, when converting a floating-point expression to a bit-vector expression, if it is feasible for the expression to be NaN, it means the converted bit-vector expression can take on many values that all represent NaN. In practice this is rarely a problem, however one of Imperial's synthetic benchmarks deliberately tries to branch on the value of the lower-order bits of a NaN, whose value is not defined by IEEE-754.

During phase II, Imperial discovered that this would sometimes crash their implementation due to inconsistent constraints. This was partly due to some bugs in Z3[7] and LLVM's APFloat[8]. This was caused by Z3 giving a model which when substituted into KLEE's expression language would be unsatisfiable (even though Z3 claimed the model was satisfiable in its own constraint language). We refer to this as "inconsistent constraints". There were two causes for this. First, sometimes Z3 would generate invalid models. This issue was reported[9] and fixed by Z3's de-

[7] `https://github.com/Z3Prover/z3/issues/740`
[8] `https://bugs.llvm.org/show_bug.cgi?id=30781`
[9] `https://github.com/Z3Prover/z3/issues/740`

velopers. Second, there is an inconsistency between KLEE's and Z3's expression languages when handling NaNs, which is partly due to a bug[10] in LLVM's APFloat which is used to evaluate KLEE's expression language under a given model. Imperial mitigated this issue by doing two things. First, using a Z3 configuration option to always use the same binary representation for NaNs (quiet NaN, where all significand bits except the least significant are zero) when casting floating-point expressions to bit-vector expressions. Second, ensuring that KLEE's expression language used the same binary representation for NaN as Z3 during constant folding. However this is not ideal because it means that during concrete execution it is only possible to have one bit pattern represent NaN, whereas during real execution many different bit patterns could represent NaN. Unfortunately the under-specified nature of IEEE-754 NaN bit patterns are a general problem. Even if KLEE's expression language was modified to have identical semantics to Z3, they may still differ from those of the target machine or other SMT solvers.

4.5 Experimental Comparison

In this section we turn to the comparison of the tools. First, we discuss problems with the benchmarks flagged by tool comparison (§4.5.1), and then the results comparing the finalized tool versions head-to-head during phase III (§4.5.2). Then we discuss the validity of our hypothesis in the context of our results (§4.5.3). Finally, we discuss threats to the validity of our work (§4.5.4).

4.5.1 Benchmark issues

Non-termination. When replaying tests generated by the tools (to gather coverage information and check that reported bugs are reproducible) we found that two benchmarks[11][12] did not terminate for certain inputs. In one case this was unintentional, due to a bug in the implementation of a binary sort algorithm used by the benchmark. We did not fix these benchmarks for our comparison because neither the benchmark specifications nor KLEE are concerned with non-termination bugs. We handled these cases by setting a timeout when replaying test cases

[10] `https://bugs.llvm.org/show_bug.cgi?id=30781`
[11] `https://github.com/delcypher/fp-benchmarks-imperial/issues/13`
[12] `https://github.com/delcypher/fp-benchmarks-imperial/pull/12`

(see §4.2.3). Due to gcov implementation details [78], branch coverage is not recorded for non-terminating tests. This affected only a few benchmarks and gave neither tool an advantage in our ranking scheme.

Unnormal values. We found several problems with a benchmark that operated on the x86_fp80 type, involving *unnormal* values (see §2.5). First, Clang would incorrectly optimize the benchmark by performing erroneous constant folding[13]. We thus disabled optimizations for this benchmark. We also discovered it is possible to exploit this problem to crash Clang[14]. Second, we found that several operations on unnormal numbers used in this benchmark behaved *inconsistently* across compilers (e.g. `isnan()` and casting to integers). We concluded that this was due to these operations exhibiting undefined behaviour, and removed them from the benchmark. Their removal did not otherwise affect this synthetic "torture test" benchmark.

The remaining issues are cases where our tools found a benchmark to be incorrect, contrary to its specification; for each issue we applied a simple fix:

Failing to account for NaNs. A benchmark that sorted an array of partially symbolic floating-point values was incorrect when infinity values were added to yield NaN values, later triggering an assertion failure when checking correctness of sorting. A benchmark performing matrix multiplication on a partially symbolic matrix was similarly incorrect. We fixed these issues by adding assumptions that inputs are not infinities or NaNs, respectively.

Failing to account for scientific notation. A benchmark that verifies the output of `atof()`[15] intended to constrain the characters of the symbolic input string to represent a small decimal value, asserting that the result of `atof()` was in the expected range. The input constraints accidentally allowed scientific notation (e.g. `1e10`), so that `atof()` could generate a value outside the expected range. We fixed this by only enforcing the assertion when the symbolic input string did not contain scientific notation.

[13] `https://bugs.llvm.org/show_bug.cgi?id=31294`
[14] `https://bugs.llvm.org/show_bug.cgi?id=31292`
[15] Converts a string to a floating-point value.

Failing to take poor approximation into account. A benchmark that checks the result of $\sqrt{x^2}$, where x is a symbolic floating-point value, did not account for denormal numbers. As the computation may cause a gradual underflow to a denormal number, its precision may be reduced to a single bit in the worst case, causing a very high relative error. We fixed the benchmark by changing its specification to state that there exists a failing path.

4.5.2 Head-to-head tool comparison

Tool ranking. Table 4.1 summarizes the number of benchmarks for which each team *won*, according to the procedure for ranking tools as described in §4.2.3.

Table 4.1: Ranking of the tools. Each count shows the number of wins for a tool except the last row which shows the number of draws.

Reason	Aachen	Imperial
Other tool has false positives	0	0
Finds more bugs	0	7
Highest branch coverage	0	8
Fewest crashes	0	2
Smallest execution time	1	34
Draws	34	

Neither tool reported false positives. For seven benchmarks Imperial found more bugs than Aachen. In cases where the tools found the same number of bugs, Imperial achieved higher branch coverage in eight cases. In six of these cases, Imperial achieves more coverage than Aachen because Aachen's tool crashes when trying to generate some of its test cases, whereas Imperial's tool doesn't crash and successfully generates all its test cases. In one case, Imperial achieves more coverage because it is able to explore more of the program. Finally, in one case Imperial achieves more coverage due to an artifact of the program under execution and the path exploration order used by the tools. The program consists of two sequential `if` statements (that both perform an early exit) and then an infinite loop. Imperial's tool generates a test case that takes the false branch of the first `if` statement and then true branch of the second `if` statement which then triggers the program to exit. Aachen's tool generates a test that take the true branch of the first `if` statement which then triggers the program to exit and so doesn't execute the second `if` statement. Imperial's test case covers more branches and so is awarded more coverage. Each tool should actually generate a test case that the other tool generates however they both spend the rest of their time stuck executing the infinite loop.

For benchmarks where the tools were as-yet indistinguishable, there were two cases where Imperial's tool did better due to Aachen's tool crashing. The crashes here are internal to Z3 which have been reported[16]. There were 35 cases where neither tool crashed but where the tools were distinguished by execution time, with Aachen and Imperial winning 1 and 34 times, respectively. Of the 34 draws, all are due to the execution times of the tools being considered indistinguishable (the confidence intervals associated with mean execution time overlap or the difference in the mean execution time is less than one second).

What these results show is that in almost all cases, Imperial's tool either outperforms (51 benchmarks), or has indistinguishable performance (34 benchmarks) to Aachen's tool. The main reason for these performance differences is the array ackermannization optimisation that Imperial's tool implements combined with the correct Z3 solver API call to make the optimisation work correctly (see §4.2.3). Without this, the performance of the tools is very similar [123]. The one benchmark where Aachen is more performant is due to the performance of the underlying constraint solver. Both tools send different (but equisatisfiable) constraints to Z3 during symbolic execution and Z3 determines satisfiability faster with Aachen's version of the constraints.

While these results are encouraging they do not allow us to answer our hypothesis because the results are a relative comparison. For that we need to look at different metrics, which we do now.

Tool complementarity and limitations. The results so far show that Imperial's tool almost always outperforms Aachen's tool. We now examine the extent to which the tools are capable of effective analysis of our benchmarks, whether they are hindered by common problems, and cases where they are complementary.

Table 4.2 shows the extent to which each tool is capable of correctly finding bugs in the benchmarks. The T^+ row shows, for each tool, the number of total bugs found, out of the number of bugs expected to be present from the benchmark specifications (in the 34 benchmarks with erroneous paths we expect to find a total of 49 bugs). The T^- row shows, for each tool, the number of bug-free benchmarks (52 total) that the tool is able to fully explore. That is, *all* feasible paths are enumerated so that correctness is exhaustively verified. In each row, tool complementarity is indicated by showing the extent to which bugs can be found, or correctness proven, by both tools or by neither tool.

[16] `https://github.com/Z3Prover/z3/issues/1251`

Table 4.2: Evaluation of the tools in terms of bug-finding, exhaustive exploration, number of crashes and number of timeouts. The T^+ and T^- rows show the number of true positives and true negatives respectively.

	Aachen	Imperial	Both	Neither
T^+	34 (69.39%)	43 (87.76%)	34 (69.39%)	6 (12.24%)
T^-	35 (67.31%)	39 (75.00%)	35 (67.31%)	13 (25.00%)
Crashes	10 (11.63%)	0 (0.00%)	0 (0.00%)	76 (88.37%)
Timeouts	20 (23.26%)	18 (20.93%)	15 (17.44%)	63 (73.26%)

Imperial's tool finds more bugs (T^+) than Aachen's tool and actually finds a superset of the bugs that Aachen's tool finds. Imperial's tool finds the majority (87.76%) of the bugs in the benchmark suite. However Imperial's tool misses six bugs (across four benchmarks) due to the tool reaching a timeout, showing that the tool has limitations.

Both the Imperial and Aachen tools manage to verify (exhaustive exploration without any bugs) the majority of the benchmarks, with Imperial's tool verifying a superset. However Imperial's tool fails to verify six benchmarks due to the tool reaching a timeout, again showing that the tool has limitations.

In terms of tool crashes, Imperial's tool does not crash on any of the benchmarks, whereas Aachen's tool crashes on ten of them. For two of these benchmarks, the crash is an internal Z3 issue[17], the others are due to KLEE detecting inconsistent constraints (see §4.4.2, NaN representation) and aborting.

Finally, in terms of timeouts both tools reach timeouts, with Imperial's tool hitting timeouts for slightly fewer benchmarks. Upon examination of these benchmarks (aside from the benchmark that does not terminate) the reason for hitting the timeout is not path explosion but the performance of the constraint solver. Over all benchmarks, our modified KLEE tools spend on average over 99.9% of their time waiting for the underlying constraint solver to return an answer. This shows that the limitations in our tools are primarily caused by the inadequate performance of the constraint solver.

[17] `https://github.com/Z3Prover/z3/issues/1251`

4.5.3 Validity of hypothesis

Recall from §4.1 that we wished to investigate our hypothesis which is that the standardised theory of floating-point arithmetic can be used by symbolic execution tools to analyse floating-point programs. Our tools and results support this hypothesis. However there are some caveats to this.

First, the performance of our tools are limited by the performance of the underlying constraint solver and in a significant number of cases (i.e. our tools reach a timeout) the performance is inadequate. Given that Z3's strategy for solving floating-point constraints is to bit-blast [104] to a SAT problem, the performance problems are not unexpected. The authors of CBMC have commented [35] that the approach is intractable in practice due to the size of SAT formulas generated. Addressing this problem is a potential avenue for future work, which we follow in Chapter 5. To encourage the improvement of constraint solvers in this area, we have collected 35,189 queries (18,033 in the existing `QF_FPBV` division and 18,033 into a new `QF_ABVFP` division[18]) from our tools and contributed them to the annual SMT-COMP solver competition.[19]

Second, the theory of floating-point arithmetic has some design issues that make it difficult to symbolically execute some aspects of the IEEE-754 standard in programs. The theory does not distinguish between signaling and quiet NaNs; does not provide a convenient way to model IEEE-754 exceptions; and does not provide a convenient way to model x86_fp80. We believe that all these issues could be worked around but at the cost of creating very complicated constraints. We have not experimentally verified this (apart from the x86_fp80 issue) and so leave this as future potential work.

Finally there is the risk that our benchmarks do not accurately represent real world programs. We have done our best to minimise this risk and discuss this more in §4.5.4.

4.5.4 Threats to validity

Our study has both internal and external threats to validity. Both tools use KLEE and Z3, so errors in these components may lead to bugs that go undetected when comparing the two implementations. However, the manual effort we put into writing specifications for the benchmarks renders this risk minimal.

[18] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_ABVFP`
[19] `http://cs.nyu.edu/pipermail/smt-comp/2017/000436.html`

Since both our tools were built on top of KLEE and Z3, our respective design decisions might be more similar than they would be had different frameworks been used. However, we found that in spite of this common infrastructure, building the extensions required important design decisions that resulted in significant differences. Moreover, having a common infrastructure made it possible to conduct a rigorous comparison that would not have been possible otherwise.

Our benchmarks might not be representative of floating-point code found in large deployed applications. However, our synthetic benchmarks are meant to systematically test challenging floating-point features that real applications would exercise, while our real-world benchmarks are based on existing applications or widely-used libraries.

Finally, due to the computational complexity of floating-point constraint solving, all benchmarks contain comparatively few floating-point operations and symbolic data. This is due to the fact that constraint checking large numeric applications is currently infeasible in the presence of floating-point numbers.

4.6 Related Work

There is a large body of existing work that performs analysis of floating-point programs. This was previously discussed in §2.5.3. The most similar work to ours are those that try to symbolically execute floating-point programs.

Quan et al. [154] extend KLEE to support symbolic execution of floating-point programs. Their extension relies on linking against a software implementation of IEEE-754 floating-point arithmetic (i.e. a *soft float* library). This effectively bypasses the lack of support for floating-point constraints in KLEE by re-expressing all floating-point constraints as bit-vector constraints. This allows them to use KLEE's existing constraint solver to check the satisfiability of path constraints. In contrast, our approach adds first class support for floating-point constraints inside KLEE and uses the theory of floating-point arithmetic [161] available in the Z3 constraint solver. We believe our approach is superior for several reasons. First, using a soft float library adds additional paths to explore, due to branch conditions in the soft float library. In our experience this is bad for performance and actually caused us to redesign our tools (see §4.4.1) to avoid this where possible. Second, not having floating-point expressions in path constraints means

that it is not possible to take advantage of floating-point specific improvements in constraint solvers, such as the work we undertake in Chapter 5. We have not empirically compared our approaches because their tool is not available.

Earl et al. [18]'s work on test case generation for floating-point exceptions shares some similarity with ours. We both use KLEE to symbolically execute floating-point programs, using Z3 to solve constraints. However our goals and approach are very different. Their work focuses on detecting floating-point exceptions and tries to find feasible paths by approximating floating-point constraints using real arithmetic and then refines this by performing a local search for floating-point inputs in the neighbourhood of the satisfying assignment (i.e. a set of real numbers). Our work is more generally applicable and offers more precision because we use Z3's support for the theory of floating-point arithmetic to symbolically execute arbitrary floating-point programs. In principle our approach could be used to solve the problem of finding inputs that trigger IEEE-754 exceptions. However, because we currently do not model these exceptions during execution, this isn't possible. We believe this is just an implementation limitation, rather than a limitation of our approach.

Collingbourne et al. [54] extend KLEE to check the equivalence of the output of two floating-point programs. Their approach does not rely on a floating-point constraint solver and instead speculatively executes program paths and then compares the expressions representing program output to check for equivalence. Their approach for comparing expressions first performs several rewrite rules to canonicalise the expressions and then performs a syntactic comparison. This approach is prone to false positives and cannot generate test cases. In contrast our approach is more general in that it can symbolically execute floating-point programs precisely, rather than just cross-checking two programs. Our approach does not suffer from false positives and can generate test cases. However we have not tried applying our tools to the problem of cross-checking and have not empirically compared our tools to theirs. We leave this as potential future work.

The Pex [171] and SPF [148] symbolic execution tools can use search-based floating-point constraint solvers (FloPSy [106] and CORAL [167, 31] respectively) to solve floating-point path constraints. Both solvers are incomplete (i.e. they can't show unsatisfiability). Pex tries to mitigate this by first approximating floating-point constraints as constraints over rational numbers and uninterpreted functions and then using FloPSy to refine the approximation if a satisfying assignment is found. In our view this is unsound because it may cause some paths to be marked

as infeasible, when in practice they are feasible. SPF does not appear to handle Coral's incompleteness meaning that bugs might be missed. In contrast our approach uses Z3's support for the theory of floating-point arithmetic which is a complete theory and can be combined with other complete theories such as the theory of `FixedSizeBitVectors` to solve constraints using a mixture of theories. Unfortunately, it is not possible to empirically compare these tools with ours because they operate on different intermediate representations, for which there is no common front-end.

The SPF symbolic execution tool has also been extended to use the REALIZER constraint solver [113] for the purpose of checking the accuracy of floating-point programs [157]. The solver models floating-point constraints exactly using reals, integers, and functions that model IEEE-754 rounding. These transformed constraints are then solved using Z3. This approach has the advantage that constraints can be written using a combination of floating-point and real value expressions, which allow constraints that concern the deviation of a floating-point value from a corresponding real value (a programmer usually intends the floating-point arithmetic to model) to be written. In comparison our tool does not allow for constraints to be written over reals, and thus our tool cannot be used to examine the deviation of floating-point expressions from their real counterparts. In principle our tool could be extended to support real expressions and leverage Z3's support. However, a theory combination of `FixedSizeBitVectors`, `FloatingPoint` and `Reals` is in general not decidable and would likely result in even worse performance that what we have already observed.

The FPSE symbolic execution tool [32, 11] targets C programs and uses an interval solver over real arithmetic combined with project functions to model floating-point arithmetic. This solver cannot reason over bit-vector constraints and thus cannot solve constraints that use a mixture of floating-point and bit-vector constraints. This limits the applicability of the tool because real programs have branches that involve more that just floating-point conditions. In contrast, because the Z3 constraint solver supports combinations of theories (namely the `FloatingPoint` and `FixedSizeBitVectors` theories) our approach can precisely reason about programs that contain symbolic machine integers and floating-point variables. Binaries for FPSE are available but we have not empirically compared our tools with it. We leave this as potential future work.

4.7 Conclusion

In this chapter we presented our work that tries to remove some of the limitations of previous symbolic execution tools by using the standardised theory of floating-point arithmetic available in several SMT solvers to solve floating-point constraints that appear in the context of symbolically executing floating-point programs. Our hypothesis was that using this theory in the context of symbolic execution would be possible. To test this hypothesis we collaborated with a team of researchers from RWTH Aachen to create two independently developed symbolic execution tools (§4.4), and a benchmark suite (§4.3) on which to evaluate these tools. We performed this work using a methodology inspired by N-version programming with the goal of eliciting different tool designs that would lead to more insights than if both teams had worked on the same tool. Our experience creating the tools and our evaluation (§4.5.2) supports our hypothesis, but with some important caveats.

The most important of these caveats is that constraint solver performance limits the performance of our tools. If we wish improve the performance of our tools, we must first improve the performance of constraint solvers on floating-point constraints. This problem inspired the work in the next chapter, where we experiment with a technique that can reduce the time to solve floating-point constraints.

Chapter 5

Solving floating-point constraints using coverage-guided fuzzing

5.1 Introduction

As noted in Chapter 4, in symbolic execution, constraint solving frequently accounts for the majority of analysis time. This is especially true when reasoning over floating-point constraints. Thus it is natural to seek ways to reduce the time required to solve constraints. This is the research problem we tackle in this chapter.

In this chapter, we present our initial investigation into applying *coverage-guided fuzzing* in the context of constraint solving. Recall from §2.4 that coverage-guided fuzzing involves generating random inputs for a program, with generation guided by previous inputs that have increased code coverage of the program being fuzzed. This idea of applying coverage-guided fuzzing to constraint solving was inspired by an approach we used to write and test specifications for some of the floating-point benchmarks used in the study of Chapter 4. For some benchmarks we found that our modified version of the KLEE symbolic execution engine was incredibly slow at proving whether or not assertions in the benchmarks could fail. To speed up this process, some benchmarks were fuzzed using a coverage-guided fuzzer, and in several cases the fuzzer found an input triggering an assertion failure much faster than our modified version of KLEE did. This notable difference in bug-finding time made us wonder whether

coverage-guided fuzzing could be used to quickly find satisfying assignments to floating-point constraints. Our hypothesis is that a constraint-solving approach based on coverage-guided fuzzing will be able to solve constraints faster than existing techniques in some cases.

To use a coverage-guided fuzzer to find satisfying assignments to a set of constraints Q[1], our key idea is to transform the constraints into a program P constructed such that:

- P initialises a set of variables from the fuzzer's input, with each variable corresponding to a free variable in Q.
- P contains a sequence of `if` statements where the truth of each branch condition corresponds exactly to one constraint in Q and every constraint is represented in P. We refer to these `if` statements as *constraint branches*.
- If the `false` branch of any of the constraint branches is taken, then the current input is not a satisfying assignment for Q; in this case the program returns, requesting another input.
- If execution of P follows a path that traverses *all* of the `true` branches of the constraint branches, the current input is a satisfying assignment for Q; in this case the program terminates.

The program P is then given to a coverage-guided mutation-based fuzzer. The fuzzer will repeatedly run P with different inputs until a satisfying assignment to Q is found or the fuzzer reaches a resource limit (e.g. timeout). The intuition behind applying a coverage-guided fuzzer is that it will try to generate inputs that cover new code. In particular, the program location where it is checked whether *all* constraint branches evaluated to true is a target for the fuzzer.

This approach to solving constraints is sound (provided the semantics of the constraints of Q are precisely modelled by program P) but is not complete because it *cannot* practically prove unsatisfiability in most cases. Proving constraints to be unsatisfiable would require the fuzzer to test every possible input, which is infeasible to do in a reasonable time budget for all but the most trivial constraints. Consequently, this approach needs to be paired with a complete solver to be useful on arbitrary constraints because the satisfiability of a set of arbitrary constraints is usually not known in advance. We envision that our approach would be run in parallel with a complete solver to form a portfolio solver.

In this chapter, we present the design and evaluation of JFS (JIT Fuzzing Solver) which is a prototype constraint solver implementing the previously mentioned idea. This chapter is structured as follows. First, to concretely illustrate our idea we provide an example translation from

[1]The satisfiability problem is the conjunction of all constraints in the set

floating-point constraints to a C++ program suitable for fuzzing (§5.2). We then present the design and implementation of our prototype, JFS (§5.3). Then we present our evaluation of JFS, which provides evidence in support of our hypothesis (§5.4). We then review related work (§5.5), and conclude (§5.6).

5.2 Example translation

To illustrate our idea of translating constraints into a program that can be fuzzed, we first start with an example query in Listing 5.1. The example contains five `assert` statements, each with a single constraint. The satisfiability problem that the example is stating is the conjunction of all five constraints. The example contains two free variables `a` and `b`, both of type `(_ FloatingPoint 11 53)`, which corresponds to the IEEE-754 binary64 type. The constraints perform division of `a` by `b` (where both `a` and `b` are constrained not to hold NaN values) using two different rounding modes and asserts that the results are not NaN and not equal. This is a satisfiable query. A possible satisfying assignment has `a` set to `0x0.410815d750e65p-1022`[2] ($\approx 5.65235 \times 10^{-309}$) and `b` set to `0x1.021c1b000e7cp+28` ($\approx 2.70648 \times 10^{8}$). In this assignment, dividing `a` by `b` rounding to nearest (ties to even) results in `0x0.0000000408001p-1022` ($\approx 2.088452 \times 10^{-317}$) and rounding toward positive infinity results in `0x0.0000000408002p-1022` ($\approx 2.088453 \times 10^{-317}$). The results are *denormal* numbers where the difference is in the least significant digit.

Listing 5.1: An example query in SMT-LIBv2.5 format made from a set of floating-point constraints.

```
1  (declare-fun a () (_ FloatingPoint 11 53))
2  (declare-fun b () (_ FloatingPoint 11 53))
3  (define-fun a_b_rne () (_ FloatingPoint 13 53) (fp.div RNE a b))
4  (define-fun a_b_rtp () (_ FloatingPoint 11 53) (fp.div RTP a b))
5  (assert (not (fp.isNaN a)))
6  (assert (not (fp.isNaN b)))
7  (assert (not (fp.eq a_b_rne a_b_rtp)))
8  (assert (not (fp.isNaN a_b_rne)))
9  (assert (not (fp.isNaN a_b_rtp)))
10 (check-sat)
```

A possible translation of these constraints into a C++ program—following the strategy outlined in §5.1—is shown in Listing 5.2. The `FuzzerTestOneInput` function on line 1 is the entry point for the fuzzer. The fuzzer will repeatedly call this function with different inputs. The inputs are a buffer of bytes pointed to by the `data` pointer of size `size` bytes.

[2] This is C++ hexfloat notation

Listing 5.2: A translation of the constraints in Listing 5.1 to a C++ program.

```
1  int FuzzerTestOneInput(const uint8_t* data, size_t size) {
2    if (size != 16) return 0;
3    uint64_t numSolved = 0;
4    double a = makeFloatFrom(data, size, 0, 63);
5    double b = makeFloatFrom(data, size, 64, 127);
6    if (!isnan(a)) ++numSolved;
7    if (!isnan(b)) ++numSolved;
8    double a_b_rne = div_rne(a, b);
9    double a_b_rtp = div_rtp(a, b);
10   if (a_b_rne != a_b_rtp) ++numSolved;
11   if (!isnan(a_b_rne)) ++numSolved;
12   if (!isnan(a_b_rtp)) ++numSolved;
13   if (numSolved == 5) {
14     return 1; // TARGET REACHED
15   }
16   return 0;
17 }
```

Line 2 An `if` statement checks whether the buffer is the right size to contain two `doubles` (16 bytes). If the buffer is not large enough, the function returns `0`, which instructs the fuzzer to try a different input. The reason for this guard is that the fuzzer is agnostic to our domain (where all interesting inputs are the same size) and therefore may try to resize the buffer as a possible mutation to an existing input. Not only is trying an input of the wrong size undesirable, it also could lead to undefined behaviour in the program if the buffer is too small because the programs would perform an out of bounds read. Note that in principle the fuzzer could be modified to avoid such mutations which would make this buffer size check unnecessary.

Line 3 Variable `numSolved` is initialised to zero. This variable is used to track the number of constraints satisfied.

Lines 4 and 5 Variables `a` and `b` are constructed from the buffer using bits 0 to 63 (`a`), and bits 64 to 127 (`b`). These variables correspond directly to the free variables `a` and `b` in Listing 5.1.

Line 6 An `if` statement checks whether or not `a` is a NaN. This line is the constraint branch that corresponds directly to the first constraint on line 5 in Listing 5.1. If `a` is not a NaN then `numSolved` is incremented by one to record that the constraint was satisfied.

Line 7 An `if` statement checks whether or not `b` is a NaN. This line is the constraint branch that corresponds directly to the second constraint on line 6 in Listing 5.1. If `b` is not a NaN then `numSolved` is incremented by one to record that the constraint was satisfied.

Line 8 The variable `a_b_rne` is declared and initialised with the result of calling `div_rne(a, b)`. This function computes the result of performing floating point division rounding the result to the nearest value (ties to even). This variable corresponds directly to the `a_b_rne` macro declared on line 3 in Listing 5.1.

Line 9 The variable `a_b_rtp` is declared and initialised with the result of calling `div_rtp(a, b)`. This function computes the result of performing floating point division rounding the result toward positive infinity. This variable corresponds directly to the `a_b_rtp` macro declared on line 4 in Listing 5.1.

Line 10 The variables `a_b_rne` and `a_b_rtp` are compared. This line is the constraint branch that corresponds directly to the third constraint on line 7 in Listing 5.1. If the constraint is true then `numSolved` is incremented by one to record that the constraint was satisfied.

Lines 11 and 12 It is checked whether `a_b_rne` and `a_b_rtp` are NaN, respectively. These lines are the constraint branches that correspond directly to the fourth (line 8) and fifth (line 9) constraint in Listing 5.1 respectively. The variable `numSolved` is incremented as appropriate if either of these constraints is satisfied.

Line 13 An `if` statement checks whether `numSolved` is equal to `5`. If the condition is true, all constraints have been satisfied and execution proceeds to line 14, where `return 1` is executed. This indicates to the fuzzer that the desired input has been found. If the condition is false, then not all constraints have been satisfied so execution proceeds to line 16 where `return 0` is executed. This indicates to the fuzzer that another input should be tried.

Even though this is a very simple example, it already shows some promise. On our test system, Z3 takes approximately four seconds to solve this query, whereas applying a coverage-guided fuzzer to the program given in Listing 5.2 takes less than a second. It is worth noting however this is a cherry-picked example. First, the constraints were picked to be satisfiable. If the constraints were unsatisfiable the fuzzer would loop forever because no input would ever cause line 14 to be reached. Second, many different inputs satisfy the constraints so it quite easy for the fuzzer to find a satisfying assignment to the constraints.

5.3 Design and Implementation

We now discuss JFS, our prototype implementation of a constraint solver based on coverage-guided fuzzing as discussed in §5.1 and §5.2.

Figure 5.1: Architecture of JFS.

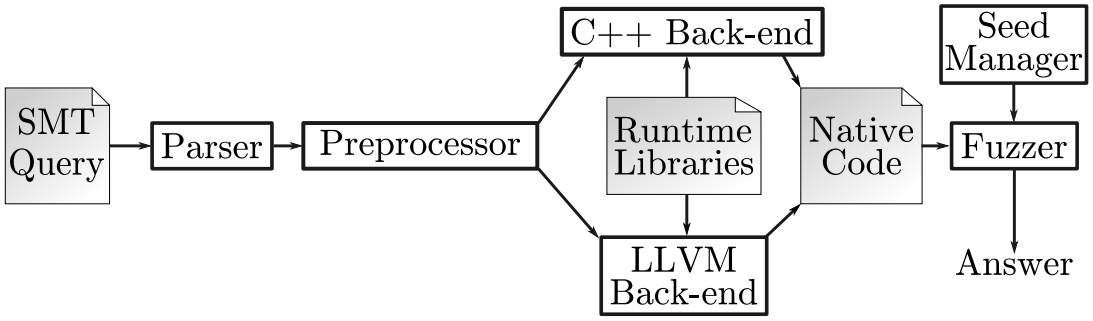


Figure 5.1 illustrates the architecture of JFS. JFS takes as input a file in the SMT-LIBv2.5 format which is consumed by the *parser*. Recall that the SMT-LIBv2.5 format consists of one or more constraints. The satisfiability problem stated in the format is the conjunction of these constraints, and as a consequence JFS takes as input a conjunction of constraints. The parser outputs an abstract syntax tree (AST) for each constraint in the query. These ASTs are then consumed by the *preprocessor*, which performs various transformations on the ASTs and extracts information from them that is useful for fuzzing. This information and the ASTs are then consumed by a *back-end*. A back-end handles generating a program from its input and invoking

a fuzzer on the generated program with seeds provided by the *seed manager*. The design of JFS allows for different back-ends to be used. This provides separation of concerns between front-end tasks (parser and preprocessor) and the back-end.

Figure 5.1 shows two back-ends, the *C++ back-end* and the *LLVM back-end*. The purpose of having different back-ends is to allow experimentation with different program generation and fuzzing methods. Both back-ends use different code generation methods that generate *native code*. These are discussed later in this section. After these back-ends have generated a program to be fuzzed, the program is then executed by a fuzzer. If the fuzzer finds a satisfying assignment, it terminates reporting the assignment, otherwise it continues to execute indefinitely unless bounded (e.g. by a timeout).

The generation of native code by JFS is "just-in-time" (JIT) in the sense that query-specific native code is generated in order to solve each query. This is why JIT appears in JFS's name. Generating native code allows us to re-use existing fuzzers built for fuzzing native code. This approach might also offer improved throughput (inputs tried per second) when compared to fuzzing constraints directly (i.e. constant folding ASTs) which has call indirection overhead and memory allocation overhead. Of course generating native code has some associated overhead as well, but this can be amortised over a sufficiently long fuzzing run.

The current implementation of JFS is written in C++11 and is built on top of several existing projects. The in-memory representation of constraints uses Z3's constraint language and corresponding API. This allows us to reuse Z3's parser and constraint simplification tactics. To compile native code, Clang and LLVM are used. To fuzz generated code, the coverage-guided mutation-based fuzzer LibFuzzer [119] is used.

We now discuss the details of the various components of JFS.

Parser. The parser parses queries written in the SMT-LIBv2 language. We use Z3's parser to parse queries and produce the in-memory representation of constraints (ASTs).

Preprocessor. The preprocessor is used to compute information from the ASTs and modify them to aid fuzzing. This is currently done in two stages: a *simplification* stage and then a *property extraction* stage that extracts information relevant to fuzzing.

The following simplification passes are currently implemented:

1. **And hoisting.** This separates the constraint `(and a b)` into two separate constraints `a` and `b` where `a` and `b` are both boolean expressions. We have implemented this externally from Z3.
2. **Constant propagation.** This detects free variables that have implied constant values and replaces all uses of those free variables with the implied constant values. This is implemented by using Z3's `propagate-values` tactic. To illustrate this, consider the expression `(= a #x00)` $\wedge$ `(bvugt a b)`. Constant propagation would infer that `a` has the value `#x00`[3] and that all uses of `a` should be replaced by this constant. Thus the resulting expression would be `(= a #x00)` $\wedge$ `(bvugt #x00 b)`.
3. **Duplicate constraint elimination.** This removes duplicate constraints from the constraint set. We have implemented this externally from Z3.
4. **Expression simplification.** This simplifies expressions by running Z3's expression simplifier. This performs actions such as constant folding.
5. **Simple contradictions to false.** This detects constraints of the form `a` and `(not a)` and replaces them with *false*. We have implemented this externally from Z3.
6. **True elimination.** This removes constraints of the form *true* from the constraint set. We have implemented this externally from Z3.

These passes are run in the order specified above to simplify the constraints as much as possible before fuzzing. After this, the *property extraction* stage runs. This stage consists of two passes. The *equality extraction* pass and *free variable to buffer assignment* pass.

The *equality extraction* pass is an optimisation based on the following observation. The fuzzer is very unlikely to guess inputs that satisfy equality constraints (e.g. `(= a #xff00ff00)` where `a` is a free variable of type 32-bit bit-vector). In these cases, the equality constraints can be removed and the program constructed in such a way that the constraint is always true, so that the fuzzer does not waste time trying to guess. This is a sound transformation. The pass works by walking over each constraint and gathering sets of equalities where each set contains the free variables and any constant values implied to be equal by the constraints. Each equality constraint used to modify the set of recorded equalities is removed from the set of constraints. The equality sets are then used during program generation and in the *free variable to buffer assignment pass*. Currently, this pass only supports finding equalities by looking for applications of the equality function with arguments that are free variables or constants. This pass could be extended in the future to handle the `fp.eq` function and arguments where a type cast is performed.

[3] This is a SMT-LIBv2.5 hexadecimal literal. It is an 8-bit wide bit-vector with value 0.

Listing 5.3: An example query in SMT-LIBv2.5 format to illustrate equality extraction.

```
1 (declare-fun a () (_ FloatingPoint 11 53))
2 (declare-fun b () (_ FloatingPoint 11 53))
3 (declare-fun c () (_ FloatingPoint 11 53))
4 (declare-fun d () (_ FloatingPoint 11 53))
5 (assert (= a b))
6 (assert (= b c))
7 (assert (= d (_ +zero 11 53)))
8 (assert (not (fp.isNaN (fp.add RNE c d))))
9 (check-sat)
```

To illustrate the *equality extraction* pass consider the constraints shown in Listing 5.3. The pass would infer two sets $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ and $\{\texttt{d}, 0.0\}$. For the first set, the constraints on lines 5 and 6 imply that the free variables `a`, `b`, and `c` must be equal to each other. For the second set, the constraint on line 7 implies that the free variable `d` must be equal to the constant `0.0`. The program that would be generated using the information gathered by the *equality extraction* pass is illustrated in Listing 5.4. There are several notable features here. First, only the free variable `a` is constructed from the buffer. The variables `b` and `c` are initialised to `a`'s value, and the `d` variable is initialised to be the constant `0.0`. Second, only one constraint is checked on line 9 (corresponds to line 8 in Listing 5.3). The other constraints are dropped because they always hold due to the way the program is constructed.

Listing 5.4: A translation of the constraints in Listing 5.3 to a C++ program using information gathered by the equality extraction pass.

```
1  int FuzzerTestOneInput(const uint8_t* data, size_t size) {
2    if (size != 8) return 0;
3    uint64_t numSolved = 0;
4    double a = makeFloatFrom(data, size, 0, 63);
5    double b = a;
6    double c = a;
7    double d = 0.0;
8    double c_plus_d = c + d;
9    if (!isnan(c_plus_d)) ++numSolved;
10   if (numSolved == 1) {
11     return 1; // TARGET REACHED
12   }
13   return 0;
14 }
```

The *free variable to buffer assignment* pass walks over the constraints and gathers all free variables. Each free variable is then assigned a chunk of the input buffer that the generated program will receive from the fuzzer, unless that free variable is part of an equality set (computed by the *equality extraction* pass described above). In this case, if the equality set that the free variable is a member of contains a constant it is not assigned a chunk of the buffer. At program generation time the free variable will be assigned the value of the constant. If the equality set that the free variable is a member of does not contain a constant, only one of the

free variables in that set will be assigned a chunk of the buffer. At program generation time all the other free variables in the equality set will be assigned the value of the free variable assigned a chunk of the buffer.

Currently chunks inside the buffer are tightly packed without consideration for alignment (i.e. there is no padding to make sure chunks are aligned to a byte boundary). Chunks are ordered by the order they appear while traversing ASTs. This makes the order deterministic (useful for reproducibility) but arbitrary. This strategy is sub-optimal for reading from the buffer because byte aligned access is usually faster. This will likely lead to sub-optimal throughput during fuzzing. However, it does optimise the number of wasted bits by keeping them to a minimum. A wasted bit is a bit that is not used to instantiate free variables during fuzzing. As such, when the fuzzer mutates those bits it will have no impact on the control flow of the program, and hence satisfiability of the constraints. Thus it is desirable to minimise the number of wasted bits. It is worth noting that the choice to optimise the number of wasted bits is due to our decision to use an off-the-shelf fuzzer. In principle the fuzzer's mutations could be modified to be aware of the structure of the fuzzing buffer so that wasted bits are not mutated or used in crossover mutations.

After all these passes run it is checked whether the resulting constraints are trivially satisfiable or unsatisfiable. If the resulting constraint set is empty that implies the constraints were satisfiable. If the resulting constraint set contains just *false* that implies the constraints are not satisfiable. In both cases there is no reason to run the fuzzer because satisfiability has already been proved and so JFS just returns the result immediately. In the case of satisfiability, the model for the resulting constraints can use any arbitrary assignment to the free variables. However, a satisfiable model for the resulting constraints does not necessarily satisfy the original constraints. This is because the equalities found by the equality extraction passes need to be enforced, and the simplification passes might remove free variables whose values are implied. To get a correct model, the model for the resulting constraints needs to be propagated through all the executed passes (in reverse) so that they can modify the model if necessary. A model that has been propagated through all the passes will then satisfy the original constraints. At the time of writing JFS does not implement model generation, but it is a feature that will be added in the future.

Runtime Libraries. Generated programs call functions from the *runtime libraries*, which implement the semantics of the `FloatingPoint` and `BitVector` types from the SMT-LIBv2 standard. The libraries also include functions to create these types from arbitrary chunks of the fuzzing buffer. There are several reasons for having runtime libraries rather than embedding the semantics directly into the generated program. First, the runtime libraries do not change between queries, so compilation time can be reduced by compiling them ahead of time (i.e. they are pre-compiled). Second, having the functions that implement the semantics of the SMT-LIBv2 theories in a runtime library means that can easily be tested. If the functions were not in a runtime library, and instead part of the generated program, it would make those functions difficult to test.

The current implementation of the runtime library only supports using natively supported types, thus the `FloatingPoint` types can only be 32-bit or 64-bit wide and `BitVector` types must be at most 64-bits wide. The implementation only supports rounding modes natively supported by `x86_64` architecture (all except round to nearest, ties to away from zero). For our initial study it was thought that restricting ourselves to natively supported types and rounding modes would be sufficient. In the future, the runtime libraries could be extended to emulate other SMT-LIBv2 types in software using arbitrary precision arithmetic libraries (e.g. LLVM's APInt for bit-vector types and GNU MPFR[4] for floating-point types). Doing this would be a very straightforward extension to JFS, however it is likely that the benefit of compiling query specific native code would be minimal due to the fact that native types are no longer being used. In this scenario we suspect that the performance of fuzzing programs that use arbitrary precision arithmetic libraries would be very similar to (or worse than) just fuzzing the in memory ASTs (i.e. the expressions representing constraints) directly.

C++ Back-end. The C++ Back-end generates a C++ program from the provided constraints and information gathered by the property extraction stage. This program is compiled by Clang instrumented with coverage instrumentation (for the fuzzer) and then linked against the runtime libraries and LibFuzzer. This produces a binary that JFS can then invoke.

The generated C++ program uses templated `Float<EB,SB>` and `BitVector<N>` types which correspond to the `(_ FloatingPoint EB SB)` and `(_ BitVec N)` types in SMT-LIBv2 respectively. The template parameters are used to pick[5] what implementation is used to implement the

[4] `http://www.mpfr.org/`
[5] Using `std::enable_if<>`

SMT-LIBv2 semantics at compile time. If the sizes are supported natively, then a native implementation of the semantics provided by the runtime libraries is used. Support for non-native sizes is currently not implemented and so a compilation error is triggered if they are used.

The reason for using this indirection via templated types is so that the implementation of the C++ program printer is decoupled from the implementation of the SMT-LIBv2 semantics. That is to say, that the C++ program printer does not need to know whether the SMT-LIBv2 types use a native or non-native implementation.

During early testing of JFS it was discovered that in some cases asking Clang to optimise generated programs resulted in long compile times that sometimes caused memory exhaustion. To avoid these issues, the current implementation of JFS does not optimise generated code by default.

There are many different ways to encode constraints as a program. The encoding we use when evaluating a single input is to evaluate all constraints, even if we realise during evaluation that some constraints are false. We refer to this encoding as the *try-all* encoding. This is the encoding used in Listing 5.2. Another possible encoding is the *fail-fast* encoding. Listing 5.5 shows the *fail-fast* encoding for the constraints in Listing 5.1. In this encoding, if any constraint branch is determined to be false then the current input is immediately discarded (`return 0` statements), without evaluating other constraint branches. When the fuzzer picks an input that satisfies a constraint for the first time, a new branch will be covered which will give the fuzzer feedback that the input is interesting. The advantage of the *try-all* encoding is that it doesn't matter in which order the program evaluates constraints because all of them are evaluated. This means that if an input satisfies any of the constraints, then the branch coverage will increase, regardless of the order of constraint evaluation. In contrast the *fail-fast* encoding is order sensitive because constraints have to be satisfied in a particular order for branch coverage to increase[6]. An advantage of the *fail-fast* encoding is that it might have higher throughput (number of inputs tried per second) in cases where there are a large number of constraints to evaluate. In our work we use the *try-all* encoding because we believe giving the fuzzer feedback is more important than optimising for throughput. We leave empirically comparing these encodings as a potential avenue for future work (see §6).

[6]JFS also supports this encoding

Listing 5.5: A translation of the constraints in Listing 5.1 to a C++ program using the *fail-fast* encoding.

```
int FuzzerTestOneInput(const uint8_t* data, size_t size) {
  if (size != 16) return 0;
  double a = makeFloatFrom(data, size, 0, 63);
  double b = makeFloatFrom(data, size, 64, 127);
  if (!isnan(a)) {} else { return 0; }
  if (!isnan(b)) {} else { return 0; }
  double a_b_rne = div_rne(a, b);
  double a_b_rtp = div_rtp(a, b);
  if (a_b_rne != a_b_rtp) {} else { return 0; }
  if (!isnan(a_b_rne)) {} else { return 0; }
  if (!isnan(a_b_rtp)) {} else { return 0; }
  return 1; // TARGET REACHED
}
```

LLVM Back-end. The LLVM back-end is currently not implemented but it is shown in Figure 5.1 to illustrate JFS's design. In principle the LLVM back-end would construct a program in memory as LLVM IR, optimise it, add coverage instrumentation, link it to the runtime libraries and LibFuzzer and then run it in-process using LLVM's JIT.

We suspect that this would provide a performance benefit by avoiding calls to `fork()` and also give very fine-grained control of how code is generated. However, for our prototype we did not implement this because this approach would be harder to debug and would require significant changes to LibFuzzer.

Seed Manager. The seed manager is in charge of deciding what seeds (initial input buffers to try) to give the fuzzer. The seeds given to the fuzzer can greatly affect how quickly the fuzzer finds a satisfying assignment (if one exists).

The seed manager allows an arbitrary number of *seed generators* to be attached. A seed generator generates a seed if it has available seeds left to supply, otherwise it reports that it has been exhausted. The seed manager iterates through the seed generators in a round-robin fashion until all seed generators are exhausted or a bound is reached. The bound can either be the maximum number of seeds or the maximum disk space used by the seeds. The intention of this design is to allow different seeding strategies to be tried in the future.

Currently we use a very simple strategy where we supply two seed generators. The first generates a single seed of the correct size where all bits are zero, and the other generates a single seed of the correct size where all bits are one. The motivation for doing this is that the fuzzer is very unlikely to guess the correct size for the inputs on its own if not given any seeds. The reason that we provide two seeds is so that there are enough seeds to perform a crossover mutation. Using more advanced seeding strategies is a potential avenue for future work (see §6).

Fuzzer. The fuzzer is currently implemented using LibFuzzer. The fuzzer is given a fixed time budget to fuzz the program and uses seeds provided by the seed manager. If the fuzzer finds its targets within the time budget it will report `sat`, otherwise it will report `unknown`.

There is one special case where the fuzzer is executed differently. If the size of fuzzing the buffer is zero (i.e. there are no free variables) the fuzzer will run a single iteration. If the target is reached the query is reported as satisfiable, otherwise it is reported as unsatisfiable. This special case occurs if the constant folding pass is disabled or fails to constant fold expressions that could be folded. We have observed failure to constant fold in several cases which is due to bugs in Z3.[7]

We currently use LibFuzzer's built-in mutators and do not provide our own. However, we have modified LibFuzzer's built-in mutators so that mutations that change the input size are not tried. Note however the buffer size check that is emitted at the beginning of generated programs is still required because during LibFuzzer's initialisation it always tries an input of zero size. Experimenting with different mutation strategies is a potential avenue for future work (see §6).

JFS's design in principle supports finding satisfying assignments to any theory using finite data types. However, our current implementation is restricted to any combination of the `Core` (i.e. Boolean), `FixedSizeBitVectors`, and `FloatingPoint` theories of SMT-LIBv2. A further restriction of the current implementation is that only a subset of the `BitVector` and `FloatingPoint` types (as mentioned above) are supported.

Despite these restrictions, it would be possible to use JFS as a constraint solver for the modified version of the KLEE symbolic execution engine presented in Chapter 4 that supported reasoning over floating-point programs. Although KLEE uses the theory of arrays which is not supported by JFS, the *array ackermannization* optimization implemented in Chapter 4 performs an equisatisfiable transform that removes uses of the array theory in certain cases. In the case that all uses of arrays are removed JFS could be used. Alternatively JFS could be modified to use Z3's `bvarray2uf` and `ackermannize_bv` tactics as a preprocessing step. These two tactics combined perform a similar transform to *array ackermannization*, except that array accesses at non-constant indices can be transformed.

[7] `https://github.com/Z3Prover/z3/issues/1242`

5.4 Evaluation

We now turn to the evaluation of JFS. We compare JFS against six state-of-the-art SMT solvers, all of which support solving floating-point constraints. These solvers are MathSAT5 [46] and Z3 [140], which are complete solvers based on converting floating-point operations into bit-vector operations and then bit-blasting to a SAT problem; CORAL [167], which is an incomplete solver based on using meta-heuristic search; goSAT [25] and XSat [77], which are incomplete solvers based on mathematical optimisation techniques; and COLIBRI [36] which is a complete solver, based on interval methods. These are discussed in more detail in §2.6.2.

First, we discuss the benchmark selection process (§5.4.1). Then we discuss details of our experimental set up (§5.4.3), after which we show and discuss the results of these experiments in the context of our hypothesis (§5.4.4).

5.4.1 Benchmark selection

We use several SMT-LIB benchmark suites as the basis for our benchmarks. A summary of these suites is shown in Table 5.1. The table shows three different benchmark suites, `QF_BV`,[8] `QF_FP`,[9] and `QF_BVFP`.[10] A subset of these benchmarks are used in the annual SMT solver competition, SMT-COMP.

`QF_BV` contains queries in the logic of quantifier free `BitVector` types. `QF_FP` contains queries in the logic of quantifier free `FloatingPoint` types. `QF_BVFP` contains queries in the logic of quantifier free `BitVector` and `FloatingPoint` types. All of these logics implicitly include the `Core` theory (Boolean type). Even though our primary goal is to solve floating-point constraints we include bit-vector constraints because JFS also supports them.

In Table 5.1 the columns describe the following information. **Suite** is the name of the benchmark suite, **Revision** is the Git SHA-1 hash corresponding to the revision we used, **Sat** is the number of benchmarks labelled as satisfiable in the suite, **Unsat** is the number of benchmarks labelled as unsatisfiable in the suite, **Unknown** is the number of benchmarks labelled as having unknown satisfiability in the suite, and **Total** is the total number of benchmarks in the suite.

[8] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV.git`
[9] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP.git`
[10] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BVFP.git`

Table 5.1: Table showing a summary of the SMT-LIB benchmark suites we use as a basis for our experiments.

Suite	Revision	Sat	Unsat	Unknown	Total
QF_BV	f7e691bf	11,280	20,991	136	32,407
QF_FP	3346ad7a	20,016	20,028	258	40,302
QF_BVFP	57d0c730	14,018	3174	23	17,215

In total across these three benchmark suites there are 89,924 benchmarks. Running this many benchmarks on a single machine (even with many cores) would lead to an unacceptably large upper bound on execution time given that (as we shall see in §5.4.3) the maximum allowed time for a single query is 900 seconds. Therefore the selection of a subset of these benchmarks is essential.

Our first step in picking a subset is to improve the benchmark labelling, which we will use in a later step. The label on a benchmark states its expected satisfiability (`sat`, `unsat`, or `unknown`). Across the benchmarks there are 417 benchmarks labelled as `unknown`. We ran both MathSAT5 and Z3 over these benchmarks, with a timeout of 900 seconds to try to relabel them as either `sat` or `unsat`. If either solver reports a benchmarks as being `sat` or `unsat` then we use that as the benchmark label unless the solvers report conflicting satisfiability. However, we did not observe any conflicts during this step. This step reduced the total number of benchmarks labelled as `unknown` to 172. Table 5.2 shows a summary of the relabelled benchmarks.

Table 5.2: Table showing summary of the SMT-LIB benchmark suites after relabelling.

Suite	Sat	Unsat	Unknown	Total
QF_BV	11,283	20,991	133	32,407
QF_FP	20,125	20,141	36	40,302
QF_BVFP	14,033	3179	3	17,215

The second step in picking a subset of benchmarks is to remove all benchmarks labelled as `unsat`. We do this because running JFS on unsatisfiable benchmarks serves no useful purpose. Recall that JFS is an incomplete solver, in particular it cannot prove unsatisfiability in non-trivial cases. Running JFS on unsatisfiable benchmarks in most cases would just lead to wasted compute time because the solver will not be able to find a satisfying assignment. This would lead to the solver running indefinitely or reaching a timeout if one is specified. Recall from §5.1 that to use JFS for arbitrary constraints (that might be unsatisfiable), that it would need to be paired with a complete solver. Although we could do this here, we choose not to because we wish to study JFS's performance in isolation. We refer to the remaining benchmarks after this second step, as the *unsat-filtered* benchmarks and the three filtered suites as $\texttt{QF\_BV}_{uf}$,

QF_FP$_{uf}$, and QF_BVFP$_{uf}$. This step leaves 45,613 benchmarks, which still has an unacceptably large upper bound on execution time, so we filtered the benchmarks further by sampling from them.

An obvious approach to try is to do uniform random sampling from the *unsat-filtered* benchmarks. This approach unfortunately would result in a very biased subset of benchmarks because the benchmark suites are not well distributed in terms of *difficultly*. Here we use the time to solve the query as the measure of *difficulty*.

We ran both MathSat5[11] and Z3[12] over the *unsat-filtered* benchmarks with a timeout of 900 seconds and recorded the wall time used by each solver. Although a timeout of 900 seconds would appear to have the same unacceptably large upper bound on execution time (which we are trying to avoid), it turns out many of the queries can be solved quickly by both MathSAT5 and Z3 which we quickly learnt when trying much smaller timeout values.

Figure 5.2 shows a histogram plot of Z3 wall clock execution time over the *unsat-filtered* QF_BV benchmark suite (QF_BV$_{uf}$). The histogram has five-second-wide-bins. The x-axis shows execution time (linear scale) and the y-axis shows histogram bin count (logarithmic scale). It can be seen that the execution time is very unevenly distributed. Some bins are incredibly large (e.g. 9946 queries were solved in under 5 seconds) and some are very small (e.g. only one query was solved in range of $[410, 415)$ seconds). We observed a very similar distribution of execution times when running with either Z3 or MathSat5 on each of the *unsat-filtered* benchmark suites.

These histograms illustrate why uniform random sampling from the set of *unsat-filtered* benchmarks is a bad approach: it is incredibly likely that benchmarks from the large bins will be selected. This would result in too many easy benchmarks being selected, resulting in a subset of benchmarks with very little diversity in terms of difficultly. Given that we would like to observe JFS's performance on benchmarks with a variety of difficulties, a different sampling strategy was required.

[11] Version 5.5.1
[12] Version 4.6

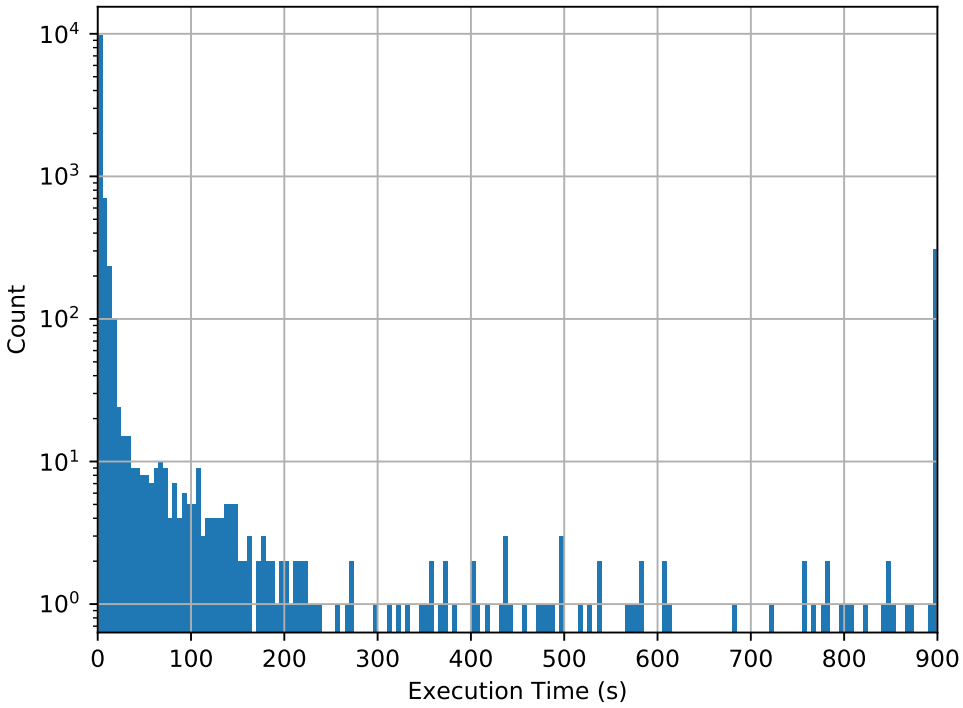


Figure 5.2: Histogram showing the approximate execution time distribution of Z3 on the *unsat-filtered* `QF_BV` benchmarks with a 900 seconds per query timeout.

We use a stratified random sampling method to sample from each of the *unsat-filtered* benchmark suites. The strata in this case are the histogram bins. The algorithm we use picks a benchmark by selecting a histogram by round-robin selection, then a bin from that histogram by uniform random selection, and then a benchmark from the selected bin by uniform random selection. The algorithm then repeats this process N times, to select N benchmarks. This sampling method makes it very unlikely that only benchmarks from large bins will be selected. Although we could make the probability of picking a bin inversely proportional to its height (thus likely avoiding selection from large bins), this would bias our selection to infrequently occurring execution times, which is not our goal.

Algorithm 1 Algorithm used to select a diverse subset of benchmarks.

```
1:  procedure SelectBenchmarks(S, N)
2:      B ← ∅                                              ▷ Set of benchmarks
3:      H ← [Z3(S), Mathsat5(S)]                           ▷ List of histograms
4:      while |B| < N ∧ |H| > 0 do
5:          i ← |B|
6:          hist ← H[i mod |H|]                            ▷ Select histogram
7:          if hist_is_empty(hist) then
8:              H ← remove_element(hist, H)                ▷ Remove histogram from list
9:              continue
10:         end if
11:         bin ←random_select_bin(hist)                   ▷ Select histogram bin
12:         if bin_is_empty(bin) then
13:             hist ←remove_bin(hist, bin)                ▷ Remove empty bin from histogram
14:             continue
15:         end if
16:         b ← random_remove_benchmark(bin)               ▷ Select & remove benchmark from bin
17:         B ← B ∪ {b}
18:     end while
19:     return B
20: end procedure
```

The algorithm we use is shown by the SelectBenchmarks procedure in Algorithm 1. The procedure SelectBenchmarks takes S, a benchmark suite (in our case QF_BV$_{uf}$, QF_FP$_{uf}$, or QF_BVFP$_{uf}$), and N, the number of benchmarks to select. On line 2, B is initialised as the empty set. Then on line 3, D is initialised as a list of histograms. Z3(S) is the histogram of measured Z3 execution times for the benchmark suite S and Mathsat5(S) is the histogram of measured MathSat5 execution times for the benchmark suite S. In our case all histograms use 5 second wide bins. Next on line 4, the algorithm loops until the required number of benchmarks have been selected or the list of histograms becomes empty. The body of the loop proceeds as follows. On line 6 the histogram to select from is chosen in a round-robin fashion. Then on line 7 it is checked whether the selected histogram is empty using the hist_is_empty(hist) function which returns true if and only if the given histogram contains no bins. If the selected histogram is empty, it is removed from the list of histograms and then the loop starts again. Next on line 11 a bin is randomly selected from the selected histogram with uniform random probability using the random_select_bin(hist) function. Then on line 12 it is checked whether the selected bin is empty using the bin_is_empty(bin) function. If the selected bin is empty then the selected histogram is updated to remove the empty bin using the remove_bin(hist, bin) function and then the loop starts again. Next on line 16 a benchmark is selected and removed from the selected bin with a uniform random probably using the random_remove_benchmark(bin)

function. Finally, on line 17 the set B is updated to contain the selected benchmark. Notice, that while selecting duplicate benchmarks is possible (due to there being multiple histograms containing the same benchmarks), collecting duplicate benchmarks is not due to B being a set.

We apply Algorithm 1 to each of the *unsat-filtered* benchmark suites with N set to 5% of the supplied benchmark suite (S), but rounded up to nearest multiple of two so that an equal number of benchmarks are selected from each histogram. Table 5.3 summarises the resulting benchmark suites which we refer to as the *final-sample* benchmarks and the three final suites as $\texttt{QF\_BV}_{fs}$, $\texttt{QF\_FP}_{fs}$, and $\texttt{QF\_BVFP}_{fs}$.

Table 5.3: Table showing summary of the SMT-LIB benchmark suites after relabelling and stratified random sampling.

Suite	Sat	Unsat	Unknown	Total
$\texttt{QF\_BV}_{fs}$	554	0	18	572
$\texttt{QF\_FP}_{fs}$	974	0	36	1010
$\texttt{QF\_BVFP}_{fs}$	699	0	3	702

5.4.2 Solver configuration

We compare JFS against six state-of-the art constraint solvers for floating-point constraints. We now discuss their configuration along with the configuration of JFS. Some solvers do not completely support the theories used in the benchmark suites and we note this where appropriate. These solvers are discussed in more detail in §2.6.2. For each of the solvers that support setting a random seed, we set a fixed value to try to get reproducible results.

COLIBRI COLIBRI [36] is a complete solver based on interval solving that supports the `FixedSizeBitVectors`, `FloatingPoint`, and `Core` theories. Thus we apply it to all three benchmark suites. We use revision 1572 available from `http://soprano-project.fr/download_colibri.html`. We have patched the bash wrapper script provided by the solver to be more compliant with the SMT-LIB standard. Specifically we output `unsat` when the solver result is unsatisfiable and only give a non-zero exit code when an error occurs. We use COLIBRI in its default `starexec_run_default` configuration.

CORAL CORAL [167] is an incomplete solver based on meta-heuristic search. The CORAL solver only supports floating-point constraints, hence we only apply it to the $\texttt{QF\_FP}_{fs}$ benchmarks. CORAL does not support the SMT-LIBv2.5 constraint format and instead has its own

constraint language. Furthermore, its constraint language lacks several features meaning it only supports a subset of the semantics of the `FloatingPoint` and `Core` theories. This makes comparing CORAL with other constraint solvers extremely challenging. Nevertheless we perform a best effort comparison by implementing a library (`smt2coral`[13]) to convert SMT-LIBv2.5 constraints into CORAL's constraint language. We then use this library to implement a wrapper tool that invokes this library to convert constraints and then pass them to CORAL to solve. Our translation of SMT-LIBv2.5 floating-point constraints to CORAL's constraint language is neither sound nor complete due to missing features in CORAL's constraint language. Specifically:

1. The `ite`, `fp.abs`, `fp.fma`, `fp.roundToIntegral`, `fp.min`, `fp.max` functions are not supported.
2. Only the `Bool`, `Float32`, and `Float64` sorts are supported.
3. Conversion operations between different floating-point sorts are not supported.
4. The `=`, `fp.neg`, `fp.isNegative`, and `fp.isPositive` functions are unsoundly translated. This is because the semantics of the SMT-LIBv2.5 `FloatingPoint` theory distinguishes between positive zero and negative zero. However, CORAL's constraint language provides no way to distinguish between the two zeros.
5. Only the "round to nearest, ties to even" rounding mode is supported.

CORAL has a large number of options available. We contacted the original authors for advice on the options to use which are as follows:

- `--cacheSolutions=false`
- `--simplifyUsingIntervalSolver=false`
- `--pcCanonicalization=false`
- `--removeSimpleEqualities=false`
- `--toggleValueInference=false`

The authors also advised that we use CORAL with the interval solver `realpaver` (version `0.4`[14]), so we compiled this (patched to work on `x86_64`) and configured CORAL to use it. The CORAL authors also recommended trying both the PSO (particle swarm optimisation) and AVM (alternating variable method) search methods, therefore we try both. We refer to the configurations as, CORAL-PSO and CORAL-AVM respectively. CORAL-PSO uses the `--nIterationsPSO=600` option to limit the number of iterations and CORAL-AVM uses the `--nIterationsAVM=20000` and `--nSelectionsAVM=10` options to limit the number of AVM iterations. Again, these options

[13] `https://github.com/delcypher/smt2coral`
[14] `http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`

were values suggested by the CORAL developers. In these configurations CORAL will exit after the number of requested iterations has been performed. Unfortunately, there is no way to force CORAL to run until a timeout has been reached. To workaround this, we increment the random seed for CORAL and start the search again if CORAL exits and a timeout has not yet been reached.

goSAT goSAT [25] is an incomplete solver that reformulates solving floating-point constraints as a mathematical optimisation problem and then applies off-the-shelf libraries to find a global minimum. The solver only supports the `FloatingPoint` and `Core` SMT-LIBv2 theories, so we only run it on the $\texttt{QF\_FP}_{fs}$ benchmarks.

We use goSAT version `b5a423cd4736bac13672b66218d7f63b10453bef` in its default configuration but with the `-smtlib-output` option so that its output is SMT-LIBv2.5 compliant. goSAT has several dependencies. We use nlopt[15] `2.4.2`, LLVM `4.0`, and Z3 `4.6.0`.

JFS We use JFS version `7caabce0feab0200652996aeb31dad420c5a611b`. JFS has several dependencies. We use LLVM `r325330`, Clang `r325114`, and compiler-rt `r324506` all on the `release_60` branch; and Z3 `4.6.0`.

We use the `try-all` branching encoding (`-branch-encoding=try-all`), with code optimisation disabled (`-O0`), and with naive seeds (`-sm-all-ones-seed`, `-sm-all-zeros-seed`, `-sm-max-num-seed=2`).

JFS supports the `FloatingPoint`, `BitVector`, and `Core` theories, but with some caveats as explained in §5.3. Thus we apply JFS to all three benchmark suites.

MathSAT5 MathSAT5 [46] is a complete solver, based on bit-blasting constraints to a SAT problem which is then solved by a SAT solver. We use MathSAT5 version `5.5.1`. MathSAT5 supports the `FloatingPoint`, `FixedSizeBitVectors`, and `Core` theories. Therefore we apply it to all three benchmarks suites. MathSAT5 comes with a set of configurations (`smtcomp2015_main.txt`) for several different SMT-LIBv2.5 logics. We apply the appropriate configuration for each of the three benchmark suites.

[15] `https://nlopt.readthedocs.io/en/latest/`

XSat XSat [77] is an incomplete solver which operates on the same principles as goSAT. It only supports the `FloatingPoint` and `Core` theories, so we only run it on the $\texttt{QF\_FP}_{fs}$ benchmarks.

XSat has not made any official public releases. However, a version of the solver was uploaded to STAR-EXEC[16] servers for the 2017 SMT-COMP competition, and so this is the version we use.

Z3 Z3 [140] is a complete solver based on bit-blasting constraints to a SAT problem which is then solved by a SAT solver. Z3 supports the `FloatingPoint`, `FixedSizeBitVectors`, and `Core` theories, so we apply it to all three benchmark suites. We use Z3 version `4.6.0`. We use Z3 in its default configuration.

5.4.3 Experimental set up

We ran the eight configurations (seven solvers, with CORAL in two configurations) on a machine with two Intel® Xeon® E5-2450 v2 CPUs (8 physical cores each) with 256GiB of RAM running Ubuntu `16.04LTS`. Each solver was run five times per benchmark with a timeout of 900 seconds for each run. The repeat runs of a solver are used to compute average execution time and observe non-deterministic behaviour. Each solver was executed in parallel over the set of benchmarks, running on at most 13 benchmarks in parallel. This was done to minimise the time taken to perform experiments.

It is important that solver execution times are reproducible. To aid in this the following steps were taken:

- Each solver was executed in a Docker [132] container to keep the running solvers isolated from each other and to control their allocated resources.
- Each container was pinned to a single CPU core. The CPU cores used for pinning were isolated from system scheduling using systemd's `CPUAffinity` parameter. This means that each solver gets exclusive access to one CPU. This partially mitigates other processes running (including other solvers) on the system interfering with the running solver.
- Each container was pinned to its CPU's nearest NUMA node. This avoids the container using a mix of different NUMA nodes which can lead to non-reproducible execution times.
- CPU hyper-threading was disabled. This is required to make CPU pinning effective because when hyper-threading is enabled, one physical CPU core pretends to be two separate CPU cores.
- CPU turbo boost was disabled. CPU turbo boost can cause large changes in the CPU clock speed during solver execution which can lead to non-reproducible execution times.

[16] `https://www.starexec.org/`

- The `pstate` CPU governor was set to "performance" requesting the same min/max frequency (2.5GHz). Using these settings tries to avoid changing the CPU clock speed as much as possible.
- Each container had a 10 GiB memory limit enforced and it was requested that no swap space be used. Using a memory limit is done to prevent any one solver from using more than its fair share of the available memory.
- Address space layout randomisation (ASLR) was disabled. ASLR can cause behaviour differences in solvers that rely on particular memory addresses.
- As noted in §5.4.2, each solver had a fixed random seed set, if supported.

To combine repeat runs of a solver on the same benchmark we took the following approach. If any run on the same benchmark is reported as satisfiable or unsatisfiable and there are no conflicts, then it is counted as having given the correct result. If the reported satisfiability does not match the benchmark's label and the benchmark's label is not `unknown`, then the solver is counted as having given a wrong result. If a run on the same benchmark is reported as a mixture of satisfiable and unsatisfiable, and the label of the benchmark is not `unknown`, then the solver is treated as having given the wrong result. Otherwise the solver is counted as reporting the benchmark as unknown. To combine the execution times (wall clock time) the arithmetic mean and confidence intervals (99.9%) are computed. When comparing execution times between solvers, mean execution times are only considered distinguishable if their confidence intervals do not overlap.

5.4.4 Results

We now discuss the results of our experiments. First, we look at the satisfiability results reported by each solver, for the three different benchmark suites. Second, we look at JFS's similarity, complementarity and limitations in terms of benchmark satisfiability, with respect to other solvers on each of the three benchmark suites. Third, we compare JFS's runtime performance to other solvers for the three different benchmarks suites. Finally we discuss these results in the context of our hypothesis (defined in §5.1).

Satisfiability results

Table 5.4: Table summarising satisfiability results reported by each solver for the $\texttt{QF\_BV}_{fs}$ benchmarks.

Solver	Sat	Unsat	Wrong result	Unknown
COLIBRI	65	0	0	507
JFS	21	0	0	551
MathSAT5	424	0	0	148
Z3	497	0	0	75

Table 5.4 shows how the benchmarks in the $\texttt{QF\_BV}_{fs}$ suite were reported by the COLIBRI, JFS, MathSAT5, and Z3 solvers. The CORAL, goSAT, and XSat solvers cannot be applied to this suite, hence their absence (see §5.4.2). The **Sat** and **Unsat** columns show the number of benchmarks that a solver reported as satisfiable or as unsatisfiable, respectively. The **Wrong result** column shows the number of benchmarks where a solver reported the satisfiability to be the opposite of the existing benchmark label. Finally, the **Unknown** column shows the number of benchmarks where a solver failed to give a result. From the table it can be seen that Z3 is the clear winner, reporting 497 benchmarks as satisfiable. MathSAT5 closely follows by reporting 424 benchmarks as satisfiable. The COLIBRI and JFS solvers reported a very low number of benchmarks as satisfiable, with JFS doing the worst. None of the benchmarks were reported as unsatisfiable and none of the solvers gave a wrong result. None of the solvers gave conflicting results.

Table 5.5: Table summarising the different reasons for each solver failing to give a result for the $\texttt{QF\_BV}_{fs}$ benchmarks.

Solver	Crash	Out of memory	Timeout	Unknown	Unsupported
COLIBRI	16	0	458	33	0
JFS	0	0	504	0	47
MathSAT5	0	19	129	0	0
Z3	0	13	62	0	0

All solvers failed to give a result for some of the $\texttt{QF\_BV}_{fs}$ benchmarks. Table 5.5 summarises the reasons for the solvers not giving a result. The **Crash** column shows for each solver, the number of benchmarks that resulted in the solver crashing. The **Out of memory** column shows for each solver, the numbers of benchmarks that resulted in the memory limit being reached. The **Unknown** column shows for each solver, the number of benchmarks where the reason for failure could not be inferred. These cases are essentially the solver exiting, without hitting any resource limit with a non-zero exit code that does not indicate a segmentation fault (one type of crash) and with empty output. Finally, the **Unsupported** column shows, for each solver, the

number of benchmarks where the solver reported that it did not support the benchmark. Here JFS reports that it does not support 47 benchmarks due to them using bit-vectors that are wider than it currently supports. This is only a limitation of JFS's current implementation because support for wider bit-vectors could be added in the future.

Unfortunately these results show that JFS is not very competitive on the $\texttt{QF\_BV}_{fs}$ benchmarks. However, we shall see later in this section that JFS can complement every solver on this benchmark suite.

Table 5.6: Table summarising satisfiability results reported by each solver for the $\texttt{QF\_BVFP}_{fs}$ benchmarks.

Solver	Sat	Unsat	Wrong result	Unknown
COLIBRI	666	1	6	29
JFS	682	0	0	20
MathSAT5	699	0	0	3
Z3	699	0	0	3

Table 5.6 shows how the benchmarks in the $\texttt{QF\_BVFP}_{fs}$ suite were reported by the COLIBRI, JFS, MathSAT5, and Z3 solvers. The CORAL, goSAT, and XSat solvers cannot be applied to this suite, hence their absence (see §5.4.2). The table uses the same columns as Table 5.4, whose meanings were previously described. From the table it can be seen that both Z3 and MathSAT5 found 699 benchmarks to be satisfiable. This is closely followed by JFS, which found 682 benchmarks to be satisfiable. Finally, COLIBRI found fewer, but a respectable 666 benchmarks to be satisfiable. COLIBRI actually found seven benchmarks to be unsatisfiable. For one benchmark the label is `unknown` and no other solver was able to contradict COLIBRI, so, according to our methodology, we consider this result to be correct. However, for six benchmarks COLIBRI reported the benchmarks as unsatisfiable even though the label stated the benchmarks are satisfiable. For three of those benchmarks, JFS reported them as satisfiable, and for all of those benchmarks MathSAT5, and Z3 reported them as satisfiable. This indicates that COLIBRI's results are wrong which suggests that its implementation either contains bugs or is unsound. None of the other solvers reported benchmarks as being unsatisfiable or gave wrong results.

Table 5.7: Table summarising the different reasons for each solver failing to give a result for the $\texttt{QF\_BVFP}_{fs}$ benchmarks.

Solver	Crash	Out of memory	Timeout	Unknown	Unsupported
COLIBRI	0	0	12	17	0
JFS	0	0	18	0	2
MathSAT5	0	0	3	0	0
Z3	0	0	3	0	0

All solvers failed to give a result for some of the QF_BVFP$_{fs}$ benchmarks. Table 5.7 summarises the reasons for the solvers not giving a result. The columns have the same meaning as in Table 5.5. Here JFS reports that it does not support two benchmarks due to them using floating-point expressions with widths it does not currently support. This is only a limitation of JFS's current implementation because support for other floating-point widths could be added in the future.

These results show that JFS is very competitive with the other solvers in terms of the number of benchmarks solved on the QF_BVFP$_{fs}$ benchmark suite. It does not beat every solver on these terms, however we will see later in this section that JFS can also complement every solver on this benchmark suite.

Table 5.8: Table summarising satisfiability results reported by each solver for the QF_FP$_{fs}$ benchmarks.

Solver	Sat	Unsat	Wrong result	Unknown
COLIBRI	956	3	5	46
CORAL-AVM	19	0	0	991
CORAL-PSO	48	0	0	962
goSAT	73	0	0	937
JFS	949	0	0	61
MathSAT5	849	1	0	160
XSat	228	17	108	657
Z3	951	0	0	59

Table 5.8 shows the how benchmarks in the QF_FP$_{fs}$ suite were reported by the COLIBRI, CORAL (in two configurations), goSAT, JFS, MathSAT5, XSat, and Z3 solvers. The table uses the same columns as Table 5.4, whose meanings were previously described. From the table it can be seen that COLIBRI reported the most number of benchmarks as satisfiable (956), followed closely by Z3 (951) and JFS (949). MathSAT5 found fewer benchmarks (849) to be satisfiable. The remaining solvers XSat (228), goSAT (73), CORAL-PSO (48), and CORAL-AVM (19) found significantly fewer benchmarks to be satisfiable.

The COLIBRI, MathSAT5, and XSat solvers reported some benchmarks as unsatisfiable.

XSat reported 125 benchmarks to be unsatisfiable. For 17 benchmarks, XSat reported them as unsatisfiable where the benchmark label was `unknown`. For these benchmarks neither MathSAT5, JFS, nor Z3 were able to contradict these results. However, goSAT contradicts some of these by reporting three of these benchmarks as satisfiable. COLIBRI agrees with XSat on two of 17 benchmarks. However, given the relative immaturity of COLIBRI and goSAT (compared

to MathSAT5 and Z3) we chose not to trust these results, so we continue to use the existing labels and say that XSat correctly reported these benchmarks as unsatisfiable. For 108 benchmarks, XSat reports them as unsatisfiable where the existing benchmark labels state that they should be satisfiable. On all these benchmarks either MathSAT5, JFS, or Z3 (or a combination of these) were able to contradict XSat by reporting these benchmarks as satisfiable. These are wrong results which demonstrates XSat's unsound behaviour. When XSat's search fails to find a solution within a reasonable period of time, it reports a benchmark as unsatisfiable even though it has no proof that this is the case.

COLIBRI reported eight benchmarks to be unsatisfiable. For three of these benchmarks, COLIBRI reported them as unsatisfiable where the benchmark label was `unknown`. No other solver was able to contradict these results so we say that COLIBRI correctly reported them as unsatisfiable. However, COLIBRI reported five benchmarks as unsatisfiable where the benchmark label stated that they should be satisfiable. For these five benchmarks either MathSAT5 or Z3 (or sometimes both) contradicted COLIBRI's results. Therefore for these five benchmarks we conclude that COLIBRI gave a wrong result. This demonstrates that COLIBRI either contains implementation bugs or is unsound.

MathSAT5 reported one benchmark as unsatisfiable where the benchmark label was `unknown` and no other solver was able to contradict this result. Therefore we say that MathSAT5 correctly reported this benchmark as unsatisfiable.

Table 5.9: Table summarising the different reasons for each solver failing to give a result for the $\texttt{QF\_FP}_{fs}$ benchmarks.

Solver	Crash	Out of memory	Timeout	Unknown	Unsupported
COLIBRI	1	0	36	1	8
CORAL-AVM	344	0	52	0	595
CORAL-PSO	344	0	23	0	595
goSAT	159	0	0	469	309
JFS	0	0	61	0	0
MathSAT5	0	0	35	0	125
XSat	645	0	12	0	0
Z3	0	4	55	0	0

All solvers failed to give a result for some of the $\texttt{QF\_FP}_{fs}$ benchmarks. Table 5.9 summarises the reasons for the solvers not giving a result. The columns have the same meaning as in Table 5.5. Many solvers reported some benchmarks as being unsupported. COLIBRI failed to parse one benchmark. For the CORAL-AVM and CORAL-PSO solvers, 594 benchmarks could not be converted from SMT-LIBv2.5 constraints to CORAL's constraint language using the `smt2coral` tool.

This is due to there being no way to perform the conversion due to CORAL's constraint language not being rich enough to support the semantics of the SMT-LIBv2.5 `FloatingPoint` theory. The other benchmark not supported by CORAL-AVM and CORAL-PSO is a benchmark that is too large to be passed to CORAL when converted by the `smt2coral` tool. The goSAT solver reports that it doesn't support 309 benchmarks. The MathSAT5 solver reports that it doesn't support 125 benchmarks. This is due to MathSAT5 not supporting the `fp.fma` and `fp.rem` functions from the `FloatingPoint` theory. It is this lack of support that caused MathSAT5 to be ranked under JFS in terms of the number of benchmarks shown to be satisfied.

These results show that JFS is very competitive with the other solvers in terms of the number of benchmarks solved on the $\texttt{QF\_FP}_{fs}$ benchmark suite. In fact it beats MathSAT5, a state-of-the-art and widely used constraint solver. However, as noted above the reason here is due to MathSAT5 not supporting some functions from the SMT-LIBv2.5 `FloatingPoint` theory. JFS does not beat every solver on these terms, however we will see later in this section that JFS can also complement every solver on this benchmark suite.

Satisfiability similarity, complementarity and limitations

So far we have only examined the absolute number of benchmarks solved by each solver. We now examine the number of benchmarks that can be shown to be satisfiable by both JFS and another solver (i.e. similarity), by JFS and not the other solver (JFS complementing another solver), not by JFS but by the other solver (a limitation of JFS), and neither (a limitation of both solvers).

Table 5.10: Table summarising the satisfiability intersection, differences, and limitations for JFS compared to other solvers for the $\texttt{QF\_BV}_{fs}$ benchmarks.

Solver	Both	Only JFS	Only other	Neither
COLIBRI	17 (2.97%)	4 (0.70%)	48 (8.39%)	503 (87.94%)
MathSAT5	21 (3.67%)	0 (0.00%)	403 (70.45%)	148 (25.87%)
Z3	20 (3.50%)	1 (0.17%)	477 (83.39%)	74 (12.94%)
All above	21 (3.67%)	0 (0.00%)	520 (90.91%)	31 (5.42%)

Table 5.10 shows JFS's capability, complementarity, and limitations for the $\texttt{QF\_BV}_{fs}$ benchmarks. The **Both** column states the number of benchmarks shown to be satisfiable by both JFS and the other solver. The **Only JFS** column shows the number of benchmarks that were shown to be satisfiable by JFS and not by the other solver. The **Only other** column shows the number of benchmarks that were shown to be satisfiable by the other solver and not by JFS. The **Neither** column shows the number of benchmarks that were shown to be satisfiable by neither JFS nor the other solver. Each row of the table corresponds to the *other* solver (specified by the **Solver** column). The "All above" row has a special meaning and is a combination of all the above results. For the "All above" row, the **Both** table cell is the union of all benchmarks that both JFS and another solver managed to solve (i.e. it is a union of intersections, not an intersection of intersections). For this row, the **Only JFS** table cell is the number of benchmarks found satisfiable by JFS and none of the other solvers. For this row, the **Only other** table cell is the union of all benchmarks found to be satisfiable by another solver and not JFS. For this row the **Neither** table cell is the number of benchmarks not found satisfiable by any solver. From this table we can see that there is very little similarity between JFS and other solvers (**Both** column) in terms of the benchmarks solved. However, the table shows that JFS complements two of the solvers by solving a few benchmarks that another solver does not (**Only JFS** column). JFS does not complement a combination of all the other solvers however. In terms of limitations of JFS, it can be seen that most of the benchmarks are in the **Only other** column for MathSAT5 and Z3, and in the **Neither** column for COLIBRI.

These results show that on the $\texttt{QF\_BV}_{fs}$ benchmarks JFS occasionally complements an existing solver, but for the majority of benchmarks other solvers are superior.

Table 5.11: Table summarising the satisfiability intersection, differences, and limitations for JFS compared to other solvers for the $\texttt{QF\_BVFP}_{fs}$ benchmarks.

Solver	Both	Only JFS	Only other	Neither
COLIBRI	660 (94.02%)	22 (3.13%)	6 (0.85%)	14 (1.99%)
MathSAT5	682 (97.15%)	0 (0.00%)	17 (2.42%)	3 (0.43%)
Z3	682 (97.15%)	0 (0.00%)	17 (2.42%)	3 (0.43%)
All above	682 (97.15%)	0 (0.00%)	17 (2.42%)	3 (0.43%)

Table 5.11 shows JFS's capability, complementarity, and limitations for the $\texttt{QF\_BVFP}_{fs}$ benchmarks. The columns have the same meanings as Table 5.10, which were previously discussed. From this table we can see a great deal of similarity between the solvers with the majority of benchmarks being solved by both JFS and another solver. In term of complementarity, we can see that JFS solves 22 benchmarks that COLIBRI does not. However, JFS does not complement

the MathSAT5 or Z3 solvers. In terms of JFS's limitations we can see that there are 17 benchmarks that JFS fails to solve that at least one of the other solvers manage to solve. The table also shows that there is a very large overlap in the capability of JFS, MathSAT5, and Z3; and that the number of benchmarks not solved by any solver is very small compared to the other benchmark suites. This suggests that the $\texttt{QF\_BVFP}_{fs}$ benchmark suite might not be challenging enough to illustrate the different capabilities of the tools. However, later in this section we will see that the runtime of these solvers on this benchmark suite is sometimes non-trivial, which indicates that the benchmark suite does contain challenging benchmarks.

These results show that JFS is very competitive with other solvers on the $\texttt{QF\_BVFP}_{fs}$ benchmarks and is able to complement the COLIBRI solver. However, JFS does show some limitations.

Table 5.12: Table summarising the satisfiability intersection, differences, and limitations for JFS compared to other solvers for the $\texttt{QF\_FP}_{fs}$ benchmarks.

Solver	Both	Only JFS	Only other	Neither
COLIBRI	938 (92.87%)	11 (1.09%)	18 (1.78%)	43 (4.26%)
CORAL-AVM	18 (1.78%)	931 (92.18%)	1 (0.10%)	60 (5.94%)
CORAL-PSO	36 (3.56%)	913 (90.40%)	12 (1.19%)	49 (4.85%)
goSAT	55 (5.45%)	894 (88.51%)	18 (1.78%)	43 (4.26%)
MathSAT5	824 (81.58%)	125 (12.38%)	25 (2.48%)	36 (3.56%)
XSat	213 (21.09%)	736 (72.87%)	15 (1.49%)	46 (4.55%)
Z3	939 (92.97%)	10 (0.99%)	12 (1.19%)	49 (4.85%)
All above	949 (93.96%)	0 (0.00%)	35 (3.47%)	26 (2.57%)

Table 5.12 shows JFS's capability, complementarity, and limitations for the $\texttt{QF\_FP}_{fs}$ benchmarks. The columns have the same meanings as Table 5.10, which were previously discussed. From this table we can see a great deal of similarity between the Z3 and COLIBRI solvers, and a high (but notably less) amount of similarity with MathSAT5. The similarity with the other search-based solvers (CORAL-AVM, CORAL-PSO, goSAT, and XSat) is quite low. In terms of complementarity, JFS complements every solver by finding benchmarks to be satisfiable that the other solver does not. However, JFS does not find any benchmarks to be satisfiable that a combination of all other solvers would. For the search-based solvers (CORAL, goSAT, JFS, and XSat) JFS finds many benchmarks to be satisfiable that the other solver does not. This shows that out of the all the search-based solvers, JFS is the most competitive, at least for the benchmark we have compared on. In terms of limitations, every solver finds some benchmarks to be satisfiable that JFS does not (i.e. every solver is able to complement JFS). There are also some benchmarks that neither JFS, nor another solver manage to show as satisfiable.

These results show that JFS is very competitive with other solvers on the $\texttt{QF\_FP}_{fs}$ benchmarks and is able to complement every solver, while showing some limitations.

Runtime performance

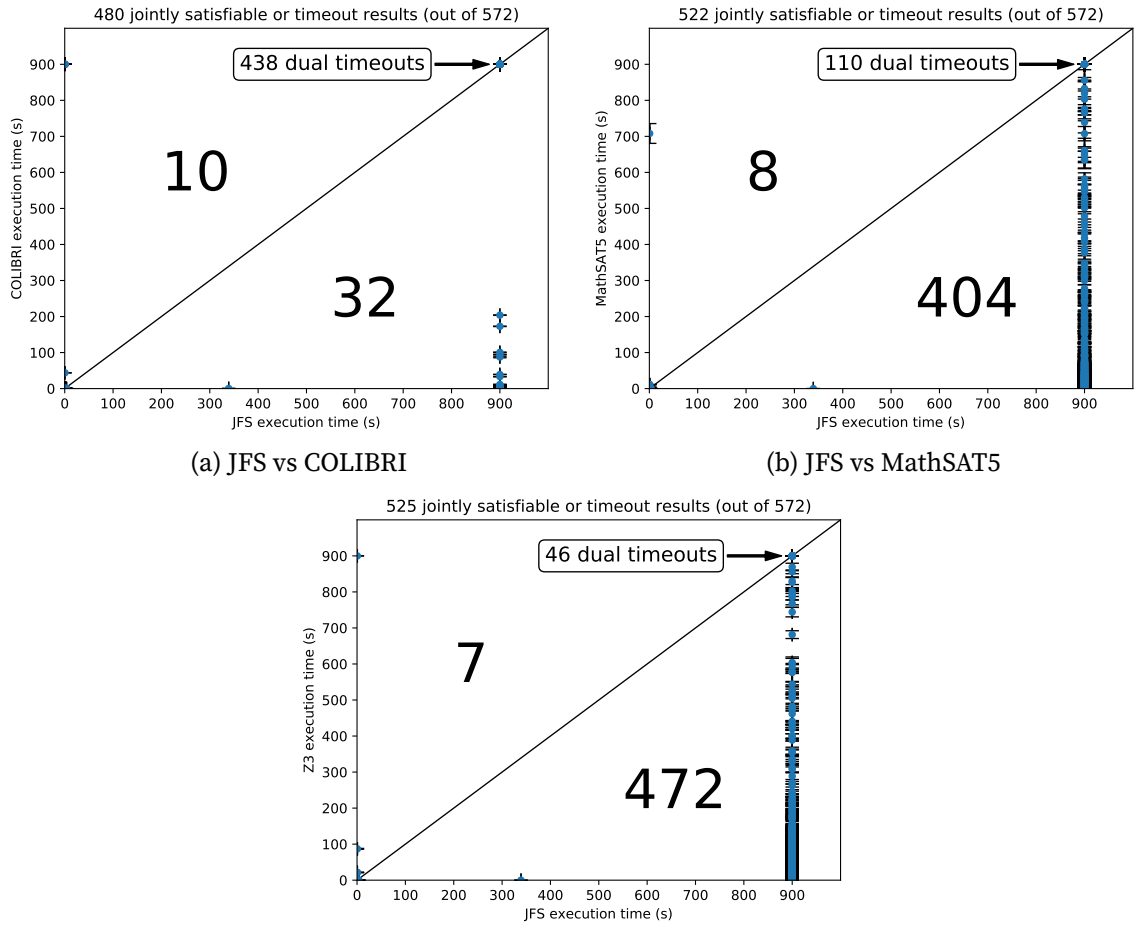
So far we have only looked at how many benchmarks were found to be satisfiable by the solvers in our study. Another important aspect is how quickly the solvers are able to solve the benchmarks. We now compare the wall clock execution times of JFS compared to every other solver for each of the three benchmark suites.

Figure 5.3 shows a set of scatter graphs comparing JFS against COLIBRI (5.3a), MathSAT5 (5.3b), and Z3 (5.3c) on the $\texttt{QF\_BV}_{fs}$ benchmarks. For every benchmark where both JFS and the other solver found the benchmark to be satisfiable or reached a timeout we plot a point. The number of points is shown above each graph (e.g. for Figure 5.3a 480 benchmarks were found to be satisfiable by both JFS and COLIBRI or a timeout was reached). We exclude all other cases because it does not make sense to compare execution times if one of the solvers crashed or gave the wrong result. A point at position (x, y) indicates that a benchmark took on average x seconds for JFS to show satisfiability and y seconds for the other solver to show satisfiability. If a solver reached a timeout its ordinate value is 900 seconds. A benchmark where both solvers reached a timeout is shown as a point at $(900, 900)$. In most cases there are multiple benchmarks where this occurs and this leads to multiple points being drawn at the same location. To allow the reader to see the number of benchmarks for which this occurs we explicitly annotate these points, stating the number of "dual timeouts". All points include error bars which show the upper and lower confidence intervals. Each graph has a diagonal line ($y = x$). Points that are above the line (i.e. $x < y$) are benchmarks where JFS was faster than the other solver. Points that are below the line (i.e. $x > y$) are benchmarks where the other solver was faster than JFS. The graphs show the number of benchmarks for which one solver was faster than the other. The number written above the diagonal line is the number of benchmarks where JFS was faster and the number written below the diagonal line is the number of benchmarks where the other solver was faster. For example for Figure 5.3a JFS was faster than COLIBRI for 10 benchmarks but COLIBRI was faster than JFS for 32 benchmarks. Note that we take confidence intervals into account when computing these numbers. If the confidence intervals of the two solver's execution times overlap, then we consider them incomparable.

From looking at the graphs in Figure 5.3 we can make several interesting observations. First, COLIBRI, MathSAT5, and Z3 are faster more often for these benchmarks. Second, there are cases where JFS is faster than the other solvers. This illustrates that JFS could be used to complement existing solvers. Third, the points are all very close to the axes (i.e. $x = 0$ and $y = 0$ lines) or solver timeout lines (i.e. $x = 900$ and $y = 900$). In most cases either JFS solves the problem very quickly with the other solver struggling, or the opposite. This again suggests that JFS is very much complementary to the other solvers in terms of runtime performance. Finally, recall from earlier in this section that Table 5.11 might suggest that the $\texttt{QF\_BVFP}_{fs}$ benchmarks are not challenging enough. However, we can see from Figure 5.4 that several of the benchmarks took a non-trivial (i.e. $>$ 10 seconds) amount of time to solve which indicates that at least some subset of the benchmarks are challenging.

Figure 5.4 shows a set of scatter graphs comparing JFS against COLIBRI (5.4a), MathSAT5 (5.4b), and Z3 (5.3c) on the $\texttt{QF\_BVFP}_{fs}$ benchmarks. For these graphs we can make the same observations that we did for the graphs in Figure 5.4. Those observations show that although the other solvers are more frequently faster than JFS, there are cases where JFS is faster, and in those cases JFS is complementary.

Figure 5.5 shows a set of scatter graphs comparing JFS against COLIBRI (5.5a), CORAL-AVM (5.5b), CORAL-PSO (5.5c), goSAT (5.5d), MathSAT5 (5.5e), XSat (5.5f), and Z3 (5.5g) on the $\texttt{QF\_FP}_{fs}$ benchmarks. We can make several interesting observations from these graphs. First, except for goSAT, JFS is faster more often ($> 70\%$) than the other solver. Second, the error bars for XSat's execution time are very large. We discovered that this is because XSat exhibits very non-deterministic behaviour, despite our best efforts. We did not provide it with a random seed because we could not find an option to do this, we suspect that this is the source of this non-deterministic behaviour. These large error bars do not affect our observations because they occur in cases where JFS reached a timeout so it is clear in those cases that XSat was faster. Finally, most of the points are close to the axes or the solver timeout lines (i.e. $x = 900$ and $y = 900$). This shows that in most cases either JFS is much faster than the other solver, or vice versa. This observation shows that JFS is complementary to the other solvers in terms of execution time.

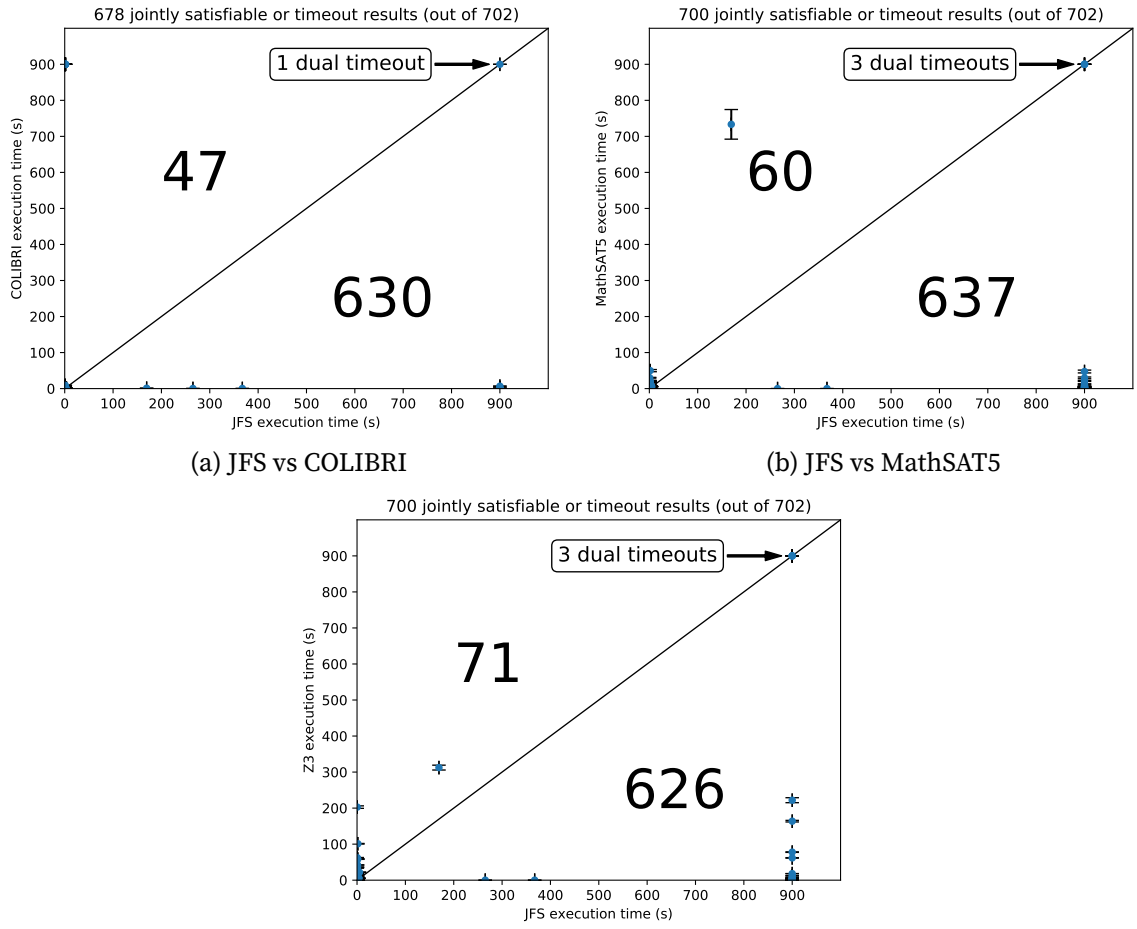


(a) JFS vs COLIBRI

(b) JFS vs MathSAT5

(c) JFS vs Z3

Figure 5.3: Scatter plots comparing JFS's execution time with that of other solvers on the $\texttt{QF\_BV}_{fs}$ benchmarks.

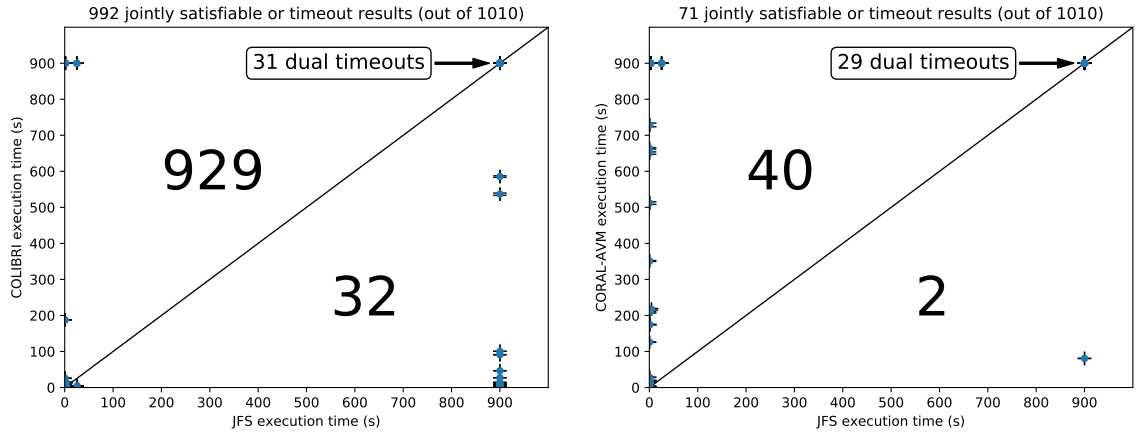


(a) JFS vs COLIBRI

(b) JFS vs MathSAT5

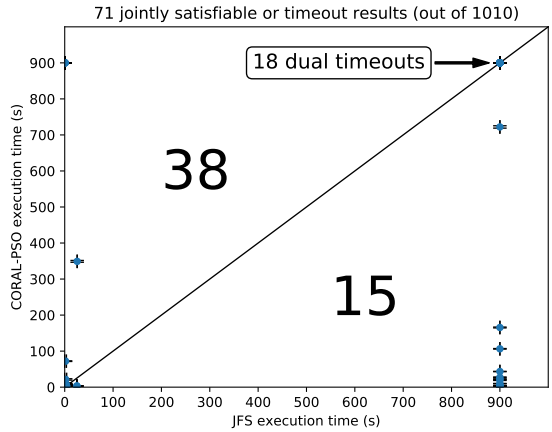
(c) JFS vs Z3

Figure 5.4: Scatter plots comparing JFS's execution time with that of other solvers on the $\texttt{QF\_BVFP}_{fs}$ benchmarks.

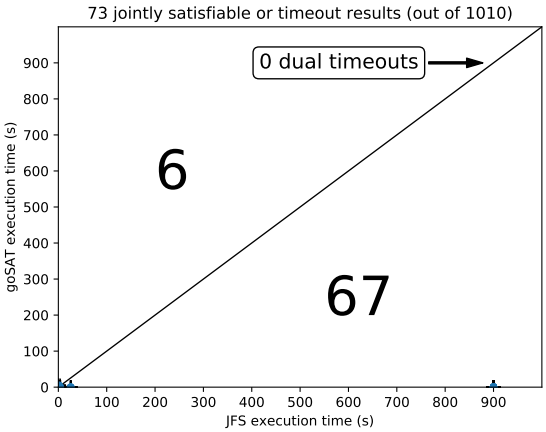


(a) JFS vs COLIBRI

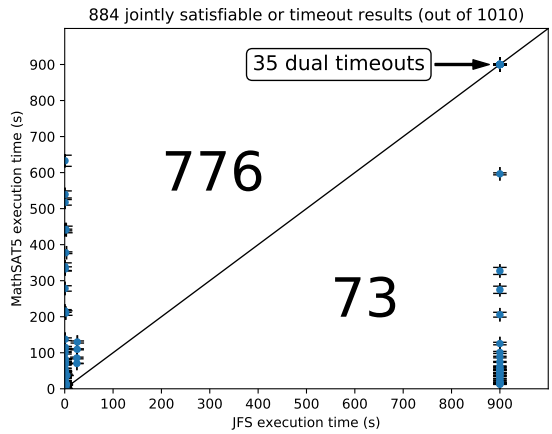
(b) JFS vs CORAL-AVM



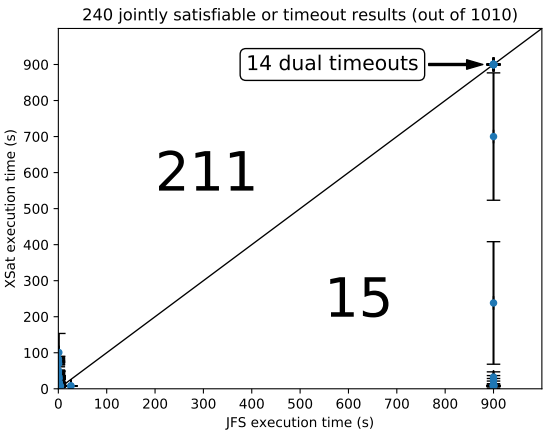
(c) JFS vs CORAL-PSO



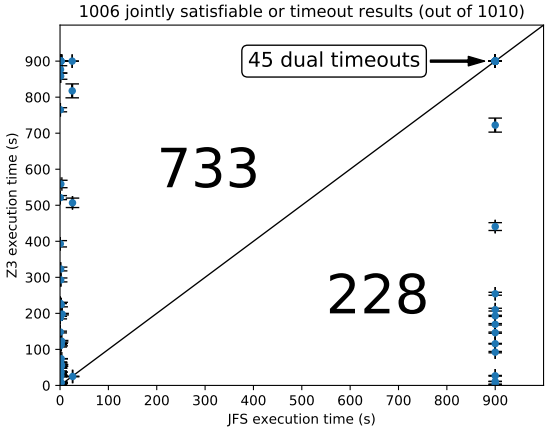
(d) JFS vs goSAT



(e) JFS vs MathSAT5



(f) JFS vs XSat



(g) JFS vs Z3

Figure 5.5: Scatter plots comparing JFS's execution time with that of other solvers on the $\texttt{QF\_FP}_{fs}$ benchmarks.

Validity of hypothesis

Recall from §5.1 that our hypothesis was that using coverage-guided fuzzing as a method for solving floating-point constraints would be faster than existing techniques in some cases. Our results show that JFS (our prototype solver implementing constraint solving via coverage-guided fuzzing) is indeed faster is some cases than existing solvers on floating-point constraints. This supports our hypothesis.

Our results are even better than our pessimistic hypothesis. Our results show that JFS is very competitive on floating-point benchmarks ($\texttt{QF\_FP}_{fs}$ benchmark suite) compared with existing solvers, is faster than all existing solvers on many benchmarks, and is superior to the existing search-based solvers (CORAL, goSAT, and XSat) in terms of number of benchmarks solved. Although we've shown JFS to be generally superior to the existing search-based solvers, that does *not* necessarily mean that JFS's coverage-guided fuzzing technique is superior to the techniques used by the other search-based solvers. Our results show that the existing search-based solvers frequently crashed. It may be the case that higher quality implementations of the ideas used by these search based solvers would be comparable to JFS or even superior. However, improving the quality of these other solvers is out of scope for our work.

Our results also show that JFS is somewhat competitive on benchmarks that use a mixture of floating-point and bit-vector constraints ($\texttt{QF\_BVFP}_{fs}$ benchmark suite) compared with existing solvers. JFS is able to show satisfiability for more benchmarks than COLIBRI, but slightly fewer than MathSAT5 and Z3. In terms of runtime performance JFS is less competitive than the other solvers due to it infrequently showing satisfiability faster than the other solvers. In these infrequent cases, JFS could be used to complement existing solvers by being able to solve the constraints faster. This supports our hypothesis.

Our results also show that JFS is generally inferior to existing solvers on bit-vector benchmarks ($\texttt{QF\_BV}_{fs}$ benchmark suite) but can still complement existing solvers by occasionally being able to solve constraints faster. Although this is disappointing, these results don't contradict our hypothesis because our hypothesis was that our approach would be faster in *some* cases, not *all* cases.

These results show that it would be very beneficial to build a solver that uses JFS in combination with another solver (i.e. a portfolio solver). If JFS were combined with a complete solver this would also remove one of its major limitations, which is the fact it can only show satisfiability (i.e. it can't show unsatisfiability). We leave experimenting with this idea as future work.

A very natural question to ask is: "Why is there such a large difference between JFS's performance on bit-vector benchmarks and floating-point benchmarks?" We leave this for future work, but we speculate that the reason is hidden bias in the benchmark suites. That is to say the floating-point benchmarks are "easier" to solve than the bit-vector benchmarks.

Bit-vector solvers have been available for over a decade which has allowed a set of difficult and challenging benchmarks to be developed over a long period of time. These benchmarks likely evolved in difficulty as bit-vector solvers gradually increased their capability. Solvers for floating-point constraints on the other hand are comparatively new and have had much less time to develop. As a consequence, the available floating-point benchmarks are a reflection of the relatively immature floating-point constraint solvers currently available.

It is also worth drawing an analogy with coverage-guided fuzzers applied to bug finding. These fuzzers are typically only good at finding very shallow bugs and can only excel at finding deep bugs with a large amount of compute time or with good seeds. It could be the case that the floating-point benchmarks that JFS has been applied to are the equivalent of shallow programs, where bugs are easy for a fuzzer to find.

5.5 Related work

There is a large body of existing work that seeks to improve solving floating-point constraints. This was previously discussed in §2.6.2.

The FloPSy [106] and CORAL [167] solvers apply meta-heuristic search techniques to try to find satisfying assignments to floating-point constraints. Like JFS these methods are incomplete because they can only show satisfiability (i.e. they cannot prove unsatisfiability). Both solvers construct a fitness function which they either attempt to minimise or maximise. JFS implicitly has a fitness function which it is trying to maximise. This function is the number of covered branches in the programs it generates. Like FloPSy and CORAL, JFS performs a search to try to find values to maximise this function. This implicit fitness function is very coarse (a branch

is covered, or it is not) in comparison to FloPSy and CORAL's fitness functions, which gradually change as candidate solutions get closer to a satisfying assignment. FloPSy's evolutionary search algorithms are very similar to the coverage-guided mutations used in JFS. Despite the coarseness of JFS's fitness function, in our work we have compared JFS with CORAL and have shown JFS to be superior, both in number of benchmarks it can show to be satisfiable, and in execution time (in most but not all cases). This may be an artifact of CORAL's implementation which was never designed to work with floating-point constraints from the SMT-LIBv2.5 `FloatingPoint` theory. We implemented a tool to convert SMT-LIBv2.5 constraints into CORAL's native constraint language but in many cases the conversion wasn't possible or CORAL crashed. We could not compare with FloPSy because it is too tightly integrated with Pex [171], the symbolic execution tool it is designed to work with. The CORAL solver supports using an interval solver to improve the quality of its initial candidate inputs. It's likely we could apply a similar approach in JFS to generate higher quality seeds for the fuzzer. These solvers don't support the theory of `FixedSizeBitVectors`, unlike JFS. However, given that they rely on meta-heuristic search, these solvers could support this theory if support was implemented.

The goSAT [25] and XSat [77] solvers both reformulate finding a satisfying assignment as a mathematical optimisation problem and apply existing mathematical optimisation algorithms to try to find a global minimum. This is very similar to FloPSy and CORAL in that the functions that goSAT and XSat seek to minimise are essentially a fitness functions. The difference is in the algorithms used to perform the search. Like JFS this strategy is incomplete. As previously mentioned JFS implicitly has a fitness function. However, JFS's fitness function is very coarse compared to the goSAT and XSat fitness functions. Despite this, in our comparison with goSAT and XSat, we have shown JFS to be superior to both solvers in terms of the number of benchmarks solved and in most cases the execution time performance. This may be an artifact of goSAT and XSat's implementations. Both tools frequently crashed or reported that they did support the supplied benchmarks. These solvers do not support the theory of `FixedSizeBitVectors`, unlike JFS.

The MathSAT5 [46], SONOLAR [149], and Z3 [140] solvers all solve floating-point constraints by transforming floating-point operations into bit-vector circuits and then bit-blasting these into a SAT problem. This problem is then solved using a SAT solver. This approach is complete, unlike JFS, but it can end up generating very large SAT problems, which are time consuming to solve. Like JFS, these solvers support a combination of the `FixedSizeBitVectors` and

`FloatingPoint` SMT-LIBv2.5 theories. In our work we have compared JFS against both MathSAT5 and Z3. Although JFS solves fewer benchmarks than Z3 (and fewer than both MathSAT5 and Z3 on some benchmark sets), we have shown that in many cases JFS's search based strategy is faster than the strategy used by MathSAT5 and Z3. This complementary behaviour of JFS suggests that these solvers would likely benefit from incorporating a search-based strategy—like JFS's—in combination with their existing strategies to form a portfolio solver. In our work we did not compare against SONOLAR. Such a comparison should be straightforward given that the solver claims to support the SMT-LIBv2.5 `FloatingPoint` theory. We did not perform a comparison simply because we were not aware that the solver supported the `FloatingPoint` theory when we started our experiments. The inclusion of SONOLAR in our experiments is unlikely to change our results for several reasons. First, SONOLAR uses essentially the same techniques as the MathSAT5 and Z3 solver which we already compare against. Second, a public release of SONOLAR has not been made since 2014, whereas MathSAT5 and Z3 are actively developed. Given this extra development time we expect that MathSAT5 and Z3 would outperform SONOLAR.

The COLIBRI [36] and FPCS [133] solvers use interval solving as a complete method for solving floating-point constraints. The COLIBRI solver like JFS supports a combination of `BitVector` and `FloatingPoint` theories. In our work we compared with COLIBRI. COLIBRI in most cases solved more benchmarks than JFS, however it also gave wrong results and was often slower than JFS. This suggests that COLIBRI might benefit from incorporating a search-based strategy—like JFS's—in combination with their existing strategies to form a portfolio solver. In our work we did not compare against FPCS because it is not publicly available. Even if the solver was publicly available it is very unlikely that it supports the SMT-LIBv2.5 `FloatingPoint` theory given that is was developed in 2001 and likely has not been updated since. This would require us to implement a constraint language converter in a similar manner to what we have done for the CORAL solver.

The REALIZER [112] solver tries to solve floating-point constraints by transforming (in an equisatisfiable manner) floating-point constraints into constraints over reals. This strategy is sound but is not complete because theories over reals are undecidable. Despite taking an undecidable approach, this strategy might be able to show some constraints as unsatisfiable which JFS can only do in very trivial cases. REALIZER's strategy is particularly suitable for working with constraints that check the accuracy of floating-point expressions compared to their real

counterparts. JFS cannot serve this use case because it cannot handle constraints over reals. Technically it would be possible to have JFS generate a program over reals. However, fuzzing them would require under-approximating the input space, which is infinite, with a finite input space.

Other researchers have compared search-based solvers to existing solving methods. Takaki et al. [169] compared the CVC3 [21] solver to a set of random solvers, heuristic solvers, and portfolio solvers. As in our work, the authors concluded that a portfolio approach would perform best. In contrast though, our comparison focuses on floating-point constraint solvers and thus includes many state-of-the-art floating-point constraint solvers.

5.6 Conclusion

In this chapter we presented our investigation into using coverage-guided fuzzing as a method for finding satisfying assignments to floating-point constraints. This work was motivated by poor floating-point constraint solver performance, observed in our work in Chapter 4. Poor floating-point constraint solver performance was the research problem we sought to tackle.

Our hypothesis was that using coverage-guided fuzzing to solve floating-point constraints would be faster than existing floating-point constraint solvers in some cases. To test this hypothesis we implemented a prototype solver (JFS) that generates programs from constraints where path reachability is equivalent to finding a satisfying assignment. JFS then applies an off-the-shelf coverage-guided fuzzer to try and find an input to trigger execution down the path that finds a satisfying assignment to the original constraints. We then compared our prototype against six existing state-of-the-art floating-point constraint solvers on three existing benchmark suites.

Our results support our hypothesis. On two benchmark suites, JFS is highly competitive with existing solvers in terms of the number of benchmarks solved. On one benchmark suite, JFS in many cases solves the benchmarks faster than each solver we compared with. On one benchmark suite (one that uses only bit-vectors and no floating-point constraints), JFS solves very few benchmarks compared to other solvers. However on this benchmark suite JFS occasionally solves benchmarks faster than each solver we compared with. The performance of JFS suggests that existing solvers would benefit incorporating JFS's strategy into their existing set of strategies to create a portfolio solver. Creating a portfolio solver would also alleviate JFS's

biggest weakness, which is that it an incomplete solver. Recall from §5.1 that JFS can only prove unsatisfiability in trivial cases and so in most cases can only prove satisfiability. Thus JFS is an incomplete solver. However, combining JFS with a complete solver creates a complete solver that does not suffer from JFS's incompleteness and has all of JFS's performance benefits.

Chapter 6

Conclusion and Future work

In this thesis we have presented three main contributions:

In Chapter 3 we investigated the problem of there being no existing comparison of symbolic execution and other program analysis techniques (see §2.1) at the level of an intermediate verification language (IVL - see §2.3.2). We had two hypotheses. First, that symbolic execution of an IVL is competitive with other techniques in terms of bug finding and verification. Second, that the state-of-the-art for symbolic execution of IVLs can be improved. To answer these hypotheses, we implemented our own symbolic execution tool (Symbooglix) and compared it to several existing program analysis tools for the Boogie IVL over two large benchmark suites. We showed that Symbooglix outperforms an existing symbolic execution tool (Boogaloo), which supports our second hypothesis. We found that Symbooglix is very competitive with existing tools on one benchmark suite but less competitive on the other. However on this benchmark suite Symbooglix was found to be complementary (i.e. some bugs are only found by Symbooglix) to existing tools. This partially supports the first hypothesis.

In Chapter 4 we investigated the problem that most symbolic execution tools do not support reasoning over symbolic floating-point programs due to the lack of support from most underlying constraint solvers. Our hypothesis was that support for the SMT-LIBv2.5 `FloatingPoint` theory [160] which is now available in some existing constraint solvers could be used by symbolic execution tools to reason over floating-point programs. Unfortunately due to the of lack support for floating-point programs in the Boogie IVL, we could not use Symbooglix for this investigation. Instead we extended the KLEE symbolic execution tool to reason over floating-point programs by using the support for the `FloatingPoint` theory in Z3. Coincidently, a differ-

ent research group started investigating the same problem and also extended KLEE at roughly the same time. We used this opportunity to collaborate and compare our implementations on a set of benchmarks developed by both research groups. Both implementations were able to symbolically execute a subset of the floating-point programs we developed to compare the implementations partially true. Although it indeed possible to use the `FloatingPoint` theory to reason over floating-point programs, we observed that such an approach does not scale well due to poor constraint solver performance. These results partially support our hypothesis.

In Chapter 5 we investigated the problem raised at the end of Chapter 4, namely that the performance of existing floating-point constraint solvers is inadequate. Our hypothesis was that, in some cases, coverage-guided fuzzing could be used as a method to solve floating-point constraints faster than existing approaches. To test this hypothesis we implemented a prototype solver (JFS) that generates programs from constraints where path reachability is equivalent to finding a satisfying assignment. JFS then applies an off-the-shelf coverage-guided fuzzer to try and find an input to trigger execution down the path that finds a satisfying assignment to the original constraints. We then compared our prototype against six existing state-of-the-art floating-point constraint solvers on three existing benchmark suites. Our results support our hypothesis. On two benchmark suites, JFS is highly competitive with existing solvers in terms of the number of benchmarks solved. On one benchmark suite, JFS in many cases solves the benchmarks faster than each solver we compared with. On one benchmark suite (one that uses only bit-vectors and no floating-point constraints), JFS solves very few benchmarks compared to other solvers. However on this benchmark suite JFS occasionally solves benchmarks faster than each solver we compared with. The performance of JFS suggests that existing solvers would benefit incorporating JFS's strategy into their existing set of strategies to create a portfolio solver.

We see several directions for possible future work related to these topics:

Symbolic execution for intermediate verification languages The main limitation of Symbooglix at the time of writing relates to its limited support for quantifiers. Given that symbolic execution works on a per-path basis, it may be possible to exploit path-specific knowledge to instantiate the quantifiers that appear in an axiom or assertion in a smart way, attempting to avoid passing quantified formulae to the underlying SMT solver. This is related to work on using "triggers" for quantifier handling in SMT solvers [138], but we speculate that path-specific

information may be useful in avoiding quantifiers altogether in some cases, so that triggers are unnecessary, or at least in optimizing the form that at a trigger takes so as to guide the SMT solver as much as possible.

Another limitation of Symbooglix at the time of writing is its lack of support for generating test cases. There are two complications in doing this. First, Symbooglix needs to support generating test cases that contain unbounded maps (an infinite data structure) and provide implementations for uninterpreted functions. This is more complicated than generating test cases for conventional programming languages because unbounded maps and uninterpreted functions do not exist in these languages. However, given that Symbooglix's underlying constraint solver can generate a model for these cases that is expressible in a finite manner, it should be possible to implement support in Symbooglix. The second complication is that the front-end used to generate the Boogie program executed by Symbooglix needs to provide a mechanism of translating a test case for the Boogie program into a test case for the original program that was used to create the Boogie program. There is currently no standard mechanism for doing this and illustrates an area where the separation of concerns provided by the Boogie IVL breaks down. Without a common interface to represent test cases, both the front-end and back-end need to be concerned with how test cases are represented.

Another direction is to apply the approach of *stratified inlining*, applied by the Coral tool [108], in the context of per-path symbolic execution. Stratified inlining involves performing analysis with procedures replaced by their specifications, inlining procedures on demand when an abstract counterexample reveals that a potential error would traverse a given procedure. It is straightforward to perform procedure summarisation during per-path symbolic execution, and we envisage using backwards program slicing to over-approximate whether a summarised procedure may contribute to a possible error: if it provably cannot, there is no need to inline the procedure in attempting to produce an input that can trigger the error. Employing procedure abstraction during symbolic execution has the potential to reduce path explosion significantly, even offering an infinite reduction in paths if a procedure that would yield an infinite number of paths can be summarised on an abstract path that proves to be error-free.

Symbolic execution of floating-point programs One obvious future direction for our work is to combine the two separate implementations of floating-point support in KLEE into a single implementation that can be contributed upstream and so used by other researchers. The im-

plementation written by Imperial showed superior performance, however the Aachen implementation more accurately modelled the x86_fp80 data type. It will require further investigation to determine how the design differences of the two tools can be unified.

Another interesting direction to take our work is to investigate the use of Z3 tactics [141] to improve performance. Tactics allow the user of Z3 to have control over the high level reasoning steps used to solve queries. This allows the solver to be tweaked to the specific problem domain being tackled. For example the array ackermannization optimization implemented in one of the forks of KLEE can actually be implemented using two Z3 tactics combined—`bvarray2uf` followed by `ackermannize_bv`. A patch for upstream KLEE sketching this idea has already been submitted for review.[1]

Another idea is to integrate the JFS solver into our modified version of KLEE. The performance of a fuzzer is highly dependent on the seeds (initial inputs) it is given. In the context of symbolic execution, inputs that satisfy previously visited paths are already known. These inputs could be used to seed the fuzzer with interesting inputs that could reduce JFS's solving time and thus improve KLEE's performance. JFS is of course an incomplete solver and so it would need to be paired with another complete solver to be used as part of a portfolio of solvers. This would allow KLEE to benefit from JFS's ability to solve some satisfiable queries quickly but still fall back to another solver when JFS fails to return an answer in a timely manner.

A natural extension of our work would be to investigate supporting features of floating-point programs that are difficult to model in the SMT-LIBv2.5 `FloatingPoint` theory.

The first feature is IEEE-754 floating-point exceptions. The SMT-LIBv2.5 `FloatingPoint` theory does not provide functions to determine if these exceptions can be raised. A consequence of this is that our tools do not support floating-point programs that query if any floating-point exceptions have been raised. To support this feature each execution state would need to maintain its own set of raised floating-point exceptions. In the case that floating-point operations are performed on symbolic data the state of each floating-point exception flag would also become symbolic. This poses a problem because the expressions representing the floating-point exception flags would likely become excessively large during execution. This could be ex-

[1] `https://github.com/klee/klee/pull/659`

tremely wasteful because it could be the case that the state of the floating-point exception flags are never read. Further investigation is required to devise techniques that allow modelling IEEE-754 floating-point exceptions to be scalable.

The second feature is signaling and quiet NaNs. The SMT-LIBv2.5 `FloatingPoint` theory does not distinguish between signaling and quiet NaNs. Consequently this means that our tools do not correctly handle floating-point programs that require this distinction to be made. One approach to resolve this would be to symbolically track NaNs being used in every floating-point operation so that it is possible to determine the type of NaN that would be produced by each operation. This would incur a large overhead and would also be wasteful in cases where the distinction between NaN types is not required. A second approach would be to change the `FloatingPoint` theory itself. It is likely there would be resistance to such a change by floating-point constraint solver authors because it would make the implementations of these solvers more complicated. Further investigation is required to find the best solution to this problem.

The final feature is the x86_fp80 type. This type is not an IEEE-754 floating-point type and therefore isn't supported by the SMT-LIBv2.5 `FloatingPoint` theory. In Chapter 4 we discussed two approaches for handling the x86_fp80 type using the SMT-LIBv2.5 `FloatingPoint` type. One approach ignored the non IEEE-754 classes (e.g. unnormal) and the other approach chose to explicitly model them. We do not know if any of these approaches (or another, yet-to-be-devised approach) are suitable because our benchmarks made limited use of the x86_fp80 type. Further investigation is required to determine an approach that is optimal for real world floating-point programs.

Constraint solving via coverage-guided fuzzing One obvious issue that arises from our work in Chapter 5 is the performance difference of JFS between the bit-vector benchmarks and the floating-point benchmarks. In Chapter 5 we speculated that the reason for this might be bias in the benchmarks suites we used. This requires further investigation.

Another issue with our work in Chapter 5 is that we do not know whether JFS's coverage guided approach is superior to just trying completely random inputs because none of the solvers use this approach. To address this we would need to implement a drop-in replacement for LibFuzzer that JFS could invoke. This drop-in replacement would just try random inputs without guidance from program coverage. This modified configuration of JFS could then be added to our comparison of existing solvers.

There are lots there are lots potential improvements to JFS that could be investigated.

Currently JFS just invokes a single fuzzing process and thus only uses a single CPU. On multi-core systems (which are now commonplace) it may be beneficial to invoke multiple copies of the fuzzer (each with a different random seed) and have them share the same corpus directory so that each fuzzer benefits from the inputs discovered by the other fuzzers.

Currently JFS just emits branches in the generated program using a *try-all* approach where every constraint is evaluated even if some constraints were determined to be false for the current input. It may be the case that a different constraint evaluation approach in JFS-generated programs would offer better performance. For example a *fail-fast* approach could stop evaluating constraints for the current input as soon as a constraint is found to be false.

In the current implementation of JFS, compiler optimisations are disabled for performance reasons. It would be interesting to investigate which optimisations are beneficial to fuzzing, and in particular in the context of JFS. It may be worth applying optimisations that complicate the control flow so that the fuzzer records inputs that traverse the additional paths. These additional inputs then indirectly act as mutation hints for the generation of more inputs. For example comparing two 64-bit integers could be split into eight separate byte comparisons. This could be beneficial and has already been tried by other researchers.[2]

The *equality extraction* pass is currently missing several opportunities to detect equalities. For example equalities through conversion operations (e.g. conversion of a free bit-vector variable to a floating-point type that is then asserted to be equal to another floating-point expression) are not supported. It might also be worth adding support for the `fp.eq` function, which acts like SMT-LIB equality except for a few special values that would need to be handled specially.

Currently JFS tries to solve all constraints together. In the case that the constraints contain multiple independent subsets (i.e. free variables are not shared between subsets) it might be beneficial to fuzz each subset of constraints separately and in parallel to increase performance.

JFS currently uses LibFuzzer's built-in mutators (with slight modification). These mutators are not aware of the structure of the data that JFS-generated programs use. It would be interesting to investigate using custom mutators that are aware of the structure of the data (e.g. that the first 8 bytes of the input represent a 64-bit IEEE-754 floating-point number).

[2] `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`

JFS currently doesn't support converting the input found by LibFuzzer into a model that satisfies the original constraints. Implementing support for this would allow JFS-generated models to be validated and would be useful to users of JFS who need the model associated with the satisfiability result.

Finally, the seeds provided to a fuzzer have a huge impact on the search progress of a fuzzer. Currently JFS just generates two seeds, one with all bits being zero, and one with all bits being one. This is a very naive strategy because the seeds are basically the upper and lower bounds of a very large search space. There are heuristics that could be used to seed the fuzzer with more interesting inputs. One heuristic would be to assign free variables values from a set of predefined interesting values. For example, if the type of the free variable was `FloatingPoint`, the set of interesting values could be set to $\{+0.0, -0.0, \text{NaN}, +\text{Infinity}, -\text{Infinity}\}$. Another heuristic would be to scan the query for constants and add them to the set of interesting values to assign to free variables. The number of seed combinations grows exponentially with the number of free variables. Therefore a limit would need to be enforced to prevent the set of seeds from becoming too large. Another heuristic would be to use the approach of the CORAL solver. CORAL uses an interval solver to find conservative bounds on satisfying assignments and uses values within these bounds to seed the search. A similar approach could be implemented in JFS.

Aside from applying JFS to symbolic execution as previously mentioned, JFS has great potential for experimenting with extensions to the SMT-LIB *FloatingPoint* theory. For example the transcendental floating-point functions such as sine and cosine are not part of the standard. However because JFS executes native code it can simply call any implementation of the transcendental floating-point functions available to it (e.g. the implementation from the host operating system's C math library) to represent these functions. These functions are typically weakly specified but this might be sufficient depending on the domain JFS is being applied to.

Bibliography

[1] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[2] Michal Zalewski. *AFL "bug-o-rama" trophy case*. `http://lcamtuf.coredump.cx/afl/#bugs`.

[3] Merav Aharoni, Sigal Asaf, Laurent Fournier, Anatoly Koyfman, and Raviv Nagel. "FP-gen - a test generation framework for datapath floating-point verification". In: *Eighth IEEE International High-Level Design Validation and Test Workshop 2003, San Francisco, CA, USA, November 12-14, 2003*. 2003, pp. 17–22.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd. Addison Wesley, 2006. ISBN: 0-321-48681-1.

[5] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation". In: *IEEE Transactions on Software Engineering* 36.6 (Nov. 2010), pp. 742–762. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.52.

[6] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. "JPF-SE: A Symbolic Execution Extension to Java PathFinder". In: *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*. Braga, Portugal, Mar. 2007.

[7] Stephan Arlt and Martin Schäf. "Joogie: Infeasible Code Detection for Java". In: *Proceedings of the 24th International Conference on Computer-Aided Verification (CAV'12)*. Berkeley, CA, USA, July 2012, pp. 767–773.

[8] Andrei Arusoaie, Ștefan Ciobâcă, Vlad Crăciun, Dragoș Gavriluț, and Dorel Lucanu. “A Comparison of Static Analysis Tools for Vulnerability Detection in C/C++ Code”. In: *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Sept. 2017.

[9] A. K. Ashish and J. Aghav. “Automated techniques and tools for program analysis: Survey”. In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. July 2013, pp. 1–7. DOI: 10.1109/ICCCNT.2013.6726693.

[10] A. Avizienis. “The N-Version Approach to Fault-Tolerant Software”. In: *IEEE Transactions on Software Engineering (TSE)* 11 (Dec. 1985), pp. 1491–1501.

[11] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. “Symbolic Path-Oriented Test Data Generation for Floating-Point Programs”. In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST’13)*. Luxembourg, Mar. 2013.

[12] Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. “A Survey of Symbolic Execution Techniques”. In: *CoRR* abs/1610.00502 (2016). URL: http://arxiv.org/abs/1610.00502.

[13] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. “A decade of software model checking with SLAM”. In: *Communications of the Association for Computing Machinery (CACM)* 54.7 (2011), pp. 68–76.

[14] Ethel Bardsley et al. “Engineering a Static Verification Tool for GPU Kernels”. In: *Proceedings of the 26th International Conference on Computer-Aided Verification (CAV’14)*. Vienna, Austria, July 2014, pp. 226–242.

[15] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. “Boogie: A Modular Reusable Verifier for Object-Oriented Programs”. In: *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO’05)*. Amsterdam, The Netherlands, Nov. 2005, pp. 364–387.

[16] Michael Barnett and K. Rustan M. Leino. “Weakest-Precondition of Unstructured Programs”. In: *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE’05)*. Lisbon, Portugal, Sept. 2005, pp. 82–87.

[17] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. “Automatic Detection of Floating-point Exceptions”. In: *Proceedings of the 40th ACM Symposium on the Principles of Programming Languages (POPL'13)*. Rome, Italy, Jan. 2013.

[18] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. “Automatic detection of floating-point exceptions”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 2013, pp. 549–560.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2015.

[20] Clark Barrett, Aaron Stump, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.0*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2010.

[21] Clark Barrett and Cesare Tinelli. “CVC3”. In: *Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07)*. Berlin, Germany, July 2007.

[22] Clark Barrett et al. “CVC4”. In: *Proceedings of the 23rd International Conference on Computer-Aided Verification (CAV'11)*. Cliff Lodge, Snowbird, UT, USA, July 2011.

[23] *BCT: Byte Code Translator for .NET code to Boogie*. `https://github.com/boogie-org/bytecodetranslator`.

[24] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. “Proofs from tests”. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*. Seattle, WA, USA, July 2008.

[25] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. “goSAT: Floating-point Satisfiability as Global Optimization”. In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'17)*. 2017, pp. 11–14. DOI: `10.23919/FMCAD.2017.8102235`.

[26] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. “GPUVerify: A Verifier for GPU Kernels”. In: *Proceedings of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)*. Tucson, AZ, USA, Oct. 2012.

[27] Bruno Blanchet et al. “A Static Analyzer for Large Safety-critical Software”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. PLDI ’03. San Diego, California, USA: ACM, 2003, pp. 196–207. ISBN: 1-58113-662-5. DOI: 10.1145/781131.781153. URL: http://doi.acm.org/10.1145/781131.781153.

[28] Sylvie Boldo and Jean-Christophe Filliâtre. “Formal verification of floating-point programs”. In: *Computer Arithmetic, 2007. ARITH’07. 18th IEEE Symposium on*. IEEE. 2007, pp. 187–194.

[29] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. “Combining Coq and Gappa for certifying floating-point programs”. In: *International Conference on Intelligent Computer Mathematics*. Springer. 2009, pp. 59–74.

[30] *Boogie from Microsoft Research*. https://github.com/boogie-org/boogie.

[31] Mateus Borges, Marcelo d’Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. “Symbolic Execution with Interval Solving and Meta-heuristic Search”. In: *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. ICST ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 111–120. ISBN: 978-0-7695-4670-4. DOI: 10.1109/ICST.2012.91. URL: http://dx.doi.org/10.1109/ICST.2012.91.

[32] Bernard Botella, Arnaud Gotlieb, and Claude Michel. “Symbolic Execution of Floating-point Computations”. In: *Softw. Test. Verif. Reliab.* 16.2 (June 2006), pp. 97–121. ISSN: 0960-0833. DOI: 10.1002/stvr.v16:2. URL: http://dx.doi.org/10.1002/stvr.v16:2.

[33] Martin Brain, Vijay D’Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. “Deciding floating-point logic with abstract conflict driven clause learning”. In: *Formal Methods in System Design* 45.2 (2014), pp. 213–245.

[34] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. *An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic*. Tech. rep. 2015. URL: http://smtlib.cs.uiowa.edu/papers/BTRW15.pdf.

[35] A. Brillout, D. Kroening, and T. Wahl. “Mixed abstractions for floating-point arithmetic”. In: *2009 Formal Methods in Computer-Aided Design*. Nov. 2009, pp. 69–76. DOI: 10.1109/FMCAD.2009.5351141.

[36] Bruno Marre and François Bobot and Zakaria Chihani. “Real Behavior of Floating Point Numbers”. In: *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories (SMT’17)*. Heidelberg, Germany, July 2017. URL: http://smt-workshop.cs.uiowa.edu/2017/papers/SMT2017_paper_21.pdf.

[37] Jacob Burnim and Koushik Sen. “Heuristics for Scalable Dynamic Test Generation”. In: *Proceedings of the 23rd IEEE International Conference on Automated Software Engineering (ASE’08)*. L’Aquila, Italy, Sept. 2008.

[38] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08)*. San Diego, CA, USA, Dec. 2008.

[39] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. “EXE: Automatically Generating Inputs of Death”. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS’06)*. Alexandria, VA, USA, Oct. 2006.

[40] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. “EXE: Automatically Generating Inputs of Death”. In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008). A shorter version of this article appeared in CCS’06., pp. 1–38. ISSN: 1094-9224. DOI: http://doi.acm.org/10.1145/1455518.1455522.

[41] Cristian Cadar et al. “Symbolic Execution for Software Testing in Practice—Preliminary Assessment”. In: *Proceedings of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact’11)*. Honolulu, HI, USA, May 2011.

[42] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. “Unleashing Mayhem on Binary Code”. In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P’12)*. San Francisco, CA, USA, May 2012.

[43] Liming Chen and Algirdas Avizienis. “N-version programming: A Fault-tolerance approach to reliability of software operation”. In: *Proceedings of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS’78)*. Toulouse, France, June 1978.

[44] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. “Rigorous Floating-point Mixed-precision Tuning”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: ACM, 2017, pp. 300–315. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009846. URL: http://doi.acm.org/10.1145/3009837.3009846.

[45] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems”. In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. Newport Beach, CA, USA, Mar. 2011.

[46] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. “The MathSAT5 SMT Solver”. In: *Proceedings of TACAS*. Ed. by Nir Piterman and Scott Smolka. Vol. 7795. LNCS. Springer, 2013.

[47] *clang: a C language family frontend for LLVM*. http://clang.llvm.org/.

[48] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logic of Programs, Workshop*. London, UK, UK: Springer-Verlag, 1982, pp. 52–71. ISBN: 3-540-11212-X. URL: http://dl.acm.org/citation.cfm?id=648063.747438.

[49] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-guided Abstraction Refinement for Symbolic Model Checking”. In: *Journal of the Association for Computing Machinery (JACM)* 50.5 (Sept. 2003), pp. 752–794.

[50] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*. Barcelona, Spain, Mar. 2004.

[51] Ernie Cohen et al. “VCC: A Practical System for Verifying Concurrent C”. In: *Proceedings of the 22nd Theorem Proving in Higher Order Logics (TPHOLs'09)*. Munich, Germany, Aug. 2009.

[52] Hélène Collavizza, Claude Michel, Olivier Ponsini, and Michel Rueher. “Generating test cases inside suspicious intervals for floating-point number programs”. In: *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, May 31, 2014*. 2014, pp. 7–11.

[53] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. “Symbolic Crosschecking of Data-Parallel Floating-Point Code”. In: *IEEE Transactions on Software Engineering (TSE)* 40.7 (2014), pp. 710–737.

[54] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. “Symbolic Crosschecking of Floating-Point and SIMD Code”. In: *Proceedings of the 6th European Conference on Computer Systems (EuroSys’11)*. Salzburg, Austria, Apr. 2011.

[55] *Competition on Software Verification (SV-COMP)*. http://sv-comp.sosy-lab.org/.

[56] Stephen A. Cook. “The Complexity of Theorem-proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. URL: http://doi.acm.org/10.1145/800157.805047.

[57] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*. Oct. 2017. URL: http://coq.inria.fr.

[58] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM Symposium on the Principles of Programming Languages (POPL’77)*. Los Angeles, CA, USA, Jan. 1977.

[59] *CVE-2014-6271*. Available from MITRE, CVE-ID CVE-2014-6271. Sept. 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271.

[60] V. D’Silva, D. Kroening, and G. Weissenbacher. “A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[61] Marc Daumas, Laurence Rideau, and Laurent Théry. “A Generic Library for Floating-Point Numbers and Its Application to Exact Computing”. In: *Theorem Proving in Higher Order Logics*. Ed. by Richard J. Boulton and Paul B. Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 169–184. ISBN: 978-3-540-44755-9.

[62] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: http://doi.acm.org/10.1145/368273.368557.

[63] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: http://doi.acm.org/10.1145/321033.321034.

[64] Kalyanmoy Deb. “An introduction to genetic algorithms”. In: *Sadhana* 24.4 (Aug. 1999), pp. 293–315. ISSN: 0973-7677. DOI: 10.1007/BF02823145. URL: https://doi.org/10.1007/BF02823145.

[65] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

[66] Niklas Eén and Niklas Sörensson. “An Extensible SAT-solver”. In: *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37. URL: https://doi.org/10.1007/978-3-540-24605-3_37.

[67] Pär Emanuelsson and Ulf Nilsson. “A Comparative Study of Industrial Static Analysis Tools”. In: *Electronic Notes in Theoretical Computer Science* 217 (2008). Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008), pp. 5–21. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.06.039. URL: http://www.sciencedirect.com/science/article/pii/S1571066108003824.

[68] Dawson Engler and Madanlal Musuvathi. “Static Analysis versus Software Model Checking for Bug Finding”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 191–210. ISBN: 978-3-540-24622-0.

[69] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution”. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE’99)*. Los Angeles, CA, USA, May 1999. URL: citeseer.nj.nec.com/ernst00dynamically.html.

[70] Felipe Manzano. “A symbolic execution engine for amd64 binaries”. In: *Ekoparty*. Buenos Aires, Argentina, Sept. 2013. URL: https://www.ekoparty.org/archive/2013/charlas/Manzano.pdf.

[71] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3 – Where Programs Meet Provers”. In: *Proceedings of the 22nd European Symposium on Programming (ESOP'13)*. Rome, Italy, Mar. 2013.

[72] Cormac Flanagan and K. Rustan M. Leino. “Houdini, an Annotation Assistant for ESC/-Java”. In: *Symposium of Formal Methods Europe*. Mar. 2001.

[73] Cormac Flanagan and James B. Saxe. “Avoiding Exponential Explosion: Generating Compact Verification Conditions”. In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: ACM, 2001, pp. 193–205. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360220. URL: http://doi.acm.org/10.1145/360204.360220.

[74] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics*. Vol. 19. American Mathematical Society. 1967, pp. 19–32.

[75] Fröhlich, Andreas and Biere, Armin and Wintersteiger, Christoph M. and Hamadi, Youssef. “Stochastic Local Search for Satisfiability Modulo Theories”. In: AAAI, Jan. 2015. URL: https://www.microsoft.com/en-us/research/publication/stochastic-local-search-for-satisfiability-modulo-theories/.

[76] Zhoulai Fu and Zhendong Su. “Achieving High Coverage for Floating-point Code via Unconstrained Programming”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 306–319. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062383. URL: http://doi.acm.org/10.1145/3062341.3062383.

[77] Zhoulai Fu and Zhendong Su. “XSat: A Fast Floating-Point Satisfiability Solver”. In: *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 187–209. ISBN: 978-3-319-41540-6. DOI: 10.1007/978-3-319-41540-6_11. URL: http://dx.doi.org/10.1007/978-3-319-41540-6_11.

[78] *gcov – A Test Coverage Program*. gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[79] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing”. In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'05)*. Chicago, IL, USA, June 2005.

[80] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: *Queue* 10.1 (Jan. 2012), 20:20–20:27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. URL: http://doi.acm.org/10.1145/2090147.2094081.

[81] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. “Automated Whitebox Fuzz Testing”. In: *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS'08)*. San Diego, CA, USA, Feb. 2008.

[82] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. “Compositional May-Must Program Analysis: Unleashing the Power of Alternation”. In: *Proceedings of the 37th ACM Symposium on the Principles of Programming Languages (POPL'10)*. Madrid, Spain, Jan. 2010.

[83] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0-201-15767-5.

[84] Anjana Gosain and Ganga Sharma. “A Survey of Dynamic Program Analysis Techniques and Tools”. In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Ed. by Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5.

[85] Yijia Gu, Thomas Wahl, Mahsa Bayati, and Miriam Leeser. “Behavioral Non-portability in Scientific Numeric Computing”. In: *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. 2015, pp. 558–569.

[86] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. “Synergy: A New Algorithm for Property Checking”. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'06)*. Graz, Austria, Mar. 2006.

[87] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamarić. “SMACK+Corral: A Modular Verifier”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 451–454. ISBN: 978-3-662-46681-0.

[88] *The Heartbleed Bug*. http://heartbleed.com/. Apr. 2014.

[89] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259.

[90] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*. Bellevue, WA, USA, Aug. 2012.

[91] *IEEE Standard for Floating-Point Arithmetic*. Standard. Institute of Electrical and Electronics Engineers, 2008. DOI: 10.1109/IEEESTD.2008.4610935.

[92] Intel. *Intel(R) 64 and IA-32 Architectures Software Devloper's Manual*. Volume 1, 8.2.2 Unsupported Double Extended-Precision Floating-Point Encodings and Pseudo-Denormals. 2016.

[93] *Information technology – Microprocessor Systems – Floating-Point arithmetic*. Standard. Geneva, Switzerland: International Organization for Standardization, 2011.

[94] *Programming languages – C*. Standard. Geneva, Switzerland: International Organization for Standardization, 1999.

[95] *Information technology – Programming languages – C*. Standard. Geneva, Switzerland: International Organization for Standardization, 2011.

[96] Ivo Colombo. *Debugging Symbolic Execution*. 2012.

[97] S. C. Johnson. *Lint, a C Program Checker*. Tech. rep. 1978, pp. 78–1273.

[98] William Kahan. “Paranoia”. In: *Available from netlib* 20 (1987).

[99] Richard Karpinski. “Paranoia-A Floating-Point Benchmark”. In: *Byte* 10.2 (1985), p. 223.

[100] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. “Comparing Verification Condition Generation with Symbolic Execution: An Experience Report”. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 196–208. ISBN: 978-3-642-27705-4.

[101] Gary A. Kildall. “A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945. URL: http://doi.acm.org/10.1145/512927.512945.

[102] James C. King. “Symbolic execution and program testing”. In: *Communications of the Association for Computing Machinery (CACM)* 19.7 (1976), pp. 385–394.

[103] R Krishnan, Margaret Nadworny, and Nishil Bharill. “Static Analysis Tools for Security Checking in Code at Motorola”. In: *Ada Lett.* XXVIII.1 (Apr. 2008), pp. 76–82. ISSN: 1094-3641. DOI: 10.1145/1387830.1387833. URL: http://doi.acm.org/10.1145/1387830.1387833.

[104] Daniel Kroening and Ofer Strichman. “Decision Procedures: An Algorithmic Point of View”. In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Bit Vectors, pp. 160–163. ISBN: 978-3-540-74105-3. DOI: 10.1007/978-3-540-74105-3_6.

[105] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. “Efficient state merging in symbolic execution”. In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'12)*. Beijing, China, June 2012.

[106] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. “FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution”. In: *Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*. Ed. by Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 142–157. ISBN: 978-3-642-16573-3. DOI: 10.1007/978-3-642-16573-3_11. URL: http://dx.doi.org/10.1007/978-3-642-16573-3_11.

[107] Akash Lal and Shaz Qadeer. “Powering the Static Driver Verifier Using Corral”. In: *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'14)*. Hong Kong, Nov. 2014, pp. 202–212.

[108] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. “Corral: A Solver for Reachability Modulo Theories”. In: *Proceedings of the 24th International Conference on Computer-Aided Verification (CAV'12)*. Berkeley, CA, USA, July 2012.

[109] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, CA, USA, Mar. 2004.

[110] Hoang M. Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. “Verifying SystemC Using an Intermediate Verification Language and Symbolic Simulation”. In: *Proceedings of the 50th Annual Design Automation Conference*. Austin, TX, USA, June 2013, pp. 1–6.

[111] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. “RAIVE: Runtime Assessment of Floating-point Instability by Vectorization”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Pro-*

gramming, Systems, Languages, and Applications. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 623–638. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814299. URL: http://doi.acm.org/10.1145/2814270.2814299.

[112] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. "Make it real: Effective floating-point reasoning via exact arithmetic". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2014, pp. 1–4. DOI: 10.7873/DATE.2014.130.

[113] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. "Make it real: Effective floating-point reasoning via exact arithmetic". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*. 2014, pp. 1–4.

[114] Daan Leijen. *Division and Modulus for Computer Scientists*. http://research.microsoft.com/pubs/151917/divmodnote.pdf. 2001.

[115] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. LPAR'10. Dakar, Senegal, Apr. 2009, pp. 348–370.

[116] K. Rustan M. Leino. *This is Boogie 2*. Tech. rep. Microsoft Research, June 2008.

[117] Rustan Leino and Peter Müller. "A Basis for Verifying Multi-Threaded Programs". In: Springer-Verlag Berlin, Heidelberg, Mar. 2009, pp. 378–393. URL: https://www.microsoft.com/en-us/research/publication/basis-verifying-multi-threaded-programs/.

[118] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (July 1993), pp. 18–41. ISSN: 0018-9162. DOI: 10.1109/MC.1993.274940.

[119] *LibFuzzer*. http://llvm.org/docs/LibFuzzer.html.

[120] *LibFuzzer trophies*. http://llvm.org/docs/LibFuzzer.html#trophies.

[121] Daniel Liew, Cristian Cadar, and Alastair Donaldson. "Symbooglix: A Symbolic Execution Engine for Boogie Programs". In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'16)*. Chicago, IL, USA, Apr. 2016, pp. 45–56.

[122] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael Zähl, and Klaus Wehlre. "Floating-Point Symbolic Execution: A Case Study in N-version Programming". In: *Proceedings of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*. Urbana-Champaign, IL, USA, Oct. 2017.

[123] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael Zähl, and Klaus Wehrle. “Floating-Point Symbolic Execution: A Case Study in N-version Programming”. In: *Proceedings of the 32nd IEEE International Conference on Automated Software Engineering (ASE’17)*. Urbana-Champaign, IL, USA, Oct. 2017.

[124] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. “Provably Correct Peephole Optimizations with Alive”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. Portland, OR, USA: ACM, 2015, pp. 22–32. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737965. URL: http://doi.acm.org/10.1145/2737924.2737965.

[125] Rupak Majumdar and Koushik Sen. “Hybrid Concolic Testing”. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*. Minneapolis, MN, USA, May 2007.

[126] M. Mantere, I. Uusitalo, and J. Roning. “Comparison of Static Code Analysis Tools”. In: *2009 Third International Conference on Emerging Security Information, Systems and Technologies*. June 2009, pp. 15–22. DOI: 10.1109/SECURWARE.2009.10.

[127] *Mars Climate Orbiter Mishap Investigation Board Phase I Report*. Tech. rep. NASA, Nov. 1999.

[128] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. “Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators”. In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’12)*. London, UK, Mar. 2012.

[129] Kenneth L. McMillan. “Lazy Abstraction with Interpolants”. In: *Proceedings of the 18th International Conference on Computer-Aided Verification (CAV’06)*. Seattle, WA, USA, Aug. 2006, pp. 123–136. ISBN: 978-3-540-37406-0. URL: http://dx.doi.org/10.1007/11817963_14.

[130] Kenneth McMillan and Andrey Rybalchenko. *Computing Relational Fixed Points using Interpolation*. Tech. rep. MSR-TR-2013-6. Microsoft Research, Jan. 2013.

[131] David Menendez, Santosh Nagarakatte, and Aarti Gupta. “Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM”. In: *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Pro-*

ceedings. Ed. by Xavier Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 317–337. ISBN: 978-3-662-53413-7. DOI: 10.1007/978-3-662-53413-7_16. URL: https://doi.org/10.1007/978-3-662-53413-7_16.

[132] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[133] C. Michel, M. Rueher, and Y. Lebbah. "Solving Constraints over Floating-Point Numbers". In: *Principles and Practice of Constraint Programming — CP 2001*. Ed. by Toby Walsh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 524–538. ISBN: 978-3-540-45578-3.

[134] Claude Michel. "Exact Projection Functions for Floating Point Number Constraints". In: *International Symposium on Artificial Intelligence and Mathematics, AI&M 2002, Fort Lauderdale, Florida, USA, January 2-4, 2002*. 2002. URL: http://rutcor.rutgers.edu/~amai/aimath02/PAPERS/21.ps.

[135] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Communications of the Association for Computing Machinery (CACM)* 33.12 (1990), pp. 32–44.

[136] Barton Miller et al. *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Tech. rep. University of Wisconsin–Madison, 1995. URL: citeseer.ist.psu.edu/miller95fuzz.html.

[137] David Monniaux. "The Pitfalls of Verifying Floating-point Computations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.3 (May 2008), 12:1–12:41. ISSN: 0164-0925. DOI: 10.1145/1353445.1353446.

[138] Michał Moskal. "Programming with Triggers". In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. SMT '09. Montreal, Canada: ACM, 2009, pp. 20–29. ISBN: 978-1-60558-484-3. DOI: 10.1145/1670412.1670416. URL: http://doi.acm.org/10.1145/1670412.1670416.

[139] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Annual Design Automation Conference*. DAC '01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017. URL: http://doi.acm.org/10.1145/378239.379017.

[140] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Budapest, Hungary, Mar. 2008.

[141] Leonardo de Moura and Grant Olney Passmore. “Automated Reasoning and Mathematics”. In: ed. by Maria Paola Bonacina and Mark E. Stickel. Berlin, Heidelberg: Springer-Verlag, 2013. Chap. The Strategy Challenge in SMT Solving, pp. 15–44. ISBN: 978-3-642-36674-1. URL: http://dl.acm.org/citation.cfm?id=2554473.2554475.

[142] Nicholas Nethercote and Julian Seward. “Valgrind: A Program Supervision Framework”. In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003).

[143] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T)”. In: *J. ACM* 53.6 (Nov. 2006), pp. 937–977. ISSN: 0004-5411. DOI: 10.1145/1217856.1217859. URL: http://doi.acm.org/10.1145/1217856.1217859.

[144] Andres Nötzli and Fraser Brown. “LifeJacket: Verifying Precise Floating-point Optimizations in LLVM”. In: *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. SOAP 2016. Santa Barbara, CA, USA: ACM, 2016, pp. 24–29. ISBN: 978-1-4503-4385-5. DOI: 10.1145/2931021.2931024. URL: http://doi.acm.org/10.1145/2931021.2931024.

[145] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. “Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach”. In: *CoRR* abs/1711.09362 (2017). arXiv: 1711.09362. URL: http://arxiv.org/abs/1711.09362.

[146] Brian S. Pak. *Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution*. 2012.

[147] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. “Automatically Improving Accuracy for Floating Point Expressions”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 1–11. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737959. URL: http://doi.acm.org/10.1145/2737924.2737959.

[148] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis”. In: *Automated Software Engineering* 20.3 (Sept. 2013), pp. 391–425.

[149] Jan Peleska, Elena Vorobev, and Florian Lapschies. “Automated Test Case Generation with SMT-Solving and Abstract Interpretation”. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 298–312. ISBN: 978-3-642-20398-5.

[150] David M. Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. “Accelerating Array Constraints in Symbolic Execution”. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'17)*. Santa Barbara, CA, USA, July 2017.

[151] Nadia Polikarpova, Carlo A. Furia, and Scott West. “To Run What No One Has Run Before”. In: *Proceedings of the 2013 Runtime Verification (RV'13)*. Rennes, France, Sept. 2013.

[152] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 978-0-521-88068-8.

[153] Sylvie Putot, Eric Goubault, and Matthieu Martel. “Static Analysis-Based Validation of Floating-Point Computations”. In: *Numerical Software with Result Verification, International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 19-24, 2003, Revised Papers*. 2003, pp. 306–313.

[154] Minghui Quan. “Hotspot Symbolic Execution of Floating-Point Programs”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 1112–1114. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983966. URL: http://doi.acm.org/10.1145/2950290.2983966.

[155] Jean-Pierre Queille and Joseph Sifakis. “Specification and Verification of Concurrent Systems in CESAR”. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. London, UK, UK: Springer-Verlag, 1982, pp. 337–351. ISBN: 3-540-11494-7. URL: http://dl.acm.org/citation.cfm?id=647325.721668.

[156] Zvonimir Rakamarić and Michael Emmi. “SMACK: Decoupling Source Language Details from Verifier Implementations”. In: *Proceedings of the 26th International Conference on Computer-Aided Verification (CAV'14)*. Vienna, Austria, July 2014, pp. 106–113.

[157] Jaideep Ramachandran, Corina S. Pasareanu, and Thomas Wahl. “Symbolic Execution for Checking the Accuracy of Floating-Point Programs”. In: *ACM SIGSOFT Software Engineering Notes* 40.1 (2015), pp. 1–5.

[158] C. Rubio-González et al. “Precimonious: Tuning assistant for floating-point precision”. In: *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2503296.

[159] Cindy Rubio-González et al. “Floating-point Precision Tuning Using Blame Analysis”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 1074–1085. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884850. URL: http://doi.acm.org/10.1145/2884781.2884850.

[160] Philipp Rümmer and Thomas Wahl. “An SMT-LIB Theory of Binary Floating-Point Arithmetic”. In: *International Workshop on Satisfiability Modulo Theories (SMT)*. 2010, p. 151. URL: http://www.cprover.org/SMT-LIB-Float/smt-fpa.pdf.

[161] Philipp Rümmer and Thomas Wahl. “An SMT-LIB Theory of Binary Floating-Point Arithmetic”. In: *Informal proceedings of 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC, Edinburgh, Scotland*. 2010.

[162] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C”. In: *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. Lisbon, Portugal, Sept. 2005.

[163] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. Boston, MA, USA, June 2012.

[164] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer—data race detection in practice”. In: *Workshop on Binary Instrumentation and Applications*. New York, NY, USA, Dec. 2009.

[165] Yan Shoshitaishvili et al. “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'16)*. San Jose, CA, USA, May 2016.

[166] João P. Marques Silva and Karem A. Sakallah. “GRASP&Mdash;a New Search Algorithm for Satisfiability”. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. ICCAD ’96. San Jose, California, USA: IEEE Computer Society, 1996, pp. 220–227. ISBN: 0-8186-7597-7. URL: http://dl.acm.org/citation.cfm?id=244522.244560.

[167] Matheus Souza, Mateus Borges, Marcelo d’Amorim, and Corina S. Păsăreanu. “CORAL: Solving Complex Constraints for Symbolic Pathfinder”. In: *Proceedings of the Third International Conference on NASA Formal Methods*. NFM’11. Pasadena, CA: Springer-Verlag, 2011, pp. 359–374. ISBN: 978-3-642-20397-8. URL: http://dl.acm.org/citation.cfm?id=1986308.1986337.

[168] Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”. In: *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS’16)*. San Diego, CA, USA, Feb. 2016.

[169] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo d’Amorim, and Ricardo B. C. Prudêncio. “Randomized constraint solvers: a comparative study”. In: *Innovations in Systems and Software Engineering* 6.3 (Sept. 2010), pp. 243–253. ISSN: 1614-5054. DOI: 10.1007/s11334-010-0124-1. URL: https://doi.org/10.1007/s11334-010-0124-1.

[170] G. Tassey. *The economic impacts of inadequate infrastructure for software testing*. Tech. rep. National Institute of Standards and Technology, 2002.

[171] Nikolai Tillmann and Jonathan De Halleux. “Pex: white box test generation for .NET”. In: *Proceedings of the 2nd International Conference on Tests and Proofs (TAP’08)*. Prato, Italy, Apr. 2008.

[172] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London mathematical society* 2.1 (1937), pp. 230–265.

[173] *Undefined Behavior Sanitizer*. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. 2017.

[174] W. Miller and D. L. Spooner. “Automatic Generation of Floating-Point Test Data”. In: *IEEE Transactions on Software Engineering (TSE)* SE-2.3 (Sept. 1976), pp. 223–226. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233818.

[175] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI’11)*. San Jose, CA, USA, June 2011.

[176] Aleksandar Zeljic, Christoph M. Wintersteiger, and Philipp Rümmer. “An Approximation Framework for Solvers and Decision Procedures”. In: *J. Autom. Reasoning* 58.1 (2017), pp. 127–147.

Appendix A

Expected number of attempts at guessing a value

A.1 Uniform random guesses

We want to calculate the expected number of attempts required to guess an integer between 0 and $N-1$, inclusive. Note this is the average (arithmetic mean) number of attempts required if this process of guessing was repeated forever.

Let us first assume that probability of guessing the integer in any single attempt is p. The expected number of attempts, $\langle A \rangle$, is then given by equation A.1.

$$\langle A \rangle = \lim_{C \to \infty} 1 \cdot p + 2(1-p)p + 3(1-p)^2 p + \ldots + C(1-p)^{C-1} p \tag{A.1}$$

$$= \sum_{i=1}^{\infty} i(1-p)^{i-1} p \tag{A.2}$$

In words, this is the probability of guessing correctly on the first go, p, multiplied by 1 , plus the probability of first guessing incorrectly and then correctly, $(1-p)p$, multiplied by 2, and so on and so forth.

Now let us assume that probability p, is uniformly random, i.e. $p = \frac{1}{N}$. Equation A.1 now becomes equation A.3.

$$\langle A \rangle = \sum_{i=1}^{\infty} i \left(\frac{N-1}{N} \right)^{i-1} \frac{1}{N} \tag{A.3}$$

We now multiply equation A.3 by N to give equation A.4, and by $(N-1)$ to give equation A.5.

$$\langle A \rangle N = 1 + 2\left(\frac{N-1}{N}\right) + 3\left(\frac{N-1}{N}\right)^2 + \dots \tag{A.4}$$

$$\langle A \rangle (N-1) = \left(\frac{N-1}{N}\right) + 2\left(\frac{N-1}{N}\right)^2 + 3\left(\frac{N-1}{N}\right)^3 + \dots \tag{A.5}$$

We now subtract equation A.5 from equation A.4 to give equation A.6, which simplifies to equation A.7.

$$\langle A \rangle \left(N - (N-1)\right) = 1 + (2-1)\left(\frac{N-1}{N}\right) + (3-2)\left(\frac{N-1}{N}\right)^2 + (4-3)\left(\frac{N-1}{N}\right)^3 + \dots \tag{A.6}$$

$$\langle A \rangle = \sum_{i=0}^{\infty} \left(\frac{N-1}{N}\right)^i \tag{A.7}$$

Equation A.7 is a geometric series. Recall that a geometric series of the form given shown on the left hand side of equation A.8 converges to the value on the right hand side if $|x| < 1$.

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad \text{if} \quad |x| < 1 \tag{A.8}$$

This allows us to simplify equation A.7 to equation A.9.

$$\langle A \rangle = N \tag{A.9}$$

That is to say, the expected number of attempts at guessing an integer between 0 and $N-1$ inclusive is N.

A.2 Uniform random guess with feedback

We now look at the expected number of attempts to randomly guess the value of an integer, but where guessed are performed a byte at a time, and feedback is received when the guess of a byte is correct. This *approximately* models a fuzzer guessing a byte at a time and getting coverage feedback when it makes a correct guess.

The expected number of guesses, $\langle A \rangle$, to guess B bytes is given by equation A.10, where P_x is the probability of correctly guessing all B bytes in x guesses (where $x >= B$).

$$\langle A \rangle = \lim_{C \to \infty} BP_B + (B+1)P_{B+1} + (B+2)P_{B+2} + \ldots + (B+C)P_{B+C} \tag{A.10}$$

We now need to compute P_x. Let us first assume that the probability of guessing any one particular byte correctly is given by p. Let us also assume that when feedback is received that a byte value was correctly guessed, that this is recorded, but which particular byte was the correct is ignored for subsequent guesses. This may seem like an odd choice, but it simplifies the calculation because it makes p constant. For example, if there were 6 bytes and we assume uniform probability of picking a byte, then $p = \frac{1}{6} \cdot \frac{1}{256}$. This means that even after correctly guessing one byte, the fraction in p does not change from $\frac{1}{6}$ to $\frac{1}{5}$. This odd choice also roughly reflects the behaviour of a coverage-guided fuzzer. The fuzzer records an input (i.e. a guess) that increases coverage (i.e. one of the byte assignments is correct) but doesn't use the fact it knows which one is correct is subsequent guesses (i.e. it may later mutate a byte that it correctly guessed). We say roughly because this ignores other behaviours that a coverage-guided fuzzer might have such as cross-over mutations and using a corpus of existing inputs.

We can now give a formulas for the probabilities.

$$
\begin{aligned}
P_B &= p^B \\
P_{B+1} &= B(1-p)p^B \\
P_{B+2} &= \frac{(B+1)B}{2}(1-p)^2p^B \\
&\dots \\
P_x &= \binom{x-1}{x-B}(1-p)^{x-B}p^B
\end{aligned}
$$

Note that $\binom{x-1}{x-B}$ is the binomial coefficient (also known as choose). This factor is required because we have to consider the various possible combinations of correct and incorrect guesses. For example, if $B = 4$ and we are considering P_{B+1}, then we need to consider that in the sequence of guesses, the one incorrect guess can occur at four different positions. Note it can't occur as the last guess because we stop guessing once all bytes have been correctly guessed.

The formula for $\langle A \rangle$ is now given by equation A.11 which can then be rewritten as equation A.12 by performing the substitution $j = i - B$.

$$\langle A \rangle = \sum_{i=B}^{\infty} i\binom{i-1}{i-B}(1-p)^{i-B}p^B \tag{A.11}$$

$$\langle A \rangle = \sum_{j=0}^{\infty}(j+B)\binom{j+B-1}{j}(1-p)^{j}p^B \tag{A.12}$$

Unfortunately this is where our algebraic treatment of this equation ends. This infinite series can be shown to converge by the "ratio test". However we do not know how to show what it convergences to.

However the Wolfram Alpha tool[1] can show what it converges to for concrete values. For the particular example that references this appendix, $p = \frac{1}{6} \cdot \frac{1}{256}$ and $B = 6$. Entering the following text into Wolfram Alpha will cause it to report that the infinite series converges to 9216.

```
sum 0 to infinity (j+6)*((j + 5) choose j) * (1 - (1/1536))^j * ((1/1536)^6)
```

[1] https://www.wolframalpha.com/