UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

Scalable and Precise Verification based on *k*-induction, Symbolic Execution and Floating-Point Theory

by

Mikhail Yasha Ramalho Gadelha

Thesis for the degree of Doctor of Philosophy

June 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Thesis for the degree of Doctor of Philosophy

by Mikhail Yasha Ramalho Gadelha

In this thesis, we describe and evaluate approaches for the efficient reasoning of real-world C programs using either Bounded Model Checking (BMC) or symbolic execution. We present three main contributions.

First, we describe three new technologies developed in a software verification tool to handle real-world programs: (1) a frontend based on a state-of-the-art compiler, (2) a new SMT backend with support for floating-point arithmetic and (3) an incremental bounded model checking algorithm. These technologies are implemented in ESBMC, an SMT-based bounded model checker for C programs; results show that these technologies enable the verification of a large number of programs.

Second, we formalise and evaluate the *bkind* algorithm: a novel extension to the *k*-induction algorithm that improves its bug-finding capabilities by performing backward searches in the state space. The *bkind* algorithm is the main scientific contribution of this thesis. It was implemented in ESBMC, and we show that it uses fewer resources compared to the original *k*-induction algorithm to verify the same programs without impacting the results.

Third, we evaluate the use of SMT solvers in a state-of-the-art symbolic execution tool to reduce the number of false bugs reported to the user. Our SMT-based refutation of false bugs algorithm was implemented in the clang static analyser and evaluated on a large set of real-world projects, including the MacOS kernel. Results show that our refutation algorithm cannot only remove false bugs but also speed up the analysis when bugs are refuted. The algorithm does not remove any true bug and only introduces a 1% slowdown if it is unable to remove any bugs.

Contents

List of Tables 9

List of Figures 11

List of Algorithms 15

List of Publications 17

Nomenclature 17

Declaration Of Authorship 21

Acknowledgements 23

1 Introduction 27
- 1.1 Scope of this Thesis . . . 29
- 1.2 Contributions . . . 31
- 1.3 Thesis Structure . . . 33

2 Background 35
- 2.1 Software Verification in Practice . . . 35
- 2.2 Program Formalization . . . 38
  - 2.2.1 Encoding Approaches . . . 40
  - 2.2.2 Encoding Programs and Checking Properties . . . 41
- 2.3 Static Analysis Approaches . . . 43
  - 2.3.1 Bounded Model Checking . . . 44
  - 2.3.2 Symbolic Execution . . . 45
- 2.4 Floating-point Arithmetic . . . 46
  - 2.4.1 Verifying Programs with Floating-point Arithmetic . . . 49
- 2.5 Constraint Solvers . . . 50
  - 2.5.1 SAT and SMT Solvers . . . 50
  - 2.5.2 Floating-point Arithmetic support in SMT Solvers . . . 52

3 Verifying C Programs: The Development of a Compiler-based Frontend to a Floating-point Backend 55
- 3.1 Illustrative Example . . . 57
- 3.2 ESBMC Architecture . . . 58
- 3.3 Frontend: Understanding C Programs . . . 59
  - 3.3.1 New supported features . . . 60

3.4 GOTO Converter: a State Transition System Generator . . . 66
3.5 Symbolic Engine: Generating SSA . . . 68
3.6 SMT Encoding of ANSI-C Programs . . . 71
3.6.1 Supported Solvers . . . 72
3.6.2 Encoding Scalars . . . 74
3.6.3 Encoding Fixed- and Floating-points . . . 74
3.6.3.1 SMT Fixed-Point Encoding . . . 74
3.6.3.2 SMT Floating-Point Encoding . . . 75
3.7 Incremental Bounded Model Checking . . . 79
3.8 Experimental Evaluation . . . 81
3.8.1 Experimental Setup . . . 81
3.8.2 Solver Performance Comparisons . . . 84
3.8.3 Floating-Point API evaluation . . . 86
3.9 Related Work . . . 88
3.10 Conclusions . . . 89

4 Correctness Proof and Bug Hunting by *k*-induction 91
4.1 Illustrative Example . . . 92
4.2 Naïve *k*-induction Algorithm . . . 94
4.2.1 Why is the *k*-induction Algorithm Naïve? . . . 98
4.3 A Smarter *k*-induction . . . 99
4.3.1 Bidirectional Bug-Finding using *k*-induction . . . 99
4.3.2 Constraint Generation using Interval Analysis . . . 103
4.3.3 Why is the *bkind* Algorithm more Efficient than *k*-induction? . . . 106
4.4 Experimental Evaluations . . . 106
4.4.1 Experimental Setup . . . 107
4.4.2 Comparison of *k*-induction-based approaches . . . 107
4.4.3 2LS Comparison . . . 110
4.5 Related Work . . . 112
4.6 Conclusions . . . 114

5 SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyser 115
5.1 The Clang Static Analyser . . . 117
5.2 Refuting False Bugs using SMT Solvers . . . 118
5.3 Experimental Evaluation . . . 120
5.3.1 Experimental Setup . . . 121
5.3.1.1 Patches to projects . . . 122
5.3.2 Bug Refutation Comparison . . . 123
5.4 Related Work . . . 124
5.5 Conclusions . . . 125

6 Conclusions 127
6.1 Future Works . . . 128

Bibliography 131

Appendices 147

A Support for the FP logic **149**

B Verification Time in the ReachSafety-Float Category **151**

List of Tables

2.1 IEEE floating-point types. 48

3.1 Formal definition of the reachability property checked in the ReachSafety category. 82

3.2 Formal definition of the memory safety properties checked in the MemSafety category. 82

3.3 Formal definition of the overflow property checked in the NoOverflows category. 82

3.4 Formal definition of the termination property checked in the Termination category. 83

5.1 Results of the analysis with and without refutation. 123

A.1 Support in each SMT solver and in the ESBMC FP API for the operations described in the SMT FP logic. A $\surd$ indicates a supported feature while $\times$ indicates an unsupported feature. 150

B.1 Verification time, in seconds, of each program in the ReachSafety category, for Z3 and MathSAT using their native floating-point API and all supported solvers using our floating-point API. A * after the name of the program indicates that it is expected to fail. 157

List of Figures

2.1 Grammar of the analyzer language. 39

2.2 Unwinding assertion and assumption representation. 42

2.3 Unwinding assertion and assumption representation. 45

2.4 Logics defined in the SMT-LIB. Source: http://smtlib.cs.uiowa.edu/logics.shtml 51

2.5 An SMT formula and one model generated by an SMT solver. 51

3.1 A small C program with a subtle error triggered if x is NaN; it will be used as a running example to explain the verification process in ESBMC. Here nondet_uint() and nondet_double() return for non-deterministic integer and double values, respectively. 57

3.2 ESBMC architecture. 58

3.3 AST of the program in Figure 3.1, generated by clang. 61

3.4 Program with float literal extensions. 63

3.5 Simple code that initializes three elements of an array out of order. . . . 63

3.6 Program with cast from unsigned long to array of function pointers. . . 63

3.7 Program using thread local storage for variables. 63

3.8 Program using compound literals. 63

3.9 Fragment of code from a program in SV-COMP. 64

3.10 Program to show the use of predefined identifiers. 65

3.11 Program to show the use of offsetof. 65

3.12 Program to show the use of alignof, sizeof and typeof. 65

3.13 Program to show the use of generic selection. 65

3.14 Program to show the use of static assertions. 66

3.15 GOTO program of the program in Figure 3.1, as printed by ESBMC. . . . 67

3.16 Unoptimized SSA generated from the program in Figure 3.1 when unwinding the program once 69

3.17 Optimized SSA generated from the program in Figure 3.1 when unwinding the program once. 70

3.18 The SMT formula generated from the SSA in Figure 3.17 when using Z3. 72

3.19 The counterexample printed by ESBMC when verifying the program in Figure 3.1 with one loop unwindings. 73

3.20 Simple floating-point program with a bug: the assertions in line 5 does not hold if x is NaN. 74

3.21 Program to demonstrate casts between floating-points and Booleans. . . 76

3.22 SMT formula generated by ESBMC to encode the casts to and from Boolean types in Figure 3.21. 76

3.23 Model for `fmax`. 77

3.24 Model for `fmin`. 77

3.25 Model for `fmod`. 78

3.26 SV-COMP results for each solver, using incremental BMC. 84

3.27 Total verification time of each solver, in seconds. 85

3.28 ReachSafety-Floats results for each solver, using the incremental BMC. The (fp2bv) next to the solver name means that our floating-point API was used to bit-blast floating-point arithmetic. 86

3.29 Total verification time of each solver in the ReachSafety-Floats category, in seconds. 87

4.1 Unbounded loop and finite unwinding. 92

4.2 Simplified illustrative examples encoding an event-condition-action system. The program in Figure 4.2a is safe since the property violation is unreachable while the program in Figure 4.2b is unsafe since the property violation is reachable after at least 5 iterations. 93

4.3 Visual representation of both searches performed by our *bkind* algorithm. Each dashed section represents the states reachable after k iterations. The arrows show the "direction" of the verification by the forward search (starting from the initial state s_1) and the backward search (from the error state ϵ). The thick line is a partial counterexample. 100

4.4 Visual representation of the state space of a program. The states inside the dashed line are the reachable states and the ones outside are unreachable states. An unconstrained over-approximation of the program assumes that all the states are reachable which might contain spurious counterexamples (counterexamples that lead to unreachable error states). An invariant is a filter of states: a strong enough invariant will remove unreachable error states from reasoning, allowing the inductive step check to prove the program correctness or to find non-spurious partial counterexamples. 104

4.5 Results of the k-induction-based algorithms in ESBMC for all SV-COMP18 benchmarks with different configurations. 108

4.6 Total verification time of the k-induction-based algorithms in ESBMC, in seconds. 109

4.7 Results of ESBMC with *bkind* algorithm and invariants, and 2LS for all SV-COMP18 benchmarks. 111

4.8 Total verification time of the *bkind* algorithm and invariants and 2LS, in seconds. 111

5.1 A small C safe program. The dereference in line 4 is unreachable because the guard in line 3 is always false. 116

5.2 The (false) bug report produced by the clang static analyser. The annotations about the path followed are inserted in a post-processing step. . 118

5.3 The refutation extension in the clang Static Analyser. 119

5.4 The SMT formula of the bug report from Fig. 5.1, using Z3. Note that the solver was able to simplify the formula to two assertions: that the first bit should be one and zero at the same time. Since this is a contradiction, the formula is unsatisfiable. 120

5.5 Number of bugs reported by the clang static analyser, with and without refutation. The refutation algorithm removed, on average, 12 bugs across seven projects. 124

List of Algorithms

4.1 The base case. 95

4.2 The forward condition. 95

4.3 The inductive step. 96

4.4 Naïve *k*-induction. 97

4.5 The base case used in the *bkind* algorithm. 101

4.6 The *bkind* algorithm. 102

4.7 The new inductive step with invariants. 105

5.1 $encodeConstraint(cs, \Phi)$ 119

List of Publications

In chronological order:

1. Renato B. Abreu, **Mikhail R. Gadelha**, Lucas C. Cordeiro, Eddie Batista de Lima Filho, and Waldir Sabino da Silva Jr. Bounded Model Checking For Fixed-point Digital Filters. *JBCS*, 22(1):1:1–1:20, 2016.

2. **Mikhail R. Gadelha**. Using Clang As A Frontend On A Formal Verification Tool. In *FOSDEM*, 2017.

3. **Mikhail R. Gadelha**, Hussama I. Ismail, and Lucas C. Cordeiro. Handling Loops In Bounded Model Checking Of C Programs Via *k*-induction. *STTT*, 19(1):97–114, 2017.

4. **Mikhail R. Gadelha**, Lucas C. Cordeiro, and Denis A. Nicole. Encoding Floating-Point Numbers Using The SMT Theory In ESBMC: An Empirical Evaluation Over The SV-COMP Benchmarks. In *SBMF*, pages 91–106, 2017.

5. **Mikhail R. Gadelha**, Jeremy Morse, Lucas C. Cordeiro, and Denis Nicole. Using Clang As A Frontend On A Formal Verification Tool. In *European LLVM Developers Meeting*, 2018.

6. **Mikhail R. Gadelha**, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An Industrial-Strength C Model Checker. In *ASE*, pages 888–891. ACM, 2018.

7. **Mikhail R. Gadelha** and Enrico Steffinlongo. SMT-Based Refutation Of Spurious Bug Reports In The Clang Static Analyzer. In *Bay Area LLVM Developers Meeting*, 2018.

8. **Mikhail R. Gadelha**, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. Towards Counterexample-guided *k*-Induction For Fast Bug Detection. In *FSE*, 2018.

Nomenclature

API	Application Programming Interface
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BFS	Breadth-First Search
BMC	Bounded Model Checking
CBMC	C Bounded Model Checker
CFG	Control Flow Graph
CNF	Conjunctive Normal Form
CSA	Clang Static Analyzer
CTL	Computational Tree Logic
DAG	Directed Acyclic Graph
DFS	Depth-First Search
DPLL	Davis-Putnam-Logemann-Loveland
ESBMC	Efficient SMT-Based Bounded Model Checker
IBMC	Incremental Bounded Model Checking
LTL	Linear-time Temporal Logic
PL	Propositional Logic
QF	Quantifier-Free Formula
QF_ABVFP	Quantifier-free formula over the theory of bit-vectors and floating-points and arrays
RT	Reachability Tree
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories
VC	Verification Condition
TDD	Test Driven Development

Declaration Of Authorship

I, **Mikhail Yasha Ramalho Gadelha**, declare that this thesis entitled as **Scalable and Precise Verification based on *k*-induction, Symbolic Execution and Floating-Point Theory** and the work presented in it are my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published.

Signed:

Date: 20/06/2019

Acknowledgements

The work in this thesis would not have been possible without the support of my wife, Loma Brito. I could not have completed this thesis without her support.

I would like to thank my supervisor Dr. Denis Nicole, for the guidance during my PhD. I also appreciate the help of Dr. Lucas Cordeiro, Dr. Jeremy Morse, Dr. Bernd Fischer, and Dr. Enrico Steffinlongo in the development of some of the ideas presented in this work.

In the last year of my PhD I participated in the Google Summer of Code with the LLVM foundation. I would like to thank my mentors George Karpenkov and Artem Dergachev for the guidance during the development of this project and, in particular, George Karpenkov for the idea for the project. Furthermore, I would like to thank Devin Coughlin and Anna Zaks for introducing me to George and Artem, and Rka Kovcs for the initial prototype of the bug refutation visitor.

Finally, I would not have achieved anything without the love and support of my family and friends. Thank you for everything.

To my dearest mother.

Chapter 1

Introduction

Computer systems are used in a variety of applications, ranging from critical systems, such as nuclear plants and automotive systems, to entertainment and games [1]. Most such systems have limited resources that force developers to use low or medium level languages, e.g., ANSI-C and C++, which are two of the major programming languages for the development of embedded systems [2]. Alternatives such as Erlang [3] have not come into widespread use.

However, limitations on computer systems are continually being exceeded because of the evolution of their hardware. Every year, embedded computer power increases. As a direct result of this evolution, more functionalities are implemented, and they become more present in our lives. However, this unavoidable trend is slowed down by the ever-increasing numbers of systems failures and security breaches.

Systems failures can be the source of catastrophic events, such as the explosion of Ariane 5, in 1996 [4]. The rocket self-destructed 40 seconds after launch, at an altitude of about 3700 meters, due to an unhandled exception thrown by a data conversion from a 64-bit floating-point to a 16-bit signed integer value [5]. System failures can also be subtler, such as the recent Heartbleed bug on OpenSSL [6] which might have leaked private information from several servers. In order to lessen such pitfalls, the computer system must be validated before release, therefore some software development methodologies include test-first programming concepts, such as Test Driven Development (TDD), i.e., a test using the desired feature is created before the development of the feature is started [7].

Of the conventional validation methods, *Peer Review* and *Software Testing* are the two main techniques to validate programs [8]. In Peer Review, a team of experienced software engineers inspects the program, provides feedback and, preferably, does not get involved with the development of the program under review. Empirical studies show that the technique can find between 31% to 93% of the errors, with an average 60% [8].

In Software Testing, the search for errors is done through extensive testing routines, called *test suites*, which force the program to be executed in different situations, followed by the analysis of the results. Empirical studies show that the technique can find up to 90% of all faults found during development, with an average of 48% [9]. The creation of the test is not an easy task though, as the test developer must have a vast knowledge of the system's behaviour and how it can fail. Tests must also be updated or removed during the development of the computer system. The effort required for such feat can be massive and impacts directly on the development of the system itself [10].

Work has been done to minimise the impact of test development on system development, mostly by moving toward automated test techniques. Fuzzing testing, a technique that involves providing random data (from an initial well-formed seed) as input to a program, is being increasingly applied to software testing and has presented good results [11] despite the random nature of the technique which may require several runs to find incorrect behaviours. Another recent advance is the advent of *sanitizers*. A sanitizer is a combination of automated code instrumentation and runtime checks in order to find invalid behaviours. So far, sanitizers have been developed to detect memory errors (Address Sanitizer) [12], data races (Thread Sanitizer) [13], uninitialized memory (Memory Sanitizer) [14], undefined behaviour (Undefined Behavior Sanitizer) [15], memory leaks (Leak Sanitizer) [16], and to perform data flow analysis (Data Flow Sanitizer) [17]. All of the sanitizers introduce a runtime and memory overhead, and they stop the program upon the discovery of an incorrect behaviour; they do, however, provide an accurate report of the trigger of the incorrect behaviour, e.g., Thread Sanitizer provides the sequence of read and write operation that triggered a data race.

Further into fully automated verification techniques, we find the *static analysis* tools. In most cases, the analysis is performed on the source code, which means that the validation is done without actually executing the program [18]. In some other cases, static analysis is performed at compile time by the compiler [19]. Static analysis tools search for patterns and behaviours, using an abstract domain to track the state of a program at a particular point (e.g., the upper and lower bounds of a variable instead of the multiple values it can assume during program execution). Static analysis tools are generally tailored towards finding common bugs such as buffer overflows and bounds violation. They are fast but might fail to prove that some safety properties hold, leading to false bugs being reported [20].

A more precise fully automated static analysis is *model checking*, part of the *Formal Methods* field, a family of techniques based on mathematical formalism for the specification, design and verification of both software and hardware [21]. Model checking is a technique that operates by converting a given program into a state transition system and exploring every path, aiming to prove that the program does not contain errors. One

advantage of model checking is that, if an error is found in the program, a counterexample (also called a witness), showing the execution trace that led to the error is presented at the end of the verification.

However, the exploration of the whole state space comes with a price: model checkers are extremely resource-hungry and might lead to exhaustion of memory or time before providing an answer. These demanding requirements allowed the growth of less precise techniques in the formal methods field, where precision is traded for smaller resource requirements [22].

This thesis focus on the usage of two such techniques – *bounded model checking* and *symbolic execution* – to verify real-world C programs.

Bounded Model Checking (BMC). Algorithms that limit the execution length to a given bound. The bound can be applied merely to the number of state transitions from the initial state or, in a more elaborated manner, to the number of loop unwindings. BMC tools based on Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solvers have been successfully applied to the verification of both sequential and parallel programs and for the discovery of subtle bugs [23, 24, 25]. Usually, they are aimed at finding bugs; the algorithm cannot prove that no bug is present in a program unless the bound is big enough to reach all states in the state space. We use the BMC algorithm as a component in our novel *bkind* algorithm and evaluate it when verifying many public benchmarks, including a large set of Linux drivers.

Symbolic Execution. Algorithms that enumerate and check the feasibility of all paths in a program, separately. Symbolic execution tools have also been applied to find errors in real-world programs [26] or, given their per-path verification approach, it has been used to generate test cases that exhaustive test all paths in a program, ensuring high coverage [27]. We extend a symbolic execution tool to use SMT solvers in order to reduce the number of false bugs reported due to imprecision in the analysis.

1.1 Scope of this Thesis

This thesis is focused on the automated verification of programs written in the C programming language and how tools and techniques can be improved to provide more accurate results in less time.

The justification for this choice is the broad adoption of the language in challenging and critical programming environments, including embedded systems, device drivers and APIs. Providing the means for verifying any application written in C will have a direct impact on the correct execution of this class of systems.

In the context of C program verification, I define my main research question in three parts:

1. Which technologies need to be developed to allow real-world programs to be modelled and verified by our in-house tool?

The development of technologies to handle real-world programs is a crucial step in verification. We present three new significant technologies in ESBMC, an SMT-based bounded model checker: a C frontend based on a state-of-the-art compiler, an SMT backend with support for floating-point arithmetic, and an incremental bounded model checking infrastructure.

The new clang-based [28] (a state-of-the-art compiler heavily used in industry) C frontend can easily parse and type-check real-world C programs; it brings several advantages but mainly it lessens the burden of maintaining a complete C frontend, allowing researchers to focus on their research. The SMT floating-point backend is a technology to improve the modelling of real-world programs using floating-point arithmetic: a limitation ESBMC had since its conception. Finally, the incremental bounded model checking algorithm is an infrastructure to develop incremental algorithms; in particular, it was the base on which we developed the *bkind* algorithm described below. The new technologies are described in Chapter 3.

2. How can software analysis and improvements to state-of-the-art algorithms reduce the amount of time required to verify programs?

Performance of verification tools is always an issue because of the state explosion problem. This problem has given risen to less accurate verification approaches, focused more on soundness than completeness. Alternatively, incomplete verification approaches such as bounded model checking, have been used as part of the induction algorithms to prove program correctness. We present the *bkind* algorithm, a novel improvement to the *k*-induction algorithm which extends the algorithm, so information produced by intermediate steps are not discarded but used to speed up program verification. Our novel *bkind* algorithm is formalized and evaluated in Chapter 4.

3. How can the quality of bug reports be improved by using more precise analysis to remove incorrect results, without slowing down the verification?

In order to apply verification tools in the industry, the performance issue was further improved by producing unsound and incomplete tools: a strange and sometimes unacceptable approach in academic environments. These tools, however, are widely used

because they have low requirements and can produce results fast; the inaccuracy produces both correct and incorrect results but results nevertheless, which is not always possible with more precise and sound tools, even with incomplete approaches. We present one incomplete and unsound symbolic execution-based tool, the clang static analyzer, and how incorrect results can be refuted using a more expensive approach (SMT solvers) while still keeping the performance acceptable by the industry standards. The new SMT-based refutation algorithm is described in Chapter 5.

1.2 Contributions

The main contribution of this thesis is the development and implementation of SMT-based tools to automatically verify C programs. In this respect, this thesis makes three significant novel contributions.

First, in Chapter 3 we describe the verification process in ESBMC, an SMT-based bounded model checker, and the new technologies developed in the tool to handle real-world programs. We evaluate the improved ESBMC using more than 9500 benchmarks and five different solvers (Z3, Boolector, MathSAT, Yices and CVC4). Regarding the new technologies developed, we present:

1. **A new clang-based frontend.** It was developed due to the high cost of maintaining a custom frontend for the ever-evolving C language. Clang is a C, C++ and Objective-C frontend for LLVM and its modular architecture allows us to reuse it as a frontend, bringing several advantages to our tool, including a stable format to represent a C program (called clang AST), compile-time program optimizations, and compilation warnings and errors consistent with a state-of-the-art C compiler. We also show some language features that were not previously supported. The clang-based frontend was presented in FOSDEM'17 [29] and the European LLVM Developers Meeting'18 [30].

2. **A new floating-point backend.** It was developed to improve the class of real-world C programs that ESBMC can handle. Since the tool's conception, ESBMC was only able to encode and verify programs using a fixed-point representation for floating-points; it is a valid approximation since fixed-points are used in a large number of applications in the embedded world, but it restricted ESBMC from verifying the broad set of programs that used floating-point arithmetic. The new floating-point API supports not only native encoding for the solvers that support the `FP` theory (Z3, MathSAT and CVC4) but can also encode floating-point arithmetic as bit-vectors, extending the `FP` theory to all solvers supported by ESBMC (including Boolector and Yices). The floating-point backend using the native encoding in Z3 and MathSAT was presented in SBMF'17 [31]. The bit-vector encoding has not yet been published.

3. **Incremental BMC (IBMC).** An algorithm used as the stepping stone for several algorithms, including *k*-induction. Two variants were developed: one that unwinds the state transition system of the program, and another that unwinds the program during the SMT encoding. ESBMC was the winner of the 2016 International Software Verification Competition in the array category using the former approach [32].

Second, in Chapter 4 we present the *k*-induction algorithm in ESBMC and the new *bkind* algorithm that improves its bug-finding capabilities. We exploit the counterexamples found by the inductive step of the *k*-induction to generate new properties and feed them back to the bounded model checking step, potentially reducing the number of iterations to find a property violation by half. We use a custom interval analyser to constraint the inductive step to lessen the number of unreachable counterexamples added to the bounded model checking step. Here, the main contribution is the use of information that is usually discarded by the *k*-induction algorithm. To the best of our knowledge, our extension has not previously been described or evaluated for software verification of C programs. Similar techniques exist, however, in other domains: Bischoff et al. [33] describe a technique called "target enlargement" which combines BDDs and SAT solvers to decrease the time to find property violations in hardware verification, and Jovanović et al. [34] describe a technique called "Property-Directed *k*-induction" to generate stronger invariants for programs written in the SALLY language. Our algorithm, however, uses SMT-based *k*-induction to generate counterexamples that are used in the verification of C programs. Our experimental evaluation shows that the new extension reduces the verification time when used in combination with the interval analyser without impacting the results. The original version of the *k*-induction algorithm (without the improve bug-finding capabilities) was published in the STTT journal in 2017 [35] and preliminary results of the *bkind* algorithm was presented in ESEC/FSE'18 [36]. The version of the *bkind* algorithm presented in this thesis has not been published yet.

Finally, in Chapter 5 we describe and evaluate a bug refutation extension for the Clang Static Analyzer (CSA) that addresses the limitations of the existing unsound built-in constraint solver. In particular, we complement CSA's existing heuristics that remove false bug reports. We encode the path constraints produced by CSA as SMT problems, use SMT solvers to precisely check them for satisfiability, and remove bug reports whose associated path constraints are unsatisfiable. We evaluated our extension when analysing twelve widely used C/C++ open-source projects of various size (tmux, Redis, OpenSSL, twin, git, PostgreSQL, sqlite3, curl, libWebM, Memcached, Xerces-c, and XNU) using five different SMT solvers (Z3, Boolector, MathSAT, Yices and CVC4). The results show that our refutation extension cannot only remove false bug reports in the majority of analysed projects but also speed-up the analysis time when bugs are refuted, while only introducing a small overhead when no bug is refuted. The new

refutation algorithm in the clang static analyzer was presented in the Bay Area LLVM Developers Meeting'18 [37].

1.3 Thesis Structure

In Chapter 2, we present the required background for the subsequent Chapters.

In Chapter 3, we present an in-depth description of the verification process in ESBMC and the technologies developed to improve it (namely the C frontend based on clang, the SMT backend with support for floating-point arithmetic, and the incremental bounded model checking infrastructure). We evaluate these technologies on a large set of benchmarks and compare the results using a variety of different SMT solvers.

In Chapter 4, we first present the *k*-induction algorithm and show how it uses bounded model checking to both find bugs and to prove correctness. We then present the *bkind* algorithm, which aims to reuse information produced by the *k*-induction algorithm to improve its bug-finding capabilities. We evaluate the *bkind* algorithm against the the *k*-induction algorithm in ESBMC and the k-induction-k-invariants algorithm in 2LS.

In Chapter 5, we present the clang static analyzer and how it can verify large real-world programs. We then present an improvement to the analyser, a more precise bug refutation algorithm, and evaluate the extension in many real-world open source projects.

Finally, in Chapter 6, we summarise the work presented in this thesis, discuss future works and conclude.

Chapter 2

Background

In this Chapter, we review the required background for the subsequent Chapters. The primary focus of this thesis is the improvements of existing verification approaches of C programs, in particular, bounded model checking and symbolic execution. In Section 2.1 we discuss the broader field of software verification to which bounded model checking and symbolic execution belong. We then define a basic notation and formalise programs in Section 2.2. In Section 2.3.1 we present the bounded model checking technique which is used in Chapters 3 and 4. We present symbolic execution in Section 2.3.2; this is important for Chapter 5. We discuss floating-point arithmetic (and some related works) in Section 2.4 which is important for Chapter 3. Finally, in Section 2.5 we present constraint solvers which are relevant to all work done in this thesis.

2.1 Software Verification in Practice

Until very recently the adoption of methods other than testing remained weak in industry, despite the development of more precise verification approaches in the 1960s [21]. Nowadays we find several companies using more sophisticated methods during the development of large software solutions: Amazon is using the TLA+ language specification to design systems, which has both found subtle bugs and proved correctness [38]; Microsoft is developing new theories to model and verify large-scale network configurations (2^{32} headers, approximately 820.000 rules) reducing verification times from more than 5 days to 2 hours [39]; the B method has been used to design correctly safe software of the metro line in Paris, generating real software that ran for 20 years without any bugs [40].

When using automated verification tools, however, is necessary to define what a software component should do and how it should do it, often referred to as functional and non-functional requirements. The requirements are defined when the system is being

designed but, in order to formally verify it, they have to be expressed in a manner that can be understandable by tools.

Research into defining a formal language for expressing formal specification first appeared in the 1960s and included languages such as TLA+ [41] and E-ACSL [42], allowing engineers to define and verify the expected behaviour of the system formally. Today, system designs lack proper formal specification, as often software engineers are not familiar with the techniques. Alternative approaches try to combine code and formal specification in a single language. These approaches formally verify specifications and generate machine code. For example, Dafny [43] and the B method [44] are programming languages that allow engineers to embed specifications in the code using common language constructs, reducing the gap between code and formal verification.

In this domain, a requirement is often called a property: an invariant in the system that should always hold [21]. A property can be expressed just as a variable that should always hold value, or as more complex behaviour such as that the program should always respond to requests. Properties can also be negatives, thus stating that a behaviour should never occur. Several programming languages also have built-in properties, stating that the behaviour is only defined under certain circumstances, such as requiring that pointers to invalid memory should never be dereferenced, or that iterators are invalidated after some operations. System and languages properties can be checked and, if a property is violated, the program may perform an illegal operation, produce a wrong output or abort during execution.

The properties can also be expressed in more elaborate ways, e.g., using temporal logic to check the evolution of the states of a program [45]. Temporal language such as *linear temporal logic* (LTL) [46] allows expression of how the system behaves over time, e.g., defining future behaviours of the system. Properties expressed using temporal logic are divided into two groups: safety and liveness [21]. A safety property expresses that no bad behaviour happens, while a liveness property expresses that something good eventually happens. Examples of safety properties include buffer overflows, memory safety, bounds checks and division by zero[1]. Termination guarantees are examples of liveness properties.

A property can also be expressed as a path through a model using LTL, in a way that describes how the path is taken through the state transition system evolves over time, e.g., a property might eventually hold in the future (a variable will hold a specific value somewhere in the future) or the next state after s_n will always be s_{n+1} (state s_2 will be the next state after s_1 for all paths). Properties can also be expressed using *computation tree logic* (CTL) [47] which quantifies over future paths that the program may take. LTL

[1]The now decommissioned USS Yorktown was fully automated in 1996 and, when a blank field was entered into a database, it was treated as zero and caused a divide-by-zero exception that crashed the operating system, a Microsoft Windows NT 4.0. The battleship was paralysed for almost 3 hours until the operating system was rebooted.

and CTL can express different properties: some properties can be described in CTL but not in LTL, and vice-versa [48]. Any property expressed in either logic, however, can be expressed in CTL* which is a superset of both logics [49].

Once properties and the systems are properly specified, some decision procedure is applied to reason about satisfiability. In particular, a decision procedure is *sound* if it never returns false answers, and it is *complete*, if it returns an answer for all inputs [21]. The halting problem, however, states that no algorithm can state if a program terminates for all inputs; this is an uncomputable problem [50]. The undecidable nature of the problem, however, does not make the field of formal methods an inapplicable science. In a way, it pushes the research in a more applied direction, similar to the field of applied mathematics: a complete and sound solution is impossible, but approximations still provide useful results.

Approximate algorithms either over-approximate or under-approximate the program behaviour. On the one hand, when an algorithm under-approximates a program, only a subset of the reachable state space is explored, stopping whenever some bound is reached (time, size, number of messages); it might be both incomplete and unsound [21]. Alternatively, over-approximation sacrifices soundness in favour of completeness; it explores a broader set of states than the set of reachable states [21].

Regardless of the chosen approach, verification tools usually terminate and provide three different verdicts given a program and a property: `TRUE`, `FALSE` or `UNKNOWN` [21]. An `UNKNOWN` result is often associated with exhausting time or memory limits.

A `FALSE` result indicates that the verification tool found an execution trace that violates the property. An execution trace is a deterministic path in the program, replacing sources of non-determinism (e.g., user input, thread scheduling) by actual values. A `TRUE` result indicates that the verification tool did not find any property violation and can provide new invariants to back this claim.

In practice, a `FALSE` result is useful because it provides means to reproduce unexpected behaviours, while a `TRUE` result is often seen as unreliable. It is challenging to trust `TRUE` results because several different pieces should come together to provide such strong claims: the property should be correctly specified, the program and property should have been correctly encoded, and the verification tool should be sound. Furthermore, even when the verification tool provides a proof, it is hard to generate this in a format that can be easily understandable by humans. These issues, however, do not demerit the verification tool's usefulness and even proving partial correctness increases the safety of the system.

A verification tool can also provide false positive and false negative results. A false negative result (a wrong `FALSE`) may indicate that the verification tool found an execution trace that violates the specification but the execution trace is unfeasible. These traces

can also be used to generate regression tests that can be quickly verified and dismissed, so their impact is relatively small.

Alternatively, a false positive result (a wrong `TRUE`) indicates that the verification tool did not find any property violation and reported that the program is safe. This is particularly bad when tools do not provide a way to check this claim. In most situations, it is better to claim an `UNKNOWN` result instead of the prospect of a false positive.

2.2 Program Formalization

We now define some mathematical formalism that will be used to formalise a program; it will be used in all Chapters.

Sets are denoted using capital Latin letters (e.g., A, B, C) and elements of these sets are denoted using lower Latin letters (e.g., a, b, c). We use different fonts to distinguish program lines and mathematical expressions. A program line uses `typewriter` font (e.g., `x = x + 1`) while a mathematical equation uses the *math* font (e.g., $x' = x + 1$).

The usual sets of numbers are denoted using blackboard letters: $\mathbb{N}$ is the set of natural numbers, $\mathbb{Z}$ is the set of integers, $\mathbb{Q}$ is the set of rational numbers and $\mathbb{R}$ is the set of real numbers. We define $\mathbb{B} = \{\top, \bot\}$ as the set of Booleans, where $\top$ is $true$ and $\bot$ is $false$, and we denote the projection operator that returns the one-indexed i^{th} element from a tuple v as $v|_i$ (e.g., $\{6, 10, 13\}|_1 = 6$).

We formalise programs by mathematically modelling the program semantics [51]. A program is formalised as a state transition system, which is equivalent to a simple programming language with no procedures. This representation is similar to the control flow graph (CFG) representation used by compilers [52] but we associate program statements with edges.

The syntax of the language we used to formalise a program is shown in Figure 2.1. In this grammar, $stmt$ defines the syntax of assignments, assertions and assumptions. An assertion statement checks if a Boolean expression holds while an assumption statement filters the state space such that a state is only reachable after an assumption if and only if the assumption Boolean expression holds. We reserve the symbol '$*$' for non-deterministic assignments. $bool\_expr$ denotes expressions over $\mathbb{B}$ and $expr$ denotes expressions over $\mathbb{Z}$, while op and rel_op are the usual operations over $\mathbb{B}$ and $\mathbb{Z}$, respectively.

We assume that every variable $v \in V$ range over the mathematical integers ($\mathbb{Z}$), however, when translating C programs the actual range of variables will be bounded by sizes defined by the C language standard [53].

$$
\begin{array}{lll}
\langle stmt \rangle & ::= \langle id \rangle := \langle expr \rangle & \text{// Assignment to a named variable} \\
& \mid \langle id \rangle := * & \text{// Non-deterministic assignment} \\
& \mid \texttt{assume}(\langle bool\_expr \rangle) & \text{// A guard} \\
& \mid \texttt{assert}(\langle bool\_expr \rangle) & \text{// A check} \\
& \mid \langle empty \rangle & \text{// No-op} \\
\\
\langle bool\_expr \rangle & ::= \langle expr \rangle \ \langle rel\_op \rangle \ \langle expr \rangle & \\
& \mid \langle bool\_expr \rangle \ \wedge \ \langle bool\_expr \rangle & \\
& \mid \langle bool\_expr \rangle \ \vee \ \langle bool\_expr \rangle & \\
& \mid \neg \langle bool\_expr \rangle & \\
\\
\langle expr \rangle & ::= \langle id \rangle & \\
& \mid \langle const \rangle & \text{// A numerical constant} \\
& \mid \langle expr \rangle \ \langle op \rangle \ \langle expr \rangle & \\
& \mid \ - \langle expr \rangle & \\
\\
\langle op \rangle & ::= + \mid - \mid * \mid / \mid \% & \\
\\
\langle rel\_op \rangle & ::= < \mid \leq \mid > \mid \geq \mid = \mid \neq &
\end{array}
$$

Figure 2.1: Grammar of the analyzer language.

Definition 2.1 (State Transition System)**.** A state transition system is a tuple (S, s_1, T), where S is the set of program states, s_1 is the starting state and T is the set of transitions $T \subseteq S \times S$.

Definition 2.2 (Error State)**.** An error state $\epsilon \in S$ represents a property violation in the original program. Reaching an error state in the state transition system is equivalent to either terminating the program unsuccessfully or to triggering undefined behaviour.

We assume that all error states are final states since any computation after these states is meaningless.

Definition 2.3 (Execution Path)**.** An execution path $[s_i, s_{i+1}, \ldots, s_j]$ is a sequence of $(j - i)$ unwindings of a transition relation $tr \in T$, such that for any consecutive state s_i and s_{i+1} there exists a transition from s_i to s_{i+1}.

Definition 2.4 (Reachable State)**.** A reachable state is any state s_i in an execution path in which the first state is the initial state s_1.

Definition 2.5 (Counterexample)**.** A counterexample π^k is an execution path $[s_1, \ldots, s_k]$ of length k such that $s_k = \epsilon$.

A counterexample is an execution path from the initial state s_1 to an error state ϵ.

Definition 2.6 (Partial counterexample)**.** A partial counterexample is an execution path $[s_i, \ldots, s_k]$ such that $s_k = \epsilon$, and $i > 1$.

Intuitively it is an execution path leading to an error state ϵ that does not start from the initial state s_1.

We introduce the term *formula* to represent the state transition system of a program and the properties it is expected to conform to. We operate over quantifier-free first-order logic formulas within a theory $\mathcal{T}$ such that the satisfiability problem is NP-hard. Suitable theories include linear real/integer arithmetic and propositional reasoning. A set of all formulas over a set of free variables is defined as $\mathcal{F}$.

A formula is said to be an *atom* if it is an indecomposable proposition, a *literal* if it is an atom (or its negation), and a *clause* if it is a disjunction of literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses.

We denote a formula P where all free occurrences of a variable y were replaced by x as $P[x/y]$. A formula $P \in \mathcal{F}$ is satisfiable if there exists a variable assignment $\mathcal{M}$ (a *model*) such that $P[\mathcal{M}/\mathrm{x}]$ is a tautology ($\mathcal{M} \vDash P$), e.g., $\{x \mapsto \top, b \mapsto \top\} \vDash a \vee b$.

Checking the satisfiability of quantifier-free formulas is a classic NP-complete problem [54]. However, modern *Boolean satisfiability* (SAT) and *satisfiability modulo theory* (SMT) solvers can in practice often solve formulas very efficiently: the worst-case computational complexity is not necessarily relevant to queries posed to the solver.

In the literature, a property is often described as a *verification condition* (VC) [24, 55, 56]. Given a program represented as a state transition system, the simplest property to be checked is reachability. The violation of a reachability property in a state transition system is a sequence of states, from the initial state until an error state; effectively it is a counterexample. While this may seem obvious, the property becomes more difficult to check if branches are taken under consideration. For instance, if a branch must be chosen depending on user input, both sides of a branch must be considered because one of them might be in the counterexample. One can also try to compute all reachable states from the initial state, but the problem might become impractically large.

2.2.1 Encoding Approaches

The first verification tools, such as SPIN [57] and SMV [58] model the program as a state transition system and check properties against them. SPIN [57] is probably the most successful verification tool available, receiving the ACM System Software Award in April 2002, and it is mostly aimed at hardware verification. SPIN uses a language called PROMELA to describe the state transition system, including variable assignments and conditional branching. SPIN also supports LTL properties expressed in-program, which are negated and then converted into Büchi automata [59] as part of the verification process. SMV also uses a custom description language to represent a transition system and allows properties to be expressed in CTL [48].

The naïve approach to verify these state transition systems is to explore all the paths in the system. However, even finite state systems with loops can have infinite paths. SMV computes fixed-points over a symbolic formula while SPIN records visited states to avoid re-exploring previously visited paths.

SPIN and SMV also differ in how they represent the state space. SPIN enumerates the states *explicitly* while SMV enumerates them *symbolically*. When enumerating the states explicitly, the state of the system is represented as the explicit value of all variables and the program counter. The biggest disadvantage of this approach is the state space explosion, as the amount of memory required to store all states is proportional to the state space. When enumerating the states symbolically, the values are not explicitly stored, but the state encoding symbolically represents their values (e.g., storing the range of values a variable can assume, instead of every single value).

McMillan [60] made one of the first attempts at implementing a symbolic verification tool. The tool encodes the system using μ-calculus [61] and a binary decision diagram (BDD) [62] is built to evaluate the reachability of states. Later, BDDs were replaced with SAT (Boolean Satisfiability) in symbolic verification [22], which shifted the verification approach to falsification since, at least for falsification, SAT-based verification tools scale better [63].

Recent researches also replaced SAT with SMT (Satisfiability Modulo Theory) [24, 64, 65]. The main reason for this change is that some programs require determining the satisfiability of formulas (e.g., integer arithmetic) which SAT solvers are not able to solve directly [8].

2.2.2 Encoding Programs and Checking Properties

We define an operator $\texttt{OP} \in \texttt{OPS}$ as a formula $\tau(\texttt{i} \cup \texttt{i'})$ over the initial variables `i` and subsequent variable `i'` such that for every pair of models $(\mathcal{M}_1, \mathcal{M}_2)$, there is $\mathcal{M}_2 \in \texttt{OP}(\mathcal{M}_1)$ if and only if $(\mathcal{M}_1 \cup \mathcal{M}_2[\texttt{i/i'}]) \vDash \tau$. This is equivalent to say that a formula is satisfiable over $(\mathcal{M}_1 \cup \mathcal{M}_2)$ if an operator applied to $\mathcal{M}_1$ generates $\mathcal{M}_2$.

As an example, let us assume a program with two variables, `i` and `j`, and a constraint $\texttt{i} < 10$. This is represented by a formula:

$$i < 10 \wedge i' = i \wedge j' = j \tag{2.1}$$

while an assignment `i := i + 1` is represented as:

$$i' = i + 1 \wedge j' = j \tag{2.2}$$

The number of *frame* assignments which state that all unmodified variables remain unchanged will increase with the number of variables; this is similar to the frame problem in the artificial intelligence field [66]. In practice, this is encoded using single static assignments (SSA) [67], which renames variables in such a way that every variable is assigned only once.

```
1 int main(){
2   i = 0;
3   y = i;
4   i = y + i;
5 }
```

(a) Simple program with assignments.

```
1 i_1 := 0;
2 y_1 := i_1;
3 i_2 := y_1 + i_1;
```

(b) SSA of the program.

Figure 2.2: Unwinding assertion and assumption representation.

Figure 2.2 shows an example of a program and its SSA form; the formula encoding of the program is:

$$(i_1 = 0 \wedge y_1 = i_1 \wedge i_2 = y_1 + i_1) \tag{2.3}$$

We can also encode strongest post-conditions by converting the operator to formulas. Given a formula $\phi(\mathtt{i})$ and a transition $\tau(\mathtt{i} \cup \mathtt{i'})$, a formula representing the strongest post-conditions is $(\exists \mathtt{i}.\phi(\mathtt{i}) \wedge \tau(\mathtt{i} \cup \mathtt{i'}))[\mathtt{i'}/\mathtt{i}]$, e.g., a post-condition $i < 10$ under a transition $i' = i + 1$ is $(\exists \mathtt{i}.\mathtt{i} < 10 \wedge \mathtt{i'} = \mathtt{i} + 1)[\mathtt{i'}/\mathtt{i}]$ which is $\mathtt{i} < 11$.

For programs without loops, encoding a program and checking for satisfiability is enough to prove its safety: the entire program can be converted to a formula $P(\mathtt{x})$, a desired property is encoded as $\phi(\mathtt{x})$ and a constraint solver can be queried for $P(\mathtt{x}) \wedge \neg\phi(\mathtt{x})$. If the formula is satisfiable, a model $\mathcal{M} \models P(\mathtt{x}) \wedge \neg\phi(\mathtt{x})$ will be produced by the constraint solver which can be used to automatically generate a failure witness [68]. Otherwise, the program is safe.

This reasoning cannot be directly applied to unbounded programs. Safety is generally proven for such infinite systems by using inductive invariants [51]. Such properties can sometimes be proven by induction: in order to prove that the invariant holds universally, first it is shown that the invariants hold for the initial states, then that they hold under the transition relation.

Definition 2.7 (Inductive invariant)**.** A property $\varphi(\mathtt{x})$ is an inductive invariant for a state transition system P if and only if φ satisfies the initial state condition and it is inductive under a state encoding of P, i.e., $\forall \mathtt{x}, \mathtt{x}' : \varphi(\mathtt{x}) \wedge tr(\mathtt{x}, \mathtt{x}') \implies \varphi(\mathtt{x}')$.

In the literature, proving properties by finding inductive invariants is often based on the notion of abstract domains, called *abstract interpretation* [69]. An abstract domain is a set where every element groups states by property of interest, e.g., an interval abstract

domain groups a range of states for every single variable in a state transition system. Several different abstract domains exist, and they differ on how the states are grouped in the set.

An *interval abstract domain* is a map $\mathtt{x} \to \mathbb{R} \times \mathbb{R}$. An element $\{x : [a, b]\}$ in this domain is a range $\{x \geq a \wedge x \leq b\}$ and the comparison is given using the component-wise comparison on tuples, applied component-wise to maps.

One of the limitations of abstract interval domains is that they are not relational: it is not possible to express the relation between variables. An *octagon abstract domain* [70] is a more expressive abstract domain that represents program variables $x, y \in X$ as bounds on expressions $\pm x \pm y \leq c$. A *polyhedra abstract domain* [71] generalises the convex abstract state even further by allowing an element in the abstract domain to be any convex polyhedron.

Template constraint domains [72] are a compromise between scalability and precision. They are parameterized by a vector of functions of $\mathtt{x}$ (called *templates*, e.g., $2x + 5y$), and an element in this domain is a vector of bounds on these functions.

2.3 Static Analysis Approaches

In this Section we present two automated static analysis approaches: *bounded model checking* and *symbolic execution*. The former is used as part of an algorithm used to find property violation and to prove correctness described in Chapter 4 and the latter is used to analyse programs in Chapter 5.

Compared to other verification techniques, static analysis offers three advantages:

- It is fully automatic. Static analysers are *push-button* tools, i.e., they require no supervision and the user is not required to know logic or theorem proving.
- When the static analyser finds a property violation on the program, it produces a counterexample (also called witness) that demonstrates the sequence of states that violated the property.
- It can reason about a program safety using symbolic values instead of concrete values.

A static analyser might generate a huge state space, often requiring large amounts of resource in order to perform the verification. State-of-the-art static analysers can handle state spaces of about 10^8 to 10^9 with explicit state-space enumeration. In specific problems, by making use of smart algorithms and simplification strategies, a larger state space (10^{20} up to 10^{420}) can be handled by static analysers [8].

Techniques have been developed in order to reduce the amount of resources required to perform the complete verification. These methods suffer from unsoundness or incompleteness (or both in some cases) but are capable of presenting results with limited resources: any result is better than no result at all. In particular, symbolic execution [73] tools often explore different paths in a program while keeping the variables symbolic, encoding the path when a final state is reached and checking the path satisfiability. Symbolic execution tools are often used to find bugs in real-world software[2]. A hybrid approach between actual execution and symbolic exploration, *concolic testing*, uses concrete inputs generated by constraints solvers to maximise coverage [75]. Bounded model checking tools explore the state transition system up to a bound k and can prove that no bug is reachable within that many iterations; bounded model checking tools were also used to find bugs in real-world single- and multithreaded programs [25]. Both methods are usually incomplete, as they tend to explore regions of the state space, rather than all of it. Unsoundness in these approaches is usually related to approximation performed during the verification.

Symbolic execution and bounded model checking tools employ different strategies to explore state transition systems: while symbolic execution tools apply a depth-first search (DFS) approach (going from an initial state all the way to a final state before backtracking), bounded model checking tools apply a breadth-first search (BFS) approach (exploring all neighbouring states before moving to the next depth level).

2.3.1 Bounded Model Checking

The bounded model checking (BMC) technique was developed to contain the complexity that a system presents to a model checker; otherwise, it might need an infinite amount of resource (e.g., processor, memory, time) to complete the verification of all possible states [21].

In bounded model checking, loops and recursions will only be unwound up to a given bound k [65]; for this reason, BMC is mainly used to prove falsification of properties as they do not check the state space for bounds greater than k.

The unwinding process is essential to the bounded model checking technique, as it is responsible for restricting the state space explored by the algorithm. Given a bound k, it removes states from the verification that are only reachable for unwindings $> k$ while preserving the program behaviour for loop unwindings $\leq k$. In practice, this is done by using either *unwinding assumptions* that symbolically encode state-space constraints or *unwinding assertions* that can produce counterexamples to indicate that the state space was not fully explored.

[2] Recently, CPPChecker was able to find a buffer overflow in libXInput that was introduced in 1991 [74].

An unwinding assertion prevents the tool from presenting a false negative result; it detects if the states beyond the loop bound were not evaluated during the verification. However, an unwinding assumption removes states beyond the loop, and if ill-chosen may remove all states, leading to false safety claims. Removing states from the verification is sometimes desirable, as described in Chapter 4.

```
1 void assert(_Bool cond)
2 {
3   if(!cond)
4     exit(EXIT_FAILURE);
5 }
```

(a) General representation of an assertion.

```
1 void assume(_Bool cond)
2 {
3   if(!cond)
4     exit(EXIT_SUCCESS);
5 }
```

(b) General representation of an assumption.

Figure 2.3: Unwinding assertion and assumption representation.

Figure 2.3 presents the general representation of an assertion and an assumption. In an assertion, a program exits with failure if the condition does not hold, while in an assumption, a program exits with success if the condition does not hold (only paths that satisfy the condition are allowed to continue). An unwinding assertion/assumption is an assertion/assumption where the condition `cond` is the loop termination condition.

Formally, given a state transition system of a program P unwound k times, the bounded model checking procedure can be formulated as:

$$\pi = init(s_1) \wedge \bigwedge_{i=1}^{k-1} tr(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\phi(s_i) \tag{2.4}$$

Where a predicate $init(s_1)$ denotes that s_1 is the initial state of an unwound program P, a transition $tr(s_i, s_{i+1}) \in T$ is a transition from s_i to s_{i+1}, $\phi(s_i)$ is a safety property and π is a counterexample (non-empty if the formula is satisfiable).

2.3.2 Symbolic Execution

Symbolic execution is another technique developed to address the complexity of verifying large systems [73]. In a symbolic execution verification, all possible paths in a program are enumerated and evaluated individually. The technique has seen renewed interest and has been implemented in several different tools [76, 77, 78, 79] and applied to many different problems, from software engineering to security [80].

Similar to bounded model checking, symbolic execution will symbolically explore the state space, collect constraints and check for the reachability of bugs. The main difference is the behaviour when a branch is found: while a bounded model checking tool

will evaluate both sides and merge the states after the branch, a symbolic execution tool will explore each branch separately, making a copy (known as *forking*) of the current state. This process will happen for every branch in the program until an error state is reached, at which point all the constraints in this path (known as *path constraints*) are encoded and checked for satisfiability. Similar to bounded model checking, a model is generated if the formula is satisfiable and a test case can be produced.

If the path constraints are unsatisfiable, the symbolic execution tool backtracks to the last visited branch (removing every constraint added when evaluating the unsatisfiable side of the branch) and explores the other side of the branch.

A symbolic execution tool will explore all these branches and encode the program twice: for the third and the fifth paths, where there is a possible property violation. The path exploration grows exponentially with the number of branches; it is often referred to as *path explosion*. Tools like KLEE [75] employ several optimisations in order to reduce the number of paths explored, including caching previous queries and checking if branch path constraints are satisfiable before exploring them.

One advantage of symbolic execution tools over bounded model checkers is that the former can be easily used for coverage test generation. Symbolic execution tools explore the state space using a DFS algorithm so a test case can be generated for each explored path of the program: every time a final state is reached, the underlying solver can be queried for a model that satisfies all the constraints in that path [81]. BMC tools, on the other hand, explore the state space using a BFS algorithm, so coverage test generation is trickier, e.g., Angeletti et al. [82] describe a method to generate a set of tests for coverage by inserting errors in the source code (i.e., `assert(0)`), and running CBMC to find counterexamples for these properties.

2.4 Floating-point Arithmetic

In this Section, we discuss floating-point arithmetic; it is relevant to Chapter 3 where we present a new SMT encoding with support for floating-point arithmetic.

The manipulation of real values in programs is a necessity in many fields, e.g., scientific programming [83]. The set of real numbers, however, is infinite and some numbers cannot be represented with finite precision, e.g., irrational numbers. Over the years, computer manufacturers have experimented with different machine representations for real numbers [84]. The two fundamental ways to encode a real number are the fixed-point representation, usually found in embedded microprocessors and microcontrollers [85], and the floating-point representation, in particular, the IEEE floating-point standard (IEEE 754-2008 [86]), which has been adopted by many processors [87].

Each encoding can represent a range of real numbers depending on the word-length and how the bits are distributed. A fixed-point representation of a number consists of an integer component, a fractional component and a bit for the sign, while the floating-point representation consists of an exponent component, a significand component and a bit for the sign. Floating-point has a higher dynamic range than fixed-point (e.g., a `float` in C has 24 bits of precision, but can have values up to 2^{127}), while fixed-point can have higher precision than floating-point [88]. Furthermore, the IEEE floating-point standard contains definitions that have no direct equivalent in a fixed-point encoding, e.g., positive and negative infinities (`+/-inf`).

In general, IEEE floating-point numbers are of the following kinds [86]:

Zeroes. Both `+0` and `-0` are defined in the standard. Most of the operations will behave identically when presented with `+0` or `-0` except when extracting the sign bit or dividing by zero (usual rules about signedness apply and will result in either `+inf` or `-inf`). Equalities will even be evaluated to true when comparing positive against negative zeroes.

NaNs. The **N**ot **a** **N**umber special values represent undefined or unrepresentable values, e.g., $\sqrt{-1}$ or `0.f/0.f`. As a safety measure, most of the operations will return NaN if at least one operator is NaN, as a way to indicate that the computation is invalid. NaNs are not comparable: with the exception of the not equal operator (`!=`), all other comparisons will return false (even comparing a NaN against itself). Furthermore, casting NaNs to integers has undefined behaviour.

Infinities. Both `+inf` and `-inf` are defined in the standard. These numbers represent overflows or the result of non-zero number divisions by zero (if the implementation of C respects Annex F of the C language specification [53]).

Normal Numbers. A non-zero number which can be represented within the range supported by the encoding.

Denormal Numbers (or Subnormal Numbers). A non-zero number representing values very close to zero, filling the gap between what can be usually represented by the encoding and zero.

The IEEE standard also defines five kinds of exceptions. These exceptions are to be raised under specific conditions:

Invalid Operation. This exception is raised when the operation produces a NaN as a result.

Overflow. This exception is raised when the result of an operation is too large to be represented by the encoding. By default, these operations return $\pm$`inf`.

Division By Zero. It is raised by `x/`$\pm$`0`, for `x`$\neq$`0`. By default, these operations return $\pm$`inf`.

Underflow. Raised when the result is too small to be represented by the encoding. The result is a denormal floating-point.

Inexact. This exception is raised when the encoding cannot represent the result of an operation unless it is rounded. By default, these operations will round the result.

The standard defines four rounding modes. Given a real number x, a rounded floating-point $r(x)$ will be rounded using:

Round Toward Positive (RTP). $r(x)$ is the least floating-point value $\geq x$.

Round Toward Negative (RTN). $r(x)$ is the greatest floating-point value $\leq x$.

Round Toward Zero (RTZ). $r(x)$ is the floating-point with the same sign of x, such that $|r(x)|$ is the greatest floating-point value $\leq |x|$.

Round To Nearest (RTN). $r(x)$ is the floating-point value closest to x; if two floating-point values are equidistant to x, $r(x)$ is the one which the least significant bit is zero.

Finally, the standard defines some arithmetic operations (add, subtract, multiply, divide, square root, fused multiply-add, remainder), conversions (between formats, to and from strings), and comparisons and total ordering.

Name	Common Name	Size (exponent + significand)
fp16	Half precision	16 (5 + 10)
fp32	Single precision	32 (8 + 23)
fp64	Double precision	64 (11 + 53)
fp128	Quadruple precision	128 (15 + 113)

Table 2.1: IEEE floating-point types.

The standard defines how floating-point are to be encoded using bit-vectors. Table 2.1 shows four primitive types usually available in the x86 family of processors that follow the standard; each type is divided into three parts: one bit for the sign, an exponent and a significand part which depends on the bit length of the type. The significands also include a hidden bit: a 1 bit that is assumed to be the leading part of the significand, unless the number is denormal.

In Annex F of the C language specification [53], fp32 and fp64 are defined as `float` and `double`. The standard does not define any types for fp16, and compilers usually implement two formats: `__fp16` as defined in the ARM C language extension (ACLE) [89] and `_Float16` as defined by the ISO/IEC 18661-3:2015 standard [90]. While `__fp16` is only a storage and interchange format (meaning that it is promoted when used in arithmetic operations), `_Float16` is an actual type, and arithmetic operations are performed using

half precision. The standard only weakly specifies how a fp128 (`long doubles`) should be implemented, and compilers usually implement it using an 80-bit long double extended precision format [87].

Floating-point numbers are represented as $(-1)^{sign} \times significand \times 2^{exponent}$. Here, $1 \leq significand \leq 2$ and $2^{exponent}$ is the scaling factor. For example, -0.75 is written as -1.5×2^{-1}, or the 32-bit bit-vector $\langle 10111111110000000000000000000000 \rangle$.

Normal floating-point numbers are encoded assuming that the leading hidden bit is 1 and the exponent is in the range $[-exponent_{max} + 1, exponent_{max}]$. Denormals are encoded assuming that the leading hidden bit is zero and the exponent is $-exponent_{max}$. Zeros are represented as an all zero bit-vector (except for the sign bit if the zero is negative). Finally, a bit-vector with the exponent equal to $exponent_{max}$ and significand all zero is an infinity, and a bit-vector with an exponent equal to $exponent_{max}$ and significand not zero is a NaN.

2.4.1 Verifying Programs with Floating-point Arithmetic

The analysis of programs with floating-point arithmetic has received much attention, especially when safety depends on the correctness of these programs: the Ariane 5 rocket exploded mid-air in 1996 due to an exception thrown by an invalid floating-point conversion [5]. It is a complex problem because the semantics may change beyond code level, including the optimisation performed by compilers [83].

Several symbolic execution tools try to verify programs with floating-point arithmetic by employing different strategies: CoverMe [91] reformulates floating-point constraints as mathematical optimisation problems and uses a specially built solver called XSat [92] to check for satisfiability. Pex [93] uses a similar approach and reasons about floating-point constraints as a search problem, and they are solved by using meta-heuristics search methods. FPSE [94] models floating-point arithmetic by using an interval solver over real arithmetic combined with projection functions. HSE [95] extends KLEE [75] to symbolically execute the program and convert floating-point numbers into bit-vectors; it then uses SMT solvers to reason about satisfiability.

Bounded model checkers have also been applied to verify programs with floating-point arithmetic: CBMC [23] and 2LS [96] convert floating-point operations to bit-vectors and use SAT solvers to reason about satisfiability. CPBPV [97] uses bounded model checking combined with their FPCS [98] interval solver to generate tests that violate output constraints in the program.

2.5 Constraint Solvers

In this Section, we introduce SAT and SMT solvers: constraint solvers with the goal of proving whether or not a given set of constraints is satisfiable. In particular, SMT solvers are the engine that makes the work in this thesis possible; every contribution we made encodes properties in SMT and checks their satisfiability using SMT solvers.

2.5.1 SAT and SMT Solvers

Boolean constraints are the simplest types of constraints that can be solved: they consist of one or more Boolean variables, operators OR ($\vee$), operators AND ($\wedge$), or operators NOT ($\neg$). Checking the satisfiability of a set of Boolean constraints in the conjunctive normal form (CNF) form is known as an SAT problem. A significant amount of research has been put in SAT solvers, as they have a wide range of direct applications (e.g., hardware verification); in particular, modern SAT solvers use the conflict-driven clause learning algorithm (CDCL) [99]. The CDCL algorithm was inspired by another algorithm, the Davis-Putnam-Logemann-Loveland algorithm (DPLL) [100] to check for satisfiability. The DPLL algorithm essentially consists of two steps: (1) choose a truth value for a literal (2) propagate the implications of this decision; it is also known as *unit propagation* and can simplify large sets of clauses. The CDCL algorithm extends the DPLL algorithm to learn with conflicts and to derive new clauses to avoid reaching the same conflict again. Example of SAT solvers include minisat [101] and cryptosat [102].

Boolean constraints, however, are not the only type of constraints that can be solved. In particular, encoding programs as constraint problems is more straightforward if the operations do not have to be converted [103] to SAT problems. Theories of bit-vectors, floating-point arithmetic, quantifiers, arrays and uninterpreted functions are available in satisfiability modulo theories (SMT) solvers; these can check for satisfiability of formulas encoded using a combination of these concepts.

In SMT, a theory is a collection of types (known as *sorts*), functions over those types, and axioms over types and functions; it also describes the language and semantics for the encoding of constraints. SMT-LIB v2.6 [104] is the most popular standard that describes an SMT language, the theories and logics (a collection of one or more theories) and example constraint sets. Theories include *ArraysEx* (functional arrays with extensionality), *FixedSizeBitVectors* (bit-vectors with arbitrary size), *Core* (core theory, defining the basic Boolean operators), *FloatingPoint* (floating-point numbers), *Ints* (integer numbers), *Reals* (real numbers) and *Reals_Ints* (Real and integer numbers).

The theories are combined in the form of logics and Figure 2.4 show all the logics defined in SMT-LIB standard. The naming convention is as follows: **QF** for quantifier free formulas, **A** or **AX** for the ArraysEx theory, **BV** for the FixedSizeBitVectors theory, **FP**

for the FloatingPoint theory, **IA** for the Ints theory, **RA** for the Reals theory, **IRA** for the Reals_Ints theory, **IDL** for Integer Difference Logic, **RDL** for Rational Difference Logic, **L** before **IA**, **RA**, or **IRA** for the linear fragment of those arithmetics, **N** before **IA**, **RA**, or **IRA** denotes the non-linear fragment of those arithmetics, **UF** for undefined function symbols.

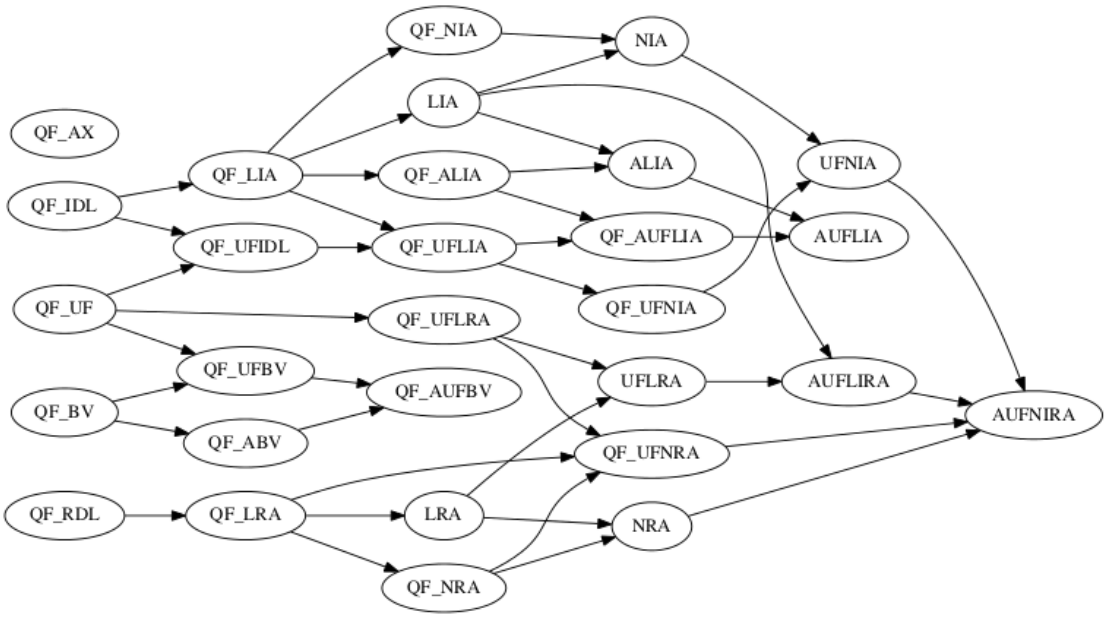


Figure 2.4: Logics defined in the SMT-LIB.
Source: http://smtlib.cs.uiowa.edu/logics.shtml

State-of-the-art SMT solvers are built on top of SAT solvers to speed up their performance. Allowing higher level constraints will impact in the solver's ability to solve a formula efficiently; that is the reason why SAT solvers are embedded in SMT solvers: an SMT formula can be reduced to SAT and checked using the efficient DPLL(T) strategy [105]. If the formula is satisfiable, the model of the SMT formula can also be reconstructed using the model of the SAT formula. SMT solvers often perform this process when the SMT formula does not contain quantifiers.

```
1 (set-logic QF_BV)
2 (declare-fun $3 () (_ BitVec 64))
3 (declare-fun $1 () (_ BitVec 64))
4
5 (assert (= $1 $3))
6
7 (check-sat)
8 (get-model)
```

(a) Simple SMT formula.

```
1 sat
2 (model
3   (define-fun $3 () (_ BitVec 64)
4     #x0000000000000000)
5   (define-fun $1 () (_ BitVec 64)
6     #x0000000000000000)
7 )
```

(b) Model generated by an SMT solver.

Figure 2.5: An SMT formula and one model generated by an SMT solver.

Figure 2.5 show a satisfiable formula and one model found by Z3 [106] when checking this formula. Line 1 defines the logic that will be used in this formula: QF_BV; it is the logic that uses bit-vectors without quantifiers. Two variables are defined in lines 2 and 3, `$3` and `$1` and they are both bit-vectors of size 64 (note that the names include a dollar sign, which is useful when we generate SSA with different indexes in Chapter 3).

In line 5, a constraint is defined to check if `$3` and `$1` are the same bit-vector. Note that these variables are not initialised: they are free variables and can assume any bit-vector pattern. Using them without initialisation is not undefined behaviour as would be using uninitialized variables in the C language; here they are defined but unknown.

Finally, line 7 tells the SMT solver to check for satisfiability and line 8 tells the SMT solver to print the model (shown in Figure 2.5b), if any.

Figure 2.5b shows the model produced by the solver. Line 1 shows the verification result: the formula is satisfiable and the model starting in line 2 shows the assignments to the variables `$1` and `$3` that satisfy the SMT formula.

2.5.2 Floating-point Arithmetic support in SMT Solvers

There are several different strategies to solve SMT formulas with floating-point arithmetic. It is tempting to use a real arithmetic strategy to solve a formula with floating-point arithmetic but, as we seen previously, the floating-point arithmetic is an approximation of the arithmetic and introduces concepts that are not found in standard real arithmetic, e.g., NaNs.

One approach (which we implement in Chapter 3) is to convert all floating-point arithmetic to bit-vector operations; a process also called as bit-blast. It is also the approach used by Z3 [106] and SONOLAR [107]. This approach, however, produces large formulas that increase the complexity of the satisfiability check [108]. Bit-blasting floating-point arithmetic also removes the possibility of high-level reasoning about the problem.

MathSAT [109] implements a different strategy called abstract conflict-driven clause learning (ACDL) combined with interval solvers [110]. This allows the reasoning to be performed in the floating-point theory itself and the authors claim that this strategy is frequently faster than the bit-blasting strategy. Oddly enough, this is not the default behaviour when solving formulas with floating-point arithmetic in MathSAT.

Another method (implemented in a fork of Z3) is a framework for applying approximations to constraints [111]. The approximated formula is then checked and iteratively refined until a model is found or until it is no longer possible to refine the approximation; it is similar to counterexample-guided abstract refinement [112]. The approximation used in this approach is a floating-point arithmetic with less precision that is

incremented after each unsatisfiable approximated model. The authors claim that this approach is on a par with the ACDL approach in MathSAT.

REALIZER uses a different approach and encodes the impact of the rounding operations in the floating-point arithmetic using real arithmetic [113]. The floating-point formula is then checked for satisfiability using an SMT solver with support for real arithmetic. The authors show that this approach is well suitable to check the satisfiability of formulas that combine floating-point and real arithmetic; in particular, authors show how this approach can be used when developing software to track the deviation of floating-point arithmetic.

Finally, XSat [92] and goSAT [114] reformulate the floating-point arithmetic as mathematical optimisation problems, then apply optimisation techniques to check the formula satisfiability. The authors claim that their approach is 700x faster than Z3. However, it can produce a small number of incorrect results.

Chapter 3

Verifying C Programs: The Development of a Compiler-based Frontend to a Floating-point Backend

Abstract. *We describe and evaluate the verification process in ESBMC, an SMT-based bounded model checker, and three new technologies developed to address the challenge of correctly modelling a C program. In particular, we developed (1) a new frontend based on a state-of-the-art C compiler, (2) a new SMT backend in ESBMC that can precisely encode floating-point arithmetic, either using native APIs (in solvers that support it) or by bit-blasting the operations (for all solvers in ESBMC), and (3) a new incremental bounded model checking infrastructure to be used a base for several incremental algorithms in ESBMC. We evaluate these technologies using a large set of publicly available benchmarks and compare the performance of ESBMC using five different solvers: CVC4, Yices, MathSAT, Z3 and Boolector. The results show that ESBMC can correctly verify a large number of benchmarks while only providing a small number of incorrect results. These results also serve as evidence to support Boolector as our default SMT solver: ESBMC can provide more answers in less time when using it to verify C programs. Furthermore, when verifying program with floating-point arithmetic, our new bit-blasting API for floating-points produced no wrong answers and Boolector (an SMT solver that does not support floating-point natively), was able to produce more answers than the other solvers while only being 6% slower than the fastest SMT solver with native support for floating-point, MathSAT.*

Software verification tools operate by converting their input (e.g., a program source code) into a format understandable by an automated theorem prover, encoding high-level program properties (e.g., pointer safety) and algorithms (e.g., bounded model

checking) into low-level equations (e.g., SMT). The encoding process of a program usually involves several intermediate steps, designed to generate a formula that can be efficiently solved by the theorem provers.

In this Chapter we present the verification process in one bounded model checker, ESBMC (**E**fficient **S**MT-based Context-**B**ounded **M**odel **C**hecker) [24, 115, 116, 117], the new technologies developed in the tool, and evaluate ESBMC using a large set of benchmarks [118].

The main improvements ESBMC are the new clang-based [28] C frontend (Section 3.3), a floating-point backend (Section 3.6), and the incremental bounded model checking infrastructure (Section 3.7). We developed the new clang-based frontend, which is now the default frontend in ESBMC. It supports all C11 features [53], and it is a fraction of the size of the previous CBMC-based frontend; this improves maintainability. The new floating-point backend is a generic floating-point SMT API that extends the floating-point feature to all solvers (including Boolector [119] and Yices [120] that currently do not support the SMT `FP` logic [121]). The incremental bounded model checking infrastructure allowed the easy development of several algorithms, including the *k*-induction and *bkind* algorithms described in Chapter 4.

For evaluation, we used the benchmarks of the International Competition on Software Verification (SV-COMP), from the 2018 edition [122] for verification. SV-COMP is, to the best of our knowledge, the most extensive public set of C programs available today, with almost 10000 benchmarks with known verification verdicts for well-defined properties, ranging from bug reachability to memory safety, integer overflow and termination. We, however, do not present the results using the SV-COMP *scores*. ESBMC reports wrong results for some benchmarks (mostly due to internal bugs), and wrong benchmarks seriously penalise the scores. In this scenario, an improvement to ESBMC might increase both the number of correct and incorrect results, but the score will be lower. As we are interested in evaluating the new improvements in ESBMC, the SV-COMP score is not helpful.

The five different solvers supported by ESBMC were evaluated (Z3 [106], Yices [120], Boolector [119], MathSAT [109], and CVC4 [123]) and ESBMC is able to evaluate more benchmarks within the usual time and memory limits (15 minutes and 15GB, respectively) when using Boolector. In particular, results show that Boolector can solve more floating-point problems using the new floating-point API than MathSAT or Z3 which have native floating-point APIs.

Organization. In Section 3.1 we present an illustrative example to be used in the remainder of the Chapter to explain the verification process in ESBMC. Section 3.2 presents the architecture of ESBMC. Section 3.3 presents the new clang-based frontend for C programs and the program abstract syntax tree (AST) of the illustrative example. Section 3.4 shows the simplification process in which the AST is converted into a state

transition system and the extra processing performed in this step. Section 3.5 shows how the state transition system is unwound and how the program and its properties are prepared to be converted to SMT. Section 3.6 presents the solvers supported by ESBMC, how scalars and fixed-points are encoded in SMT using bit-vectors, and the new floating-point API, which encodes floating-point arithmetic according to the IEEE floating-point standard. Section 3.7 discusses the advantages of incremental bounded model checking and its usage in ESBMC. Section 3.8 shows the comparative results for all the benchmarks of SV-COMP18 with five different solvers. Section 3.9 presents some related tools and their verification process, and Section 3.10 draws conclusions.

3.1 Illustrative Example

We use the C program in Figure 3.1 as a running example to illustrate the verification process in ESBMC. It is a simple program that multiplies an initial `double` number greater than zero by 2, `N` times, and checks after each multiplication if the number is still positive.

```
1  #include<assert.h>
2  int main() {
3    double x = nondet_double();
4    if(x <= 0.)
5      return 0;
6
7    unsigned int N = nondet_uint();
8    unsigned int i = 0;
9    while(i < N) {
10     x = (2 * x);
11     assert(x > 0);
12     ++i;
13   }
14   assert(x > 0);
15   return 0;
16 }
```

Figure 3.1: A small C program with a subtle error triggered if `x` is NaN; it will be used as a running example to explain the verification process in ESBMC. Here `nondet_uint()` and `nondet_double()` return for non-deterministic integer and double values, respectively.

Note that the program contains one subtle bug. The asserts in the program (lines 11 and 14) always hold if the initial value of `x` is a normal floating-point or `+/-inf` (floating-points overflow to `+/-inf`). However, the assertions will fail if the initial value of `x` is a NaN (**N**ot **a** **N**umber, used to represent an undefined or unrepresentable value). The guard in line 4 will not prevent the program from running, since comparing NaN to any floating-point (including itself) is always false. If the `x` is NaN, the first assertion will fail if `N` is greater than zero, otherwise, the second assertion will fail.

3.2 ESBMC Architecture

Figure 3.2 shows the current architecture of ESBMC; white rectangles represent input and output while grey rectangles represent the verification steps. The tool is composed of several modules, each with a specific goal.

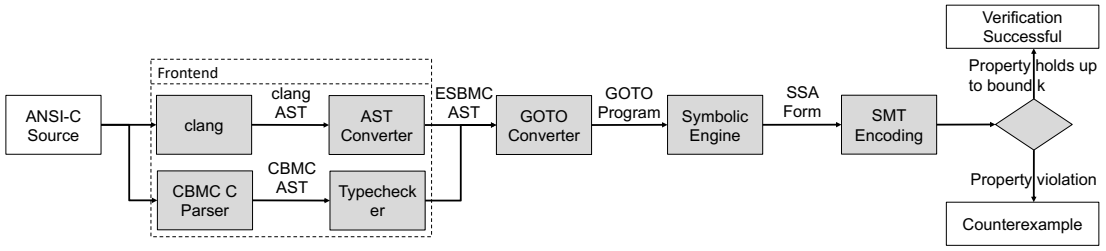


Figure 3.2: ESBMC architecture.

Frontend. Converts the program into an abstract syntax tree (AST) [124]. ESBMC has three frontends, one clang-based [19] C frontend, one CBMC-based C frontend and one CBMC-based C++ frontend [23] (in Figure 3.2 the C++ frontend is omitted but it works similarly to the C frontend, with a parser [125] and a type-checker [126]). An in-depth description of this module is presented in Section 3.3.

GOTO converter. Converts the AST generated by the frontend into a state transition system (as described in Chapter 2) called GOTO program. In this module, the GOTO program can be changed to add property checks and k-induction specific instructions. An in-depth description of this module is presented in Section 3.4.

Symbolic Engine. Converts the GOTO program into a sequence of static single assignments (SSA) [67]. This module unwinds the loops of the GOTO program [127], propagating constants to generate a minimal set of SSAs. This module can also add property checks, most of them related to dynamically allocated memory. An in-depth description of this module is presented in Section 3.5.

SMT encoding. Converts the set of SSAs into SMT and checks for satisfiability. If the formula is satisfiable, then the SMT solver is queried for relevant information in order to build the counterexample. The backend currently supports five solvers: Z3 [106], MathSAT [109], Boolector [119], Yices [120], and CVC4 [123]. An in-depth description of this module is presented in Section 3.6.

The following Sections provide an in-depth description of each of these modules, the improvements developed in them and how they convert and verify the illustrative program. The information presented in these Sections may overlap slightly with the bounded model checking description provided in Chapter 2 but the goal here is to describe the technological improvements and the bounded model checking process in practice.

3.3 Frontend: Understanding C Programs

One of the significant challenges in software verification is the development and maintenance of an infrastructure which can handle real-world programs; it is an issue that intensifies given the ever-evolving programming language standards. The C and C++ programming languages are problematic in this sense, with the standards now being updated every three years, after a 13 years gap between the C/C++98 and C/C++11 major releases.

The increasing number of features introduce both semantic and syntactic changes, which researchers must support in their tools, reducing the amount of effort that can be spent addressing research questions.

The frontend is then an important piece of technology that should facilitate the transition between the program under verification and a format the tool can work upon. Different software verification tools implement different solutions for this problem: CPAChecker, a software verification platform [128], uses the parser from CDT, a C/C++ IDE plugin for Eclipse [129]; SMACK [130], a bounded software verifier, transforms the LLVM bytecode [131] into the Boogie intermediate verification language [132], while CBMC [23] uses a modified ANSI-C parser written by James Roskind and a C++ parser based on OpenC++ [133].

Numerous verification tools are now available that use the LLVM infrastructure to verify programs. In SV-COMP, several tools (e.g., SMACK [130], DIVINE [134], Predator [135], VVT [136], Seahorn [137], and Forest [138]) used the LLVM infrastructure in the verification process, often verifying the LLVM bytecode or converting it to a intermediate representation (e.g., Boogie [132]), a trend that is likely to grow in the future.

Using the LLVM bytecode in the verification process brings some advantages: the bytecode is highly optimised by LLVM and will simplify the verification, thus reducing the costs of the verification. However, there are disadvantages in this process, including losing context information (e.g., variable, class and function names are mangled in C++ [139]), source location information and, in rare cases, optimisations can generate wrong results [140]. While the latter can be avoided by disabling optimisations (and thus losing one of the advantages of the approach), the other two are not easily avoided.

ESBMC (being derived initially from CBMC v2.9 [23]) had two frontends, one for C and another for C++. C and C++ programs were parsed and processed by the different frontends to generate the abstract syntax tree (AST) of the program. The C frontend is based on a C parser produced in 1998 by James Roskind, and the C++ frontend is based on OpenC++ [133], which had its last stable release in 2004. Although the code has received updates from both CBMC and ESBMC, both frontends (spanning more than 62 KLOC in total) are outdated technologies.

Maintaining such outdated technology is a substantial task. We chose to use the LLVM infrastructure to implement a new frontend but using a different approach from other verification tools. Instead of working on the LLVM bytecode, ESBMC accesses the AST that generates the bytecode. The AST is guaranteed to remain unchanged across minor releases (e.g., for all 3.x.x releases).

We developed a new frontend based on clang. We use clang to parse the C program, which generates the clang AST, and we convert it into ESBMC AST. The new clang frontend currently lives alongside the old frontends, and it is enabled by default. The clang frontend brings several advantages to the verification process. First, clang has a powerful static analyser, and it can provide a state-of-the-art compiler diagnostics of the program, thus issuing the same warnings and errors that one would expect from a conventional compilation. Second, the support for new language features is simplified, because only a new AST node converter needs to be added to the converter class in ESBMC, instead of layering in the feature from the flex parser to the type-checker, as must be done in the old frontends. Finally, by choosing the AST instead of the bytecode, a complete representation of the original program is available, including the original names of every class, function and variable. The disadvantage is that ESBMC is not able to access the optimisations that LLVM performs on the bytecode. When considering the size of the new frontend, it is more than eight times smaller (4014 LOC against 35879 LOC). The smaller size eases maintenance and the implementation of new features.

Figure 3.3 shows the AST of our running example in Figure 3.1, as generated by clang. We omitted several nodes of the tree to keep it short (e.g., the `nondet_double` and `nondet_uint` function declarations). The AST contains several nodes, for declarations (`FunctionDecl`, `VarDecl`), expressions (`DeclRefExpr`, `CallExpr`, `ImplicitCastExpr`), literals (`IntegerLiteral`, `FloatingLiteral`), statements (`CompoundStmt`, `ReturnStmt`, `IfStmt`, `WhileStmt`) and operations (`BinaryOperator`, `UnaryOperator`). This is only a subset of possible nodes and many more exist, representing every feature in C, plus a number of different extensions (e.g., blocks in C).

3.3.1 New supported features

There follows an incomplete list of features that were not supported or that were only partially supported by the old frontends, but that now are completely supported by the clang-based frontend:

1. Float literals extensions: the new C standards define a number of prefixes and suffixes that can be added to float literals that will define their type. The code in Figure 3.4 shows a program from SV-COMP[1] that ESBMC was not able to verify

[1] floats-cbmc-regression/float-no-simp7_true-unreach-call.c

```
1  `-FunctionDecl <main.c:2:1, line:16:1> line:2:5 main 'int ()'
2    `-CompoundStmt <col:12, line:16:1>
3      |-DeclStmt <line:3:3, col:29>
4      | `-VarDecl <col:3, col:28> col:10 used x 'double' cinit
5      |   `-CallExpr <col:14, col:28> 'double'
6      |     `-DeclRefExpr <col:14> 'double ()' Function 'nondet_double'
7      |-IfStmt <line:4:3, line:5:12>
8      | |-BinaryOperator <line:4:6, col:11> 'int' '<='
9      | | |-ImplicitCastExpr <col:6> 'double' <LValueToRValue>
10     | | | `-DeclRefExpr <col:6> 'double' lvalue Var 0x7b6bff0 'x'
11     | | `-FloatingLiteral <col:11> 'double' 0.000000e+00
12     | |-ReturnStmt <line:5:5, col:12>
13     | | `-IntegerLiteral <col:12> 'int' 0
14     |-DeclStmt <line:7:3, col:33>
15     | `-VarDecl <col:3, col:32> col:16 used N 'unsigned int' cinit
16     |   `-CallExpr <col:20, col:32> 'unsigned int'
17     |     `-DeclRefExpr <col:20> 'unsigned int ()' Function 'nondet_uint'
18     |-DeclStmt <line:8:3, col:21>
19     | `-VarDecl <col:3, col:20> col:16 used i 'unsigned int' cinit
20     |   `-ImplicitCastExpr <col:20> 'unsigned int' <IntegralCast>
21     |     `-IntegerLiteral <col:20> 'int' 0
22     |-WhileStmt <line:9:3, line:13:3>
23     | |-BinaryOperator <line:9:9, col:13> 'int' '<'
24     | | |-ImplicitCastExpr <col:9> 'unsigned int' <LValueToRValue>
25     | | | `-DeclRefExpr <col:9> 'unsigned int' lvalue Var 'i'
26     | | `-ImplicitCastExpr <col:13> 'unsigned int' <LValueToRValue>
27     | |   `-DeclRefExpr <col:13> 'unsigned int' lvalue Var 'N'
28     | `-CompoundStmt <col:16, line:13:3>
29     |   |-BinaryOperator <line:10:5, col:13> 'double' '='
30     |   | |-DeclRefExpr <col:5> 'double' lvalue Var 'x'
31     |   | `-ParenExpr <col:9, col:13> 'double'
32     |   |   `-BinaryOperator <col:10, col:12> 'double' '*'
33     |   |     |-ImplicitCastExpr <col:10> 'double' <IntegralToFloating>
34     |   |     | `-IntegerLiteral <col:10> 'int' 2
35     |   |     `-ImplicitCastExpr <col:12> 'double' <LValueToRValue>
36     |   |       `-DeclRefExpr <col:12> 'double' lvalue Var 'x'
37     |   |-CallExpr <line:11:5, col:15> 'void'
38     |   | | `-DeclRefExpr <col:5> 'void (_Bool)' Function 'assert'
39     |   | `-ImplicitCastExpr <col:12, col:14> '_Bool' <IntegralToBoolean>
40     |   |   `-BinaryOperator <col:12, col:14> 'int' '>'
41     |   |     |-ImplicitCastExpr <col:12> 'double' <LValueToRValue>
42     |   |     | `-DeclRefExpr <col:12> 'double' lvalue Var 'x'
43     |   |     `-ImplicitCastExpr <col:14> 'double' <IntegralToFloating>
44     |   |       `-IntegerLiteral <col:14> 'int' 0
45     |   `-UnaryOperator <line:12:5, col:7> 'unsigned int' prefix '++'
46     |     `-DeclRefExpr <col:7> 'unsigned int' lvalue Var 'i'
47     |-CallExpr <line:14:3, col:13> 'void'
48     | | `-DeclRefExpr <col:3> 'void (_Bool)' Function 'assert'
49     | `-ImplicitCastExpr <col:10, col:12> '_Bool' <IntegralToBoolean>
50     |   `-BinaryOperator <col:10, col:12> 'int' '>'
51     |     |-ImplicitCastExpr <col:10> 'double' <LValueToRValue>
52     |     | `-DeclRefExpr <col:10> 'double' lvalue Var 'x'
53     |     `-ImplicitCastExpr <col:12> 'double' <IntegralToFloating>
54     |       `-IntegerLiteral <col:12> 'int' 0
55     `-ReturnStmt <line:15:3, col:10>
56       `-IntegerLiteral <col:10> 'int' 0
```

Figure 3.3: AST of the program in Figure 3.1, generated by clang.

with the old frontend. In this program, the prefix `0x` means that the constant is hexadecimal, the prefixes `e` (or `E`, `p` or `P`) define an exponent and the suffix `f` (or `F`) sets the constant type to `float` (suffixes `l` or `L` set the constant type to `long double` and no suffix sets the constant type to `double`).

2. Designated initialisers: designated initialisers were introduced in the C90 standard and define semantics to initialise elements in a struct or array. Figures 3.5 and 3.9 show two examples of code that was not previously supported. Figure 3.5 shows that designated initialisers can be used for out-of-order initialisation of elements in an array. Figure 3.9 shows a piece of code from a program in SV-COMP[2]. which ESBMC had not been able to verify due to the usage of designated initializers when initializing members of a structure.

3. Complex cast expressions: Figure 3.6 shows a program from SV-COMP[3] containing casts that ESBMC did not support. In this code, an `unsigned long` is being cast to an array of function pointers.

4. C11 thread local keyword: the C11 standard defines a new keyword to change the storage duration of a variable: `thread_local`. It is a mechanism by which variables are allocated such that there is one instance of the variable per thread. Figure 3.7 shows a piece of code from a program from SV-COMP[4] that, although it does not use the `thread_local` keyword, uses the non-standard version of the keyword `__thread`. ESBMC's concurrent model does not yet support the behaviour of a variable with thread local storage; however, the first support step has been made and at least a correct internal representation of the behaviour is generated.

5. Compound literals: compound literals were defined in the C99 standard; they construct an unnamed object of an in-place specified type for arrays, structs or unions. Figure 3.8 shows a program that uses compound literals to create two arrays and assign their addresses to pointers.

6. Statically determined statements/expressions: by using clang, ESBMC is able to determine results for several statements, expressions and operators statically. Figures 3.10–3.14 show programs, where clang is used to determine the output of statements, expressions and operators. Figure 3.10 shows the usage of predefined identifiers, as described in the C99 standard; these are macros that return standardised names for functions and methods. Figure 3.11 shows the usage of the operator `offsetof`, which returns the offset of a given member of a struct. Figure 3.12 shows the usage of the operators `alignof`, `typeof` and `sizeof`; `alignof`

[2] ldv-linux-4.2-rc1/linux-4.2-rc1.tar.xz-08_1a-drivers-net-ethernet-sfc-sfc.ko-en try_point_true-unreach-call.cil.out.c

[3] ldv-linux-3.4-simple/32_7_cilled_false-unreach-call_const_ok_linux-32_1-drivers-net-wireless-p54-p54usb.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.ot.c

[4] pthread-ext/43_NetBSD__sysmon_power__sliced_true-unreach-call.c

```
int main (int argc, char **argv) {
  float f = 0x1.9e0c22p-101f;
  float g = -0x1.3c9014p-50f;
  float target = -0x1p-149f;

  float result = f * g;
  if(!(result == target)) assert(0);

  return 0;
}
```

Figure 3.4: Program with float literal extensions.

```
int main() {
  int a[6] = { [3] = 29, [0] = 15, [1] = 2 };
  assert(a[3] == 29);
  return 0;
}
```

Figure 3.5: Simple code that initializes three elements of an array out of order.

```
typedef unsigned char u8;

int main() {
  u8 ((*__cil_tmp255))[4096] ;
  unsigned long __cil_tmp254 ;

  __cil_tmp255 = (u8 ((*))[4096])__cil_tmp254;
  return 0;
}
```

Figure 3.6: Program with cast from `unsigned long` to array of function pointers.

```
__thread _Bool COND = 0;
```

Figure 3.7: Program using thread local storage for variables.

```
int main()
{
  int *p = (int[]){2, 4};
  assert(*(++p) == 4);

  const float *pc = (const float []){1e0, 1e1, 1e2};
  assert(*(pc++) == 1e0);

  return 0;
}
```

Figure 3.8: Program using compound literals.

```
typedef unsigned int __u32;
typedef unsigned long long __u64;

typedef __u64 __le64;
typedef __u32 __le32;

union efx_dword {
   __le32 u32[1U] ;
};

typedef union efx_dword efx_dword_t;

union efx_qword {
   __le64 u64[1U] ;
   __le32 u32[2U] ;
   efx_dword_t dword[2U] ;
};

typedef union efx_qword efx_qword_t;

union efx_oword {
   __le64 u64[2U] ;
   efx_qword_t qword[2U] ;
   __le32 u32[4U] ;
   efx_dword_t dword[4U] ;
};

typedef union efx_oword efx_oword_t;

struct efx_farch_register_test {
   unsigned int address ;
   efx_oword_t mask ;
};

static struct efx_farch_register_test const falcon_b0_register_tests[18U]
    =
  {      {0U, {.u32 = {262143U, 262143U, 262143U, 262143U}}},
        {2048U, {.u32 = {4294967294U, 98303U, 0U, 0U}}},
        {2640U, {.u32 = {2147418167U, 0U, 0U, 0U}}},
        {2688U, {.u32 = {4294901376U, 536870911U, 33554686U, 8388607U}}},
        {3200U, {.u32 = {4294901760U, 0U, 0U, 0U}}},
        {1568U, {.u32 = {2097151U, 0U, 0U, 0U}}},
        {2112U, {.u32 = {15U, 0U, 0U, 0U}}},
        {2128U, {.u32 = {1023U, 0U, 0U, 0U}}},
        {592U, {.u32 = {4095U, 0U, 0U, 0U}}},
        {3600U, {.u32 = {29495U, 0U, 0U, 0U}}},
        {3872U, {.u32 = {7967U, 0U, 0U, 0U}}},
        {4640U, {.u32 = {3176U, 0U, 0U, 0U}}},
        {4656U, {.u32 = {524644U, 0U, 0U, 0U}}},
        {4672U, {.u32 = {118491660U, 0U, 0U, 0U}}},
        {4832U, {.u32 = {8184U, 0U, 0U, 0U}}},
        {4720U, {.u32 = {4294901761U, 0U, 0U, 0U}}},
        {4608U, {.u32 = {4294967295U, 0U, 0U, 0U}}},
        {4880U, {.u32 = {261903U, 0U, 0U, 0U}}}};
```

Figure 3.9: Fragment of code from a program in SV-COMP.

```c
#include <string.h>

int main() {
  assert(strcmp(__func__, "main") == 0);
  assert(strcmp(__FUNCTION__, "main") == 0);
  assert(strcmp(__PRETTY_FUNCTION__, "int main()") == 0);
}
```

Figure 3.10: Program to show the use of predefined identifiers.

```c
#include <stddef.h>

struct S {
  float f;
  double d;
};

struct T {
  int i;
  struct S s[10];
};

int main() {
  assert(offsetof(struct T, s[2].d) == 48);
}
```

Figure 3.11: Program to show the use of `offsetof`.

```c
#include <stdalign.h>

int main() {
  assert(alignof(float[10]) == 4);
  assert(alignof(struct {char c; int n;}) == 4);

  struct s { int a; } x;
  typeof(x) y;

  assert(sizeof(y) == sizeof(x));
}
```

Figure 3.12: Program to show the use of `alignof`, `sizeof` and `typeof`.

```c
#include<math.h>

#define cbrt(X) _Generic((X), \
              long double: cbrtl, \
                  default: cbrt, \
              const float: cbrtf, \
                    float: cbrtf \
)(X)

int main() {
  assert(cbrt(8.0) == 2.0f);
}
```

Figure 3.13: Program to show the use of generic selection.

```
1 static_assert(sizeof(long) == 4, "Code relies on int being 4 bytes");
2
3 int main(void) {
4   static_assert(sizeof(long) == 4, "Code relies on int being 4 bytes");
5   return 0;
6 }
```

Figure 3.14: Program to show the use of static assertions.

returns the alignment, in bytes, required by an object of the given type; `typeof` returns the type of the argument, and `sizeof` returns the size, in bytes, of the given object. Note that, when those operators are unable to determine the result statically (e.g., with variable length arrays), ESBMC is used to generate the correct values. Figure 3.13 shows the use of a generic selection expression; generic selection expressions were introduced in the C11 standard and provide a way to choose among several expressions at compile time. Figure 3.14 shows the usage of static asserts; static asserts were defined in the C11 standard; they check assertions at compile-time. In this case, ESBMC completely ignores any static assertions in the code and does not generate verification conditions, as they are checked by clang when generating the AST. If they fail, clang will output the error after the generation of the AST. If clang did not find an assertion violation, there is no error, and there is no need to recheck it [53].

3.4 GOTO Converter: a State Transition System Generator

After the ESBMC AST is generated, the next step is to generate a state transition system from the program, called GOTO program. In this process, the state transition system is simplified, and new property checks can be added.

The textual representation of the GOTO program of our running example in Figure 3.1, as printed by ESBMC, is shown in Figure 3.15. The GOTO program is a simplified version of the program: `for` and `while` are replaced by a branch (line 24) and a backward GOTO (line 36). It is very similar to a C program, containing assignments, function calls and returns, and also contains location information (shown as comments).

Once the GOTO program is generated, the following transformations can be applied, depending on the set of options given to ESBMC[5]:

1. Function inlining: replaces function calls with the instructions of the called function. Partial inlining is enabled by default (it inlines functions with fewer than 10 instructions), but full inlining can be enforced. For obvious reasons, it is disabled for recursive functions.

[5]To check all available options of the ESBMC tool, run `$esbmc --help`

```
main (main):
    // file main.c line 3 function main
    return_value$_nondet_double$1=NONDET(double);

    // file main.c line 3 function main
    x=return_value$_nondet_double$1;

    // file main.c line 4 function main
    IF !(x <= 0.000000) THEN GOTO 1

    // file main.c line 5 function main
    RETURN: 0

    // file main.c line 7 function main
1:  return_value$_nondet_uint$2=NONDET(unsigned int);

    // file main.c line 7 function main
    N=return_value$_nondet_uint$2;

    // file main.c line 8 function main
    i=0;

    // file main.c line 9 function main
2:  IF !(i < N) THEN GOTO 3

    // file main.c line 10 function main
    x=IEEE_MUL(2.000000, x);

    // file main.c line 11 function main
    ASSERT (_Bool)(x > 0.000000)

    // file main.c line 12 function main
    i=i + 1;

    // file main.c line 9 function main
    GOTO 2

    // file main.c line 14 function main
3:  ASSERT (_Bool)(x > 0.000000)

    // file main.c line 15 function main
    RETURN: 0

    // 23 file main.c line 16 function main
    END_FUNCTION
```

Figure 3.15: GOTO program of the program in Figure 3.1, as printed by ESBMC.

2. Interval analysis: performs an interval analysis for integer and floating-point variables, which may be reachable from the `main` function. The intervals are added back into the GOTO program as assumptions. A more detailed explanation of the interval analysis is presented in Chapter 4.

3. *k*-induction transformations: new instructions are introduced to be used by the *k*-induction algorithm, including non-deterministic assignments to variables written inside the loop, and assumption of loop entry conditions. A more detailed

explanation of the *k*-induction algorithm is presented in Chapter 4.

4. Property checks: many property checks can be inserted in the GOTO program, including division by zero checks, integer and float overflow checks, NaN checks (if the result of an operation can be NaN), data race checks, deadlock checks and atomicity checks. All these properties are encoded as asserts in the GOTO program.

3.5 Symbolic Engine: Generating SSA

The next step in the verification process is the static single assignment (SSA) generation. This process unwinds the GOTO program generated in the previous step and can also add property checks, including dynamic memory checks (bounds check, memory alignment, offset pointer-free and double free) and unwinding assertions.

Figure 3.16 shows the SSA generated by ESBMC when unwinding once our running example in Figure 3.1 with no optimizations. A suffix is added to each SSA variable and intended to disambiguate the following situations:

different variable valuations at different points in the program.

@ different pieces of variable storage, i.e., instances of the same lexical variable, but that have different locations in memory. The obvious example is when the same function is called twice, and it uses a variable `i`; two distinct `i` variables can exist at different locations (in the case of recursion, there will be multiple copies of `i`).

& different assignments in different multithreaded interleavings [115]. One of the concurrency modes in ESBMC (schedule) encodes all program traces into the same SMT formula: each path through the program gets its ID number, which is put after `&` to distinguish variables.

! local variables in different threads. If a function runs in two different threads, then the variable `i` has to be distinguished.

Note that new variables are inserted in the program. Variables `guard@0!0&0#1` and `guard@0!0&0#2` were created (line 5 and 16) to represent the first branch condition and the loop entry condition, respectively; assignments will be guarded by them depending on the branch taken (e.g., the assignment `x = 2 * x` is guarded by `!guard@0!0&0#1 && !!guard@0!0&0#2`, line 20). The variables `i@1!0&0#4`, `x@1!0&0#4` and `N@1!0&0#2` are phi variables: conditional assignments that merge two paths together, as described in Chapter 2. Also note that a new assertion was added in line 31; it is an unwinding assertion (as described in Section 2.3.1).

```
// file main.c line 3 function main
(01) x@1!0&0#1 == nondet_symbol(symex::nondet0)

// file main.c line 5 function main
(02) guard@0!0&0#1 == !(!(x@1!0&0#1 <= 0.000000))

// file main.c line 7 function main
(03) N@1!0&0#1 == nondet_symbol(symex::nondet1)
     guard: !guard@0!0&0#1

// file main.c line 8 function main
(04) i@1!0&0#1 == 0
     guard: !guard@0!0&0#1

// file main.c line 10 function main
(05) guard@0!0&0#2 == !(!(0 < N@1!0&0#1))

// file main.c line 10 function main
(06) x@1!0&0#2 == IEEE_MUL(2.000000, x@1!0&0#1)
     guard: !guard@0!0&0#1 && !(!guard@0!0&0#2)

// file main.c line 11 function main (assertion)
(07) (assert) !guard@0!0&0#1 && !(!guard@0!0&0#2) => (_Bool)(x@1!0&0#2 >
    0.000000))
     guard: !guard@0!0&0#1 && !(!guard@0!0&0#2)

// file main.c line 12 function main
(08) i@1!0&0#2 == 0 + 1
     guard: !guard@0!0&0#1 && !(!guard@0!0&0#2)

// file main.c line 9 function main (unwinding assertion loop)
(09) (assert) !(!guard@0!0&0#1 && !(!guard@0!0&0#2))
     guard: !guard@0!0&0#1 && !(!guard@0!0&0#2)

// file main.c line 14 function main
(10) x@1!0&0#3 == x@1!0&0#1

// file main.c line 14 function main (assertion)
(11) (assert) (!guard@0!0&0#1 && !guard@0!0&0#2 => (_Bool)(x@1!0&0#3 >
    0.000000))
     guard: !guard@0!0&0#1 && !guard@0!0&0#2

// file main.c line 16 function main
(12) i@1!0&0#4 == (!(!guard@0!0&0#1) ? i@1!0&0#0 : i@1!0&0#1)

// file main.c line 16 function main
(13) x@1!0&0#4 == (!(!guard@0!0&0#1) ? x@1!0&0#1 : x@1!0&0#3)

// file main.c line 16 function main
(14) N@1!0&0#2 == (!(!guard@0!0&0#1) ? N@1!0&0#0 : N@1!0&0#1)
```

Figure 3.16: Unoptimized SSA generated from the program in Figure 3.1 when unwinding the program once

The symbolic engine in ESBMC also performs many optimisations when generating the SSA, including constant propagation (to further simplify expressions) and instruction slicing (to remove unnecessary instruction); the simplification process is an important step and speeds up the program verification considerably in some cases [141].

There are two slicing strategies: one removes all instructions after the last assert in the set of SSA, and the other collects all the symbols (and their dependent symbols) in assertions, and removes instructions that do not depend on them. Both slicing strategies ensure that unnecessary instructions are ignored in the next step, the SMT encoding.

```
1  // 10 file main.c line 3 function main
2  (01) x@1!0&0#1 == nondet_symbol(symex::nondet0)
3
4  // 12 file main.c line 5 function main
5  (02) guard@0!0&0#1 == x@1!0&0#1 <= 0.000000
6
7  // 15 file main.c line 7 function main
8  (03) N@1!0&0#1 == nondet_symbol(symex::nondet1)
9       guard: !guard@0!0&0#1
10
11 // 17 file main.c line 8 function main
12 (04) i@1!0&0#1 == 0
13      guard: !guard@0!0&0#1
14
15 // 19 file main.c line 10 function main
16 (05) guard@0!0&0#2 == 0 < N@1!0&0#1
17
18 // 19 file main.c line 10 function main
19 (06) x@1!0&0#2 == IEEE_MUL(2.000000, x@1!0&0#1)
20      guard: !guard@0!0&0#1 && guard@0!0&0#2
21
22 // 20 file main.c line 11 function main (assertion)
23 (07) (assert) (!guard@0!0&0#1 && guard@0!0&0#2 => x@1!0&0#2 > 0.000000)
24      guard: !guard@0!0&0#1 && guard@0!0&0#2
25
26 // 21 file main.c line 12 function main
27 (08) i@1!0&0#2 == 1
28      guard: !guard@0!0&0#1 && guard@0!0&0#2
29
30 // 22 file main.c line 9 function main (unwinding assertion loop)
31 (09) (assert) !(!guard@0!0&0#1 && guard@0!0&0#2)
32      guard: !guard@0!0&0#1 && guard@0!0&0#2
33
34 // 23 file main.c line 14 function main
35 (10) x@1!0&0#3 == x@1!0&0#1
36
37 // 23 file main.c line 14 function main (assertion)
38 (11) (assert) (!guard@0!0&0#1 && !guard@0!0&0#2 => x@1!0&0#3 > 0.000000)
39      guard: !guard@0!0&0#1 && !guard@0!0&0#2
```

Figure 3.17: Optimized SSA generated from the program in Figure 3.1 when unwinding the program once.

Figure 3.17 shows the optimised SSA generated by ESBMC for our running example, unwound once. Arithmetic and relational operations were simplified, and their values propagated, and instructions after the last assert were removed. Note, however, that the assignments to variable `i` were not removed; although it is not directly related to the assertions in the program, it is kept so the counterexample is easier to understand.

3.6 SMT Encoding of ANSI-C Programs

After the set of SSA is created, the next step is to encode every (not sliced) assignment in SMT and check for satisfiability. The formula is generated using the `QF_ABVFP` logic, where `QF` stands for quantifier-free formulas, `A` stands for the theory of arrays, `BV` stands for the theory of fixed-sized bit-vectors and `FP` stands for the theory of floating-points [142].

We follow a notation of constraints C and properties P when encoding the set of SSA. A constraint is an assignment or an assumption in the program, which constrains the value of a variable, while a property is an assertion in the program, a property that needs to hold given the set of constraints. Formulae 3.1 and 3.2 show the constraints and properties derived from the SSA in Figure 3.17. We shortened the names for simplicity, i.e., we only used the `#` number.

$$C := \begin{bmatrix} & x_1 == nondet\_double_1 \\ \wedge & N_1 == nondet\_uint_1 \\ \wedge & g_1 == (x_1 \leq 0.0f) \\ \wedge & i_1 == 0 \\ \wedge & g_2 == (0 < N_1) \\ \wedge & x_2 == 2.0f * x_1 \\ \wedge & i_2 == 1 \\ \wedge & x_3 == x_1 \end{bmatrix} \tag{3.1}$$

$$P := \begin{bmatrix} & \neg g_1 \wedge g_2 \implies x_2 > 0.0 \\ \wedge & \neg(\neg g_1 \wedge g_2) \\ \wedge & \neg g_1 \wedge \neg g_2 \implies x_3 > 0.0 \end{bmatrix} \tag{3.2}$$

The constraints and properties are encoded in the form $C \wedge \neg P$, meaning that, given the set of constraints, the SMT solver will try to find an assignment to variables that violate at least one property and satisfy the constraints.

Figure 3.18 shows the SMT formula generated by Z3 after converting the SSA in Figure 3.16. It is the final step in the verification of our running example in Figure 3.1. Note that the generated SMT in Figure 3.18 uses the `FP` logic, e.g., `fp.mul`, `fp.leq`, `fp.eq`, etc. The operations can also be bit-blasted but the generated formula is more than 1400 lines long; despite the bigger formula, however, both encodings are solved by Z3 in the same amount of time.

Finally, as the program will fail, ESBMC will present a counterexample: a set of assignments and the violated property in the program. Figure 3.19 shows the counterexample as printed by ESBMC, when verifying the program with one loop unwinding. It shows the set of assignments to the variables `i`, `N` and `x`, and the violated property: `x >= 0` when `x` is NaN.

```
(declare-fun |nondet$symex::nondet0| () (_ FloatingPoint 11 53))
(declare-fun |main::main::1::x@1!0&0#1| () (_ FloatingPoint 11 53))
(declare-fun |goto_symex::guard@0!0&0#1| () Bool)
(declare-fun |nondet$symex::nondet1| () (_ BitVec 32))
(declare-fun |main::main::2::N@1!0&0#1| () (_ BitVec 32))
(declare-fun |main::main::3::i@1!0&0#1| () (_ BitVec 32))
(declare-fun |goto_symex::guard@0!0&0#2| () Bool)
(declare-fun |main::main::1::x@1!0&0#2| () (_ FloatingPoint 11 53))
(declare-fun |main::main::3::i@1!0&0#2| () (_ BitVec 32))
(declare-fun |main::main::3::i@1!0&0#3| () (_ BitVec 32))
(declare-fun |main::main::1::x@1!0&0#3| () (_ FloatingPoint 11 53))
(declare-fun |execution_statet::\\guard_exec@0!0| () Bool)

(assert (= |nondet$symex::nondet0| |main::main::1::x@1!0&0#1|))
(assert (= (fp.leq |main::main::1::x@1!0&0#1| (fp #b0 #b00000000000
#x0000000000000))
   |goto_symex::guard@0!0&0#1|))
(assert (= |nondet$symex::nondet1| |main::main::2::N@1!0&0#1|))
(assert (= #x00000000 |main::main::3::i@1!0&0#1|))
(assert (= (bvult #x00000000 |main::main::2::N@1!0&0#1|)
|goto_symex::guard@0!0&0#2|))
(assert (= (fp.mul roundNearestTiesToEven
            (fp #b0 #b10000000000 #x0000000000000)
            |main::main::1::x@1!0&0#1|)
   |main::main::1::x@1!0&0#2|))
(assert (= #x00000001 |main::main::3::i@1!0&0#2|))
(assert (= #x00000000 |main::main::3::i@1!0&0#3|))
(assert (= |main::main::1::x@1!0&0#1| |main::main::1::x@1!0&0#3|))
(assert (let ((a!1 (=> |execution_statet::\\guard_exec@0!0|
                (=> (and (not |goto_symex::guard@0!0&0#1|)
                         |goto_symex::guard@0!0&0#2|)
                    (fp.gt |main::main::1::x@1!0&0#2|
                           (fp #b0 #b00000000000 #x0000000000000)))))
      (a!2 (=> |execution_statet::\\guard_exec@0!0|
               (not (and (not |goto_symex::guard@0!0&0#1|)
                         |goto_symex::guard@0!0&0#2|))))
      (a!3 (=> |execution_statet::\\guard_exec@0!0|
               (=> (and (not |goto_symex::guard@0!0&0#1|)
                        (not |goto_symex::guard@0!0&0#2|))
                   (fp.gt |main::main::1::x@1!0&0#3|
                          (fp #b0 #b00000000000 #x0000000000000))))))
  (or (not (=> true a!1)) (not (=> true a!2)) (not (=> true a!3)))))
(check-sat)
(exit)
```

Figure 3.18: The SMT formula generated from the SSA in Figure 3.17 when using Z3.

3.6.1 Supported Solvers

Boolector is an SMT solver for quantifier-free theories of bit-vectors and arrays [119]. The solver uses bit-blasting for bit-vectors and lemmas on demand for arrays, converting the formulae to SAT formulae; it supports several SAT solver as backends, including lingeling and PicoSAT [143], MiniSAT [101] and CryptoMiniSAT [102]. The solver, however, does not support floating-point. Boolector focuses on continuous refinement

```
State 1 file main.c line 3 function main thread 0
------------------------------------------------------------
  x = +NaN
      (01111111111100000000000000000000000000000000000000000000000001)

State 2 file main.c line 7 function main thread 0
------------------------------------------------------------
  N = 0 (00000000000000000000000000000000)

State 3 file main.c line 8 function main thread 0
------------------------------------------------------------
  i = 0 (00000000000000000000000000000000)

State 4 file main.c line 14 function main thread 0
------------------------------------------------------------
Violated property:
  file main.c line 14 function main
  assertion
  (_Bool)(x > 0.000000)
```

Figure 3.19: The counterexample printed by ESBMC when verifying the program in Figure 3.1 with one loop unwindings.

of an initial abstraction of the SMT formula, based on the satisfying assignments produced by the SAT solver. It is released under an MIT-like license with non-commercial and no-competition-use clauses. Boolector won 5 categories in SMT-COMP'18 [144].

Z3 is an SMT solver developed by Microsoft Research [106]. It supports a wide range of theories, including several non-standard extensions, such as tuples, lists, sets and recursively defined sorts. Z3 usually scores highly in SMT-COMP and won 68 categories in SMT-COMP'18 [144]. Since SMT-COMP'12 [145] the solver only joins the competition as a non-competing participant. Z3 is released under an MIT license.

Yices is an SMT solver developed at SRI international [120]. It supports several theories, including extensions for MAX-SMT problems and tuples. However, it does not support floating-point. In SMT-COMP'18 [144] it won 33 categories. It is currently available under a GPL3 license.

MathSAT is a general purpose SMT solver developed by FBK-IRST and the University of Trento [109]. The solver supports several SMT theories, including floating-points, but also supports the creation of Craig-interpolants [146] and partial assignment enumeration. In SMT-COMP'18 [144] it won 6 categories, and it is available under a non-commercial, academic free license.

CVC4 is a theorem prover with support for SMT and includes support for quantifiers and partial support for floating-points [123]. The theorem prover is a collaboration between the New York University and the University of Iowa, and it is released under a BSD license. In SMT-COMP'18 [144], it won 81 categories.

3.6.2 Encoding Scalars

In ESBMC, signed and unsigned integers are encoded using bit-vectors, with a length equal to the bit width of the type, as defined by the C standard and the GNU extension [147]. In particular, we support all C built-in types and modifiers, for 32-bit and 64-bit architectures. Custom bit-vector sizes are currently not supported due to a frontend limitation. We cannot easily change the parser in clang to support a custom semantic for types, like `__CPROVER_bitvector[]` in CBMC [148]. There is no obvious solution to this problem, but type attributes are a candidate to address this limitation [149]. Forking clang and change its parser to support a custom semantic will defeat the goal of having a small and maintainable frontend.

3.6.3 Encoding Fixed- and Floating-points

In this Section, we will show the SMT fixed-point and floating-point encoding. We focus on the latter, as it is one of the new technologies developed in ESBMC.

Initially, ESBMC was only able to encode `float`, `double` and `long double` using a fixed-point encoding (used in a wide range of applications in the verification of digital filters [150, 151] and controllers [152, 153, 154]). The lack of a proper floating-point encoding, however, meant that ESBMC was not able to find an entire class of bugs, such as the one shown in Figure 3.20.

```
1 int main()
2 {
3   float x = nondet_float();
4   float y = x;
5   assert(x==y);
6   return 0;
7 }
```

Figure 3.20: Simple floating-point program with a bug: the assertions in line 5 does not hold if x is NaN.

The program shown in Figure 3.20 will never fail if verified with a fixed-point encoding. However, when using a floating-point encoding, `x` can be NaN and comparing a NaN (even with itself) is always false [86]. In this scenario, the assertion in line 5 does not hold if `x` is NaN. Supporting floating-point arithmetic in ESBMC allows the accurate verification of the program.

3.6.3.1 SMT Fixed-Point Encoding

A fixed-point number is represented in ESBMC as a pair (m, n) where m is the total number of bits and $n \leq m$ is the number of fractional bits, e.g., the number 0.125 is

represented as $\langle 0000.0010 \rangle$ (assuming it is 8 bits long) in the fixed-point format. The fixed-point arithmetic is performed similarly to the bit-vector arithmetic, except that the operations are applied separately to the integral and fractional parts of the fixed-points and concatenated at the end (overflow in the fractional parts are treated accordingly). Differently from floating-points, all bit-vectors represent one number in the real domain.

3.6.3.2 SMT Floating-Point Encoding

A floating-point number is encoded in SMT using a single bit-vector and follows the IEEE-754 standard for the size of the exponent and significand precision, e.g., a half-precision floating-point (16 bits) has 1 bit for the sign, 5 bits for the exponent and 11 bits for the significand (1 hidden bit) [86]. It is interpreted as $(-1)^b \times s \times 2^e$, where b is the sign, s is the significand and e is the exponent, e.g., the number 0.125 is represented as $\langle 0011000000000000 \rangle$ in the floating-point format. Differently from the fixed-point format, in the floating-point format some bit-vectors do not represent numbers in the real domain, e.g., NaNs are encoded with a nonzero significand and all-ones exponent.

The SMT `FP` logic is an addition to the SMT standard, first proposed in 2010 by Rümmer and Wahl [155]. The current version of the theory largely follows the IEEE standard 754-2008 [86] and formalises floating-point arithmetic, positive and negative infinities and zeroes, NaNs, relational and arithmetic operators, and five rounding modes: round nearest with ties choosing the even value, round nearest with ties choosing away from zero, round towards positive infinity, round towards negative infinity and round towards zero.

There are, however, some functionalities from the IEEE standard that are not yet supported by the `FP` logic as described by Brain et al. [121]; however, when encoding C programs using the `FP` logic, most of the process is a one-to-one conversion, except for casts to Booleans and the equality operators.

Casts to Boolean. The SMT standard defines conversion operations to and from signed and unsigned bit-vectors, and other floating-point types, but does not define a conversion operation for Boolean types. ESBMC, however, generates these operations if they are present in a program, as in the one shown by the program in Figure 3.21.

The program in Figure 3.21 forces ESBMC to generate two casts: one from *Boolean* to *double* (line 5) and another one from *double* to *Boolean* (line 8). Figure 3.22a and Figure 3.22b present the SMT formulae generated by these lines, respectively.

When casting from Booleans to floating-point numbers (Figure 3.22a), an `ite` operator is used, such that the result of the cast is 1.0 if the Boolean is true; otherwise the result is 0.0. When casting from floating-point numbers to Booleans (Figure 3.22b), we encode as

```
1  int main(){
2    _Bool c;
3    double b = 0.0f;
4
5    b = c;
6    assert(b != 0.0f);
7
8    c = b;
9    assert(c != 0);
10 }
```

Figure 3.21: Program to demonstrate casts between floating-points and Booleans.

```
(assert (= (ite |main::c|
        (fp #b0 #b01111111111 #x0000000000000)
        (fp #b0 #b00000000000 #x0000000000000))
   |main::b|))
```

(a) SMT generated when casting from Boolean to floating-point.

```
(assert (= (not (fp.eq |main::b|
        (fp #b0 #b00000000000 #x0000000000000)))
   |main::c|))
```

(b) SMT generated when casting from floating-point to Boolean.

Figure 3.22: SMT formula generated by ESBMC to encode the casts to and from Boolean types in Figure 3.21.

a conditional assignment: the result is false if the floating-point value is 0.0. Otherwise, the result is true.

The `fp.eq` operator. Figure 3.22b shows the second special case when encoding C programs. When encoding the program, both assignments and comparison operations are encoded using representational equality, i.e., their representations are identical, but the standard enhances equality comparisons with richer semantics for floating-points:

```
:note
"(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero,
or vice versa.  fp.eq and all the other comparison operators evaluate
to false if one of their arguments is NaN."
```

In this case, the operator is defined to handle the special symbols from the IEEE floating-point standard, in particular, zeroes and NaNs. It would not be correct to use the ordinary operator equality for comparisons; it should only be used for assignments, while `fp.eq` is used for comparing floating-point numbers.

Unused operators from the SMT standard. When implementing the floating-point encoding, we did not use four operators defined by the SMT standard: `fp.max`, `fp.min`, `fp.rem` and `fp.isSubnormal`; we already had internal models for `fp.max`, `fp.min` and `fp.rem` so there was no reason to rewrite them, while we could not find any use for `fp.isSubnormal` in the C standard.

1. `fp.max`: returns the larger of two floating-point numbers; equivalent to the `fmax`, `fmaxf`, `fmaxl` functions. Our model of the function is shown in Figure 3.23.

```
1  double fmax(double x, double y) {
2    // If both argument are NaN, NaN is returned
3    if(isnan(x) && isnan(y)) return NAN;
4
5    // If one arg is NaN, the other is returned
6    if(isnan(x)) return y;
7    if(isnan(y)) return x;
8
9    return (x > y ? x : y);
10 }
```

Figure 3.23: Model for `fmax`.

2. `fp.min`: returns the smaller of two floating-point numbers; equivalent to the `fmin`, `fminf`, `fminl` functions. Our model of the function is shown in Figure 3.24.

```
1  double fmin(double x, double y) {
2    // If both argument are NaN, NaN is returned
3    if(isnan(x) && isnan(y)) return NAN;
4
5    // If one arg is NaN, the other is returned
6    if(isnan(x)) return y;
7    if(isnan(y)) return x;
8
9    return (x < y ? x : y);
10 }
```

Figure 3.24: Model for `fmin`.

3. `fp.rem`: returns the floating-point remainder of the division operation `x/y`; equivalent to the `fmod`, `fmodf`, `fmodl` functions. Our model of the function is shown in Figure 3.25. The preconditions were extracted from the C standard [53], which follows the IEEE standard [86]. The operation in line 18 follows the Z3 implementation of the modulus operation for floating-point [106].

4. `fp.isSubnormal`: checks if a number is denormalised, i.e., a non-zero floating-point number with magnitude less than the magnitude of that format's smallest normal number. A subnormal number does not use the full precision available to normal numbers of the same format [86]. We could not find any use case for it when modelling C11 standard functions.

```
double fmod(double x, double y) {
  // If either argument is NaN, NaN is returned
  if(isnan(x) || isnan(y)) return NAN;

  // If x is +inf/-inf and y is not NaN, NaN is returned
  if(isinf(x)) return NAN;

  // If y is +0.0/-0.0 and x is not NaN, NaN is returned
  if(y == 0.0) return NAN;

  // If x is +0.0/-0.0 and y is not zero, returns +0.0/-0.0
  if((x == 0.0) && (y != 0.0))
    return signbit(x) ? -0.0 : +0.0;

  // If y is +inf/-inf and x is finite, x is returned.
  if(isinf(y) && isfinite(x)) return x;

  return x - (y * (int)(x/y));
}
```

Figure 3.25: Model for `fmod`.

SMT Solver support for the `FP` logic. Encoding programs using the SMT floating-point theory has several advantages over a fixed-point encoding, but the main one is the correct modelling of ANSI-C/C++ programs that use IEEE floating-point arithmetic. ESBMC ships with models for most of the current C11 standard functions [156]; floating-point exception handling, however, is not yet supported.

The encoding algorithms, however, can be very complex and it is not uncommon to see the SMT solvers struggling to support every corner case [157, 158]. Currently, three SMT solvers support the SMT floating-point theory: Z3 [106], MathSAT [109] and CVC4 [123], and ESBMC implements the floating-point encoding for all of them using their native API. Regarding the support from the solvers, Z3 implements all operators, MathSAT implements all but two: `fp.rem` (remainder operator) and `fp.fma` (fused multiply-add) and CVC4 implements all but the conversions to other sorts.

The three solvers offer two (non-standard) functions to reinterpret floating-point numbers to and from bit-vectors: `fp_as_ieeebv` and `fp_from_ieeebv`, respectively. These functions can be used to circumvent any lack of operators, and only require the user to write the missing operators. Note that this is different from converting floating-point numbers to bit-vectors: converting to bit-vectors follows the rounding modes defined by the IEEE-754 standard while reinterpreting floating-point as bit-vectors returns the bit-vector format of the floating-point number.

ESBMC also provides another option when verifying programs with floating-point: a generic floating-point API, which bit-blasts the floating-point operations thus extending the floating-point support to the SMT solvers that do not support floating-points (Boolector and Yices), while also implementing some of the missing features for MathSAT and CVC4.

A detailed table with all the supported features for each solver and the ESBMC FP API can be found in Appendix A.

3.7 Incremental Bounded Model Checking

Bounded Model Checking (BMC) was initially proposed to solve the problem of the scalability limitations of BDD-based Model Checking [21]. BMC tools drop completeness (i.e., the ability to prove that a program does not contain a bug) in favour of falsification [22]. They are used mainly to find bugs, as they are only able to prove the absence of bugs if the whole state space is explored (e.g., all loops have been fully unwound). The technique has already been applied successfully in the verification of real software, including software written in languages usually used for low and medium level development, such as C [23, 24, 45] and C++ [159].

When running a BMC tool, one usually has to specify a bound k explicitly; this will be used to limit the visited regions of data structures (e.g., arrays) or the number of loop iterations. This limits the state space to be explored during verification, leaving enough that real errors in applications [23, 159, 160] can be found; BMC tools are still susceptible to exhaustion of time or memory limits for programs with loops whose bounds are too large, but unlike unbounded model checking tools, this can be prevented with a smaller value of k.

Since the best value is usually not known, one must repeatedly run the BMC tool with increasing values of k; every increase of k will increase the number of loop iterations, the recursion depth, and the time and memory requirements. In order to check whether the value of k is big enough, BMC tools insert unwinding assertions after each loop. Failing these assertions does not mean that the program has a bug, but that the verification is incomplete.

BMC tools can try to determine statically the value of k that will fully unwind all the loops; if such a value is found, there is no need for an unwinding assertion. However, determining this value is not always possible (e.g., for infinite loop programs or loop conditions that depend on user input), and seeking it leads to BMC tools trying to unwind loops to maximum possible values, often causing exhaustion of resources and failing in their original purpose: finding bugs despite limited resources.

Incremental BMC was proposed to avoid this pitfall [161]. In an incremental BMC, the program is incrementally unwound until a bug is found or until the completeness threshold is reached, ensuring that smaller problems are solved sequentially instead of guessing an upper bound for the verification. The incremental algorithm, however, has its limitations. In particular, the BMC has to redo all the parsing, generation and solving for each bound k, and no record of previous step *1* to *k-1* is used when solving

for k. Even though incremental solving first appeared in the 1990s [161], the problem of how to efficiently reuse information learnt from previous instances remains.

The problem is not only related to the effort already spent on the verification of previous instances (e.g., reusing internal state or learnt clauses); the removal of learnt clauses that are no longer true in the new instance is trickier [96]. There has been work on defining the conditions for reusing learnt information, but a detailed dependency analysis must be carried out, which weakens the benefit from incremental solving [162, 163].

Instead of expensive book-keeping, solvers implement their own strategies to improve performance when incrementally solving instances of a problem. SAT solvers often make use of assumptions of literals, where the search procedure assumes literals derived from previous instances as properties of the problem [162]. SMT solvers offer interfaces for the pushing and popping of clauses in a stack-like manner. Current state-of-the-art SMT solvers, however, do not perform well when using those interfaces and usually perform worse than non-incremental approaches [164].

In ESBMC, there are two versions of incremental BMC:

1. We extended the work from Donaldson et al. [165] in which the CFG is unwound before the SSA generation, by creating copies of the loop body and removing the backward GOTO instruction. The final SSA is similar to the one generated from plain BMC, however, unwinding the CFG prove to be faster, at the expense of higher memory usage. The main difference between the static incremental BMC algorithm in ESBMC and the one described by Donaldson the handling of nested loops. In both algorithms, the number of loop unwindings is set globally. However, nested inner loops are unwound first in ESBMC while in the work by Donaldson outermost loops are unwound first. The limitation with the latter is that the creation of copies of the loop body will replicate nested loops, requiring further loop unwindings of the nested loops. That algorithm should keep track of the newly created loops in order to replicate them correctly, but it does not, scaling poorly with loop depth. By working in the opposite direction, this problem is avoided.

2. We developed an algorithm that incrementally unwinds the state transition system by running the symbolic engine with incremental numbers of unwinds; it is an infrastructure to allow the incremental verification in a number of different configurations, including: Falsification (incrementally unwinds the program searching for property violations; cannot prove correctness), Termination (incrementally unwinds the program and checks if all the loops were completely unwound; it can prove termination, but not no-termination and cannot find bugs), Incremental Verification (a combination of falsification and termination, an incremental algorithm that finds bugs and proves correctness if all loops were unwound), and k-induction (similar to the incremental BMC but also tries to prove

correctness without fully unwinding the loops). Our approach generates the full set of SSA from scratch for every new unwinding: one would expect that using incremental SMT solving offers better results but empirical experiments show otherwise, as described by Henning et al. [164]. In contrast, Schrammel et al. [96] generate the SSA up to the end of an unbounded loop, and then incrementally extends and adds the increment to the existing SAT instance, while Brain et al. [166] splice only the new SSA generated by the new unwinding into the set of SSAs; the splice process is very complex and error-prone, and there is no clear advantage in doing it since we are not using an incremental SMT encoding in ESBMC.

3.8 Experimental Evaluation

The experimental evaluation of the verification tool described in this Section consists of three parts. In Section 3.8.1, we present the benchmarks used to evaluate ESBMC, including the different properties evaluated. In Section 3.8.2, we compare the verification results of ESBMC using Boolector [119], CVC4 [123], Z3 [106], Yices [120] and MathSAT [109] (the number of results produced and the verification time).Finally, in Section 3.8.3 we compare the results of the new floating-point API against the two solvers that support floating-point encoding natively. The purpose of this Section is to evaluate ESBMC with the new technologies and supported solvers, and identify the most suitable solver for further experiments.

Our experimental evaluation aims to answer three research questions:

RQ1 **(Frontend Coverage)** is our clang-based frontend able to parse C programs?

RQ2 **(Floating-point soundness)** is our floating-point bit-blasting backend sound?

RQ3 **(SMT solvers performance)** which solver produces better results?

3.8.1 Experimental Setup

We evaluate our approach using all the verification tasks in SV-COMP18 [122]. The competition consists of 9539 programs (6149 programs with no property violation and 3390 program with property violations), grouped into six categories plus two macro categories:

ReachSafety. In these programs, a function call is checked for reachability. It includes several sub-categories used to test specific program features, e.g., arrays, floating-points, recursion. The category contains 2943 benchmarks: in 1799 of these programs, the function call is unreachable, and in 1144 the function call is reachable. The property checked in this category is formally defined in Table 3.1.

LTL [45] Formula	Definition
`G ! call(__VERIFIER_error())`	The function "__VERIFIER_error" is not called in any finite execution of the program

Table 3.1: Formal definition of the reachability property checked in the ReachSafety category.

ConcurrencySafety: This category checks the same property as the ReachSafety category (Table 3.1) but only contains multi-threaded programs. It contains 1047 benchmarks: in 246 the function call is unreachable, and in 801 the function call is reachable.

MemSafety. The programs in this category use dynamically allocated memory, and the bugs are related to both pointer safety (e.g., null pointer dereference, out-of-bounds access, or double free) and memory leaks. It contains 326 benchmarks: 188 bug-free programs and 138 programs with at least one property violation. Formally, the properties checked in this category are defined in Table 3.2.

LTL Formula	Definition
`G valid-free`	All memory deallocations are valid (counterexample: invalid free). More precisely: There exists no finite execution of the program on which invalid memory deallocation occurs.
`G valid-deref`	All pointer dereferences are valid (counterexample: invalid dereference). More precisely: There exists no finite execution of the program on which an invalid pointer dereference occurs.
`G valid-memtrack`	All allocated memory is tracked, i.e., pointed to or deallocated (counterexample: memory leak). More precisely: There exists no finite execution of the program on which the program loses track of some previously allocated memory. (Comparison to Valgrind: this property is violated if Valgrind reports "definitely lost").

Table 3.2: Formal definition of the memory safety properties checked in the MemSafety category.

NoOverflows. The benchmarks in this category are checked for signed integer overflow (undefined behaviour in the C standard [156]) in various expressions. It contains 359 benchmarks: 182 with no signed overflow and 177 with signed overflow. Formally, the property checked in this category is defined in Table 3.3.

LTL Formula	Definition
`G ! overflow`	It is never the case that the resulting type of an operation is a signed integer type but the resulting value is not in the range of values that are representable by that type.

Table 3.3: Formal definition of the overflow property checked in the NoOverflows category.

Termination. This category checks if all paths inside a program are finite and the program terminates. It contains 2009 benchmarks: in 1274 all paths terminate while in 735 they do not terminate. Formally, the property checked in this category is defined in Table 3.4.

LTL Formula	Definition
`F end`	Every path finally reaches the end of the program. The proposition "end" is true at the end of every finite program execution (exit, abort, return from the initial call of main, etc.). A counterexample for this property is an infinite program execution.

Table 3.4: Formal definition of the termination property checked in the Termination category.

SoftwareSystems. In this category, several benchmarks are checked for both bug reachability and memory safety, as defined in Table 3.1 and Table 3.2, respectively. It is a separate category because all the programs in this category are real software systems, while most of the programs in other categories were crafted with the goal of checking a property. It contains several Linux device drivers [167] and pre-processed busybox [168] applications.

The two macro categories are called **Overall** and **Falsification**. In both, the results all of the categories are evaluated together, with the exception that true positives and false positives are discarded in the latter.

ESBMC was run on each benchmark in the competition (equivalent to the Overall category) once per solver, with the following set of options: `--floatbv`, which enables SMT floating-point encoding; `--no-div-by-zero-check`, which disables the division by zero check (an SV-COMP requirement); `--incremental-bmc`, which enables the incremental BMC; `--unlimited-k-steps`, which removes the upper limit of iteration steps in the incremental BMC algorithm; and `--force-malloc-success`, which forces all dynamic allocations succeed to (also an SV-COMP requirement). In addition, ESBMC sets the `--32` or `--64` options depending on the category architecture, and it sets the following options depending on the property to be checked: `--no-pointer-check`, `--no-bounds-check` for reachability verification, `--memory-leak-check` for memory verification, `--overflow-check` for overflow verification, and `--termination` for termination verification. In order to select an SMT solver for verification, the options `--boolector`, `--z3`, `--cvc`, `--mathsat`, and `--yices` are used.

All experiments were conducted on IRIDIS4, the supercomputer at the University of Southampton [169]. The compute node used are equipped with Intel Sandy Bridge processors running at 2.6GHz and 24GB of RAM.

For each benchmark, we set time and memory limits of 900 seconds and 16GB, respectively, as per the competition definitions. We, however, do not present the results as

scores (as it is done in SV-COMP) but show the number of correct and incorrect results, and the verification time.

Finally, given the large amount of data involved in the experiments, we use four groups to present the results in this Section: *Correct true* is the number of correct positive results (i.e., the tool reports SAFE correctly), *Correct false* is the number of correct negative results (i.e., the tool reports UNSAFE correctly), *Incorrect true* is the number of incorrect positive results (i.e., the tool reports SAFE incorrectly), *Incorrect false* is the number of incorrect false results (i.e., the tool reports UNSAFE incorrectly). All the other results are *Unknown*; these are inconclusive results when the tool does not finish the verification within the time and memory limits.

3.8.2 Solver Performance Comparisons

Figure 3.26 show the results of using incremental BMC to verify the programs in SV-COMP. Boolector is the best performing solver, producing correct results in the highest number of tests (2991 in total) while also producing a small number of incorrect results (16 in total). ESBMC can correctly parse and generate AST of all the programs in the competition, answering research question RQ1: yes, our clang-based frontend can parse C programs correctly[6].

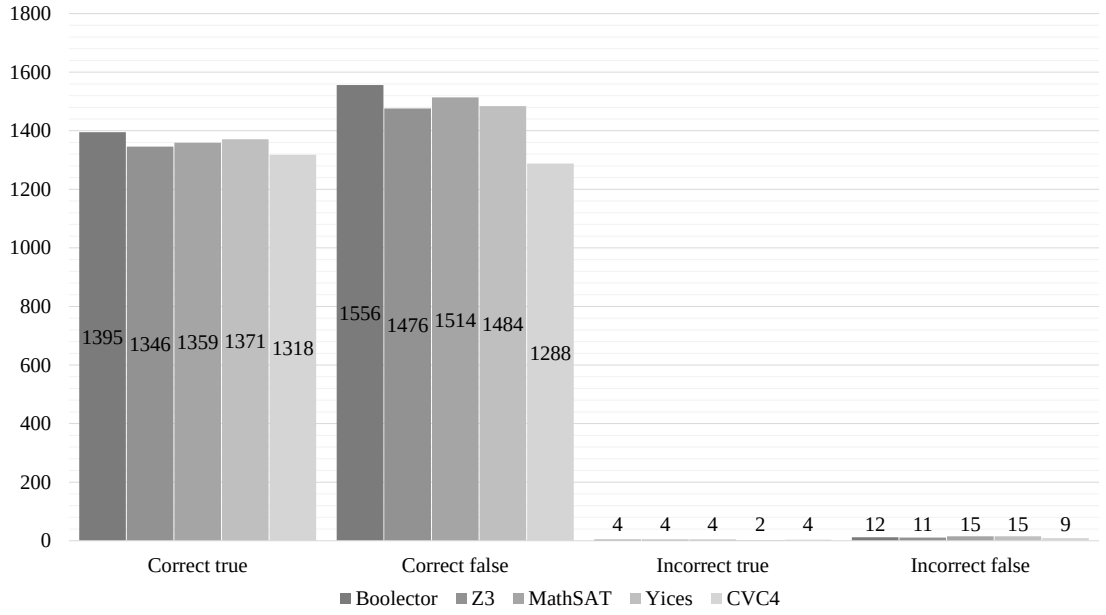


Figure 3.26: SV-COMP results for each solver, using incremental BMC.

An in-depth analysis of the (correct and incorrect) results show that the solvers are usually able to solve the same class of programs which explains the small difference in

[6]The clang frontend even found cases where the programs are not standard compliant (i.e., they do not compile). We submitted patches to the competition repository to address these mistakes.

the results, i.e., if we consider the total number of benchmarks in SV-COMP (9539), the difference between the number of correct results reported by Boolector (the best performing solver with 2991 correct results) and CVC4 (which reported the fewest number of correct results, 2606), is small and represents less than 5% of the total number of benchmarks.

All the incorrect true unsound results were produced when verifying the same four benchmarks in the ConcurrencySafety category and are related to bugs in our concurrency model. The incorrect false results, however, are spread across several categories and most solvers seem to report the same false results, except for MathSAT which reports bugs that no other solver reports. We reported the issue to the developer of MathSAT and they confirmed an issue when using two of MathSAT's simplification rules together.

Similarly, the difference in verification time is minimal. As shown in Figure 3.27, Boolector was the faster solver and verified all the programs in 5,277,216 seconds (approximately 61 days of CPU time), while CVC4 was the slowest and took 5,541,845 seconds (approximately 64 days of CPU time), 5% slower than Boolector. These numbers also explain why CVC4 reported a fewer number of incorrect results; the solver simply could not finish the verification within the time and memory limits. These results reaffirm previous finding by our group: SAT/SMT solvers available today are very efficient but are reaching a plateau [141] and the best way forward with the current technology is to create new algorithms and smarter encodings.

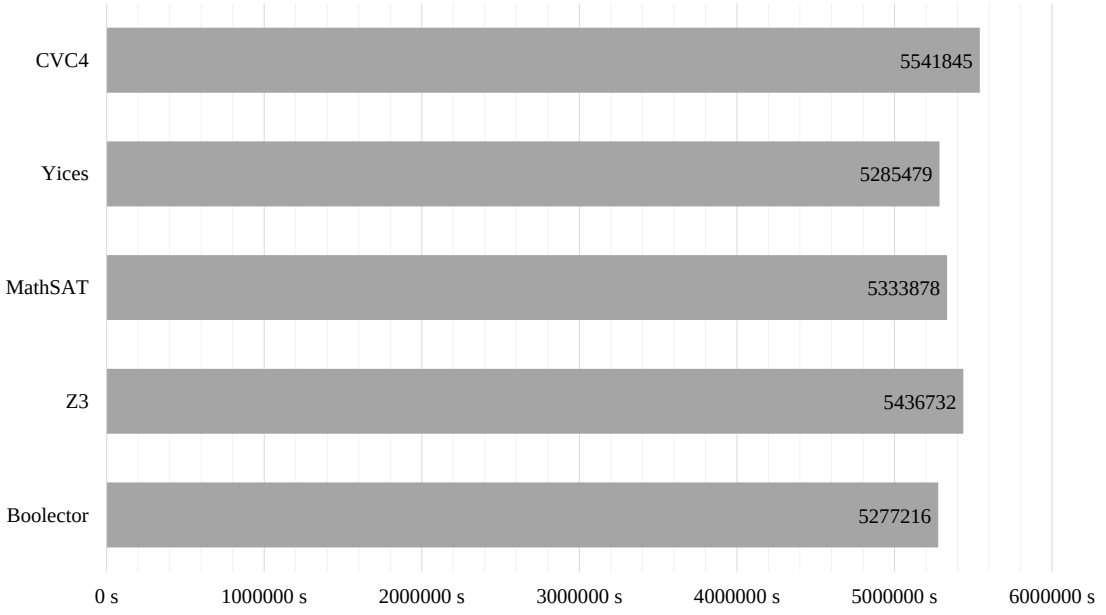


Figure 3.27: Total verification time of each solver, in seconds.

These data show that Boolector outperforms all the other currently supported solvers,

both in the number of correctly solved problems and verification time. Note, however, that Boolector does not support the SMT theory of floating-point: in these cases, ESBMC bit-blasts all the operations; we evaluate the bit-blast algorithm in Section 3.8.3. These results answer our research question RQ3: Boolector is the solver that provides the best results.

3.8.3 Floating-Point API evaluation

In SV-COMP there is a sub-category part of the ReachSafety category, called ReachSafety-Floats, which consists of 172 benchmarks designed to check for bug reachability in programs with floating-point arithmetic. We evaluated these benchmarks using two solvers that offer native support to floating-point (Z3 and MathSAT) and all the solvers using the floating-point API in ESBMC, as shown in Figure 3.28. We do not present the results of using the native floating-point API of CVC4 because the lack of support for converting sorts causes the solver to abort in a majority of the cases.

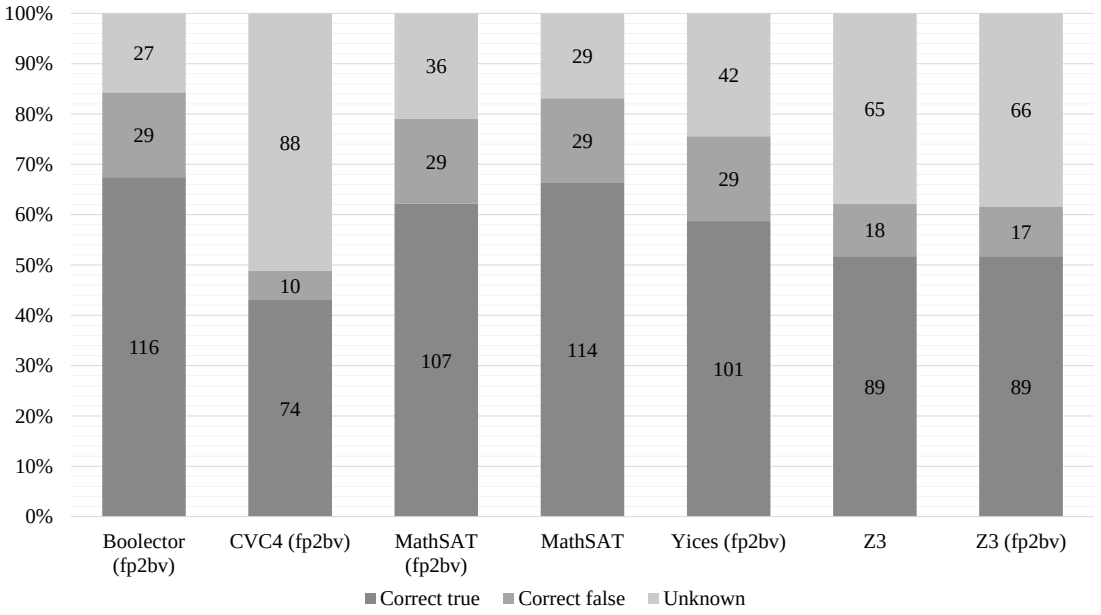


Figure 3.28: ReachSafety-Floats results for each solver, using the incremental BMC. The (fp2bv) next to the solver name means that our floating-point API was used to bit-blast floating-point arithmetic.

In Figure 3.28, *Correct true* is the number of correct positive results (i.e., the tool reports SAFE correctly), *Correct false* is the number of correct negative results (i.e., the tool reports UNSAFE correctly) and *Unknown* is the number of unknown results (i.e., the tool was stopped after 900 seconds or reached the memory limit of 15GB). There is no case where ESBMC reports an incorrect result, so we omit this from the Figure; this answers our research question RQ2: yes, our floating-point encoding appears sound.

Boolector (fp2bv) reports the highest number of correct results (145), followed by MathSAT using their native floating-point API (143). This slightly better result follows our previous evaluation of these solvers, where MathSAT can solve floating-point problems quickly but suffers slowdowns in programs with arrays [31]. CVC4 presented the fewest number of correct results, and it was the only solver to exhaust the memory limit of 15GB (in two benchmarks).

The results show that Z3 with its native floating-point API and Z3 with our fp2bv API produce very similar results; this is expected since our fp2bv API is heavily based on the bit-blasting performed by Z3 when solving floating-points. The number of variables and clauses generated in the CNF format when using Z3 with its native floating-point API is 1%-2% smaller than the number generated when using our fp2bv API. The smaller number explains the one extra correct result in Figure 3.28 and the shorter verification time in Figure 3.29; we assume this is the result of internal optimisations when performing the bit-blasting internally.

MathSAT results show that their API can solve more problems within the time and memory limits, compared to MathSAT (fp2bv). In particular, the seven extra results when using their native API involved programs with chains of multiplications, which leads us to believe that their ACDL algorithm is somehow optimised for this operation; unfortunately, MathSAT is a free but closed source tool, so we cannot confirm this.

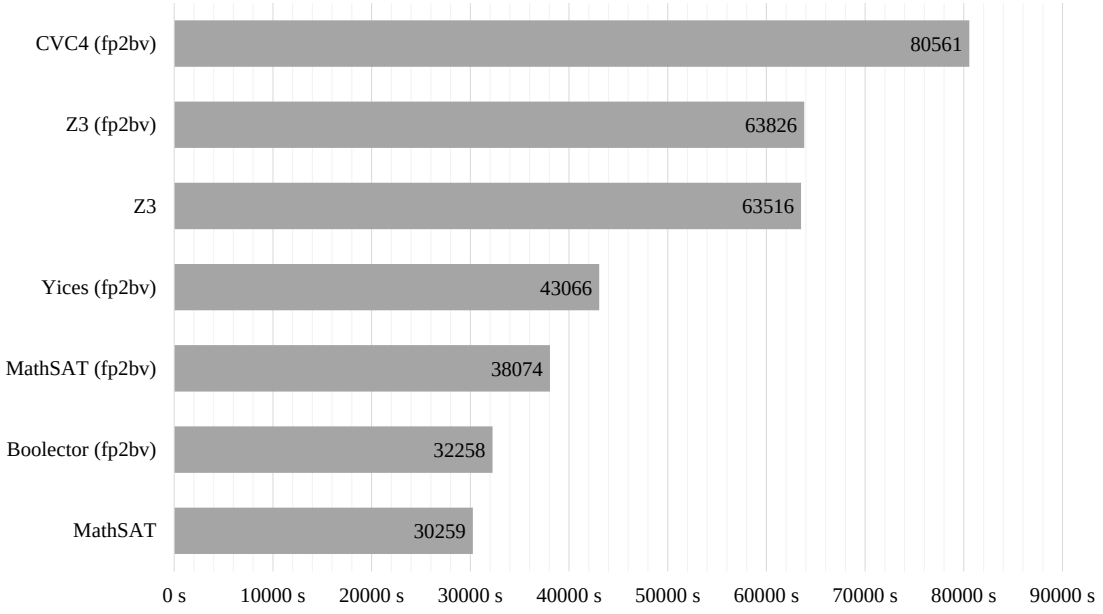


Figure 3.29: Total verification time of each solver in the ReachSafety-Floats category, in seconds.

The total verification time for each solver is shown in Figure 3.29. Although Boolector (fp2bv) was the solver that correctly verified the highest number of benchmarks, it was slower than MathSAT with its native API. Further analysis of the results shows that Boolector (fp2bv) is slightly slower in programs where the property holds. CVC4 was

the slowest solver in our evaluation. This result also confirms our previous evaluation of CVC4, where we noticed that this solver is substantially slower than the other solvers supported by ESBMC [141].

Overall, Boolector presented the best results in this category, despite not being the fastest solver. It provided slightly better results (finds more correct results than the other solvers), while only being 6% slower than the fastest solver (MathSAT); this reaffirms the answer to our research question RQ3.

A detailed table with the verification time of each benchmark is shown in Appendix B.

3.9 Related Work

There has been much work on the verification of C programs, applying different techniques in order to find bugs or to prove correctness. CBMC [23] is probably the most well-known bounded model checker available today. It is an SAT-based bounded model checker for sequential and concurrent C, C++ and Java programs [170]. The tool takes a program, possibly with user assertions, and automatically inserts assertions in order to check for violated properties. The resulting program is flattened into SSA form, then bit-blasted to an SAT formula that is satisfiable upon assertion violation. ESBMC is a tool derived from CBMC and aims to use SMT solvers instead of SAT solvers to verify C and C++ programs. CBMC also bit-blasts floating-point operations, similarly to our floating-point API. We initially implemented our floating-point API following their bit-blast algorithm but, in some cases, it would run for days when trying to convert simple programs due to the size of the generated formula[7]; for that reason, we decided to follow the bit-blast algorithms of Z3.

Merz et al. [159] describe LLBMC, a sequential bounded model checker for C and C++ programs. The tool first converts the program into an LLVM intermediate representation, using clang [19]. This conversion removes high-level information about the structure of the programs (e.g., the relationship between classes), but several optimisations are performed by LLVM which facilitate the verification process. From the LLVM intermediate representation, LLBMC generates a logical formula. This formula is further simplified and passed to an SMT solver for verification. In comparison to ESBMC, LLBMC works by bounded model checking the LLVM bytecode, instead of the AST generated by clang; the advantages and disadvantages of this approach were already discussed in Section 3.3 but, in summary, while this approach will generate smaller SMT problems, it might also introduce bugs.

[7] The problem was reported to the developers who later confirmed the issue: https://github.com/diffblue/cbmc/issues/1944.

Wang et al. [171] describe Ceagle, an automated verification tool for C programs. The tool applies 4 different approaches when verifying a program: (1) a bounded model checking with a fixed unwind approach that uses SMT to check for satisfiability; (2) a predicate lazy abstraction engine which verifies the program with a predicate-based abstract model and uses CEGAR to refine spurious counterexamples; (3) a structural abstraction engine which tries to reason about the program behaviour based on the program structure; and (4) an execution engine, which is used when all parameters are deterministic. The tool competed in SV-COMP17 and was ranked 2nd, if we consider only the *ReachSafety-Floats* sub-category. Ceagle was the only tool to verify three programs that ESBMC could not handle in that year[8]; it was even able to verify each one of them in less than 10 seconds.

3.10 Conclusions

In this Chapter, we have described the verification process in ESBMC, an SMT-based bounded model checker for C programs and compared the results of the verification of a large set of benchmarks using five different solvers. ESBMC was extensively modified as part of our research and the technologies we developed (in particular, the clang frontend, SMT backend with the floating-point support and the incremental bounded model checking infrastructure) were required to broaden the class of programs that could be handled by ESBMC.

The creation of the clang-frontend was essential for the future of ESBMC, as maintaining a C frontend is a herculean task: even more massive if a C++ frontend is desired. The new frontend simplifies the work of supporting new language features while improving diagnostic warning and error messages. As a bonus, clang can also simplify some expressions, making the verification less complicated.

Supporting floating-point encoding was also a significant improvement in ESBMC. In particular, ESBMC was already extensively used to verify digital systems [151, 152, 153]. However, these projects were limited to fixed-point arithmetic; supporting floating-point encoding will allow researchers to expand their activities.

The incremental bounded model checking infrastructure was developed as a base for several algorithms and made it easier to implement new strategies of verification in ESBMC. In particular, the algorithm described in Chapter 4 would be much harder to implement without this infrastructure.

Finally, the extensive evaluation performed during the development of these technologies also identified areas to be improved in clang, the solvers and other verification

[8] `sin_interpolated_index_true-unreach-call.c`, `sin_interpolated_bigrange_loose_true-unreach-call.c` and `sin_interpolated_bigrange_tight_true-unreach-call.c`

tools. In particular, we submitted patches to clang to improve naming generation for types[9] (accepted, part of clang 7.0), fixed naming generation for parameters[10] (under review), and submitted patches to Z3 to optimize the generation of unsigned less-than operations during the bit-blast of floating-points[11] (accepted, part of Z3 4.6.1). We also reported bugs to both CBMC[12] and MathSAT, which were later confirmed by the developers.

The new technologies developed for ESBMC, the bug reports to the various projects and the patches were my small contributions to help the growth of the compiler and verification communities.

[9] https://reviews.llvm.org/D36610
[10] https://reviews.llvm.org/D42966
[11] https://github.com/Z3Prover/z3/pull/1501
[12] https://github.com/diffblue/cbmc/issues/1944

Chapter 4

Correctness Proof and Bug Hunting by *k*-induction

Abstract. *In this Chapter, we describe and evaluate a novel algorithm, called bkind, implemented in ESBMC. Our new algorithm is an extension to the k-induction algorithm and is aimed at extending its bug-finding capabilities. We exploit the counterexamples generated by an over-approximation step to derive new properties and feed them back to the bug-finding step during the verification process. We show that, in combination with a new invariant analyser also developed for this work, the bkind algorithm can improve the number of correct results found by ESBMC in a large set of public benchmarks. Furthermore, our algorithm can reduce the verification time compared to the naïve k-induction algorithm as it only requires half the number of steps to find a property violation in an unsafe program.*

As we have previously discussed, bounded model checking (BMC) is a promising technique to verify software [22] but has limitations. BMC techniques are only able to falsify properties up to a given bound k; they are not able to prove the correctness of the system unless the tool can unwind all loops and recursive functions to their maximum possible bound.

Consider for example the simple program in Figure 4.1a, in which the loop in line 2 runs an unknown number of times, depending on the initial non-deterministic value of `x`. The assertion in line 3 holds independent of `x`'s initial value. Unfortunately, BMC tools like CBMC [23], LLBMC [159] or ESBMC [24] typically fail to verify programs that contain such loops. Soundness requires the insertion of the unwinding assertion, after the loop, as in Figure 4.1b, line 5. Verification of this program using a BMC tool will fail the *unwinding assertion* if k is too small (i.e., $k < 2^{32} - 1$).

In mathematics, one usually approaches such unbounded problems using *proof by induction*. A variant called k-induction uses BMC as a "component" to prove partial correctness; it has been successfully combined with continuously-refined invariants [172],

```
1 unsigned int x;
2 while(x>0) x--;
3 assert(x==0);
```

(a) Simple unbounded loop program.

```
1 unsigned int x;
2 if (x>0)
3   x--;
4 ...
5 assert(!(x>0));
6 assert(x==0);
```

(b) Finite unwinding done by BMC.

Figure 4.1: Unbounded loop and finite unwinding.

to prove that C programs do not contain data races [165, 173], or that design-time constraints are respected [162]. Additionally, *k*-induction is a well-established technique in hardware verification, where it is applied to the transition relations present in hardware designs [162, 174, 175].

In this Chapter we describe and evaluate the *bkind* (bidirectional *k*-induction) algorithm: an extension of the *k*-induction algorithm that improves its bug-finding capabilities, reducing the number of iterations to find a property violation in half. In practice, the *bkind* algorithm performs a bidirectional search for bugs in the program state space. Our results show that the *bkind* algorithm is particularly useful when combined with invariants produced by interval analysis, producing a higher number of results in less time than the *k*-induction algorithm.

Organization. In Section 4.1 we present an illustrative example that will be used to explain the algorithms described in this Chapter. Section 4.2 presents the *k*-induction algorithm while Section 4.3 presents the new *bkind* algorithm and discusses the new interval analysis implemented in ESBMC. Section 4.4 shows the comparative results of the new *bkind* algorithm against the *k*-induction algorithm. We also compare the *bkind* algorithm against *kIkI*, another extension to the *k*-induction algorithm implemented in 2LS. Finally, Section 4.5 presents some related work and Section 4.6 draws conclusions.

4.1 Illustrative Example

We use the programs in Figure 4.2 as examples to illustrate the strengths and weaknesses of the *k*-induction algorithm and how our *bkind* algorithm addresses the weaknesses. They are simplified (and modified) versions of a program in SV-COMP18, from the ReachSafety-ECA subcategory[1]; the original version of the program is 549 LOC, so it was manually reduced for simplicity.

The programs in this sub-category are reactive systems, using Event-Condition-Action (ECA) rules [176]. The systems have a finite set of states, encoded into a variable `s`, and one control `input`. The systems read the `input` variable at each iteration and update the

[1] `eca-rers2012/Problem01_label15_false-unreach-call_false-termination.c`

system's state if some conditions are satisfied. The function `VERIFIER_nondet_int()` returns a non-deterministic value and the function `VERIFIER_error()` is a property violation (e.g., a call to `assert(0)`). In both programs a property violation in line 16 is guarded by a condition in line 15. The program in Figure 4.2a is safe because the guard never holds. The program in Figure 4.2b, however, is unsafe and at least five iterations are required to reach the property violation.

```
1  unsigned int s = 1;
2  int main() {
3    while (1) {
4      unsigned int input = __VERIFIER_nondet_int();
5      if (input > 5) {
6        return 0;
7      } else if (input == 1 && s == 1) {
8        s = 2;
9      } else if (input == 2 && s == 2) {
10       s = 3;
11     } else if (input == 3 && s == 3) {
12       s = 4;
13     } else if (input == 4 && s == 4) {
14       s = 5;
15     } else if (input == 5 && s > 5) { // unsatisfiable
16       __VERIFIER_error(); // property violation
17     }
18   }
19 }
```

(a) Simplified safe program extracted from SV-COMP18.

```
1  unsigned int s = 1;
2  int main() {
3    while (1) {
4      unsigned int input = __VERIFIER_nondet_int();
5      if (input > 5) {
6        return 0;
7      } else if (input == 1 && s == 1) {
8        s = 2;
9      } else if (input == 2 && s == 2) {
10       s = 3;
11     } else if (input == 3 && s == 3) {
12       s = 4;
13     } else if (input == 4 && s == 4) {
14       s = 5;
15     } else if (input == 5 && s >= 5) { // satisfiable
16       __VERIFIER_error(); // property violation
17     }
18   }
19 }
```

(b) Simplified unsafe program extracted from SV-COMP18.

Figure 4.2: Simplified illustrative examples encoding an event-condition-action system. The program in Figure 4.2a is safe since the property violation is unreachable while the program in Figure 4.2b is unsafe since the property violation is reachable after at least 5 iterations.

4.2 Naïve *k*-induction Algorithm

The first version of the k-induction algorithm was proposed by Sheeran et al. [175]. They proposed to use BMC algorithms to prove correctness by induction. Consider a state transition system P with a loop and a safety property $\phi(s)$. BMC algorithms can only show that no counterexample exists for k loop unwindings but not that longer counterexamples do not exist. The k-induction algorithm then tries to prove by induction that if ϕ holds for any given iteration through the loop, then ϕ holds for the next iteration. In particular, the base case tries to find a counterexample where ϕ does not hold, and the inductive step tries to prove that there exist no counterexamples. A formal definition of the standard induction principle for natural numbers is shown in Eq. 4.1 [172].

$$(\phi(0) \wedge \forall n : (\phi(n) \Rightarrow \phi(n+1))) \Rightarrow \forall n : \phi(n) \tag{4.1}$$

The k-induction algorithm then extends standard induction by asserting the safety property k times instead of only once, as shown in Eq. 4.2 [172].

$$\left(k > 0 \wedge \bigwedge_{i=0}^{k-1} \phi(i) \wedge \forall n : \left(\left(\bigwedge_{i=0}^{k-1} \phi(n+i)\right) \Rightarrow \phi(n+k)\right)\right) \Rightarrow \forall n : \phi(n) \tag{4.2}$$

Since the inductive step assumes the safety property ϕ more than once, a less general case is checked after each iteration, thus it is more likely to succeed [177].

In a previous work [35], we extended the k-induction algorithm to check program completeness in a separate step and referred to it as an iterative deepening algorithm [178] consisting of three independent checks: base case, forward condition and inductive step.

Base case. The base case is the plain BMC algorithm applied without checking the completeness threshold; it unwinds k times a state transition system P and checks all the reachable states in P from the initial state s_1 with k loop unwindings. It is satisfiable if and only if the BMC algorithm finds a counterexample of length at most k unwindings [21], as defined in Algorithm 4.1. If a counterexample is found, the program is unsafe, and the k-induction algorithm terminates.

Algorithm 4.1: The base case.

1 **Function** `base_case(`P, k`):`
2 **if** $init(s_1) \wedge \bigwedge_{i=1}^{k-1} tr(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\phi(s_i)$ **then**
3 `Let` $\epsilon = s_i$ `such that` $\neg\phi(s_i)$;
4 **return** $[s_1, \ldots, \epsilon]$;
5 **else**
6 **return** $\emptyset$;
7 **end**

Lemma 4.1 (Base case)**.** *If the function* `base_case(`P, k`)` *returns a sequence of states, then the program is unsafe and the sequence of states is a counterexample.*

Proof. The execution path returned in line 4 of the Algorithm 4.1 is a counterexample since it is an execution path (ensured by the transition relation tr in line 2) that starts with the first state of the program (ensured by $init(s_1)$ in line 2) and ends with an error state (in line 3); this follows the definition of a counterexample (c.f. Chapter 2). This is also a non-spurious counterexample because the base case is a precise check: it encodes all reachable states up to k and checks for satisfiability. If the base case returns a real counterexample then the program is unsafe.

Forward Condition. The forward condition checks if the completeness threshold $\psi(s)$ holds for the current k. It unwinds k times a state transition system P and checks if all the states that are reachable from s_1 are reachable in the current unwound state transition system. This is established by first checking that no counterexample is produced by the Algorithm 4.2. No safety property $\phi(s)$ is checked in the forward condition as they are checked in the base case for the current k. This step can only prove completeness for programs with finite loops.

Algorithm 4.2: The forward condition.

1 **Function** `forward_condition(`P, k`):`
2 **if** $init(s_1) \wedge \bigwedge_{i=1}^{k-1} tr(s_i, s_{i+1}) \wedge \neg\psi(s_k)$ **then**
3 **return** $[s_1, \ldots, s_k]$;
4 **else**
5 **return** $\emptyset$;
6 **end**

Lemma 4.2 (Forward condition)**.** *If the function* `forward_condition(`P, k`)` *returns an empty sequence of states then the program is safe.*

Proof. The forward condition checks if the completeness threshold was reached in the current unwound state transition system P, i.e., all loops were completely unwound.

This is encoded as the completeness threshold property check in line 2 of Algorithm 4.2. In practice, these checks are encoded as unwinding assertions (as described in Chapter 2) and they check if the termination condition of all loops are satisfiable for the current number of unwindings. This step can prove partial correctness if the base case did not find any bug for the current unwinding since no safety property is checked. We guarantee this precedence in k-induction by checking the base case before checking the forward condition. We conclude that if no bug was found by the base case and the completeness threshold holds for the current number of unwindings, all states were explored and the program is safe.

Inductive Step. The inductive step checks if a property violation holds in the k-th iteration given any sequence of $k-1$ loop unwindings without a property violation. Note that in practice the inductive step is very similar to the plain BMC algorithm but it tries to prove that there is no counterexample of length k. Differently from the plain BMC algorithm, however, the inductive step assumes any sequence of $k-1$ loop unwindings without a property violation instead of the sequences starting in the initial state. In other words, the inductive step checks all execution paths from any state in the program, not only the initial state. This is the inductive hypothesis of the k-induction algorithm.

In practice, the inductive step is implemented as an over-approximation of loop variables: when verifying a program, variables written inside a loop are assumed non-deterministic before loop unwinding; this is equivalent to check every path through the loop, even spurious ones.

Algorithm 4.3: The inductive step.

```
1 Function inductive_step(P, k):
2     if ∃n ∈ ℕ⁺. ⋀_{i=n}^{n+k-1}(φ(s_i) ∧ tr(s_i, s_{i+1})) ∧ ¬φ(s_{n+k}) then
3         Let ϵ = s_{n+k} such that ¬φ(s_{n+k});
4         return [s_n, ..., ϵ];
5     else
6         return ∅;
7     end
```

Lemma 4.3 (Inductive step)**.** *If the function* `inductive_step(`P,k`)` *returns an empty sequence of states, then the program is safe.*

Proof. Similarly to the forward condition check, the program is safe up to k loop unwindings because the base case did not find any reachable error state. This is guaranteed in the k-induction algorithm by running the base case before the inductive step. The inductive step then tries to find any counterexample of length k in the state space by first assuming that there was no property violation in $k-1$ iterations. The inductive

step over-approximates the state space so if no counterexample is found then this is sufficient to prove that there is no reachable bug in the program.

The naïve version of the *k*-induction algorithm is shown in Algorithm 4.4. The algorithm tries to either find a property violation or to prove correctness for an increasing number of k loop unwindings, starting from $k = 1$. If it reaches a maximum number of iterations k_{max}, the algorithm terminates with an *unknown* answer.

Algorithm 4.4: Naïve *k*-induction.

```
1  Function kind(P, k_max, k):
2      if k > k_max then return unknown;
3
4      π := base_case(P,k);
5      if π ≠ ∅ then return π;
6
7      π := forward_condition(P,k);
8      if π = ∅ then return ∅;
9
10     π := inductive_step(P,k);
11     if π = ∅ then return ∅;
12
13     return kind(P, k_max, k+1);
```

Theorem 4.4 (Soundness of the *k*-induction algorithm)**.** *If the k-induction algorithm returns:*

1. *a sequence of states* $[s_i, \ldots, s_j]$*: the program is unsafe, and the sequence is a non-spurious counterexample.*
2. $\emptyset$*: the program is safe.*
3. *unknown: the program is safe up to* k_{max} *iterations.*

Proof. The first item is ensured by lemma 4.1, if there is a property violation reachable after k unwindings, the program is unsafe and the algorithm terminates returning the counterexample (line 5).

The second item is ensured by lemmas 4.2 and 4.3, if no counterexample is found then the program is safe and an empty execution path is returned in lines 8 and 11.

Finally, the third item is ensured in line 2, if the algorithm reached the maximum number of defined iterations without terminating, an unknown answer is given.

We can then conclude that the *k*-induction algorithm always terminates either with a counterexample (if the program is unsafe), an empty execution path (if the program is safe), or with an unknown answer otherwise.

Theorem 4.5 (Partial completeness of the *k*-induction algorithm)**.** *If* $\exists\, k : 1 \leq k \leq k_{max}$ *such that the shortest counterexample is* $\pi^k_{min} = [s_1, \ldots, s_k]$ *and* $s_k = \epsilon$ *then the k-induction algorithm will find the counterexample in at most k iterations.*

Proof. This is ensured by always starting the *k*-induction algorithm as defined in Algorithm 4.4 with one loop unwinding; it always increments the number of loop unwindings by one (line 13). Furthermore, if the program is unsafe, neither the forward condition nor the inductive step will terminate the verification before the counterexample is found. Also note that, if a property violation requires zero loop unwindings (e.g., a property violation before a loop), the *k*-induction algorithm will still unwind the program once and the base case will find the property violation since it checks all states reachable with one loop unwinding (line 2).

4.2.1 Why is the *k*-induction Algorithm Naïve?

The inductive step assumption of all possible sequences of states is what makes the *k*-induction algorithm naïve; these sequences often include large unreachable regions of the state space. Safety properties might not hold in these regions of the state space, but they are irrelevant for the safety of the program. For example, when verifying the safe program in Figure 4.2a using *k*-induction, the inductive step will try to prove that the program is safe for all possible values that both `input` and `s` variables can assume; this will result in a series of spurious partial counterexamples since these variables only assume a small range of values. The version of the *k*-induction algorithm as presented in Algorithm 4.4 assumes that all partial counterexamples produced by the inductive step are spurious.

Let us use an illustrative example to show the verification process using *k*-induction. First, consider the safe program in Fig. 4.2a. The property violation is reachable if the transition condition $[input = 5 \wedge s > 5]$ holds and, since the state space is over-approximated, several states will satisfy this condition. In this case, the *k*-induction as defined in Algorithm 4.4 will eventually reach the maximum number of loop unwindings[2] and terminate with an *unknown* answer because the base case will not find a property violation (the program is safe), the completeness threshold will never be reached as the program contains an infinite loop, and the inductive step will keep finding spurious counterexamples.

Now, let us consider the unsafe program in Fig. 4.2b. The *k*-induction algorithm will need at least five iterations until the base case finds a counterexample. During these iterations, both the forward condition and the inductive step are executed and any reasoning performed in these steps are discarded **but what if the inductive step finds**

[2]The maximum unwinding depth is user-defined in many implentations of the *k*-induction algorithm, e.g., in our tool the value is 50 by default but can be changed by the user.

an actual partial counterexample? This useful information (a *partial counterexample*) is ignored as all counterexamples found by the inductive step are assumed to be spurious.

When using the naïve *k*-induction to verify the programs in Fig. 4.2, it will either produce an unknown result or will discard useful information.

4.3 A Smarter *k*-induction

The *k*-induction algorithm can be applied to solve various verification problems [162, 165, 173], but it can be further improved by taking advantage of two crucial observations:

1. *Partial counterexamples are ignored*: useful counterexamples may be generated by the inductive step, and the algorithm ignores them;
2. *Unconstrained state space*: the inductive step may find spurious counterexamples if the over-approximation is unconstrained.

Several authors address the latter by generating program invariants to rule out unreachable regions of the state space, either as a pre-processing step where invariants are introduced in the program before verification [179] or during the verification itself [172, 180, 166]. Our algorithm is the first to address the former in the context of software verification.

In this Section, we will present our two contributions: we will show (1) our novel extension to the *k*-induction algorithm to reuse information from the inductive step in Section 4.3.1 and (2) how we used interval analysis to constrain the inductive step in Section 4.3.2. Finally, in Section 4.3.3 we show how these two techniques can help the verification of our illustrative example.

4.3.1 Bidirectional Bug-Finding using *k*-induction

In this Section we define our *bkind* algorithm: a bidirectional *k*-induction. Figure 4.3 shows a visual representation of the verification of the program in Figure 4.2b using *bkind*.

The *bkind* algorithm extends the bug-finding capabilities of the *k*-induction algorithm by performing two alternating bug searches, one forward (i.e., from the initial state s_1) and one backward (i.e., from any error state ϵ) and stopping if the forward search finds a state in a counterexample produced by the backward search. Our proposed algorithm is similar to the bidirectional search algorithm from graph theory [181].

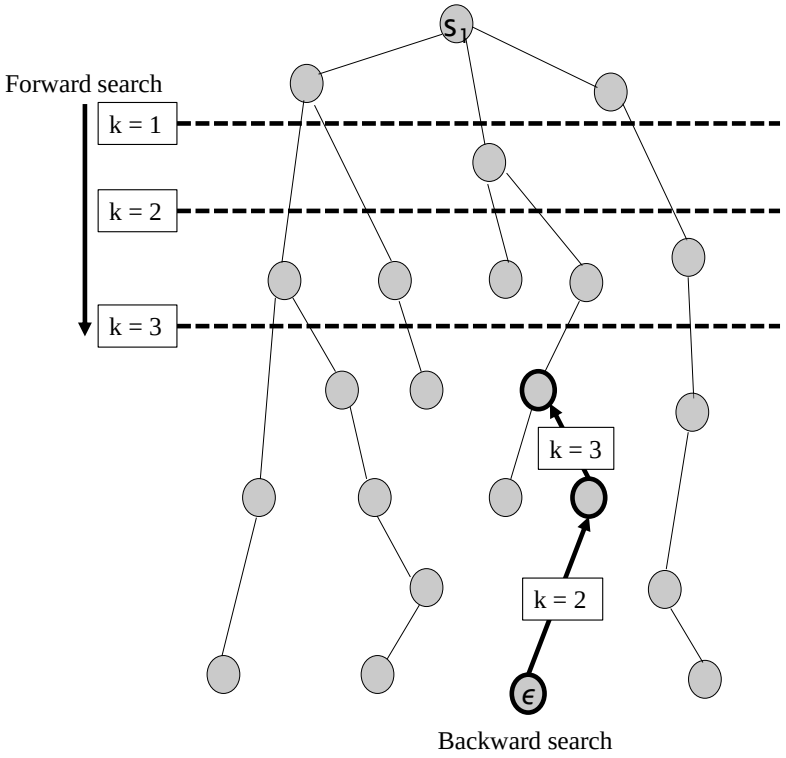


Figure 4.3: Visual representation of both searches performed by our *bkind* algorithm. Each dashed section represents the states reachable after k iterations. The arrows show the "direction" of the verification by the forward search (starting from the initial state s_1) and the backward search (from the error state ϵ). The thick line is a partial counterexample.

The *bkind* algorithm relies on two checks from the k-induction algorithm to implement the searches. The base case is the forward search since it tries to find a counterexample, $\pi^k = [s_1, \ldots, \epsilon]$, while the inductive step is the backward search and it tries to find any partial counterexample $\pi^k = [s_i, \ldots, \epsilon]$. We shall refer to the base case and inductive step as forward and backward searches, respectively.

***bkind* base case.** In order to perform the forward search, we need to extend the base case as shown in Algorithm 4.5. First we define a new function `starts_counterexample` that given a state s and a counterexample π, returns *true* if s is the first state in π otherwise returns $false$; this function will be used to perform the bidirectional search. The first condition in the new algorithm (line 10) is the same condition in the base case from the naïve k-induction algorithm and returns a counterexample if a bug was found in k iterations. The second (and new) condition (line 13) uses the function `starts_counterexample` to check if any of the states reachable by the base case start the counterexample π_{back} found by the backward search. If it holds, the new base case function

returns the execution path found by the forward search concatenated with the counterexample found by the backward search[3] (line 14). The "$\cdot$" operator concatenates two sequences. If no bug is found, the algorithm returns an empty sequence.

Algorithm 4.5: The base case used in the *bkind* algorithm.

```
1  Function starts_counterexample(s, π):
2      if π ≠ ∅ ∧ π = [s, ...] then
3          return true;
4      else
5          return false;
6      end
7
8  Function bkind_base_case(P, π_back, k):
9      BC := init(s_1) ∧ ⋀_{i=1}^{k-1} tr(s_i, s_{i+1});
10     if BC ∧ ⋁_{i=1}^{k} ¬φ(s_i)) then
11         Let ε = s_i such that ¬φ(s_i);
12         return [s_1, ..., ε];
13     else if BC ∧ ∃ i : 1 ≤ i ≤ k ∧ starts_counterexample(s_i, π_back) then
14         return [s_1, ..., s_{i-1}] · π_back;
15     else
16         return ∅;
17     end
```

Lemma 4.6 (Base case in *bkind*)**.** *If Algorithm 4.5 returns a sequence of states $[s_i, \ldots, s_j]$, this is a non-spurious counterexample.*

Proof. The first condition in the new base case (line 10) is identical to the condition in the original base case and lemma 4.1 ensures that this is a real counterexample. We only need to prove that the execution path returned in line 14 is a counterexample. The returned sequence is a counterexample because it is a concatenation of an execution path starting from the initial state in the state space with a counterexample. We know that the sequence of states in the concatenation is an execution path because of the transition in line 13 and $init(s_1)$ in line 9 ensures that it starts from the initial state in the state space. The counterexample in the concatenation is a partial counterexample because of lemma 4.7. Finally, we know that this is a non-spurious counterexample because of the partial order property of the state space which ensures the equality of two states: this is sufficient to allow the concatenation of the execution path and the counterexample.

The *bkind* algorithm is shown in Algorithm 4.6. Similarly to the naïve *k*-algorithm, the *bkind* algorithm tries to either find a property violation or to prove correctness for an

[3] In practice, the concatenation only occurs when showing the counterexample to the user: the partial counterexample found by the forward search is printed first without showing the property violation (since it was inserted by our algorithm), then the partial counterexample from the backward search is printed, showing the real property violation.

increasing number of k unwindings. The pre-conditions of the algorithm are $\pi_{back} = \emptyset$ and $k = 1$. If it reaches a maximum number of iterations k_{max}, the algorithm terminates with an *unknown* answer. The novel contribution in the new *bkind* algorithm is the bidirectional bug-finding technique that uses the counterexample produced by the backward search from the previous iteration and checks if it is reachable in the forward search in the next iteration (line 4).

Algorithm 4.6: The *bkind* algorithm.

```
1  Function bkind(P, k_max, k, π_back):
2      if k > k_max then return unknown;
3
4      π := bkind_base_case(P, π_back, k);
5      if π ≠ ∅ then return π;
6
7      π := forward_condition(P, k);
8      if π = ∅ then return ∅;
9
10     π := inductive_step(P, k);
11     if π = ∅ then return ∅;
12
13     return bkind(P, k_max, k + 1, π);
```

Lemma 4.7 (Partial counterexample from the inductive step)**.** *If Algorithm 4.3 returns a sequence of states $[s_i, \ldots, s_j]$, this is a partial counterexample of length k.*

Proof. We know this is a partial counterexample because it is an execution path (the transition in line 2 ensures that) and the last state in the path is an error state (ensured in line 3); this follows the definition of a partial counterexample as defined in Section 2. Finally, the counterexample has length k because the inductive step always tries to find a counterexample of length k; this is performed by checking if the property violation is reachable in k iterations, assuming that it holds for $k - 1$ iterations.

Theorem 4.8 (Partial completeness of the *bkind* algorithm)**.** *If $\exists\, k : 1 \leq k \leq k_{max}$ such that the shortest counterexample is $\pi^k_{min} = [s_1, \ldots, s_k]$ and $s_k = \epsilon$ then the k-induction algorithm will find the counterexample in at least $\lfloor \frac{k}{2} \rfloor + 1$ iterations.*

Proof. In order to prove this theorem, we assume that the inductive step always returns the same non-spurious partial counterexample for every k; we will show how to give partial guarantees to this assumption in Section 4.3.2.

First, we show that no more than $\lfloor \frac{k}{2} \rfloor + 1$ iterations are required to find a property violation. By contradiction, assume that the number of iterations required to find the property violation is greater than $\lfloor \frac{k}{2} \rfloor + 1$.

Let us assume a π^k_{min} where k is even. In the iteration $\frac{k}{2}$, the new base case will have explored all states up to $\frac{k}{2}$ and the inductive step will have provided a partial counterexample $\pi^{\frac{k}{2}}_{back}$. In the iteration $\frac{k}{2}+1$, the new base case will not reach any state in $\pi^{\frac{k}{2}}_{back}$ if either (1) the counterexample π_{back} is spurious (contradicting our initial assumption) or (2) the counterexample π_{back} is not spurious and there is at least one state $s_u \in \pi^k_{min}$ such that $\pi^k_{min} = [s_1, \ldots, s_{\frac{k}{2}+1}] \cdot [s_u, \ldots] \cdot \pi^{\frac{k}{2}+1}_{back}$, which has a length greater than k contradicting our assumption about the length of π^k_{min}.

Now, assume a π^k_{min} where k is odd. The proof is similar to the one where k is even, except that we consider the sequences at iteration $\frac{k-1}{2}$. If the counterexample is not found in the iteration $\frac{k-1}{2}+1$, then either (1) the counterexample is spurious or (2) a higher number of iterations is required to find the property violation, contradicting our initial assumptions.

We then generalise and conclude that no more than $\lfloor \frac{k}{2} \rfloor + 1$ iterations are required to find a property violation.

Now we show that at least $\lfloor \frac{k}{2} \rfloor + 1$ iterations are required to find a property violation. By contradiction, assume that the number of iterations required to find the property violation is less than $\lfloor \frac{k}{2} \rfloor + 1$. This means that either (1) there exists a smaller counterexample that was not found the base case, which is ensured by lemma 4.6 or (2) there is a state $s_v \in \pi^k_{min}$ such that $\pi^k_{min} = [s_1, \ldots, s_v] \cdot [s_{v+1}, \ldots, s_k]$ which has a length smaller than k contradicting our assumption about the length of π^k_{min}.

Given that at least $\lfloor \frac{k}{2} \rfloor + 1$ iterations are needed to find the property violation and no more than $\lfloor \frac{k}{2} \rfloor + 1$ iterations are needed to find the property violation, we can conclude that the *bkind* algorithm will find a counterexample π^k_{min} in exactly $\lfloor \frac{k}{2} \rfloor + 1$ iterations.

Theorem 4.9 (Soundness of the *bkind* algorithm)**.** *If the bkind algorithm returns:*

1. *a sequence of states $[s_i, \ldots, s_j]$: the program is unsafe, and the sequence is a non-spurious counterexample.*
2. *$\emptyset$: the program is safe.*
3. *unknown: the program is safe up to k_{max} iterations.*

Proof. The proof follows the same proof as the Theorem 4.5, except that lemma 4.6 ensures the first item instead of lemma 4.1.

4.3.2 Constraint Generation using Interval Analysis

In this Section, we follow the same approach adopted by several other researchers to improve the *k*-induction results: to use invariants to constraint the state and filter unreachable states evaluated by the inductive step [172, 179, 180, 166]. Figure 4.4 shows

an example of the usage of invariants (dashed line) to constraint the state space. The invariants reduce the number of states explored by the backward search by constraining the over-approximation.

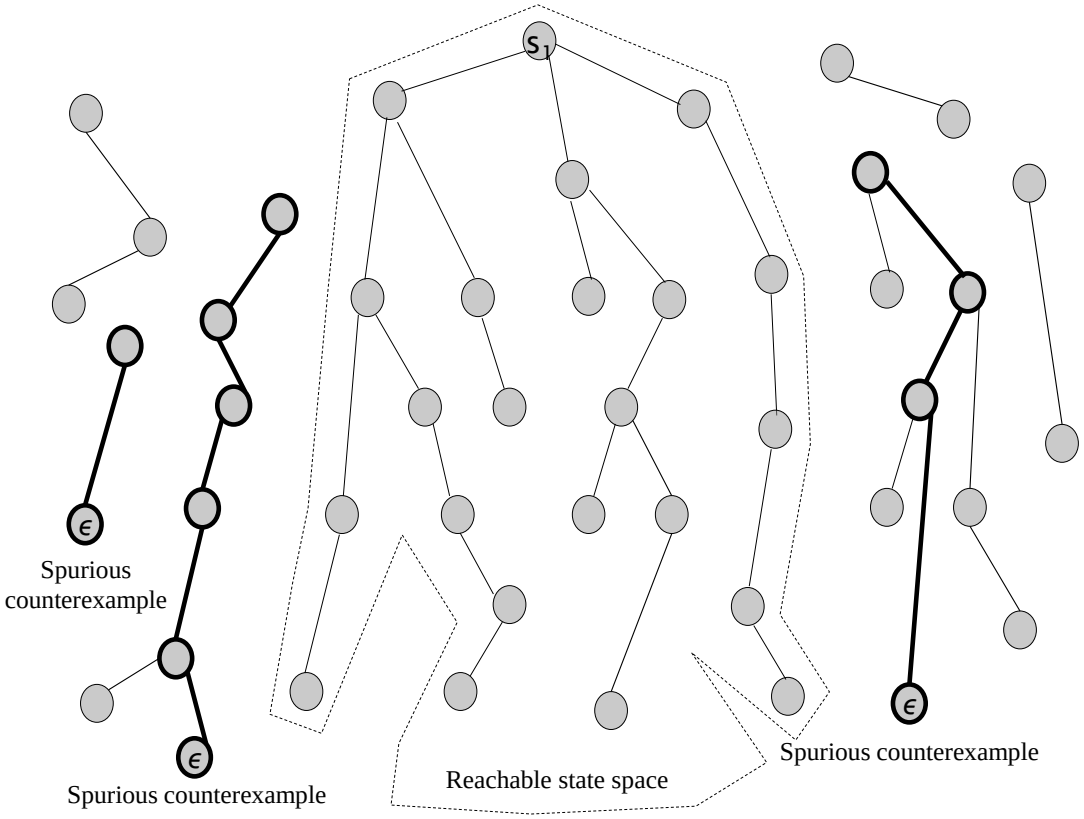


Figure 4.4: Visual representation of the state space of a program. The states inside the dashed line are the reachable states and the ones outside are unreachable states. An unconstrained over-approximation of the program assumes that all the states are reachable which might contain spurious counterexamples (counterexamples that lead to unreachable error states). An invariant is a filter of states: a strong enough invariant will remove unreachable error states from reasoning, allowing the inductive step check to prove the program correctness or to find non-spurious partial counterexamples.

We follow an approach similar to the one used by DepthK [179]: we perform a static program analysis before encoding the program and estimate the intervals that a variable can assume. Differently from DepthK, however, ESBMC does not rely on external tools and implements the invariant generation as a pre-processing step of the verification. Here we use the abstract interpretation component from CPROVER [148], which implements an abstract domain based on expressions over intervals; it associates each variable with an upper and lower bound. Our abstract domain is non-relational, i.e., we do not track intervals of variables changed through pointers and nor if the intervals are defined in terms of other variables.

Inductive step with invariants. In order to use the invariants, we need to extend the inductive step as shown in Algorithm 4.7. The algorithm is similar to the inductive step as defined in Algorithm 4.3, but it now takes an extra argument: a set of invariants φ. These invariants will constraint the state space and filter unreachable states from the inductive step check, thus reducing the number of the spurious path that might be explored.

Algorithm 4.7: The new inductive step with invariants.

```
1 Function inductive_step_invariants(P, φ, k):
2     if ∃n ∈ ℕ⁺. φ(s_n) ⋀_{i=n}^{n+k-1}(φ(s_i) ∧ tr_P(s_i, s_{i+1})) ∧ ¬φ(s_{n+k}) then
3         Let ϵ = s_{n+k} such that ¬φ(s_{n+k});
4         return [s_n, ..., ϵ];
5     else
6         return ∅;
7     end
```

Lemma 4.10 (Inductive step with invariants)**.** *If* `inductive_step_invariants(`P, φ, k`)` *returns an empty sequence of states the program is safe.*

Proof. This follows the same reasoning as lemma 4.3. We know that the program is safe up to k iterations because the base case did not find any property violation and that the inductive step over-approximates the state space when it tries to find a property violation. The new inductive step with invariants will constraint the over-approximation to be closer to the reachable state space of the program.

The lemma 4.10 is defined so the new inductive step with invariants can be used with the k-induction algorithm. Theorem 4.4 needs to be changed if invariants are used to prove correctness; lemmas 4.2 and 4.10 guarantee that the theorem is sound.

Lemma 4.11 (Partial counterexample from the inductive step with invariants)**.** *If Algorithm 4.7 returns a sequence of states* $[s_i, \ldots, s_j]$*, this is a partial counterexample of length* k*.*

Proof. This follows the same reasoning as lemma 4.7. We know this is a counterexample because of the sequence of transitions defined by tr and that the last state is an error state. Again, the invariants here will only constrain the state space, so the over-approximation is closer to the set of reachable states of the program.

The lemma 4.11 is defined so the new inductive step with invariants can be used with the *bkind* algorithm. Theorem 4.9 needs to be changed if invariants are used to prove correctness to use lemma 4.11 instead of lemma 4.3.

4.3.3 Why is the *bkind* Algorithm more Efficient than *k*-induction?

First, consider that we wish to verify the safe program in Figure 4.2a using the *bkind* algorithm. The state transition system is analysed and the following intervals are estimated based on the assignments: $\varphi = (input \geq 0, input \leq \texttt{UINT\_MAX}, s \geq 1, s \leq 5)$. The invariants are introduced in the program and are sufficient to prove that the program is safe with two loop unwindings: there will be no counterexample of size two that leads to a property violation.

Now, consider that we wish to verify the unsafe program in Figure 4.2b. Here, the same set of constraints are introduced in the program, but now the inductive step will find a set of assignment that satisfies:

$$input \geq 0 \wedge input \leq \texttt{UINT\_MAX} \wedge s \geq 1 \wedge s \leq 5 \wedge input = 5 \wedge s \geq 5$$

which is:

$$input = 5 \wedge s = 5$$

This is the program state before the error state; the reachability of this state is introduced in the program as a new property and checked in the base case. This is then extended further back for every loop iteration, effectively performing the backward search.

In conclusion, *bkind* can correctly verify both programs in Fig. 4.2: the program in Fig. 4.2a can be proven to be safe and the program in Fig. 4.2b requires fewer number of steps to find the property violation.

4.4 Experimental Evaluations

The experimental evaluation of the *bkind* algorithm and the invariant generation in ESBMC consists of three parts. In Section 4.4.1, we present the benchmarks used to evaluate ESBMC. In Section 4.4.2 we compare our *bkind* algorithm and the invariant generation with the *naïve* *k*-induction, while in Section 4.4.3 we compare the *bkind* algorithm with invariants against another state-of-the-art BMC tool that uses *k*-induction and invariant generation to verify C programs, 2LS. The tools are compared in terms of number of refuted bugs and verification time.

Our experimental evaluation aims to answer three research questions:

RQ1 **(soundness)** Does our approach produce correct results?

RQ2 **(performance I)** Does our approach improve results compared to the naïve *k*-induction algorithm?

RQ3 **(performance II)** How does our approach compare against other verification tools that use *k*-induction?

4.4.1 Experimental Setup

We use 5591 benchmarks from SV-COMP18 to evaluate the algorithms described in this Chapter. The benchmarks were extracted from the subcategories *Arrays*, *BitVectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, *Sequentialized* and *Systems_DeviceDriversLinux64*. The remaining categories were excluded because they use features that our *k*-induction does not support (termination, recursion, concurrency). When verifying programs with such features, ESBMC disables the inductive step and uses only the base case and the forward condition (for both the *k*-induction and *bkind* algorithms). Out of the 5591 benchmarks, 4134 are safe while 1457 are unsafe programs.

All experiments were conducted on IRIDIS4, the supercomputer at the University of Southampton [169]. The compute node used are equipped with Intel Sandy Bridge processors running at 2.6GHz and 24GB of RAM. We used Boolector as the SMT backend for all the verification tasks since we concluded in Chapter 3 that it is the best performing solver in ESBMC.

For each benchmark, we set time and memory limits of 900 seconds and 15GB, respectively, as per the competition definitions. We, however, do not present the results in scores (as how it is done in SV-COMP), but instead show the number of correct and incorrect results and the verification time.

Finally, given the large amount of data involved in the experiments, we use four groups to present the results in this Section: *Correct proofs* is the number of correct positive results (i.e., the tool reports SAFE correctly), *Correct alarms* is the number of correct negative results (i.e., the tool reports UNSAFE correctly), *Incorrect proofs* is the number of incorrect positive results (i.e., the tool reports SAFE incorrectly), *Incorrect alarms* is the number of incorrect false results (i.e., the tool reports UNSAFE incorrectly).

4.4.2 Comparison of *k*-induction-based approaches

In this Section we evaluate five different *k*-induction approaches. In particular, "original naïve *k*-induction" is the first version of the algorithm implemented in ESBMC (described in [35]); we have added the results from this version to show that not only we proposed and implemented the new algorithm *bkind* but also improved ESBMC's current implementation of the *k*-induction algorithm. The other *k*-induction approaches

evaluated in this Section are: "'naïve *k*-induction" (the *k*-induction algorithm described in Sec. 4.2), "naïve *k*-induction + invariants" (the *k*-induction algorithm described in Sec. 4.2 and the invariants described in Sec. 4.3.2), "*bkind*" (the *bkind* algorithm described in Sec. 4.3.1) and "*bkind* + invariants" (the *bkind* algorithm described in Sec. 4.3.1 and the invariants described in Sec. 4.3.2).

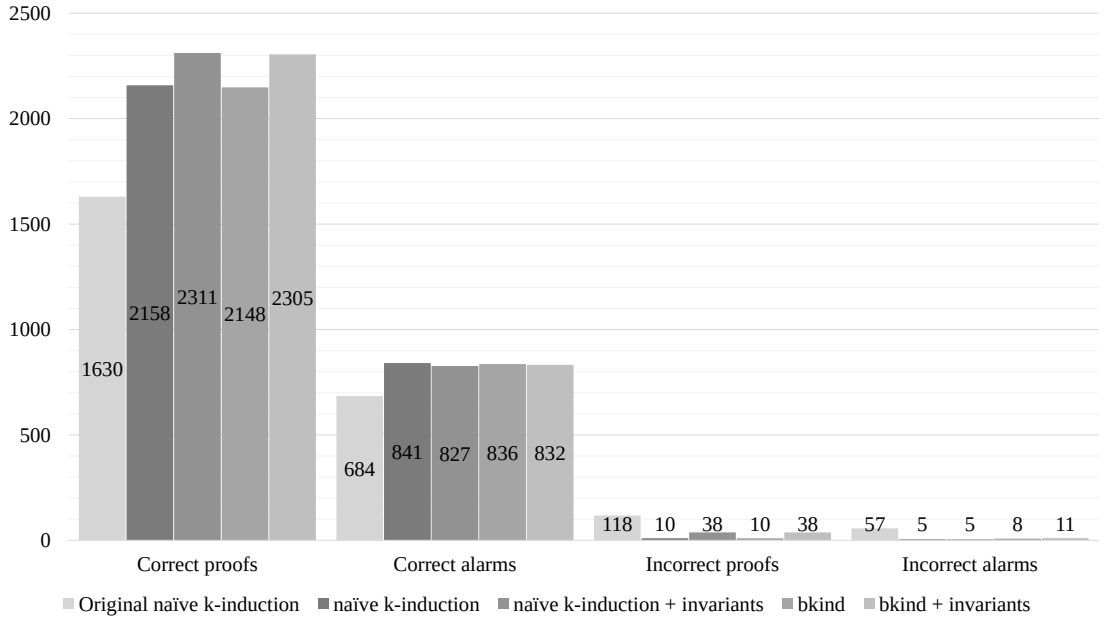


Figure 4.5: Results of the *k*-induction-based algorithms in ESBMC for all SV-COMP18 benchmarks with different configurations.

Figure 4.5 shows the results of using the *k*-induction-based approaches described in this Chapter to verify the programs in SV-COMP. First, let us compare the results of the original naïve *k*-induction [35] and the current naïve *k*-induction: the number of correct proofs and correct alarms increased by 25% and 20%, respectively, while the number of incorrect proofs and incorrect alarms decreased by 91% and 92%, respectively. Most of the wrong results in the original *k*-induction came from the fact the original algorithm (1) could not reason about early loop exits (e.g., a `break` inside a loop) and (2) would assume wrong safety conditions in the inductive step due to bugs in the implementation. Furthermore, the original *k*-induction did not support floating-point encoding which resulted in 45 incorrect alarms in the *Floats* category. It did not have the clang-based frontend and could not verify about 500 benchmarks due to parsing errors. We can only speculate, but we believe the original *k*-induction (if it could parse the programs) would report an even higher number of incorrect results, especially in the *Systems_DeviceDriversLinux64* category.

Now let us compare the current *k*-induction algorithm against the new *bkind* algorithm (with and without invariants). First, we notice that the invariants increase the number of correct proofs for both the *k*-induction and *bkind* in about 7%. This, however, comes at a cost: due to bugs in our implementation, the number of incorrect proofs is 3 times

higher when invariants are used in combination with the algorithms. In particular, our algorithm does not track intervals of variables changed through pointers and neither if the intervals are defined in terms of other variables. Fixing these limitations would greatly improve the presented results. The number of incorrect results, however, is still low: we only report incorrect proofs in about 2.5% of the 1457 incorrect benchmarks.

All the approaches (excluding the original naïve *k*-induction) report almost the same number of correct alarms with or without invariants; this is expected in the *k*-induction algorithm since the invariants are supposed only to improve the correctness proof. The *bkind* results are somewhat disappointing in this sense because we expected a higher number of bugs to be found by the algorithm. The technique indeed finds bugs in benchmarks that could not be found by the *k*-induction algorithm, but in the end, it reported a slightly smaller number of correct alarms and a higher number of incorrect alarms. An in-depth analysis of the wrong results showed that (1) when the invariants are incorrect, the *bkind* algorithm ends up finding an incorrect counterexample and (2) when the program contains arrays, the algorithm ends up generating incomplete partial counterexamples, which also lead to incorrect alarms. Despite the number of incorrect results, however, the wrong alarms only amount to 0.1% of the 4134 correct benchmarks analysed by the *k*-induction approaches.

These numbers allow us to affirm our research question RQ1 partially: the new *bkind* algorithm produces correct results for a large set of benchmarks. There are certain kinds of programs where the algorithm will provide incorrect results, but this is due to bugs in the implementation.

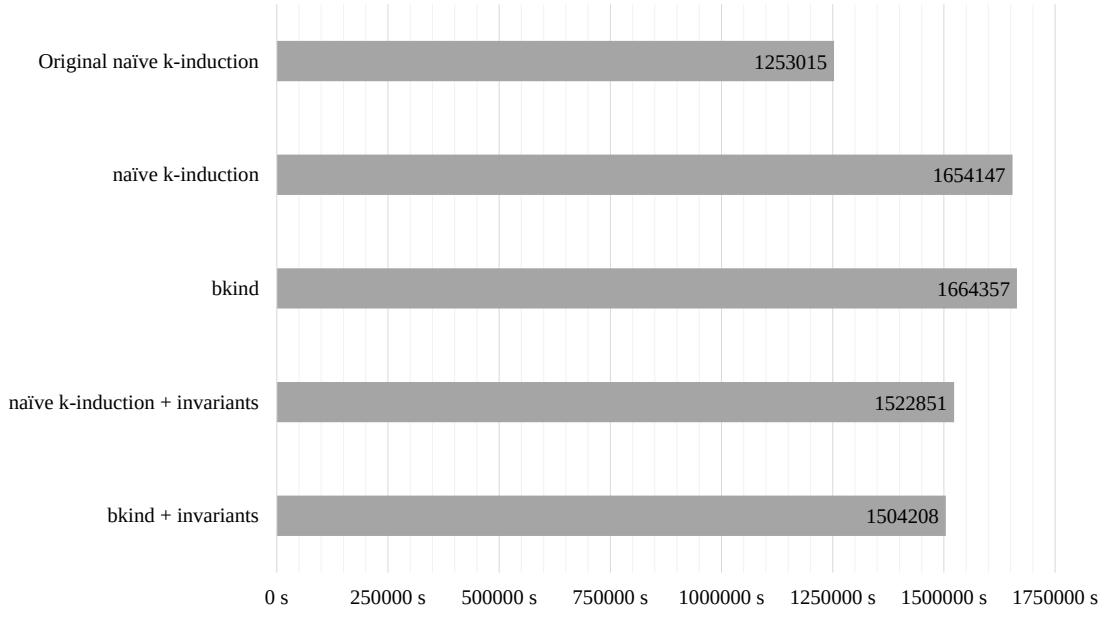


Figure 4.6: Total verification time of the *k*-induction-based algorithms in ESBMC, in seconds.

Figure 4.6 show the total verification time of the *k*-induction-based approaches. First, let us evaluate the original and the current *k*-induction: the original one is 25% faster the current *k*-induction; this can be easily explained due to the limitations in the algorithm: a number of benchmarks are not parsed by the tool, and the greater number of incorrect results allow the original *k*-induction to finish the analysis faster.

Regarding the current naïve *k*-induction and *bkind* without invariants, the latter is slightly slower (0.6%): this is expected since the inductive step is most likely to find spurious counterexamples. The slowdown presented by the *bkind* is the impact of introducing unreachable verification conditions, and it is negligible.

The results when invariants are used, however, are much better. The verification time decreases considerably in these benchmarks, making both algorithms 10% faster. Consider that the *bkind* algorithm takes 1,664,357 seconds to verify all the programs; this is equivalent to 19.2 days of continuous processing, while the *bkind* algorithm with invariants takes 1,504,208 or 17.4 days. The *bkind* algorithm with invariants speeds up the verification by almost two full days in our experiments.

These results allow us to affirm our research question RQ2: the *bkind* algorithm when used with invariants speeds up the verification compared to the naïve *k*-induction, without impacting the results.

4.4.3 2LS Comparison

We now compare the *bkind* algorithm with invariants against 2LS v0.6.0, a state-of-the-art bounded model checker with support for *k*-induction. In particular, 2LS uses the algorithm *kIkI* [180] and combines the *k*-induction algorithm with continuous invariant generation. We used the same configuration from SV-COMP18 in which 2LS is configured to generate interval constraints similar to the ones generated by ESBMC.

Figure 4.7 shows the comparative results of ESBMC against 2LS. ESBMC with the *bkind* algorithm and invariants reports more than twice correct proofs and about 35% more correct alarms compared to 2LS when analysing the same set of benchmarks. An in-depth analysis of the results shows that 2LS aborts the verification of a large number of benchmarks in the *Systems_DeviceDriversLinux64* category with the message `Irreducible control flow not supported`. Alternatively, ESBMC can prove the correctness of 1249 benchmarks in this category, significantly improving our results.

2LS provides much fewer incorrect results when compared to ESBMC in this set of benchmarks. 2LS always had a strong focus in the invariant generation since its first version; their last version in SV-COMP18 extended it even further by introducing invariant generation for termination proofs and pointer safety. The invariant generation in ESBMC is still in its first version and needs improvements.

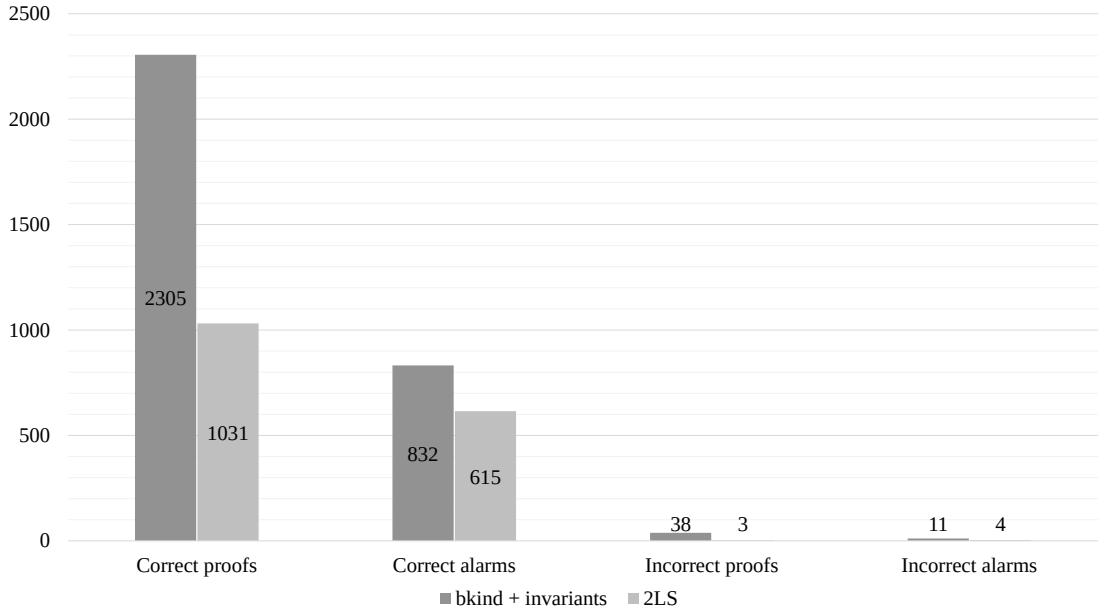


Figure 4.7: Results of ESBMC with *bkind* algorithm and invariants, and 2LS for all SV-COMP18 benchmarks.

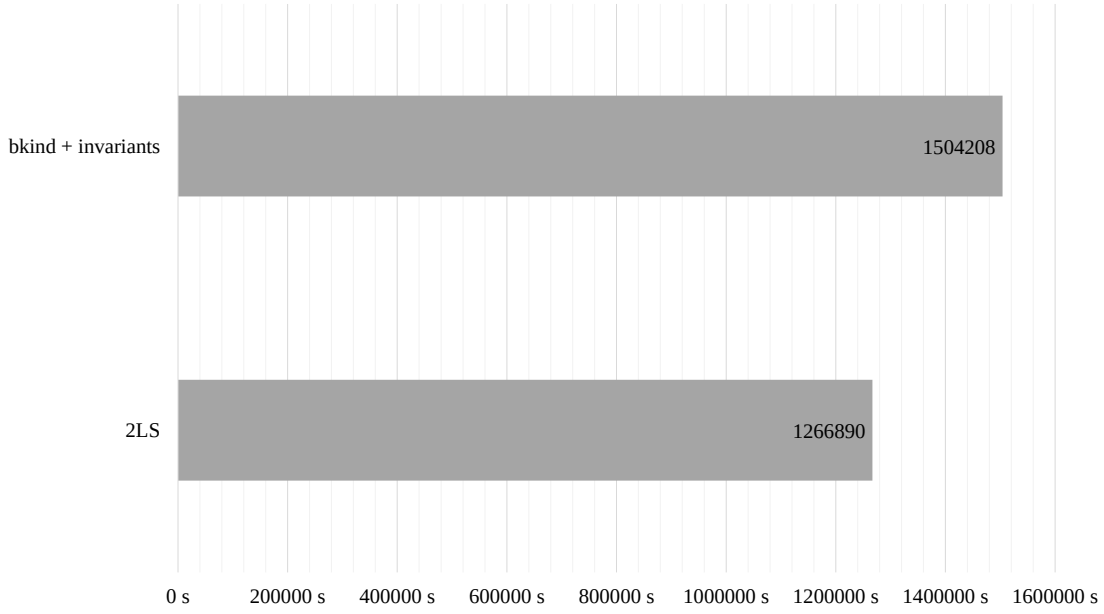


Figure 4.8: Total verification time of the *bkind* algorithm and invariants and 2LS, in seconds.

Figure 4.8 show the total verification time of ESBMC with *bkind* and invariants, and 2LS. 2LS is about 15% faster than the ESBMC even if we do not consider the *Systems_-DeviceDriversLinux64* category where 2LS would abort early in the verification. In particular, 2LS was more than ten times faster in two categories, *Arrays* and *Heap*, most likely due to the stronger invariants generated by the tool which allowed it to prove correctness faster; ESBMC would just run out of time on a large number of benchmarks

in these categories.

ESBMC with *bkind* and invariants is not the fastest verification tool using *k*-induction, but it is the one with the highest number of correct results. These results allow us to answer our research question RQ3: our novel contribution is an improvement over the current state-of-the-art verification using *k*-induction and it can report correct results for a large number of different benchmarks faster than naïve *k*-induction.

4.5 Related Work

The application of the *k*-induction method is gaining popularity in the software verification community. Donaldson *et al.* described a verification tool called Scratch to detect data races during Direct Memory Access (DMA) in the CELL BE processor from IBM [173], using *k*-induction. Properties (in the form of assertions) are automatically inserted in the program to model the behaviour of the control-flow, and the algorithm tries to find violations of those properties or prove that they hold indefinitely, using a base case and an inductive step, respectively; their proposed algorithm does not check the completeness threshold. The method also requires the code to be manually annotated with loop invariants, whereas our approach generates and adds them to the program. Finally, the tool can prove the absence of data races in many benchmarks, but it is restricted to verify that specific class of problems for a particular type of hardware, while our approach is evaluated over a more general class of programs, using the SV-COMP benchmarks.

In another related work, Donaldson *et al.* described two tools for proving the correctness of programs: K-Boogie and K-Inductor [165]. The former is an extension of the Boogie language, aimed to prove correctness (using *k*-induction) of programs written in a number of languages (Boogie, Spec, Dafny, Chalice, VCC, and Havoc), while the latter is a bounded model checker for C programs, which is built on top of the CBMC tool [23]. Both K-Boogie and K-Inductor use a *k*-induction algorithm that consists of a base case and an inductive step. The completeness threshold is not separately checked, and they rely only on the inductive step to prove correctness. Their *k*-induction algorithm has a pre-processing step, but while we introduce interval invariants during the pre-processing, their approach removes all nested loops leaving only non-nested loops. The authors compare the results of K-Inductor with Scratch and show that the new approach maintains the same coverage (regarding correctly verified programs) while being faster. However, similar to the previous work [173], the programs need to be manually changed in order to insert loop invariants while our approach does it automatically.

Malík et al. [180] describe 2LS, a C SAT-based bounded model checker. 2LS is a tool developed using the CPROVER framework [23] and aims to combine a *k*-induction algorithm with abstract interpretation. As CBMC, 2LS uses SAT solvers but instead of a fixed unwind approach, 2LS uses an incremental bounded model checking approach, where it first checks for property violations for a given bound, then tries to generate (and refine) invariants using abstract interpretation. It then builds a proof using the *k*-induction algorithm. While ESBMC is a fork of CBMC, 2LS is a tool built on top of the CPROVER framework. Their *k*-induction algorithm, called *kIkI* [166], is similar to the one implemented in ESBMC but adds an extra step to generate and refine invariants. Differently from our invariant generation algorithm that only supports interval domains, 2LS supports several abstract domains for numerical values and a shape domain for pointers [182]. 2LS also offers approaches to prove non-termination, while ESBMC has no algorithm to prove non-termination and can only prove termination by checking the unwinding assertions.

Bischoff et al. [33] propose a methodology to use BDDs and SAT solvers for the verification of programs in a bidirectional form similar to our *bkind* algorithm. In their work, they refer to the technique as *target enlargement*: the property violation is "enlarged" by checking if the states around the property violation are reachable. The BDDs are responsible for the target enlargement, collecting the under-approximated reachable state sets, followed by the SAT-based verification with the newly computed sets. The authors implemented the technique in the Intel BOolean VErifier (BOVE) and showed that the verification time of a set of public benchmarks was up to five times shorter. Compared to this work, we only use *k*-induction and SMT solvers; the inductive step in the *k*-induction algorithm is responsible for enlarging the target and the SMT solver checks for satisfiability.

Jovanović et al. [34] present a reformulation of IC3, separating the reachability checking from the inductive reasoning. They replace the conventional induction algorithm by the *k*-induction algorithm and show that it provides more concise invariants. The authors implemented the algorithm in the SALLY model checker using Yices2 to do the forward search and MathSAT5 to do the backward search. They showed that the new algorithm could solve many real-world benchmarks at least as fast as other approaches. Compared to this work, our proposed extended *k*-induction uses consecutive BMC calls to find a solution. We also implement our approach independent of solvers, and it can be used with any SMT solver supported by ESBMC; both searches, however, will be done with the same solver.

4.6 Conclusions

In this Chapter, we have described the *k*-induction algorithm and a novel contribution that extended the bug-finding capabilities of the algorithm. The new algorithm, called *bkind*, was implemented in ESBMC and evaluated in a large set of benchmarks.

k-induction is a powerful verification technique implemented in several different tools and was successfully used to verify a large number of different programs and properties. It is, however, a quite recent technique and every year we see different implementation and improvements to the algorithm. In this Chapter, we proposed and evaluated a novel improvement: useful information can be extracted for the various checks in the algorithm and can be used to improve the results of the algorithm.

In particular, the *bkind* algorithm uses information extracted from the inductive step to shorten the number of steps required to find a property violation; with strong enough invariants the *bkind* algorithm requires roughly half of the number of loop unwindings a BMC algorithm requires to find a property violation. We also have implemented an interval invariant generator that runs as a pre-processing step: invariants are automatically introduced in the program and, although the implementation has bugs, it strengths the *bkind* algorithm results. The invariant generator is based on the abstract interpretation component from the CPROVER framework [148] and it runs after the GOTO program is generated but before the SSA generation.

The results show *bkind* algorithm alone is not enough to improve the results against naïve *k*-induction, however, when combined with invariants it can reduce in 10% the verification time of a large number of benchmarks: in our experiments, this is equivalent to almost two days reduction in the verification time. In our results, most of the speedup comes from introducing invariants in the program (around 7% speed up), but the *bkind* extension can speed up the verification even further.

Chapter 5

SMT-Based Refutation of Spurious Bug Reports in the Clang Static Analyser

Abstract. *We describe and evaluate a bug refutation extension for the Clang Static Analyser (CSA) that addresses the limitations of the existing built-in constraint solver. In particular, we complement CSA's existing heuristics that remove false bug reports. We encode the path constraints produced by CSA as satisfiability modulo theory (SMT) problems, use SMT solvers to precisely check them for satisfiability, and remove bug reports whose associated path constraints are unsatisfiable. Our refutation extension refutes false bug reports in 8 out of 12 widely used open-source applications; on average, it refutes ca. 7% of all bug reports, and never refutes any accurate bug report. It incurs only negligible performance overheads, and on average adds 1.2% to the analysis time of the CSA.*

In this Chapter, we present the work done in a different tool from the previous Chapters (ESBMC). The technique described here was implemented in the Clang Static Analyser (CSA) [183], a tool part of the LLVM project. LLVM comprises a set of reusable components for program compilation [131], unlike other popular compilers, e.g., GCC and its monolithic architecture [184]. Clang [28] is an LLVM component that implements a frontend for C, C++, Objective-C and their various extensions. Clang and LLVM are used as the primary compiler technology in several closed- and open-source ecosystems, including being the primary compilation technology in MacOS and OpenBSD [185].

The CSA is an open-source project built on top of clang that can perform context-sensitive interprocedural analysis for programs written in the languages supported by clang. CSA symbolically executes the program, collects constraints, and reasons about bug reachability using a built-in constraint solver. It was designed to be fast so that it can detect common mistakes (e.g., division by zero or null pointer dereference) even in

complex programs. However, its speed comes at the expense of precision, and it cannot handle some arithmetic (e.g., remainder) and bitwise operations. In such cases, the CSA will explore execution paths which constraints might not hold, which can lead to incorrect results being reported.

```
1 unsigned int func(unsigned int a) {
2   unsigned int *z = 0;
3   if ((a & 1) && ((a & 1) ^ 1))
4     return *z;
5   return 0;
6 }
```

Figure 5.1: A small C safe program. The dereference in line 4 is unreachable because the guard in line 3 is always false.

Consider the program in Fig. 5.1. This program is safe, i.e., the unsafe pointer dereference in line 4 is unreachable because the guard in line 3 is never true; `a & 1` holds if the last bit in `a` is one, and `(a & 1) ^ 1` inverts the last bit in `a`. The analyser, however, produces the following (false) bug report when analysing the program:

```
main.c:4:12: warning: Dereference of null
 pointer (loaded from variable 'z')
    return *z;
           ^~
1 warning generated.
```

The null pointer dereference reported here means that CSA claims nevertheless to have found a path where the dereference of `z` is reachable.

Such false bug reports are in practice common; in our experience, about 50% of the reports in large systems are false. Junker et al. [186] report similar numbers for a similar symbolic execution technology. Identifying false bug reports and refactoring the code to suppress them puts a significant burden on developers and runs the risk of introducing actual bugs; these issues negate the purpose of a lightweight, fast static analysis technology.

Here we present a solution to this conundrum. We first use the fast but imprecise built-in solver to analyse the program and find potential bugs, then use slower but precise SMT solvers to refute (or validate) them; a bug is only reported if the SMT solver confirms that the bug is reachable. We implemented this approach inside clang and evaluated it over twelve widely used C/C++ open-source projects of various size (tmux, Redis, OpenSSL, twin, git, PostgreSQL, sqlite3, curl, libWebM, Memcached, Xerces-c, and XNU) using five different SMT solvers: Z3 [106], MathSAT [109], Boolector [119], Yices [120], and CVC4 [123]. Our experiments show that our refutation extension can remove false bug reports from 8 out of the 12 analysed projects; on average, it refuted 11 (or approximately 7% of all) bug reports per project, with a maximum of 51 reports

refuted for XNU; it never refuted any true bug report. Its performance overheads are negligible, and in the worst case scenario, our extension adds only 1.2% to the runtime of the static analyser.

5.1 The Clang Static Analyser

The CSA performs a context-sensitive interprocedural symbolic execution via graph reachability [187] and relies on a set of checkers, which implement the logic for detecting specific types of bugs [183, 188]. Each path in a function is explored; this includes taking separate branches and different loop unwindings. Function calls on these paths are inlined whenever possible, so their contexts and paths are visible from the caller's context.

Real-world programs, however, usually depend on external information, such as user inputs or results from library components, for which source code is not always available [189]. These unknown values are represented by free variables and the built-in constraint solver in the static analyser reasons about reachability based on expressions containing these symbols.

The CSA relies on a set of checkers that are engineered to find specific types of bugs, ranging from undefined behaviour to security property violations, e.g., incorrect usage of insecure functions such as `strcpy` and `strcat` [190]. The checkers subscribe to events (i.e., specific operations that occur during symbolic execution) that are relevant to their bug targets; for example, the nullability checker subscribes to pointer dereferences. They then check the constraints in the current path and throw warnings if they consider a bug to be reachable.

The checkers can report incorrect results as the symbolic analysis is incomplete (i.e., they can miss some true bugs) and unsound (i.e., they can generate false bug reports). The sources of these incorrect results are approximations in two components, the control-flow analyser and the constraint solver.

The control-flow analyser evaluates function calls inside a single translation unit (TU); if the symbolic execution engine finds a function call implemented in another TU, then the call is skipped, and pointers and references passed as parameters are invalidated while the function return value is assumed to be unknown. Cross translation unit support (CTU) is under development [188] but it is not part of the CSA main branch yet.

The built-in constraint solver (based on interval arithmetic) was built to be fast rather than precise and removes expressions from the reasoning if they contain unsupported operations (e.g., remainder and bitwise operations) or are deemed too complicated (e.g., containing more than one operator symbol).

The bug reports generated by the checkers are then post-processed before they are reported to the user. In this final step, some heuristics are applied to remove false bug reports and to beautify the reports. The reports are also deduplicated so that different paths that lead to the same bug only generate one report.

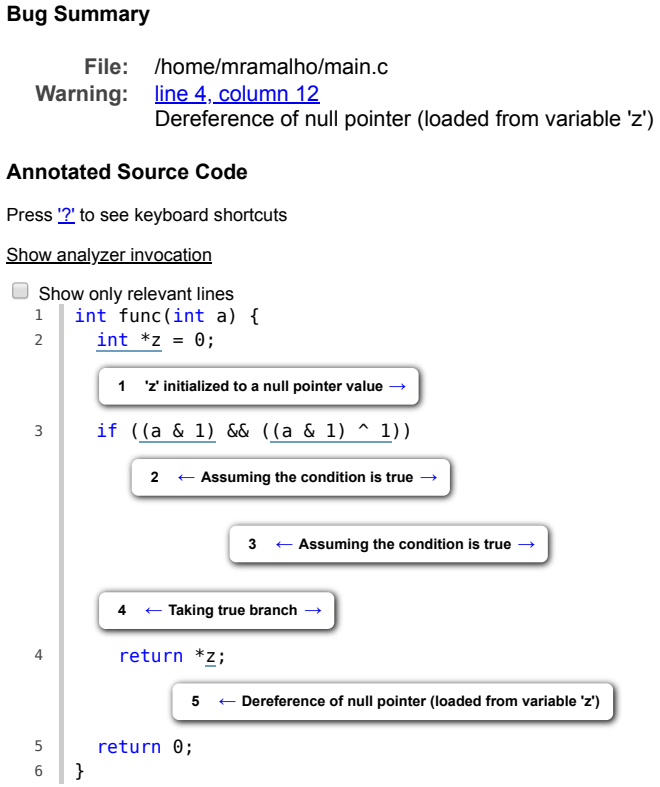


Figure 5.2: The (false) bug report produced by the clang static analyser. The annotations about the path followed are inserted in a post-processing step.

The (incorrect) annotated report is shown in Figure 5.2. Note the annotations regarding the `if` in line 3. The `if` is assumed to be true because the built-in constraint solver cannot reason about it correctly.

5.2 Refuting False Bugs using SMT Solvers

One approach to address the limitations of the built-in constraint solver is to replace it with an SMT solver. This approach has been implemented in clang, but empirical evaluations show that this approach can be up to 20 times slower.[1]

We developed an alternative solution: we use the more precise SMT solvers to reason about bug reachability only in the post-processing step. The CSA already has heuristics

[1] https://reviews.llvm.org/D28952

in place to remove false bug reports, so we extended those heuristics to precisely encode the constraints in SMT and to check for satisfiability. The built-in constraint solver runs first, and the SMT solver later validates the bug reports; a bug is only reported if both solvers agree that it is a reachable bug.

Fig. 5.3 illustrates the architecture of our solution. The CSA performs the analysis without interference from our extension: it symbolically executes the program AST, calling registered checkers which in turn call the built-in constraint solver. If a set of constraint is satisfiable, the checker produces a new bug report and return it to the CSA. After the CSA finishes the symbolic execution, the SMT-based refutation extension will encode the constraints of each bug report as SMT formulas and check them for satisfiability. CSA already supports constraint encoding in SMT using Z3 [106] but we also implemented support for Boolector [119], Yices [120], MathSAT [109], and CVC4 [123].

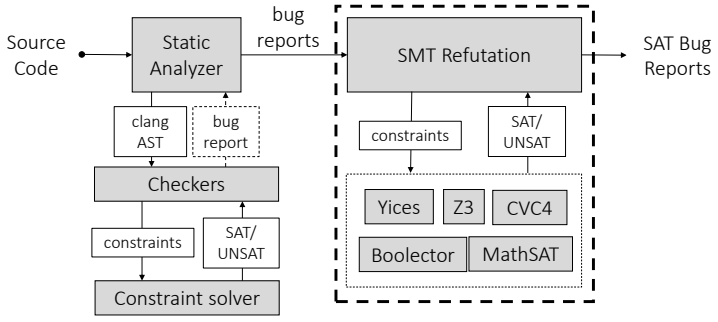


Figure 5.3: The refutation extension in the clang Static Analyser.

A bug report BR is a path to a property violation (i.e., an ϵ state). Our refutation extension walks backwards through all states s_i in BR, collects their constraints, and checks their conjunction for satisfiability. If the formula is unsatisfiable, the bug is unreachable and the bug report is discarded. In the CSA, a constraint c is a map $var \to \mathbb{Z} \times \mathbb{Z}$.

Algorithm 5.1: *encodeConstraint*(cs, Φ)

Input: `A set of constraints` C `and an SMT formula` Φ
Output: `The formula` Φ `with all constraints` c `encoded in SMT`

```
1 foreach c ∈ C do
2     if c.var ∈ Φ then
3         continue
4     else if c.interval.lower == c.interval.upper then
5         encode(c.var == c.interval.lower, Φ)
6     else
7         encode(c.var ≥ c.interval.lower ∧ c.var ≤ c.interval.upper, Φ)
8     end
9 end
```

Our constraints encoding algorithm is shown in Algorithm 5.1. Let us assume a set of constraints C, an SMT formula Φ, and a method $encode(expr, \Phi)$, which encodes an expression $expr$ in the SMT formula Φ. Algorithm 5.1 contains two optimizations when encoding constraints: duplicated symbol constraints are ignored (line 3) and if the constraint is a concrete value (the lower bound is equal to the upper bound), the constraint is encoded as an equality (line 4). The former decreased the total verification time so that the overhead introduced by the refutation algorithm was less than 20% across the large set of projects evaluated in Section 5.3 while the latter improved verification time by 3% per SMT query. Note that ignoring symbol constraints in line 3 is only possible because the refutation extension runs from the last state in a bug report (the property violation) to the initial state; any new symbol constraint found when walking backwards will always be weaker than the symbol constraints already encoded.

Fig. 5.4 shows the SMT formula of the bug found when analysing the program in Fig. 5.1. Here, `$0` is the variable `a`. In practice, the formula is equivalent to checking if the guard in line 3 of the program in Figure 5.1 holds. Since the formula is unsatisfiable, CSA will not produce a bug report for this path.

```
1 (declare-fun $0 () (_ BitVec 32))
2 (assert (= ((_ extract 0 0) $0) #b1))
3 (assert (= ((_ extract 0 0) $0) #b0))
```

Figure 5.4: The SMT formula of the bug report from Fig. 5.1, using Z3. Note that the solver was able to simplify the formula to two assertions: that the first bit should be one and zero at the same time. Since this is a contradiction, the formula is unsatisfiable.

5.3 Experimental Evaluation

The experimental evaluation of the refutation algorithm in the CSA has two parts. In Section 5.3.1, we present the projects evaluated and the environment setup, while in Section 5.3.2 we compare the analysis results from the clang static analyser with and without the bug refutation algorithm; the two approaches are compared regarding the number of refuted bugs and verification time.

Our experimental evaluation aims to answer two research questions:

RQ1 **(soundness)** Is our approach sound and can the refuted bugs be confirmed?

RQ2 **(performance)** Is our approach able to refute spurious bug reports in a reasonable amount of time?

5.3.1 Experimental Setup

We evaluated the new bug refutation algorithm in twelve open-source C/C++ projects:

1. tmux (v2.7): a terminal multiplexer.
2. Redis (v4.0.9): an in-memory database.
3. OpenSSL (v1.1.1-pre6): a software library for secure communication.
4. twin (v0.8.0): a windowing environment.
5. git (v2.17.0): a version control system.
6. postgreSQL (v10.4): an object-relational database management system.
7. SQLite3 (v3230100): a relational database management system.
8. curl (v7.61.0): a command-line tool for transferring data.
9. libWebM (v1.0.0.27): a WebM container library.
10. Memcached (v1.5.9): a general-purpose distributed memory caching system.
11. xerces-c++ (v3.2.1): a validating XML parser.
12. XNU (v4570.41.2): the operating system kernel used in Apple products.

Clang provides some scripts to analyse projects using the clang static analyser. The main scripts are:

- `SATestAdd.py`: analyses a project, creates reference results and adds the project to the list of projects to be analysed.
- `SATestBuild.py`: generates comparative results or regenerates reference results.
- `CmpRuns.py`: given two results, shows comparative statistics.

In order to use these scripts, the projects first need to be added to the list of projects to be analysed, using `SATestAdd.py <project-directory>`, e.g.:

```
$ ./SATestAdd.py tmux
```

In each project directory, the `SATestAdd.py` script expects the following files:

- `cleanup_run_static_analyzer.sh`: a script executed before analysis to remove any compilation object from previous compilations.

- `download_project.sh`: a script to download the project and unpack it to a folder called CachedSource.
- `run_static_analyzer.cmd`: a script with the commands to configure and build the project.
- `changes_for_analyzer.patch`: an optional file. It is applied to the project's source code before the `run_static_analyzer.cmd` is executed.

The script `SATestAdd.py` will use these scripts to download, apply the patch to the project (optional), analyse the project, generate the reference reports, and add the project path to a file called `projectMap.csv` (first creating the file if it does not exist).

Once `projectMap.csv` exists, `SATestBuild.py` can be used to generate reference reports:

```
$ ./SATestBuild.py -r
```

or to generate comparative results:

```
$ ./SATestBuild.py
```

`SATestBuild.py` will analyse all the projects in the `projectMap.csv` and, if generating comparative results, it will print the number of bugs added or removed.

All experiments were conducted on a computer with an Intel Core i7-2600 running at 3.40GHz and 24GB of RAM. We used clang v7.0 and a time limit of 15s per bug report was set for the projects. All the scripts required to analyse all the projects are available in https://github.com/mikhailramalho/analyzer-projects.

5.3.1.1 Patches to projects

Two projects had to be patched so the analyser could analyse them, but the patches do not change the program behaviour:

LibWebM. The patch (1) adds `#include ''cstring''` to the 'webm2pes.cc' file to fix an error that `''std::memcpy''` could not be found, and (2) adds `static_cast` to the same file to fix some integer narrowing errors.

XNU. The patch (1) changes two makefiles do disable compilation with "-Werror", (2) adds `#include ''stddef.h''` to fix a missing declaration of `ptrdiff_t`, and (3) removes the `const` modifier of a member that causes the clang static analyser to complain about an assignment to a constant member and abort.

Projects	Time (s) (no ref)	Time (s) (with ref)	Reported bugs (no ref)	Refuted bugs
tmux	86.5	89.9	19	0
redis	347.8	338.3	93	1
openSSL	138.0	128.0	38	2
twin	225.6	216.7	63	1
git	488.7	405.9	70	11
postgreSQL	1167.2	1112.4	196	6
SQLite3	1078.6	1058.4	83	15
curl	79.8	79.9	39	0
libWebM	43.9	44.2	6	0
memcached	96.0	96.2	25	0
xerces-c++	489.8	433.2	81	2
XNU	3441.7	3405.1	557	51
Total	7683.7	7408.5	1270	89

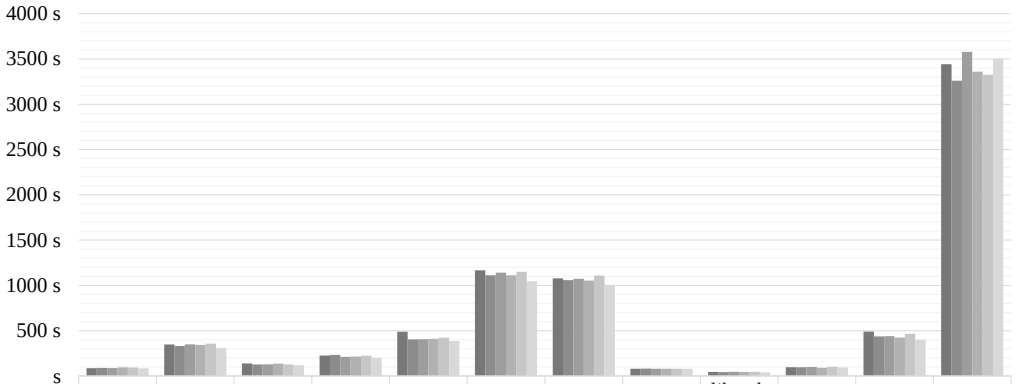
Table 5.1: Results of the analysis with and without refutation.

5.3.2 Bug Refutation Comparison

Table 5.1 shows the results of CSA with and without bug refutation enabled. Here, *Time (s) (no ref)* is the analysis time without refutation, *Time (s) (with ref)* is the analysis time with refutation enabled. Both are averaged times for all supported solvers (Z3, Boolector, MathSAT, Yices and CVC4). *Reported bugs (no ref)* is the number of bug reports produced without refutation and *Refuted bugs* is the number of refuted bugs. All solvers refuted the same bugs. There were bugs refuted in 8 out of the 12 analysed projects: Redis, OpenSSL, twin, git, PostgreSQL, sqlite3, Xerces and XNU. On average, 11 bugs were refuted when analysing these projects, with 51 bugs refuted in XNU.

In total, 89 bugs were refuted, and an in-depth analysis of them show that all of them were false positives, thus affirming RQ1. Our technique, however, is not able to refute all false bugs as the interprocedural analysis is another source of false positives in CSA, and it was not addressed in this work.

Figure 5.5 shows the analysis time of the projects for each solver. Note that the time difference between them is minimal. The average time to analyse the projects with refuted bugs was 35.0 seconds faster, a 6.25% speed up, thus affirming RQ2. The static analyser generates HTML reports for each bug report which involves a lot of IO (e.g., the HTML report produced for the program in Figure 5.1 is around 25kB), and by removing these false bugs, fewer reports are generated, and the analysis is slightly faster. Out of the four projects where no bug was refuted (tmux, curl, libWebM and Memcached), the analysis was 1.0 seconds slower on average, only a 1.24% slowdown.



	tmux	redis	openssl	twin	git	postgresql	sqlite3	curl	libwebm	memcached	xerces	xnu
Analysis time (no refutation)	86.5	347.8	138	225.6	488.7	1167.2	1078.6	79.8	43.9	96	489.8	3441.7
Z3	89.1	331.7	127.7	232.5	403.7	1111.8	1058.9	81.7	43.8	97.3	436.9	3260.5
MathSAT	86.9	350	129.5	212.1	406.1	1141.7	1074.3	79.8	46.4	99.5	440.3	3577.9
Yices	94.7	343.4	136.6	214.3	410.7	1112.4	1054.3	79.9	43.9	90.6	423.6	3359.6
Boolector	93.9	357.6	126.8	223.4	423.5	1150.6	1108.3	79.5	46.3	100.6	465.8	3326.1
CVC4	85	309	119.5	201.4	385.8	1045.7	996.4	78.7	40.5	93.2	399.5	3501.5

Figure 5.5: Number of bugs reported by the clang static analyser, with and without refutation. The refutation algorithm removed, on average, 12 bugs across seven projects.

5.4 Related Work

Static analysis of programs has seen significant improvement in the last few years and in many cases it has been applied for the analysis of big real-world projects. Cppcheck [18] focuses on detecting dangerous coding practices and undefined behaviours in C and C++ programs. The tool offers both visual and command-line interfaces to analyse projects, with the command line interface being more tailored for automated analysis environments. Cppcheck implements a context-sensitive interprocedural data-flow analysis similar to the CSA but also performs abstract interpretation when handling programs with loops. The significant difference to CSA is that Cppcheck uses a custom parser and lexical analyser to build the AST of the program, and for that reason it fails more often due to not being conformant with all of the C++ standards; the CSA does not suffer from this since it relies on clang to generate the program AST. Cppcheck, however, is perhaps the most successful static analyser freely available and it was used to detect vulnerabilities in many widely used projects, including a 23 years old stack overflow in X.Org [74].

Infer [191] is an open-source static code analysis tool used for the verification of the Facebook code base and on many mobile apps (e.g., WhatsApp); many companies adopt it (e.g., Mozilla, Spotify). It supports programs written in Java, C and Objective-C, and reason about bug feasibility using a combination of separation logic [192] and bi-abduction [193]. The tool is mainly focused on finding resource leaks and null pointers exceptions in mobile apps. The most significant feature in Infer, however, is its

incremental analysis which builds a summary of changes in a commit and analyses only functions affected by the changes, making it easy to integrate the analysis in a continuous integration environment. Compared to CSA, Infer uses different logic to reason about bugs. The analysis, however, is still imprecise and produces both false positives and false negatives. Differently from CSA, Infer was developed to be easily integrated into a continuous development environment and provides more facilities to filter false results reported in previous analyses.

An approach similar to the one adopted in this paper has also been used in Goanna [186], a C and C++ static analyser able to scale up to real-world programs (e.g., OpenSSL, Wireshark). After detecting the bugs, false bugs are eliminated by analysing the feasibility of error paths using SMT solvers. The most significant difference between Goanna and our approach is in the first (imprecise) analysis: Goanna uses NuXmv and MathSAT5 to generate the bug reports, while we use a custom-built constraint solver. Similarly to our approach, they use Z3 to encode and refute false bug reports, but we offer a more comprehensive selection of SMT solvers to choose from.

5.5 Conclusions

In this Chapter, we have described a new bug refutation algorithm implemented in the clang static analyser. The bug refutation algorithm was implemented as an optional flag in the analyser and it is not enabled by default due to license issues, i.e., even the free and open-source SMT solvers are MIT-licensed (e.g., Z3, Boolector) and would require that its license is shipped with clang; discussions to incorporate MIT-licensed projects have been going for a few years in the clang community and reached no conclusion.

Our SMT-based bug refutation extension in the clang static analyser is a simple but powerful extension; it can prevent the class of false bugs generated by the unsound built-in constraint solver from being reported to the user and introduces only a minimal overhead to the analysis. The bug refutation algorithm, however, cannot remove all the false bugs in the analysis since the incomplete interprocedural analysis is another source of incorrect results; support for a complete interprocedural analysis is currently under development [188].

The false positives, however, can be reduced if developers are willing to change their programs; many cases of false positives are caused by fragile code which probably should be refactored, or at least protected at runtime by assertions. Expecting such commitment from developers, however, is probably impractical; triaging and fixing a single false positive easily takes multiple days, a cost that is usually too high to be paid by companies. The refutation algorithm is helpful in such scenarios where a company or developer wants to analyse their projects; the fewer false positives in their reports, the better.

Our empirical evaluation shows that the bug refutation algorithm can consistently reduce the analysis time by 6.25% if bugs are removed, while only slowing down the verification by 1.24% if no bug is refuted. We used five different solvers in our experiments (Z3, Boolector, MathSAT, Yices and CVC4) and their performance is comparable. Our refutation extension using Z3 is already part of clang version 7 and support for the other solvers (Boolector, MathSAT, Yices and CVC4) are under review for clang version 8.

Chapter 6

Conclusions

This thesis examined the automated verification of programs written in the C program language using bounded model checking and symbolic execution. Both techniques were developed to address the problem of state space explosion, which is one of the significant limitations in software verification nowadays. In particular, these techniques explore the state space (or a region of the state space) differently and with different degrees of precision.

As a first step, we addressed the problem of modelling and verifying a real-world C program. In particular, we developed a new clang-based frontend, an SMT backend with support for floating-point arithmetic, and an incremental bounded model checking infrastructure in ESBMC, an SMT-based bounded model checker. The clang-based frontend addresses the issue of supporting the ever-evolving C standard, the floating-point backend allows the correct encoding of floating-point arithmetic, an essential feature for real-world C programs verification, and the incremental bounded model checking infrastructure serves as a base for several other algorithms, including *bkind*. The new floating-point backend was further extended to encode all operations using bit-vectors, extending the floating-point support to all SMT solvers supported by ESBMC. We evaluated these technologies using more the 9500 publically available benchmarks, and the results show that the new frontend can correctly parse and type-check all programs, including previously unsupported and complex programs. We evaluated these benchmarks using five state-of-the-art SMT solvers (Boolector, Z3, MathSAT, CVC4 and Yices) and we provided evidence for the choice of the default solver in ESBMC for other experiments. Results also show that our floating-point backend is sound and, when using Boolector, our floating-point bit-vector encoding outperforms other solvers with native floating-point encoding.

Once these technologies were ready, we developed the *bkind* algorithm: an extension to the *k*-induction algorithm, to improve its bug-finding capabilities. In particular, we extended the algorithm to use information already available but otherwise discarded

during verification, effectively turning the algorithm into a bidirectional bug-finding technique that potentially reduces the numbers of loop unwindings required to find a property violation in a program by half. To strengthen the technique, we also implemented an interval analyser in ESBMC. We evaluated the new algorithm using over 5000 benchmarks, and results show that the *bkind* algorithm can reduce the verification time when combined with invariants. Furthermore, we compared the new algorithm against another state-of-the-art *k*-induction tool, and our novel algorithm was able to verify more programs, while only being 15% slower.

Finally, we show an extension to the clang static analyser, aimed to reduce the number of false bugs reported by the tool. We have extended the support for SMT solvers in the clang static analyser and extended the heuristics in the analyser to precisely encode the constraints in a bug report in SMT and check for satisfiability; this addresses one problem in the analyser which has a fast but unsound built-in constraint solver. We evaluated our approach using twelve open source projects of various sizes, including the MacOS kernel XNU. Results show that our approach was able to remove several false bugs and, when false bugs are removed, the analysis is faster since the analyser has fewer bug reports to generate (around 6% faster). Furthermore, when no false bug is removed, the overhead introduced by our approach is negligible (around 1% slowdown).

6.1 Future Works

The *bkind* algorithm presented in this thesis can be further improved. The algorithm still generates information that is discarded during verification: the counterexample generated by the forward condition. The counterexample generated by this check can provide hints about the complexity threshold of a program or at least can provide information about minimal loop unwindings required to explore more region of the state space.

There are many engineering tasks to be done with ESBMC that could lead to better results in the future. In particular, bugs in the interval analyser should be fixed, or it should be dropped altogether in favour of better technology. DepthK offer options to annotate the program with invariants and 2LS generates invariants during program verification.

It would also be worthwhile to develop a clang-based C++ frontend for ESBMC. We currently suffer from the same limitation we had with the C frontend: it is massive and hard to maintain. A clang-based C++ frontend would allow ESBMC to verify C++ programs, including programs that use STL correctly; currently, we achieve this by using operational models, but the results vary greatly. By using a clang-based C++ frontend, we would have the ability to verify the STL implementation itself.

In the clang static analyser, supporting cross translation unit analysis would significantly increase its effectiveness. There are some proposed solutions, but none seem to scale for industrial needs yet.

Regarding the SMT solvers in LLVM, our SMT API in the compiler can open several doors, from optimisations to soundness checks. In particular, we are in the process of moving the SMT API from clang to LLVM so that it can be used to validate the transformation of the scalar evolution pass [194].

Bibliography

[1] Steve Heath. *Embedded Systems Design*. Newnes, Oxford, United Kingdom, 2003.

[2] Hermann Koptez. *Real-Time Systems: Design Principles For Distributed Embedded Applications*. Springer, New York, EUA, 2011.

[3] Joe Armstrong. A History Of Erlang. In *History Of Programming Languages*, pages 6–26, New York, NY, USA, 2007. ACM.

[4] Mark Dowson. The Ariane 5 Software Failure. *Software Engineering Notes*, 22(2): 84–, 1997.

[5] Report by the Inquiry Board. ARIANE 5 Flight 501 Failure, 19 July 1996.

[6] Zakir Durumeric, James Kasten, David Adrian, John Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter Of Heartbleed. In *Internet Measurement Conference*, pages 475–488, 2014.

[7] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[8] Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*. MIT Press, Cambridge, United Kingdom, 2008.

[9] Lionel C. Briand. A Critical Analysis Of Empirical Research In Software Testing. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–8, 2007.

[10] Beizer Boris. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[11] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core Compiler Fuzzing. In *Programming Language Design And Implementation*, pages 65–76, 2015.

[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX*, pages 28–38, 2012.

[13] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection In Practice. In *WBIA*, pages 62–71, 2009.

[14] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast Detector Of Uninitialized Memory Use In C++. In *CGO*, pages 46–55, 2015.

[15] Undefined Behavior Sanitizer. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2016. [Online; accessed September-2018].

[16] Leak Sanitizer. http://clang.llvm.org/docs/LeakSanitizer.html, 2016. [Online; accessed September-2018].

[17] Data Flow Sanitizer. http://clang.llvm.org/docs/DataFlowSanitizer.html, 2016. [Online; accessed September-2018].

[18] Cppcheck - A Tool For Static C/C++ Code Analysis. http://cppcheck.sourceforge.net, 2018. [Online; accessed September-2018].

[19] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A Memory Model For Static Analysis Of C Programs. In *Symposium On Leveraging Applications Of Formal Methods, Verification And Validation*, pages 535–548, 2010.

[20] David Monniaux. Formal Methods And Static Analysis : An Overview, 2012.

[21] Armin Biere. *Handbook Of Satisfiability*, volume 185. IOS Press, 2009.

[22] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 1633 of *LNCS*, pages 193–207, 1999.

[23] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool For Checking ANSI-C Programs. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 2988 of *LNCS*, pages 168–176, 2004.

[24] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-Based Bounded Model Checking For Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.

[25] Shaz Qadeer and Jakob Rehof. Context-Bounded Model Checking Of Concurrent Software. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 3440 of *LNCS*, pages 93–107, 2005.

[26] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution For Automated Real-world Software Testing. In *European Conference On Computer Systems*, pages 183–198, New York, NY, USA, 2011. ACM.

[27] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. KLOVER: A Symbolic Execution And Automatic Test Generation Tool For C++ Programs. In *Computer-Aided Verification*, pages 609–615, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Chris Lattner. *Clang Documentation*. The Clang-LLVM Project, 2015. [Online; accessed September-2018].

[29] Mikhail R. Gadelha. Using Clang As A Frontend On A Formal Verification Tool. In *FOSDEM*, 2017.

[30] Mikhail R. Gadelha, Jeremy Morse, Lucas C. Cordeiro, and Denis Nicole. Using Clang As A Frontend On A Formal Verification Tool. In *European LLVM Developers Meeting*, 2018.

[31] Mikhail Y. R. Gadelha, Lucas C. Cordeiro, and Denis A. Nicole. Encoding Floating-Point Numbers Using The SMT Theory In ESBMC: An Empirical Evaluation Over The SV-COMP Benchmarks. In *Simpsio Brasileiro De Mtodos Formais*, pages 91–106, 2017.

[32] Dirk Beyer. Reliable And Reproducible Competition Results With BenchExec And Witnesses (Report On SV-COMP 2016). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9636 of *LNCS*, pages 887–904, 2016.

[33] Gabriel P. Bischoff, Karl S. Brace, G. Cabodi, and S. Nocco, S.and Quer. Exploiting Target Enlargement And Dynamic Abstraction Within Mixed BDD And SAT Invariant Checking. *Electronic Notes in Theoretical Computer Science*, 119(2):33–49, 2005.

[34] Dejan Jovanović and Bruno Dutertre. Property-directed *k*-induction. In *Formal Methods In Computer-Aided Design*, pages 85–92, 2016.

[35] Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro. Handling Loops In Bounded Model Checking Of C Programs Via *k*-induction. *International Journal on Software Tools for Technology Transfer*, 19(1):97–114, 2017.

[36] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole. Towards Counterexample-guided *k*-Induction For Fast Bug Detection. In *ACM Joint European Software Engineering Conference And Symposium On The Foundations Of Software Engineering*, 2018.

[37] Mikhail R. Gadelha and Enrico Steffinlongo. SMT-Based Refutation Of Spurious Bug Reports In The Clang Static Analyzer. In *Bay Area LLVM Developers Meeting*, 2018.

[38] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[39] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling Network Verification Using Symmetry And Surgery.

In *Symposium On Principles Of Programming Languages*, pages 69–83, New York, NY, USA, 2016. ACM.

[40] Thierry Lecomte. Safe And Reliable Metro Platform Screen Doors Control/Command Systems. In *Symposium On Formal Methods*, pages 430–434, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[41] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The TLA + Proof System: Building A Heterogeneous Verification Platform. In *Confederation For Thermal Analysis And Calorimetry*, pages 44–44, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[42] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common Specification Language For Static And Dynamic Analysis Of C Programs. In *ACM Symposium On Applied Computing*, pages 1230–1235, New York, NY, USA, 2013. ACM.

[43] K. Rustan M. Leino. Dafny: An Automatic Program Verifier For Functional Correctness. In *Logic For Programming, Artificial Intelligence, And Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[44] Vangalur Alagar and Kasilingam Periyasamy. *The B-Method*, pages 577–633. Springer London, London, 2011.

[45] Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Model Checking LTL Properties Over ANSI-C Programs With Bounded Traces. *Software and System Modeling*, 14(1):65–81, 2015.

[46] Amir Pnueli. The Temporal Logic Of Programs. In *Symposium On Foundations Of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[47] Aravinda Prasad Sistla and Edmund Clarke. The Complexity Of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, 1985.

[48] Edmund Clarke and Ernest Allen Emerson. Design And Synthesis Of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic Of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[49] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" And "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33 (1):151–178, 1986.

[50] Alan M. Turing. On Computable Numbers, With An Application To The Entscheidungsproblem. *London Mathematical Society*, 2(42):230–265, 1936.

[51] Robert W. Floyd. *Assigning Meanings To Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[52] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, And Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[53] ISO. ISO/IEC 9899:2011 – Programming Languages – C. Standard, International Organization for Standardization, Geneva, CH, 2011.

[54] Stephen A. Cook. The Complexity Of Theorem-proving Procedures. In *Symposium On The Theory Of Computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[55] John Matthews, J Strother Moore, Sandip Ray, and Daron Vroon. Verification Condition Generation Via Theorem Proving. In *Logic For Programming Artificial Intelligence And Reasoning*, pages 362–376, 2006.

[56] Ewen Denney and Bernd Fischer. Explaining Verification Conditions. In *Algebraic Methodology And Software Technology*, pages 145–159, 2008.

[57] Gerard J. Holzmann. *Spin Model Checker, The: Primer And Reference Manual*. Addison-Wesley Professional, 1st edition, 2003.

[58] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre Mckenzie. *SMV — Symbolic Model Checking*, pages 131–138. Springer Berlin Heidelberg, 2001.

[59] Moshe Y. Vardi and Pierre Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[60] Kenneth L. McMillan. *The SMV System*, pages 61–85. Springer Science+Business Media New York, 1993.

[61] John R. Burch, Edmund Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 1020 States And Beyond. *Information and Computation*, 98(2):142–170, 1992.

[62] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, 27 (6):509–516, 1978.

[63] Robert P. Kurshan. Verification Technology Transfer. In *25 Years Of Model Checking*, pages 46–64. Springer-Verlag, Berlin, Heidelberg, 2008.

[64] Randal E. Bryant, Steven M. German, and Miroslav N. Velev. Exploiting Positive Equality In A Logic Of Equality With Uninterpreted Functions. In *Computer-Aided Verification*, pages 470–482, 1999.

[65] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking Of Software Using SMT Solvers Instead Of SAT Solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.

[66] Patrick J. Hayes. The Frame Problem And Related Problems In Artificial Intelligence. Technical report, Stanford, CA, USA, 1971.

[67] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method Of Computing Static Single Assignment Form. In *Symposium On Principles Of Programming Languages*, pages 25–35, 1989.

[68] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating Tests From Counterexamples. In *Software Engineering*, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[69] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model For Static Analysis Of Programs By Construction Or Approximation Of Fixpoints. In *Symposium On Principles Of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[70] Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[71] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery Of Linear Restraints Among Variables Of A Program. In *Symposium On Principles Of Programming Languages*, pages 84–96, New York, NY, USA, 1978. ACM.

[72] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable Analysis Of Linear Systems Using Mathematical Programming. In *Verification, Model Checking, And Abstract Interpretation*, pages 25–41, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[73] James C. King. Symbolic Execution And Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[74] X.Org Security Advisory: CVE-2013-6462: Stack Buffer Overflow In Parsing Of BDF Font Files In LibXfont. https://lists.x.org/archives/xorg-announce/2014-January/002389.html, 2018. [Online; accessed September-2018].

[75] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted And Automatic Generation Of High-coverage Tests For Complex Systems Programs. In *Symposium On Operating Systems Design And Implementation*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[76] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Programming Language Design And Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[77] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine For C. In *ACM Joint European Software Engineering Conference And Symposium*

On The Foundations Of Software Engineering, pages 263–272, New York, NY, USA, 2005. ACM.

[78] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: A Symbolic Execution Extension To Java PathFinder. In *Tools And Algorithms For The Construction And Analysis Of Systems*, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[79] Jacob Burnim and Koushik Sen. Heuristics For Scalable Dynamic Test Generation. In *Automated Software Engineering*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[80] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic Execution For Software Testing In Practice: Preliminary Assessment. In *Software Engineering*, pages 1066–1071, New York, NY, USA, 2011. ACM.

[81] Patrice Godefroid. Compositional Dynamic Test Generation. In *Symposium On Principles Of Programming Languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[82] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Computer Aided Systems Theory - EUROCAST 2009. chapter Automatic Test Generation for Coverage Analysis Using CBMC, pages 287–294. Springer-Verlag, Berlin, Heidelberg, 2009.

[83] David Monniaux. The Pitfalls Of Verifying Floating-point Computations. *ACM Transactions on Programming Languages and Systems*, 30(3):12:1–12:41, 2008.

[84] George W. Gerrity. Computer Representation Of Real Numbers. *IEEE Transactions on Computers*, C-31(8):709–714, 1982.

[85] Gene Frantz and Ray Simar. Comparing Fixed- And Floating-Point DSPs. *SPRY061, Texas Instruments*, 2004.

[86] IEEE Standard For Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[87] David Goldberg. What Every Computer Scientist Should Know About Floating Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[88] Zoran Nikolić, Ha Thai Nguyen, and Gene Frantz. Design And Implementation Of Numerical Linear Algebra Algorithms On Fixed Point DSPs. *European Association for Signal Processing*, 2007(1), 2007.

[89] ARM C Language Extensions 2.1. pages 1–74, 216.

[90] Technical Committee ISO/IEC JTC 1/SC 22/WG 14. ISO/IEC 18661-3:2015 – Floating-point Extensions: Interchange And Extended Types. Standard, International Organization for Standardization, Geneva, CH, 2015.

[91] Zhoulai Fu and Zhendong Su. Achieving High Coverage For Floating-point Code Via Unconstrained Programming. In *Programming Language Design And Implementation*, pages 306–319, New York, NY, USA, 2017. ACM.

[92] Zhoulai Fu and Zhendong Su. XSat: A Fast Floating-Point Satisfiability Solver. In *Computer-Aided Verification*, volume 9780 of *LNCS*, pages 187–209, 2016.

[93] Nikolai Tillmann and Jonathan De Halleux. Pex: White Box Test Generation For .NET. In *Tests And Proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[94] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic Execution Of Floating-point Computations: Research Articles. *Software Testing, Verification & Reliability*, 16(2):97–121, 2006.

[95] Minghui Quan. Hotspot Symbolic Execution Of Floating-point Programs. In *Symposium On Foundations Of Software Engineering*, pages 1112–1114, New York, NY, USA, 2016. ACM.

[96] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental Bounded Model Checking For Embedded Software (Extended Version). *Formal Aspects of Computing*, 29(5):911–931, 2017.

[97] Hélène Collavizza, Claude Michel, Olivier Ponsini, and Michel Rueher. Generating Test Cases Inside Suspicious Intervals For Floating-point Number Programs. In *Constraints In Software Testing Verification And Analysis*, pages 7–11, New York, NY, USA, 2014. ACM.

[98] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving Constraints Over Floating-Point Numbers. In *Principles And Practice Of Constraint Programming*, pages 524–538, Berlin, Heidelberg, 2001. Springer-Verlag.

[99] João P. Marques Silva and Karem A. Sakallah. GRASP – A New Search Algorithm For Satisfiability. In *Computer-aided Design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[100] Martin Davis, George Logemann, and Donald Loveland. A Machine Program For Theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[101] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[102] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers To Cryptographic Problems. In *SAT*, pages 244–257, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[103] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point Of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[104] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. [Online; accessed September-2018].

[105] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT And SAT Modulo Theories: From An Abstract Davis–Putnam–Logemann–Loveland Procedure To DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[106] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 4963 of *LNCS*, pages 337–340, 2008.

[107] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated Test Case Generation With SMT-solving And Abstract Interpretation. In *NASA Formal Methods*, pages 298–312, Berlin, Heidelberg, 2011. Springer-Verlag.

[108] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed Abstractions For Floating-point Arithmetic. In *2009 Formal Methods In Computer-Aided Design*, pages 69–76, 2009.

[109] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 7795 of *LNCS*, pages 93–107, 2013.

[110] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding Floating-point Logic With Abstract Conflict Driven Clause Learning. *Formal Methods in System Design*, 45(2):213–245, 2014.

[111] Aleksandar Zeljić, Christoph M. Wintersteiger, and Philipp Rümmer. An Approximation Framework For Solvers And Decision Procedures. *Journal of Automated Reasoning*, 58(1):127–147, 2017.

[112] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer-Aided Verification*, pages 154–169, 2000.

[113] Miriam Leeser, Saoni Mukherjee, Jaideep Ramachandran, and Thomas Wahl. Make It Real: Effective Floating-point Reasoning Via Exact Arithmetic. In *Design, Automation And Test In Europe Conference And Exhibition*, pages 117:1–117:4, 3001 Leuven, Belgium, 2014. European Design and Automation Association.

[114] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. GoSAT: Floating-point Satisfiability As Global Optimization. In *Formal Methods In Computer-Aided Design*, pages 11–14, Austin, TX, 2017. FMCAD Inc.

[115] Lucas C. Cordeiro and Bernd Fischer. Verifying Multi-threaded Software Using SMT-based Context-bounded Model Checking. In *Software Engineering*, pages 331–340, 2011.

[116] Jeremy Morse, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. Handling Unbounded Loops With ESBMC 1.20 (Competition Contribution. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 7795 of *LNCS*. Springer, 2013.

[117] Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 8413 of *LNCS*, pages 405–407. Springer Berlin Heidelberg, 2014.

[118] Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Automated Software Engineering*, pages 888–891. ACM, 2018.

[119] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver For Bit-Vectors And Arrays. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 5505 of *LNCS*, pages 174–177, 2009.

[120] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver, August 2006.

[121] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. An Automatable Formal Semantics For IEEE-754 Floating-Point Arithmetic. In *Symposium On Computer Arithmetic*, pages 160–167, 2015.

[122] SoSy-Lab. SV-Comp 2018. https://sv-comp.sosy-lab.org/2018/, 2018. [Online; accessed September-2018].

[123] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer-Aided Verification*, volume 6806 of *LNCS*, pages 171–177, 2011.

[124] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[125] John Levine. *Flex & Bison*. O'Reilly Media, Inc., 1st edition, 2009.

[126] Benjamin C. Pierce. *Types And Programming Languages*. The MIT Press, 1st edition, 2002.

[127] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey Of Symbolic Execution Techniques. *ACM Computing Survey*, 51(3):50:1–50:39, 2018.

[128] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool For Configurable Software Verification. In *Computer-Aided Verification*, volume 6806 of *LNCS*, pages 184–190, 2011.

[129] Andreas Ibing. SMT-Constrained Symbolic Execution For Eclipse CDT/Codan. In *Software Engineering And Formal Methods*, pages 113–124, 2014.

[130] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. SMACK Software Verification Toolchain. In *Software Engineering*, pages 589–592, 2016.

[131] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework For Lifelong Program Analysis & Transformation. In *Symposium On Code Generation And Optimization*, pages 75–96, 2004.

[132] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier For Object-oriented Programs. In *Symposium On Formal Methods For Components And Objects*, pages 364–387, 2006.

[133] Shigeru Chiba. A Metaobject Protocol For C++. In *Object-oriented Programming, Systems, Languages, And Applications*, pages 285–299, 1995.

[134] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model Checking Of C And C++ With DIVINE 4. In *Automated Technology For Verification And Analysis*, pages 201–207, 2017.

[135] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A Practical Tool For Checking Manipulation Of Dynamic Data Structures Using Separation Logic. In *Computer-Aided Verification*, pages 372–378, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[136] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna Verification Tool: IC3 For Parallel Software. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9636 of *LNCS*, pages 954–957, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[137] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *Computer-Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.

[138] Pablo Gonzalez-de Aledo and Pablo Sanchez. FramewORk For Embedded System Verification. In *Tools And Algorithms For The Construction And Analysis Of Systems*, pages 429–431, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[139] Kevin Atkinson, Matthew Flatt, and Gary Lindstrom. ABI Compatibility Through A Customizable Language. In *GPCE*, pages 147–156, New York, NY, USA, 2010. ACM.

[140] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Practical Verification Of Peephole Optimizations With Alive. *Communications of the ACM*, 61(2):84–91, 2018.

[141] Jeremy Morse. *Expressive And Efficient Bounded Model Checking Of Concurrent Software*. PhD thesis, University of Southampton, Southampton, UK, 2015.

[142] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krsti, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0. Technical report, 2010.

[143] Armin Biere. Lingeling, Plingeling, PicoSAT And PrecoSAT At SAT Race 2010. Technical report, Institute for Formal Models and Verification, 2010.

[144] Matthias Heizmann, Aina Niemetz, Giles Reger, and Tjark Weber. SMT-COMP 2018. http://smtcomp.sourceforge.net/2018/results-toc.shtml, 2018. [Online; accessed September-2018].

[145] David R. Cok, Alberto Griggio, Roberto Bruttomesso, and Morgan Deters. The 2012 SMT Competition. http://smtcomp.sourceforge.net/2012/reports/SMTCOMP2012.pdf, 2012. [Online; accessed September-2018].

[146] Stefan Kupferschmid and Bernd Becker. Craig Interpolation In The Presence Of Non-linear Constraints. In *Formal Modeling And Analysis Of Timed Systems*, pages 240–255, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[147] Trevis Rothwell and James Youngman. The GNU C Reference Manual. Technical report, International Organization for Standardization, Geneva, CH, 2015.

[148] Daniel Kroening. CProver Manual. http://www.cprover.org/cprover-manual/, 2018. [Online; accessed September-2018].

[149] Attributes In Clang. https://clang.llvm.org/docs/AttributeReference.html, 2018. [Online; accessed September-2018].

[150] Hussama Ismail, Iury Bessa, Lucas C. Cordeiro, Eddie Batista de Lima Filho, and Joao Edgar Chaves Filho. DSVerifier: A Bounded Model Checking Tool For Digital Systems. In *Symposium On Model Checking Software*, volume 9232 of *LNCS*, pages 126–131, 2015.

[151] Renato B. Abreu, Mikhail R. Gadelha, Lucas C. Cordeiro, Eddie Batista de Lima Filho, and Waldir Sabino da Silva Jr. Bounded Model Checking For Fixed-point Digital Filters. *Journal of the Brazilian Computer Society*, 22(1):1:1–1:20, 2016.

[152] Iury Bessa, Hussama Ismail, Lucas C. Cordeiro, and João Edgar Chaves Filho. Verification Of Fixed-point Digital Controllers Using Direct And Delta Forms Realizations. *Design Automation for Embedded Systems*, 20(2):95–126, 2016.

[153] Iury Bessa, Hussama Ismail, Reinaldo Palhares, Lucas C. Cordeiro, and Joao Edgar Chaves Filho. Formal Non-Fragile Stability Verification Of Digital Control Systems With Uncertainty. *IEEE Transactions on Computers*, 66(3):545–552, 2017.

[154] Lennon Chaves, Iury Bessa, Lucas C. Cordeiro, Daniel Kroening, and Eddie Batista de Lima Filho. Verifying Digital Systems With MATLAB. In *Symposium On Software Testing And Analysis*, pages 388–391, 2017.

[155] Philipp Rümmer and Thomas Wahl. An SMT-LIB Theory Of Binary Floating-Point Arithmetic. In *SMT Workshop*, 2010.

[156] Richard Smith. *Working Draft, Standard For Programming Language C++*, 2016. [Online; accessed September-2018].

[157] Levent Erkk. Bug In Floating-point Conversions. https://github.com/Z3Prover/z3/issues/1564, 2018. [Online; accessed September-2018].

[158] Andres Noetzli. Failing Precondition When Multiplying 4-bit Significand/4-bit Exponent Floats. https://github.com/CVC4/CVC4/issues/2182, 2018. [Online; accessed September-2018].

[159] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded Model Checking Of C And C++ Programs Using A Compiler IR. In *Verified Software: Theories, Tools, And Experiments*, volume 7152 of *LNCS*, pages 146–161, 2012.

[160] Franco Ivančić, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model Checking C Programs Using F-SOFT. *Computer Design*, pages 297–308, 2005.

[161] John N. Hooker. Solving The Incremental Satisfiability Problem. *The Journal of Logic Programming*, 15(1):177–186, 1993.

[162] Niklas Eén and Niklas Sörensson. Temporal Induction By Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[163] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Design Automation Conference*, pages 542–545, 2001.

[164] Henning Günther and Georg Weissenbacher. Incremental Bounded Software Model Checking. In *Symposium On Model Checking Software*, pages 40–47, 2014.

[165] Alastair Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using *k*-Induction. In *Static Analysis Symposium*, pages 351–368, 2011.

[166] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety Verification And Refutation By *k*-Invariants And *k*-Induction. In *Static Analysis*, pages 145–161, 2015.

[167] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[168] Karim Yaghmour, Jonathan Masters, and Gilad Ben. *Building Embedded Linux Systems, 2Nd Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 2008.

[169] University of Southampton. The Iridis Compute Cluster. https://www.southampton.ac.uk/isolutions/staff/iridis.page, 2018. [Online; accessed September-2018].

[170] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. JBMC: A Bounded Model Checking Tool For Verifying Java Bytecode. In *Computer-Aided Verification*, volume 10981 of *LNCS*, pages 183–190. Springer, 2018.

[171] Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jia-Guang Sun. C Code Verification Based On The Extended Labeled Transition System Model. In *Model Driven Engineering Languages And Systems*, pages 48–55, 2016.

[172] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting *k*-Induction With Continuously-Refined Invariants. In *Computer-Aided Verification*, volume 9206 of *LNCS*, pages 622–640, 2015.

[173] Alastair Donaldson, Daniel Kroening, and Philipp Rümmer. SCRATCH: A Tool For Automatic Analysis Of DMA Races. In *Symposium On Principles And Practice Of Parallel Programming*, pages 311–312, 2011.

[174] Daniel Große, Hoang Le, and Rolf Drechsler. Induction-Based Formal Verification Of SystemC TLM Designs. In *Workshop On Microprocessor Test And Verification*, pages 101–106, 2009.

[175] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction And A SAT-Solver. In *Formal Methods In Computer-Aided Design*, pages 108–125, 2000.

[176] Julio Cano, Gwenaël Delaval, and Eric Rutten. Coordination Of ECA Rules By Verification And Control. In *Coordination Models And Languages*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[177] Thomas Wahl. The *k*-induction Principle. http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf, 2013. [Online; accessed September-2018].

[178] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2nd edition, 2003.

[179] Williame Rocha, Herbert Rocha, Hussama Ismail, Lucas C. Cordeiro, and Bernd Fischer. DepthK: A *k*-Induction Verifier Based On Invariant Inference For C Programs - (Competition Contribution). In *Tools And Algorithms For The Construction And Analysis Of Systems*, pages 360–364, 2017.

[180] Viktor Malík, Štefan Martiček, Peter Schrammel, Mandayam Srivas, Tomáš Vojnar, and Johanan Wahlang. 2LS: Memory Safety And Non-termination. In *Tools And Algorithms For The Construction And Analysis Of Systems*, pages 417–421, Cham, 2018. Springer International Publishing.

[181] Nathan R. Sturtevant and Ariel Felner. A Brief History And Recent Achievements In Bidirectional Search. In *Conference On Artificial Inteligence*. AAAI Press, 2018.

[182] Viktor Malik, Martin Hruska, Peter Schrammel, and Tomas Vojnar. Template-Based Verification Of Heap-Manipulating Programs. In *Formal Methods In Computer-Aided Design*, pages 103–111, 2018.

[183] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. An User Configurable Clang Static Analyzer Taint Checker. In *Conference Of The Chilean Computer Science Society*, pages 1–12, 2016.

[184] Diego Novillo. GCC- An Architectural Overview, Current Status And Future. In *The Linux Symposium*, 2006.

[185] Michael Larabel. OpenBSD Switches To Clang Compiler For I386/AMD64. https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Default-Clang, 2017. [Online; accessed September-2018].

[186] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. SMT-based False Positive Elimination In Static Program Analysis. In *Formal Engineering Methods*, pages 316–331, Berlin, Heidelberg, 2012. Springer-Verlag.

[187] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis Via Graph Reachability. In *Symposium On Principles Of Programming Languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[188] Gábor Horváth, Péter Szécsi, Zoltán Gera, Dániel Krupp, and Norbert Pataki. Implementation And Evaluation Of Cross Translation Unit Symbolic Execution For C Family Languages. In *Software Engineering*, pages 428–428, New York, NY, USA, 2018. ACM.

[189] Artem Dergachev. Clang Static Analyzer: A Checker Developer's Guide. https://github.com/haoNoQ/clang-analyzer-guide, 2018. [Online; accessed September-2018].

[190] Available Checkers. https://clang-analyzer.llvm.org/available_checks.html, 2018. [Online; accessed September-2018].

[191] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast With Software Verification. In *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[192] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A Local Shape Analysis Based On Separation Logic. In *Tools And Algorithms For The Construction And Analysis Of Systems*, pages 287–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[193] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis By Means Of Bi-Abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.

[194] Javed Absar. Scalar Evolution - Demystified. In *European LLVM Developers Meeting*, 2018.

Appendices

Appendix A

Support for the FP logic

SMT FP operations	Z3 v4.7.1	MathSAT v5.5.1	CVC4 v1.6-prerelease	ESBMC FP API
Create floating point sort	√	√	√	√
Create rounding mode sort	√	√	√	√
Create floating point literal	√	√	√	√
Create plus and minus infinity	√	√	√	√
Create plus and minus zeroes	√	√	√	√
Crete NaN	√	√	√	√
Absolute value operator	√	√	√	√
Negation operator	√	√	√	√
Addition operator	√	√	√	√
Subtraction operator	√	√	√	√
Multiplication operator	√	√	√	√
Division operator	√	√	√	√
Fused multiply-add operator	√	×[1]	√	√
Square root operator	√	√	√	√
Remainder operator	√	×	√	×
Rounding to Integral operator	√	√	√	√
Minimum operator	√	√	√	√
Maximum operator	√	√	√	√
Less than or equal to operator	√	√	√	√
Less than operator	√	√	√	√
Greater than or equal to operator	√	√	√	√
Greater than operator	√	√	√	√
Equality operator	√	√	√	√
IsNormal check	√	√	√	√

[1] In ESBMC, a remainder operation fallback to the FP API.

SMT FP operations	Z3 v4.7.1	MathSAT v5.5.1	CVC4 v1.6-prerelease	ESBMC FP API
IsSubnormal check	√	√	√	√
IsZero check	√	√	√	√
IsInfinite check	√	√	√	√
IsNaN check	√	√	√	√
IsNegative check	√	√	√	√
IsPositive check	√	√	√	√
Convert to FP from real	√	√	×	×
Convert to FP from signed BV	√	√	×	√
Convert to FP from unsigned BV	√	√	×	√
Convert to FP from another FP	√	√	×	√
Convert to unsigned BV from FP	√	√	×	√
Convert to signed BV from FP	√	√	×	√
Convert to real from FP	√	√	×	×
Convert to IEEE BV from FP[2]	√	√	√	√
Convert to FP from IEEE BV[2]	√	√	√	√

Table A.1: Support in each SMT solver and in the ESBMC FP API for the operations described in the SMT FP logic. A √ indicates a supported feature while × indicates an unsupported feature.

[2]Not part of the SMT FP logic.

Appendix B

Verification Time in the ReachSafety-Float Category

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
addsub_double_exact.c	0	0	0	1	0	0	0
addsub_float_exact.c	0	0	0	0	0	0	0
addsub_float_inexact.c	1	0	0	0	0	0	0
arctan_Pade.c	345	895	312	149	370	895	895
bary_diverge.c	895	895	895	895	895	895	895
cast_float_ptr.c*	0	0	0	0	0	0	0
cast_float_union.c	0	2	0	0	0	1	0
cast_union_loose.c*	0	1	0	0	0	0	1
cast_union_tight.c*	1	2	1	1	1	1	1
ceil_nondet.i	1	553	1	0	0	1	3
ceil.i	0	0	0	0	0	0	0
copysign.i	1	0	0	0	1	1	0
cos_polynomial.c	154	895	63	18	32	895	895
digits_bad_for.i*	1	1	1	0	1	1	1

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
digits_bad_-while.i*	1	0	1	1	1	1	1
digits_for.i	1	0	1	0	1	1	1
digits_while.i	1	1	1	1	1	0	1
divmul_buf_-diverge.c	895	895	896	895	896	895	895
divmul_di-verge.c	896	896	895	895	873	895	895
Double_div_-bad.i*	896	896	895	895	895	896	895
Double_div.i	316	317	319	318	314	314	317
drift_tenth.c	0	0	0	0	0	1	0
exp_loop.c	20	896	102	40	7	469	366
fabs.i	0	0	0	0	0	0	0
fdim.i	0	0	0	1	0	0	0
feedback_di-verge.c	895	896	895	895	896	895	895
filter1.c	895	895	895	895	895	895	895
filter2_alt.c	896	895	895	895	895	895	895
filter2_iter-ated.c	895	895	895	895	895	895	895
filter2_reinit.c	895	895	895	895	895	895	895
filter2_set.c	895	895	895	895	895	895	895
filter2.c	895	895	895	895	895	895	895
filter_iir.c	895	895	895	895	895	895	895
float-div1.i	4	895	4	1	0	18	12
float-flags-simp1.i	0	0	0	1	0	0	1
float-no-simp1.i	0	0	0	0	0	0	1
float-no-simp2.i	3	895	4	3	6	90	69
float-no-simp3.i	0	0	0	0	0	0	0
float-no-simp4.i	1	0	0	0	0	0	1
float-no-simp6.i	1	0	0	0	1	1	0

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
float-no-simp7.i	0	0	0	0	0	0	0
float-no-simp8.i	0	0	0	0	0	0	0
float-rounding1.i	0	1	0	0	0	0	0
float-to-double1.i	0	895	1	1	0	8	8
float-to-double2.i	0	0	0	0	0	0	0
float-zero-sum1.i	0	0	0	0	0	0	0
float11.i	0	0	0	0	0	0	0
float12.i	0	2	0	0	0	1	2
float13.i	0	0	0	0	0	0	0
float14.i	0	0	0	0	0	1	1
float18.i	0	0	0	0	0	1	0
float19.i	0	1	0	0	0	0	1
float1.i	0	0	0	0	0	0	0
float20.i	0	1	1	1	0	8	9
float21.i	0	895	0	1	0	10	14
float22.i	0	1	0	1	0	0	0
float2.i	0	0	0	0	0	0	0
float3.i	0	1	0	1	0	1	1
float4.i	2	895	4	9	8	106	77
float5.i	1	1	1	0	0	2	4
float6.i	1	0	0	0	0	1	0
float8.i	0	0	0	1	0	0	0
Float_div_-bad.i*	3	4	4	3	3	4	4
Float_div.i	1	1	1	1	1	1	1
float_double.c	0	0	0	0	0	0	0
float_int_inv_-square.c*	0	6	0	0	0	2	2
float_lib1.i	1	1	1	1	1	1	1
float_lib2.i	0	0	0	0	0	0	0
floor_nondet.i	1	865	1	0	0	2	2
floor.i	0	0	1	0	0	0	0

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
fmax.i	1	0	0	0	0	0	0
fmin.i	0	0	0	0	0	0	0
fmod2.i	0	0	0	0	0	0	0
fmod3.i	0	0	0	0	0	0	0
fmod.i	0	0	0	0	1	1	0
image_filter.c	718	895	895	879	895	895	895
interpolation2.c	12	895	118	6	5	423	543
interpolation.c	6	895	27	3	3	107	93
inv_Newton.c*	895	895	895	895	895	895	895
inv_sqrt_-Quake.c	1	895	10	1	0	81	114
inv_square.c*	1	1	0	0	0	0	0
inv_square_-int.c	0	17	0	1	0	2	3
inv_square.c	0	895	0	1	0	1	0
isgreaterequal.i	0	0	0	1	0	0	0
isgreater.i	0	0	0	0	0	0	0
islessequal.i	0	1	1	0	1	0	0
islessgreater.i	0	0	0	0	0	0	0
isless.i	0	0	0	0	0	0	0
isunordered.i	0	0	0	0	0	0	0
loop.c	895	895	895	895	895	895	895
lrint.i	0	0	0	0	0	0	0
mea8000.c	895	895	896	896	896	895	895
modf.i	0	0	0	0	0	0	0
Muller_Ka-han.c	1	2	2	2	2	2	2
nan_double.c*	0	0	0	0	0	0	0
nan_double_-range.c	0	0	0	1	1	0	0
nan_float.c*	0	0	0	0	0	0	0
nan_float_-range.c	0	0	0	0	1	0	0
nan.i	0	0	0	0	0	0	0
nearbyint2.i	0	0	0	0	0	1	0
nearbyint.i	0	0	2	0	0	0	1
newton_1_1.i	49	895	110	24	871	895	895
newton_1_2.i	52	895	148	30	430	895	895

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
newton_1_3.i	60	895	341	35	895	895	895
newton_1_4.i*	35	895	31	18	5	895	895
newton_1_5.i*	46	895	51	11	7	895	895
newton_1_6.i*	58	895	17	4	1	895	895
newton_1_7.i*	13	895	11	6	1	895	895
newton_1_8.i*	54	895	25	10	3	895	895
newton_2_1.i	202	895	863	87	811	895	895
newton_2_2.i	214	895	895	108	895	895	895
newton_2_3.i	200	895	834	153	895	895	895
newton_2_4.i	201	895	811	127	895	895	895
newton_2_5.i	228	895	895	264	895	895	895
newton_2_6.i*	169	895	51	16	6	895	895
newton_2_7.i*	154	895	34	9	54	895	895
newton_2_8.i*	82	895	47	12	7	895	895
newton_3_1.i	575	895	895	313	895	896	895
newton_3_2.i	847	895	895	266	895	895	895
newton_3_3.i	576	895	895	469	895	895	895
newton_3_4.i	721	895	895	294	895	895	895
newton_3_5.i	693	895	895	895	895	895	895
newton_3_6.i*	139	896	137	46	9	895	895
newton_3_7.i*	161	895	58	20	28	895	895
newton_3_8.i*	178	895	74	38	15	895	895
remainder.i	0	0	0	0	0	0	1
rint2.i	1	0	0	0	0	0	0
rint.i	1	1	1	0	1	0	0
rlim_exit.c	895	895	895	895	728	895	895
rlim_invariant.c	895	895	895	895	895	895	895
rounding_functions.i	0	0	1	0	1	0	0
round_nondet.i	1	895	1	1	0	9	4
round.i	0	1	0	0	0	1	0
Rump_double.c	0	0	0	0	1	0	0
Rump_float.c	1	0	0	0	0	0	0
sine_1.i*	27	895	10	3	1	175	168
sine_2.i*	27	895	36	2	0	64	111

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
sine_3.i*	26	895	11	5	0	273	251
sine_4.i	45	895	145	191	895	895	895
sine_5.i	23	895	36	11	895	895	895
sine_6.i	28	895	35	8	628	895	895
sine_7.i	19	895	24	9	895	895	895
sine_8.i	24	895	43	7	16	895	895
sin_interpolated_bigrange_loose.c	895	895	895	895	895	895	895
sin_interpolated_bigrange_tight.c	895	895	895	895	895	895	895
sin_interpolated_index.c*	8	895	15	6	5	509	373
sin_interpolated_index.c	312	895	895	895	895	895	895
sin_interpolated_negation.c	895	895	896	895	895	895	895
sin_interpolated_smallrange.c	895	895	895	895	895	895	895
sqrt_biNewton_pseudoconstant.c	895	895	895	895	895	895	895
sqrt_Householder_constant.c	1	2	1	1	2	2	2
sqrt_Householder_interval.c	895	895	895	895	895	895	895
sqrt_Householder_pseudoconstant.c	895	895	895	895	895	895	895
sqrt_Newton_pseudoconstant.c	896	895	895	895	895	895	895
sqrt_poly2.c*	79	895	137	5	1	260	340

Testname	Boolector (fp2bv)	CVC4 (fp2bv)	MathSAT (fp2bv)	MathSAT	Yices (fp2bv)	Z3	Z3 (fp2bv)
sqrt_poly.c	22	895	25	7	5	895	895
square_1.i*	6	895	18	3	2	477	430
square_2.i*	9	895	29	6	4	166	656
square_3.i*	11	895	39	3	4	887	895
square_4.i	37	895	200	48	769	895	895
square_5.i	19	895	106	137	570	895	895
square_6.i	16	895	214	18	333	895	895
square_7.i	10	895	49	8	134	895	895
square_8.i	7	895	19	3	3	596	526
trunc_non-det_2.i	4	895	12	1	161	176	193
trunc_nondet.i	1	1	1	1	0	1	1
trunc.i	0	0	0	0	0	1	0
water_pid.c	7	7	7	7	7	6	7
zonotope_2.c	895	895	895	895	896	895	895
zonotope_3.c	895	896	895	895	896	895	895
zonotope_-loose.c	4	895	4	1	0	35	17
zonotope_-tight.c	4	895	4	1	1	34	17
Total (s)	32258	80561	38074	30259	43066	63516	63826
Total (correct results) (s)	8089	1795	5851	4303	7261	5339	4756

Table B.1: Verification time, in seconds, of each program in the ReachSafety category, for Z3 and MathSAT using their native floating-point API and all supported solvers using our floating-point API. A * after the name of the program indicates that it is expected to fail.