TEMPORAL VERIFICATION OF NONLINEAR PROGRAMS

by

Yuandong Cyrus Liu

A DISSERTATION

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

_____

Yuandong Cyrus Liu, Candidate

ADVISORY COMMITTEE

Eric Koskinen, Chairman	Date
David Naumann	Date
Jun Xu	Date
Daniel Dietsch	Date
Hang Liu	Date

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2022

TEMPORAL VERIFICATION OF NONLINEAR PROGRAMS

ABSTRACT

Practical software systems often have nonlinear behaviors due to expressions or statements in their source code that involve bitvectors, polynomials, exponentials, etc. Verifying temporal properties of these systems is challenging, going beyond the power of many existing temporal verification tools, which are more focused on linear programs.

This dissertation aims to improve existing temporal verification techniques to support nonlinear programs. We introduce techniques to transform nonlinear program behaviors into a linear space, where existing temporal verification tools can then be exploited. For bitvector programs, we present a theory called bitwise branching that introduces new paths to over-approximate bitvector operations with linear constraints in common cases, falling back on the original bitvector expression otherwise. With these new paths, often the fallback bitvector path is infeasible and we show that we can better exploit the well-developed integer reasoning and interpolation of verification tools. We further develop a novel binary verification toolchain to verify temporal properties of decompiled programs by retargeting the decompilation process and employing our bitwise branching.

We next look beyond bitvectors to other types of nonlinear programs like polynomials, and explore an approach that combines dynamic analysis with static temporal verification. To that end, we dynamically infer candidate invariants at program locations involving polynomial behaviors, and use static analysis to validate those candidates. These candidates may not hold, so we then describe a refinement algorithm to iteratively refine a Boolean combination of linear expressions that captures the original polynomial behavior. We show that nonlinear expressions in programs can then be replaced with our synthesized linear expressions, and it enables temporal verification tools to be applied to this nonlinear

domain.

Author: Yuandong Cyrus Liu

Advisor: Eric Koskinen

Date: December 15, 2022

Department: Computer Science

Degree: Doctor of Philosophy

To All Scientists.

Acknowledgments

This dissertation would not have reached completion without the help and support from the people I have been connected with. Firstly, I would like to thank and express my sincere gratitude to my advisor Prof. Eric Koskinen for his support during my PhD study, for his patience, motivation, and immense knowledge. He came up with the interesting subject of study and has taught me to conduct research, write clearly, prepare interesting presentations, and think both about the big picture and minute details. He also never hesitated to share his knowledge regarding both scientific topics and a multitude of soft skills to help and support me. He encourages me to step out of my comfort zone and pushes me forward. His guidance helped me in all the time of research and writing of this dissertation. I could not have imagined a better advisor and mentor for my PhD study, I am forever grateful! Besides my advisor, I would like to thank the rest of my committee members: Prof. David Naumann, Prof. Jun Xu, Prof. Daniel Dietsch, and Prof. Hang Liu for their insightful comments and encouragement.

I also had the opportunity to meet amazing scholars, researchers, and fellow students. I am grateful to Chanh for always being supportive and kind not only towards me, but towards all other people, he has guided me through technical details and helped me implement our theories and methodologies. To Chengbin for helping me implement binary analysis and he is always patient with my questions. To Daniel who helped me with my experiments and showed me a rigorous research attitude but he can always convey them with humor. To Timos and Vu for being generous with their time and insights to our research problems, technical details, and paper writing, for many times both of them stayed up late with me before our deadlines.

During my studies, I have been lucky to share an office with extraordinary col-

leagues and take courses from our faculty members. Thanks to my lab peers Ramana, Parisa, Mihai, and Ayomide for helping me practice before my defense talk, thanks to Michael and the whole Cypress group for the weekly seminar, giving me chance to present, practice and have valuable feedback on all aspects, which helped me improve a lot. Also thanks to Eduardo, Sandeep, Dave, and Georgios, it's always been a pleasure taking their courses. And thanks to our best graduate academic advisor Jannine, who has always been helpful, for countless times I knocked on her door and she is always there for help, greeting me with her warm smile, thank you for that!

I thank my family and my friends for their infinite support throughout my PhD study. To my sister and my parents whom I'm always happy to call on weekends, update them on my progress and my personal life. To my good friend Evaristo for the great discussions about culture and research in our own fields. We workout, running, and travel together, imagining each other's future life, and sharing the pressure from our research, I can always seek positive energy from him. To Adrien who is always considerable and kind. To Aida who can always gather everyone for interesting festivals and celebrations. To Estelle who stayed with me and encouraged me during my early PhD years, she is always warm to me. I'm fortunate to meet a group of international friends Nina, Bogdan, Jezabel, Marc, Lara, Mojtaba, Sheri, Farbod, Darwin, Juanra, and Koss, we had great time during the weekends' gatherings, dinners, and lunches.

Thank you all for being part of this incredible journey of my life!

Yuandong Cyrus

Liu

November 2022

Contents

List of Tables

List of Figures

Chapter 1
Introduction

In general, a software verification task is, given a program model $\mathcal{P}$ and a property φ, showing that all behaviors of $\mathcal{P}$ satisfy φ. Verification challenges arise from different types of programs and types of properties to be verified. Program properties can be decomposed into safety properties and liveness properties. A safety property describes that nothing bad happens to the program, while a liveness property states that something good will eventually occur in the program. Both safety and liveness properties can be formally characterized [13]. For liveness properties like termination, non-termination, and Linear Temporal Logic (LTL), existing tools [159, 16, 72, 110, 86, 32, 27] are effective in proving programs with linear arithmetic assignments and loop guards, but suffer with nonlinear programs that have, for example bitvector and polynomials expressions. In this dissertation, we tackle verification challenges in nonlinear programs (e.g. bitvector, polynomial.) and temporal properties. In this chapter, we first introduce background about static and dynamic analysis techniques in program verification, and discuss various types of nonlinear programs in Section 1.1, we address research challenges of interest in Section 1.2, and related work in Section 1.4. Common foundations are introduced in Chapter 2, and we dedicate the remaining chapters to the details of our approaches that solve the problems being addressed.

1.1 Background

There are many domains like medical/financial systems and automatic controlling systems that require high assurance of safety and reliability. Testing is an effective way to find program bugs before it's released. However, traditional program testing (requiring human

developers to design testing cases and run the program on them) is error-prone and the coverage of testing cases is limited. Automatic verification techniques that use static program analysis to determine program properties, provide a precise and effective way to verify program properties and expose program vulnerabilities. Static program analysis in general relies on a wide range of formal methods like type systems, abstract interpretation, satisfiability modulo theories (SMT), model checking, etc. Some of these methods have started to scale to solve problems of interesting size, like system-level programs, and they can also verify properties of programs written in various languages like `C` and `Java`. SMT solvers serve as a foundation for these automatic verification tools, and have increased theories[1] to support for different types of programs. Recent research work [19, 133, 104, 123, 150] also have improved the efficiency of SMT solvers, for example, the Z3 solver can work with millions of free variables and still find solutions in seconds.

Software in many situations uses bitwise operations heavily, such as system level software which needs to interact with device drivers and check the state of certain drivers using bitwise operations. Bitwise operations are also more efficient in computing and therefore used for code optimization. Another pervasive domain of bitvector programs is binaries. Binary code executes directly on computers, and compilers that transform and optimize source code to machine code could introduce unintended behaviors to the program. Verifying binary code can find bugs, in the running systems, that are impossible to be exposed in source code level verification. Verifying binary code directly is challenging, as high level structure and type information from source program get lost. One common way to tackle this information lost is to de-compile the binary into a high level representation like `C`, a de-compiled program that is heavy in bitwise operations. Although there are active research and effective tools in binary de-compilation, not much work has been done with formal verification of binaries, in recent years, there has been an increasing interest in

[1] http://smtlib.cs.uiowa.edu/theories.shtml

binary analysis combined with formal methods. Existing verification tools face hurdles in verifying de-compiled code, due to the heavy use of bitwise operations in low level code, in which besides code optimization, another common case to use bitwise operations is to simulate the instruction jumping state.

Non-linear arithmetic (polynomials, square root, logarithm, etc.) is common in scientific computing software, reactive systems and practical distributed systems that often have infinite temporal behaviors. For example, a real-time server protocol responding to clients, or a distributed ledger performing transactions correctly on time with desired conditions etc., verifying temporal properties against them is critical. In summary, in this dissertation, we focus on the temporal verification of non-linear programs that have bitwise operations and higher-order of polynomial operations.

1.2 Challenges

Despite the importance of verifying temporal properties in practical systems discussed in the previous section, temporal verification tools are very much restricted to linear integer programs, temporal logics like LTL express properties that necessarily involve a notion of time, however it can easily face state explosion problem. For bitvector programs, verification tools [152, 159] rely on SMT solvers with bit-vector theory support. Yet fix-sized bitvector theory in practice, using bit-blasting technique to encode integers, introduces fresh variables for every single bit of that integer, which can lead to an exponential growth of bit size [105]. Finding valid program invariants is crucial for both safety and liveness properties like termination and LTL, yet verification tools are often unable to prove invariants for loops over the bitvector domain (requires more complicated invariants), as well as non-linear arithmetic, and they abort the analysis once facing the nonlinear program. LTL verification of bitvector program is still somewhat unexplored, as seen by the lack of LTL

verification tasks listed in SV-COMP [8] prior to our work. For other types of verification tasks (e.g. reachability, termination), there are limited bitvector operations, often irrelevant to verification. Therefore, for a better evaluation of verification tools in liveness verification tasks over non-linear programs, a richer set of benchmarks are required.

Bitvectors are also heavily used in decompiled binaries, besides difficulties in tackling bitwise operations, there are well-known open questions in binary decompilation ranging from instruction recovery, function boundary discovery in disassembly (mapping from machine code to assembly code), to precise control flow graph reconstruction in code lifting (mapping from assembly code to intermediate representation). Although existing binary lifting tools (like McSema [63]) and binary analysis frameworks [37] have achieved good performance and accuracy, in both binary disassembly and lifting, their goal of lifted code is targeted for recompilation instead of verification. Software verification tools are unable to handle the de-compiled/lifted code for various reasons like metadata and register information inherited from machine code. There is no complete tool having capabilities in binary disassembly, and lift assembly into decompiled code for LTL verification.

Programs that have a higher order of polynomial operations are another class of nonlinear programs that are out of reach by existing temporal verification tools. Static analysis also struggles on finding nonlinear invariants. Dynamic analysis supports more expressive invariants and scales well to large and complex programs [137]. It can learn about program behaviors from concrete data sets, free of reasoning about abstract expressions from program source code. Dynamic analysis is effective finding convex shape of polynomial expressions in Integer domain. Unfortunately, dynamic analysis focus on exploring a few program executions therefore it is only correct with respect to the explored paths, while static analysis is exploring all possible program paths but is limited to certain kinds of target programs and simple invariants. One solution we present is to combine them to leverage benefits of both analyses, however, more questions arise as to how we

can use dynamic analysis to learn invariants of non-linear programs precisely and hook back the learned invariants to original programs, preserving the program semantics, so that verification tools can return a sound result.

1.3 Solutions

While verification tools that work with static analysis facing challenges (discussed in 1.2) in temporal verification of bitvector programs, we introduce a new bitwise branching theory to approximate bitvector programs with linear constraints. Our bitwise branching theory introduces rewriting rules for bitvector programs. It helps transform bitvector programs with linear branches soundly, verification task can be performed so that reasoning bitwise operation is often unneeded. We show that bitwise branching enables LTL verification of bitvector programs, and we implement bitwise branching in software analysis framework ULTIMATE [159]. We incorporate this implementation as a sub-routine in our complete binary verification toolchain DARKSEA, which is able to decompile the binary into a C source program, and verify LTL properties against it.

In this dissertation, for temporal verification of polynomial programs, we present a hybrid program analysis that leverage primary advantages of both static and dynamic analysis for non-linear programs. We use dynamic analysis to infer candidate linear invariants at the location of nonlinear expression, then replace those expressions with inferred linear invariants, and then use static analysis to validate the inferred results. On top of this dual analysis, we design a refinement algorithm to refine the dynamic results with counterexamples from static validation. When refinement converges, the algorithm terminates with a precise set of linear expressions that cover the whole nonlinear behaviors in the integer domain.

1.4 Related Work

We have discussed that bitvector operations and nonlinear arithmetic are prevalent in practical systems, especially in decompiled programs and critical reactive systems. One of the biggest challenges in verification of bitvector programs is solving constraints in bitvectory theory that is used by SMT solvers. There are research works focused on effective bitvector solving in source code, as well as formally verified de-compilation in binary analysis (heavily involved with bitvecor operations). On the other side, using constraint solving with dynamic analysis for program testing is gaining more attention in security research, more research efforts have been put on introducing static analysis in dynamic program testing like symbolic execution, achieving better performance code coverage. In this section, we discuss related research work in bitvector reasoning, binary de-compilation with formal methods, verification of non-linear programs (including bitvector programs and nonlinear arithmetic programs), and dynamic analysis / hybrid analysis in automatic program verification.

1.4.1 Verification of Bitvector Programs

Static verification tools often rely on SMT solvers as backend, recently with the increasing support of fix-sized bitvector theory in SMT solvers, there are active works in improving the efficiency of bitvector reasoning, with many automatic verification tools have expanded the capabilities to termination verification of bitvector programs, but still, LTL verification over bitvector and nonlinear programs are missing.

Bitvector reasoning. Outside of the context of verifying lifted binaries, many works have investigated reasoning techniques for bitvector expressions and operations. Numerous works provide support for bitvector reasoning inside SMT solvers (*e.g.* [170]). Kroening

et al. [106] describe a method of predicate image over-approximation, the algorithm can be used to compute images of predicates using bit-vector logic, but with the sacrifice of some precision. He and Rakamarić [88] examine spurious counterexamples introduced by over-approximation of bitwise operations in SMT solving, the algorithm performs type analysis to transform both a counterexample and program. However, in the case of binary decompilation, type information are lost during the compilation, performing type analysis is challenging. Mattsen *et al.* [117] describe how to use a BDD-based abstract domain to improve reasoning about indirect jumps and integrated their work into Jakstab. Bryant *et al.* [35] describe an SAT-based decision procedure that iteratively constructs an abstraction of a bit vector formula.

Termination and LTL for bitvector programs. Other works have targeted reasoning about *termination* of bit-precise *source code* programs. Cook *et al.* [51] use Presburger arithmetic for representing bitvector programs and rank functions. Chen *et al.* [40, 41] employ lexicographic rank function synthesis for bit precision in an interprocedural termination analysis, through a mixture of under- and over-approximation. They rely on the bit-precision of an underlying SMT solver. David *et al.* [58] propose termination and non-termination analysis which is able to generate nonlinear, lexical ranking functions and non-linear recurrence sets for floating-point arithmetic, and proving termination of programs with pointer arithmetic is discussed in Ströder's work [155]. Earlier, Falke *et al.* [69] derive linear arithmetic approximations of bitvector operations by introducing rewriting rules that create a large disjunction of cases, while for temporal verification, this puts a large burden on the solver. By contrast, our bitwise-branching approach creates an if-then-else expression for each scenario, shifting the burden to more efficient automata reasoning instead. The approach of Falke *et al.* is implemented in the KITTEL tool with the only support for termination. Hensel *et al.* [95] extends their previous work on automatic termination of C

programs over mathematical integers to bitvectors, instead of using SMT solving in fixed-size bitvector theory, they represent the relation between bitvetors through corresponding relations over $\mathbb{Z}$, this approach is implemented in verification tool APROVE [76]. Urban *et al.* [161] present an approach to learn the bits of information from terminating executions and synthesize ranking functions from under-approximation of these terminating program states, it has been implemented in SEAHORN tool[2]. Similar to SEAHORN (a LLVM based verification framework), SMACK [143] translates LLVM compiler's intermediate representation (IR) into Boogie [23] intermediate verification languages with limited support for bitvector programs. However, not all the LLVM IR can be correctly translated into Boogie, especially when dealing with lifted binaries. Besides compiler intermediate representation, there are also works focus on the termination of virtual machines like `Java` bytecode [141, 12]. Spin [3] is a well-known open-source software verification tool, it provides a rich LTL libraries [65] potentially can be used for other temporal verification techniques of high level programs. Despite many related works on bitvector reasoning, no prior works have focused on verifying temporal properties of lifted binaries and bitvector programs.

1.4.2 Binary Analysis with Formal Methods

From binary code to high level de-compiled code, binary de-compilation, involves multiple steps, i.e. disassembly that recovers instructions from raw binary, function boundary discovery, control flow construction, and decompiling (lifting) that translates assembly code to a structured intermediate representation (IR), and finally, we can map IR to decomplied program. Recent research works have started to focus on formal decompilation, lifting validation/verification [167], precise control flow, and binary verification in general.

[2] `https://seahorn.github.io/`
[3] `https://spinroot.com/spin/whatispin.html`

Disassembly. Disassembly is the process of recovering instructions and structural information (*e.g.* functions and control flow) from the binary, providing the basis for all types of binary analysis. For verification, an essential property of disassembly is correctness and many recent works strive for this property. The Jakstab project [103, 102] focuses on accurate control flow reconstruction in the disassembly process. This work proposes an abstract domain called Bounded Address Tracking to resolve targets of control flow transfers, it is focused on correctness in control flow recovery instead of verification. Brumley *et al.* [34] described BAP, a tool and framework for static disassembly of stripped binaries. It models the semantics of the instructions and provides all-sided APIs to support various analyses, high-level verification analysis is still missing. The UCSB group released Angr [151]. Angr includes comprehensive symbolic execution and value-set analysis to facilitate the correctness of binary disassembly (in particular control flow reconstruction). Commercial IDA Pro [71] demonstrates higher accuracy than the above disassemblers [15] and partly it also leverages sound approaches such as value-set-analysis [20] to resolve the complex constructs that make disassembly challenging. Casinghino *et al.* [37] compared BAP with Angr [151] in terms of the ability to perform value-set analysis and call graph reconstruction.

Precise de-compilation. Raw disassembly output is highly architecture-specific, which is complex for verification. A widely adopted strategy to improve matters is to further *de-compile* to higher-level representations such as LLVM IR or C code. Hex-Rays Decompiler [4], Ghidra [136], and Snowman [59] endeavor to reconstruct the abstractions that were present in the original source code. Although these tools produce de-compiled results that may seem ready for verification, they often use aggressive but error-prone inferences. As a consequence, these de-compilers are rarely used for verification purposes. Contrary to the above de-compilers, many recent papers are focused on "low-level" aspects of the

binary and aim at precise de-compilation. Roessle *et al.* [147] described a new strategy for de-compiling x86-64 into a big step semantics. The authors present a formal semantics of x86-64, prove equivalence, and describe a method for automatically de-compiling x86-64 into the big-step semantics. Earlier, others proposed the idea of decompilation-into-logic (DiL) [120, 121, 122] that directly translates assembly code into logic, which can be implemented using interactive theorem provers such as HOL4 [4]. While DiL provides a rich environment for precise reasoning about fine-grained instruction-level details, it incurs high complexities for reasoning about more coarse-grained properties (reasoning bloated instruction level details is unnecessary in these cases.) such as reachability, termination, and temporal logic. In more recent work, Verbeek *et al.* [168] build on the formal semantics of Roessle *et al.* [147] and describe techniques and their tool FoxDec to decompile into re-compilable code, exploiting semantics for more soundness guarantees. In our works, we focus on coarse-grained properties such LTL and doing so automatically by employing automatic abstraction techniques. The close de-compilation technique is open source project McSema [63] which aims at coarse-grained goals such as re-compilation, it simulates the effects of instructions without aggressive inferences, this avoids introducing errors to de-compilation. McSema gained popularity in the reverse engineering community due to its re-compilation (disassembling and lifting binaries with the goal of compiling it back to binary again) capabilities, however, it prevents verification tools from handling the low-level complexities (e.g. inline-assembly) by abstracting over some details.

Verify decompiled programs. Other recent works focus on the correctness of the decompilation/lifting process itself. This direction requires a form of refinement reasoning that the lifted code simulates the binary, akin to translation validation and certified compilation. These works do not target verifying the behavior of the binary itself but are nonetheless im-

[4] `https://hol-theorem-prover.org/`

portant orthogonal research steps since our LTL verification of decompiled binaries relies on sound lifting. Dasgupta *et al.* [56] describe a translation validation on x86-64 instructions that employs their prior work on extensive formal semantics for x86-64 (Dasgupta *et al.* [57]). They observe that instruction-level validation can then be used to enable program-level translation validation and have applied their work to improve McSema (which part of our work relies on). Metere *et al.* [119] use HOL4 to verify a translation from ARMv8 to the BAP [34] IR. Hendrix *et al.* [93] discuss their ongoing work on verifying the translation performed by their lifting tool reopt on a block-by-block basis with some block- and function-level annotations. Other works (*e.g.* the Sail language [175, 17]) have also targeted formal semantics of instruction set architectures.

1.4.3 Dynamic Verification

Nonlinear polynomial relations arise in many safety-and security-critical applications [29], an alternative way to software verification of non-linear programs is dynamic verification, adapting traditional model checking to systematic testing. Dynamic analysis is effective to execute any complex programs but has partial correct results (it is correct with respect to execution paths). Dynamic model checking, also referred to as systematic testing, was first proposed for the concurrent programs [80], and more recently Godefroid [81] discussed the pros and cons of dynamic model checking. Previously, Groce *et al.* [82] extends model checking with dynamic analysis.

Dynamic inference of invariants. Program invariant is essential in proving safety and further liveness property, in the last decades, there are much work with static analyses inferring non-linear invariants [70, 146, 145, 84, 85, 83], but with restriction of program types like polynomial equalities or non-nested loops. With a dynamic approach, inferring more complex non-linear invariants have been studied. The well-known dynamic invariant tool

Daikon [67, 66] infers candidate invariants under various templates over concrete program states. The tool comes with a large set of templates that it tests against observed traces, removing those that fail and returning the remaining ones as candidate invariants. DIG [129] focuses on numerical invariants and therefore can compute more expressive (e.g., nonlinear) numerical relations than those supported by Daikon's templates. The pure dynamic analysis in DIG supports numerical invariants of the forms nonlinear equalities [128], octagonal inequalities [129], and max/min-plus inequalities [131]. Sharma *et al.* [149, 139] proposed a hybrid dynamic and static analysis to generate equality invariants, they use DIG to compute nonlinear equality invariants from concrete traces and verify the candidate invariants with SMT solving. The works from NumInv [126] and SymInfer [127] combined the dynamic analysis from DIG to infer nonlinear invariants with symbolic execution to remove spurious results. PIE [139] and ICE [75] also use a guess and check approach to infer invariants to prove a given specification. To prove a property, PIE iteratively infers and refined invariants by constructing necessary predicates to separate (good) states satisfying the property and (bad) states violating that property. ICE uses a decision learning algorithm to guess inductive invariants over predicates separating good and bad states. The checker produces good, bad, and "implication" counterexamples to help learn more precise invariants. For efficiency, they focus on octagonal predicates and only search for invariants that are boolean combinations of octagonal relations. In general, these techniques focus on invariants that are necessary to prove a given specification, and the quality of the invariants is dependent on the target specification. More recently, [171] described a method for inferring invariants through a form of neural networks. The technique uses a Continous Logic Network to learn SMT formulas directly from program traces. The authors show that this approach can learn more general nonlinear invariants (equalities, inequalities, and disjunction).

Liveness verification (Termination, LTL, CTL). Termination and LTL properties are two common liveness properties in software verification, today, numerous theories, techniques, and tools exist for proving termination and non-termination [142, 78, 52, 53, 54, 60], as well as static tools for computation tree logic (CTL) properties. Tools include Terminator [52], Ultimate Automizer [159], HIPTNT+ [110], FUNCTION [72], CPACHECKER [152], and APROVE [16]. There is a category on termination in the Software Verification Competition (SV-COMP) [26], although in this repository there are some simple property files that can be specified in LTL, a category for LTL verification task is still missing, and not mature, ULTIMATE is state-of-art verifier that supports LTL verification but over mathematical integers. Along the way, some have shown methods for *conditional* termination, whereby preconditions are found that specify the portion of traces that terminate [45, 110]. Like invariant is essential to safety verification, ranking functions, which demonstrate well-founded relations of the program, is critical to proving program termination. An active line of research has focused on flavors of ranking functions, including piece-wise [160], ordinals [162], size-change [112], and lexicographic [31]. Babić *et al.* [18] focused on proving termination of a restricted class of nonlinear loops, called NAW loops, which have special properties to allow their termination to be proved via analyzing the divergence of variables influencing the loop conditions. All these techniques work well in linear programs, yet when facing non-linear programs they are limited. On the one hand, several static techniques are able to infer polynomial *resource bounds* [99, 98, 97]. Nguyen *et al.* [125] describe runtime contracts for enforcing termination, using the size-change strategy for termination. The TiML functional language [169] allows a user to specify time complexity as types and then uses type checking to verify the specified complexity. The WISE tool [36] uses concolic execution to search for a path policy that leads to an execution path with high resource usage. On the other hand, a number of works have exploited dynamic information to infer termination reasoning. [135] showed that linear regression can

be used to infer bounds of program loops from systematic testing suites and these bounds imply termination. They then attempt to validate those bounds and use counterexamples to improve the precision of inference. There are some other approaches that attempt to reason program termination and non-termination at the same time [87, 109, 110]. Le *et al.* [107] proposed an algorithm that incorporates learning-based reasoning for recurrent sets (for non-termination) and rank functions (for termination) into an integrated algorithm that samples inputs and partitions them into terminating versus potentially non-terminating traces. It then performs learning on these partitions separately and combines counterexamples from one partition to inform the other. The algorithm is implemented in their tool DYNAMITE, which is capable of proving termination and non-termination of non-linear programs. Kroening *et al.* [140] introduced a novel automated program termination analysis by using neural networks to represent ranking functions, this method succeeds over the programs that use disjunctions in their loop conditions and programs that include nonlinear expressions. Going beyond, temporal verification is not discussed in these works.

There are also works [1, 22, 153, 46] focusing on temporal reasoning on distributed system, and works on temporal specification mining [100, 124, 114], however, these temporal properties are still limited, and program operations are subject to linear behaviors mostly. Recently, works [138, 79, 82, 55] on data driving model checking bringing improvement in efficiency, discuss benefits of static, dynamic and hybrid analysis. Inspired by these works, our dual rewriting for branching-time verification uses dynamic learning to mitigate the burden of static analysis, shifting NLA reasoning to linear constraints, helping solvers [158] find sound results for even more complicated cases. In general, there are many techniques for static verification of temporal properties [53, 50, 25, 48, 156, 163]. However, to our knowledge, there are no verification techniques with support for CTL programs' general non-linear expressions. Ultimate [159] has support for some non-linear expressions for proving reachability, termination, or linear temporal logic (LTL). LTL is a

trace-based logic and so traces can be taken one path at a time and, for each case, over-approximated. The dynamic invariant inference work such as [171, 132] can find NLA invariants, but does not analyze CTL properties.

1.5 Contributions and Dissertation Organization

Our work focuses on temporal verification of nonlinear programs. We now list individual items of the work presented in this dissertation, and highlight the individual contributions.

- We present a novel theory of source-level bitwise branching to verify bitvector programs by approximating operations with linear constraints. We implemented bitwise branching within Ultimate Automizer, and our implementation has been merged into the ULTIMATE [159] program analysis framework.

- We released new benchmark suites for reachability, termination, LTL verification of bitvector programs, which have been submitted to SVComp [8].

- We present extensive experimental results showing that bitwise branching enables reachability, termination, and LTL verification of bitvector programs, respectively, they are competitive with state-of-the-art SMT solvers, termination tools (*e.g.*AProVe, T2, FUNCTION), we also show that bitwise branching leads to an effective technique for verifying LTL of bitvector programs.

- With bitwise branching applied to the decompiled program, we develop a complete toolchain called DARKSEA for decompiling ("lifting") and verifying binaries. We implement a series of translations that re-target lifted binaries to verification rather than re-compilation, and we devise a new benchmark suite for LTL of binary executables. Experimental results showing that DARKSEA is effective in verifying LTL properties of binary executables, and it's the first tool to do so.

- While LTL specifies program properties over a single computation trace (implicitly describes all the computation paths), CTL describes properties over program paths including branching conditions, requiring more reasoning power to handle different branches at the same time. We present a novel dual rewriting algorithm for branching-time verification of nonlinear arithmetic programs. This approach combines both static and dynamic analysis to find the exact equivalent of nonlinear behavior with linear expressions (considering both positive and negative branches). We implement the algorithm in our tool DRNLA, which is able to transform nonlinear programs into linear programs with dynamic analysis, at the same time, providing soundness proof from static analysis. We evaluate DRNLA with nonlinear benchmarks from SVComp [8]. Experimental results show that DRNLA is effective in CTL verification of nonlinear programs.

Parts of this dissertation have been presented and published in the FMCAD student forum [115] (bitwise branching theory), APLAS [116] (proving LTL properties of btivector programs and DARKSEA), and submitted to PLDI 2023 conference (CTL verification for nonlinear programs with dynamic analysis).

Throughout this dissertation, we consider bitvector operations along with other nonlinear arithmetics as nonlinear behaviors in the program, we refer to them as nonlinear programs. The rest of the dissertation is organized in the following way: Chapter 2 introduces foundations and preliminaries for the research work. In Chapter 3, we present our work on proving linear temporal logic properties of bitvector programs, through a theory of bitwise branching, that approximates bitvector programs with linear constraints. Bitwise branching allows us to exploit the well-developed integer reasoning and interpolation of verification tools, and enables more programs to be verified, in this chapter, we also present all the experimental results regarding reachability, termination, and LTL verification of bitvector

programs.

In Chapter 4, we first present a practical case application of bitwise branching theory: LTL verification of de-compiled programs. We then summarize challenges in binary de-compilation and verification, and to tackle these challenges, we implement multiple passes of LLVM byte code translation in our binary verification toolchain DARKSEA. In the end, we evaluate DARKSEA with termination and LTL benchmarks that are compiled into binaries.

In Chapter 5, we present a dynamic approach to prove branching-time properties (CTL) of nonlinear programs. We show an approach that uses dynamic learning combined with static analysis to tackle other type of non-linear programs that have a higher order of polynomials. In high level, we introduce dual refinement for the linear approximation of nonlinear expressions, when the algorithm terminates, it entails the exact mapping from linear expressions to their polynomial counterpart. We execute the program and collect concrete traces at locations of interest, infer program candidate invariants from concrete traces, then use these linear invariants to approximate polynomial expressions. Then we discuss static validation for dynamic inference, for the counterexample discovered from static validation, we present an approach to dynamically generalize the counterexample, and sent the results back to dual refinement. We implement the algorithm in our dual analysis tool DRNLA, it enables verification tools to verify CTL properties of polynomial programs, we then evaluate our tool DRNLA with a wide range of nonlinear benchmarks and compare its results with the latest CTL verification tools. In the final Chapter 6, we summarize our research work that has been done and lay out future directions of the work.

Chapter 2
Foundations

In this chapter, we introduce concepts, terms, and definitions of static program verification and dynamic analysis used in this dissertation. We also list well-known verification and dynamic analysis tools. For theoretical fundamentals, we introduce the formalization of transition systems, program syntax/semantics, linear temporal logic (LTL), computation tree logic (CTL), and Büchi automaton.

2.1 Program Semantics and Transition Systems

Static analysis algorithms traverse target programs and, various program representations of those programs are proposed in the literature. In many static verification approaches, the program is transformed into an intermediate representation that is independent of domain-specific languages. In this section, we first set up various formal definitions for program semantics, program modeling, and Boogie programs [24] which are well-known verification languages for static analysis like reachability, termination, LTL, and CTL.

2.1.1 Boogie Program

The majority of our work is on `C` source programs, however, verification tools (introduced in later Section 2.4) parse languages into internal representations, Boogie programs, a widely used intermediate language for verification program. In our implementations, input source programs (or binaries de-compiled to C) that may have nonlinear operations are then translated into Boogie programs. Figure 2.1 shows the complete syntax for a boogie program P from ULTIMATE [159] tool.

In our work, we are focused on techniques for encoding reasoning about nonlinear

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Lit} \mid \mathit{Id} \\
& \mid & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid \mathit{UnOp}\ \mathit{Expr} \mid \texttt{(}\mathit{Expr}\texttt{)} \\
& \mid & \texttt{(if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr}\texttt{)} \\
& \mid & \texttt{old(}\mathit{Expr}\texttt{)} \\
& \mid & \mathit{Id}\ \texttt{(}(\mathit{Expr})^{*}\texttt{)} \\
& \mid & \mathit{Expr}\ \texttt{[}\mathit{Expr}(\texttt{,}\ \mathit{Expr})^{*}(\texttt{:=}\ \mathit{Expr})^{?}\texttt{]} \\
& \mid & \mathit{Expr}\ \texttt{[}\mathit{Lit}\ \texttt{:}\ \mathit{Lit}\texttt{]} \\
& \mid & \texttt{(}\mathit{QOp}\ \mathit{TypeArgs}^{?}\ \mathit{VarList}(\texttt{,}\ \mathit{VarList})^{*}\ \texttt{::}\ \mathit{TrigAttr}^{*}\ \mathit{Expr}\texttt{)} \\
\mathit{BinOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \\
& \mid & \texttt{\&\&} \mid \texttt{||} \mid \texttt{==>} \mid \texttt{<==>} \\
& \mid & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
& \mid & \texttt{++} \mid \texttt{<:} \\
\mathit{UnOp} & ::= & \texttt{-} \mid \texttt{!} \\
\mathit{QOp} & ::= & \texttt{forall} \mid \texttt{exists} \\
\mathit{Lit} & ::= & \texttt{true} \mid \texttt{false} \\
& \mid & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
& \mid & \texttt{0.0} \mid \texttt{0.01}\ldots \mid \texttt{0.001}\ldots \mid \ldots \mid \texttt{1.0} \mid \texttt{1.01}\ldots \mid \ldots \\
& \mid & \texttt{0bv0} \mid \texttt{0bv1} \mid \texttt{1bv1} \mid \texttt{0bv2} \mid \texttt{1bv2} \mid \texttt{2bv2} \mid \texttt{3bv2} \mid \ldots \\
\mathit{TrigAttr} & ::= & \texttt{\{}\ \mathit{Expr}(\texttt{,}\ \mathit{Expr})^{*}\ \texttt{\}} \mid \mathit{Attr}
\end{array}
$$

Figure 2.1: Syntax of Boogie expressions in the ULTIMATE program analysis framework.

programs in the Integer domain. We thus introduce non-terminal $UninterpFn$ as uninterpreted functions for all bitwise unary and binary operations. Figure 2.2 shows our Boogie semantics extension on top of the semantics from Figure 2.1, omitting the similar syntax that showed in Figure 2.2.

$$
\begin{array}{rcl}
e & ::= & \mathit{Lit} \mid \mathit{Id} \mid \ldots \mid \mathit{UninterpFn} \\
\mathit{BinOp} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \ldots \\
\mathit{UnOp} & ::= & \mid \texttt{~} \mid \texttt{!} \\
& \ldots. & \\
\mathit{UninterpFn} & ::= & \mathit{bwAnd} \mid \mathit{bwOr} \mid \mathit{bwXor} \mid \mathit{bwShL} \mid \mathit{bwShR} \mid \mathit{bwCompl}
\end{array}
$$

Figure 2.2: Syntax of Boogie expressions with bitwise operators.

Figure 2.3 shows boogie statement syntax, as used in ULTIMATE.

```
Stmt       ::= assume Expr;                                  | assert Expr;
             | call forall Id (NondetExpr);                  | Id : Stmt
             | Lhs(, Lhs)* := Expr(, Expr)*;                  | break Id;
             | if (NondetExpr){ Stmt* } Else                 | goto Id(, Id)*;
             | while (NondetExpr) LoopInv* {Stmt* }          | call CallLhs Id ();
             | call CallLhs Id (Expr(, Expr)*);              | havoc Id(, Id)*;
             | call forall Id (Expr(, Expr)*);               | return;
Lhs        ::= Id                                            | Id[Expr(, Expr)*]
NondetExpr ::= * | Expr
Else       ::= else if (NondetExpr){ Stmt* }Else             | else { Stmt* }
CallLhs    ::= Id(, Id)* :=
LoopInv    ::= free invariant Expr;
```

Figure 2.3: Boogie statement syntax.

2.1.2 Semantics

For program (denoted P) semantics, we assume a state space $\Sigma : \mathsf{Vars} \to \mathsf{Vals}$, mapping variables (denoted V) to values. We let $[\![e]\!] : \Sigma \to \mathsf{Vals}$ and $[\![s]\!] : \Sigma \to \mathcal{P}(\Sigma)$ be the semantics of expressions and nondeterministic statements, respectively, and $[\![P]\!]$ denotes execution traces of P. A **trace** π of a program P is a sequence $\pi = \sigma_0, \sigma_1, \ldots$ such that σ_0 is an initial state, with a transition relation R of the program, and $\forall i \geq 0. R(\sigma_i, \sigma_{i+1})$, *i.e.* the transitions are in the operational semantics of P. The set of all traces of a program P is denoted by $[\![P]\!]$ and for a trace $\pi = \sigma_0, \sigma_1, \ldots$, the j-th state σ_j in the trace is denoted by $\pi[j]$.

We will also work with Boolean conditions $[\![e_b]\!] : \Sigma \to \mathbb{B}$ and expressions $[\![e]\!] : \Sigma \to \mathsf{Vals}$. To represent conditions, we use logical formulae for states, denoted $\mathcal{C}$, where $[\![\mathcal{C}]\!] : \Sigma \to \mathbb{B}$.

A transition system can be presented as an automaton. Control-flow automata (CFA) are one common such automata to model a program P as a transition system. Various verification tools [159, 152] build on top of CFAs.

Definition 2.1.1 (Control-flow automaton)**.** *A (deterministic)* control flow automaton *(CFA) [96] is a tuple $\mathcal{A} = \langle Q, q_0, X, s, \rightarrow \rangle$ where Q is a finite set of control locations and q_0 is the initial control location, X is a finite sets of typed variables, s is the loop/branch-free statement language and $\rightarrow \subseteq Q \times s \times Q$ is a finite set of labeled edges.*

2.1.3 Logical Transition Relation

We use the notation V' to denote a second set of primed versions of the same variables V, i.e. $V' = \{v' | v \in V\}$, We also work with logical state transition relations denoted $\mathcal{T}$, where $[\![\mathcal{T}]\!] \subseteq \Sigma \times \Sigma$.

2.2 Temporal Logic and Büchi Automaton

Computation tree logic (CTL) and linear temporal logic (LTL) are widely studied logics in temporal verification. Here we lay out definitions and notations for CTL and LTL used throughout our work.

2.2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a modal logic that reasons over linear program traces through time, starting in the current time and progressing into the future infinitely. LTL is per trace so, at each time instance, there is only one future timeline that can occur. We denote $Prop$ as a set of atomic propositions. LTL formulae are then composed of a finite set $Prop$, the Boolean operators $\neg, \wedge, \vee, \implies$ and the temporal operators U ("until"), R ("release"), X ("next time"), G ("globally") and F ("in the future"). Intuitively, formula $\varphi \mathsf{U} \psi$ states that either ψ is true now or φ is true now and φ remains true until such a time when ψ holds. For the case $\varphi \mathsf{R} \psi$, it means φ releases ψ, states that ψ must be true now and remains true until such a time when φ is true, thus releasing ψ. $\mathsf{X}\varphi$ means that φ is true in

the next time step after the current one. Lastly, $\mathsf{G}\varphi$ means that φ is always true in any time step while $\mathsf{F}\varphi$ designates that φ must either be true now or at some future time step. We now define LTL formulae inductively:

Definition 2.2.1 (LTL formulae)**.** *For every $p \in Prop$, p is a formula, LTL formula φ is defined as following:*

$$\varphi ::= p \in \mathsf{AP} \mid \neg\varphi \mid \mathsf{G}\varphi \mid \mathsf{F}\varphi \mid \varphi \implies \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \varphi\mathsf{U}\varphi \mid \varphi\mathsf{R}\varphi$$

Definition 2.2.2 (LTL formulae interpretation of program computation)**.** *ω denotes the set of non-negative integers, we have a computations form $\pi : \omega \to 2^{Prop}$, we use $\iff$ to denote "if only if". For computation π at the time instant $i \in \omega$ satisfies LTL formula ω, we define $\pi, i \models \varphi$ as following:*

$$\begin{aligned}
\pi, i \models p \; for \; p \in Prop &\iff p \in \pi(i). \\
\pi, i \models \neg\varphi &\iff \pi, i \not\models \varphi. \\
\pi, i \models \mathsf{G}\varphi &\iff \forall j \geq i, \pi, j \models \varphi. \\
\pi, i \models \mathsf{F}\varphi &\iff \exists j \geq i, such\ that\ \pi, j \models \varphi. \\
\pi, i \models \mathcal{X}\varphi &\iff \pi, i+1 \models \varphi. \\
\pi, i \models \varphi \wedge \psi &\iff \pi, i \models \varphi\ and\ \pi, i \models \psi. \\
\pi, i \models \varphi \vee \psi &\iff \pi, i \models \varphi\ or\ \pi, i \models \psi. \\
\pi, i \models \varphi\mathcal{U}\psi &\iff \exists j \geq i\ such\ that\ \pi, j \models \psi\ and\ \forall k, i \leq k < j, \pi, k \models \varphi. \\
\pi, i \models \varphi\mathcal{R}\psi &\iff \forall j \geq i, \iff \pi, j \not\models \psi,\ then\ \exists k, i \leq k < j\ such\ that\ \pi, k \models \varphi. \\
\pi, i \models \varphi \implies \psi &\iff \pi, i \models \neg\varphi \vee \psi.
\end{aligned}$$

2.2.2 Computation Tree Logic

There are two classes of temporal constructors in CTL formula: the AF and AW constructors quantify universally over paths and the EF and EW constructors quantify existential over paths. We use F and W as our base temporal operators and assume that formulae are written in negation normal form, in which negation only occurs next to atomic propositions. A formula that is not in negation normal form can be easily normalized. A CTL formula therefore is defined as:

$$\varphi ::= p \in \mathsf{AP} \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathsf{AF}\varphi \mid \mathsf{EF}\varphi \mid \mathsf{A}[\varphi \mathsf{W} \varphi] \mid \mathsf{E}[\varphi \mathsf{W} \varphi]$$

Note that $\mathsf{AG}p$ can be defined as $\mathsf{A}[p\ \mathsf{W}\ \mathsf{false}]$, and $\mathsf{EG}p$ can be defined as $\mathsf{E}[p\ \mathsf{W}\ \mathsf{false}]$. The semantics of each type of formula is shown below, where $[\![p]\!]$ denotes the set of states where p holds:

$$\begin{aligned}
M, \sigma \vDash p &\iff \sigma \in [\![p]\!] \\
M, \sigma \vDash \varphi_1 \wedge \varphi_2 &\iff M, \sigma \vDash \varphi_1 \wedge M, \sigma \vDash \varphi_2 \\
M, \sigma \vDash \varphi_1 \vee \varphi_2 &\iff M, \sigma \vDash \varphi_1 \vee M, \sigma \vDash \varphi_2 \\
M, \sigma \vDash \mathsf{AF}\varphi &\iff \forall(\sigma_0, \sigma_1, ...) \in \Pi(\Sigma, R, \{\sigma\}).\ \exists i \geq 0.\ M, \sigma_i \vDash \varphi \\
M, \sigma \vDash \mathsf{EF}\varphi &\iff \exists(\sigma_0, \sigma_1, ...) \in \Pi(\Sigma, R, \{\sigma\}).\ \exists i \geq 0.\ M, \sigma_i \vDash \varphi \\
M, \sigma \vDash \mathsf{A}[\varphi_1 \mathsf{W} \varphi_2] &\iff \forall(\sigma_0, \sigma_1, ...) \in \Pi(\Sigma, R, \{\sigma\}).\ (\forall i \geq 0.\ M, \sigma_i \vDash \varphi_1) \vee \\
&\qquad (\exists j \geq 0.\ M, \sigma_j \vDash \varphi_2 \wedge \forall i \in [0, j).\ M, \sigma_i \vDash \varphi_1) \\
M, \sigma \vDash \mathsf{E}[\varphi_1 \mathsf{W} \varphi_2] &\iff \exists(\sigma_0, \sigma_1, ...) \in \Pi(\Sigma, R, \{\sigma\}).\ (\forall i \geq 0. M, \sigma_i \vDash \varphi_1) \vee \\
&\qquad (\exists j \geq 0.\ M, \sigma_j \vDash \varphi_2 \wedge \forall i \in [0, j).\ M, \sigma_i \vDash \varphi_1)
\end{aligned}$$

2.2.3 ω Language and Büchi Automaton

LTL can express a strict subset of ω-regular expressions. In practical verification, linear time logic and operators make the expression of specific properties easier and more flexible to extend the specification language to full ω-regularity.

Definition 2.2.3 (ω-regular expression)**.** *An* ω***-regular expression*** *is an expression in the form* $\cup_i \alpha_i(\beta_i)^\omega$*, where* i *is non-zero and finite integers,* α *and* β *are regular expressions over the alphabet* Σ*.*

We can define ω-regular language with ω-regular language as following.

Definition 2.2.4 (ω-regular language)**.** *An* ω***-regular language*** *is the language that can be described by* ω*-regular expressions, and a language is* ω*-regular if and only if there is a Büchi automaton accepts it. The family of* ω*-regular language is closed under union, intersection, and complementation.*

The automata-theoretic approach is an effective way to accomplish the LTL verification task [164, 165]. Static analysis tools (ULTIMATE, CPACHECKER, etc.) are based on this approach, which translates LTL formulae into Büchi Automata [166]. This step is performed automatically by model checkers. An infinite word w is accepted by the Büchi automaton if the run over w visits at least one state in $\mathcal{F}$ infinitely often.

Definition 2.2.5 (Büchi Automaton)**.** *A Büchi Automaton is a quintuple* $(\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F})$ *where:*

- $\mathcal{Q}$ *is a finite set of states.*
- Σ *is a finite alphabet.*
- $\delta : \mathcal{Q} \times \Sigma \times \mathcal{Q}$ *is a transition relation.*

- $q_0 \in \mathcal{Q}$ *is the initial state.*
- $\mathcal{F} \subseteq \mathcal{Q}$ *is a set of final states.*

2.3 LTL Verification

To verify temporal properties that involve eventualities such as $\mathsf{AF}\varphi$ or $\mathsf{F}\varphi$, we need to reason about the loop body for its termination and non-termination, which requires ranking functions and recurrent sets. Additionally, building Büchi product automata is an effective fundamental approach for automata-theoretic LTL analysis, we define these fundamental ideas in this section.

Definition 2.3.1 (Ranking function)**.** *For a state space $\mathcal{S}$, a ranking function f is a total map from $\mathcal{S}$ to a well-ordered set with ordering relation $\prec$. A relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is well-founded if and only if there exists a ranking function f such that $\forall(s, s') \in \mathcal{T}.f(s') \prec f(s)$.*

Definition 2.3.2 (Recurrent set)**.** *For sets of states $\mathcal{X}$ and transition relation $\mathcal{T}$, $\mathcal{X}$ is a recurrent set if*

1. *$\mathcal{S} \neq \emptyset$,*
2. *$\mathcal{T}$ is total on $\mathcal{X}$,*
3. *the image of $\mathcal{T}$ on $\mathcal{X}$ is contained within $\mathcal{X}$*

Definition 2.3.3 (Büchi program product)**.** *Let $\mathcal{P} = (Loc, l_0, \delta_{\mathcal{P}})$ be a program (written in Boogie programming language) over the set of statements $Stmt$, AP a set of atomic propositions over the program's variables Var, and let $\mathcal{A} = (\Sigma, \mathcal{Q}, q_0, \rightarrow, \mathcal{F})$ be a Büchi automaton whose alphabet is $\Sigma = 2^{AP}$. The Büchi program product $\mathcal{P} \otimes \mathcal{A}$ is a Büchi program $B = (Stmt_{\mathcal{B}}, Loc_{\mathcal{B}}, l_{0_{\mathcal{B}}}, \delta_{\mathcal{B}}, Loc_{\mathcal{F}_{\mathcal{B}}})$ such that the set of statements consists of all sequential compositions of two statements where the first element is a statement of $\mathcal{P}$*

and the second element is a statement that assumes that a subset of atomic propositions is satisfied, where:

- $Stmt_{\mathcal{B}} = \{s;\ assume\ a | s \in Stmt, a \in 2^{AP}\}$.
- $Loc_{\mathcal{B}} = \{(l, q) | l \in Loc\ and\ q \in Q\}$.
- $l_{0_{\mathcal{B}}} = (l_0, q_0)$.
- $\delta_{\mathcal{B}} = \{((l, q), s;\ assume\ a, (l', q')) | (l, s, l') \in \delta_{\mathcal{P}}\ and\ (q, a, q') \in \rightarrow\}$ *(l', q' are next location/state obtained from the transition relation).*
- $Loc_{\mathcal{F}_{\mathcal{B}}} = \{(l, q) | l \in Loc\ and\ q \in \mathcal{F}\}$.

2.4 Principal Verification Tools and Dynamic Verification

In Chapter 1, we introduced various verification tools based on static analysis and dynamic analysis. In this section, we summarize more details about tools that are competitive among their peer tools and have been used in our experiments.

2.4.1 Static Temporal Verification

Among static verification tools, ULTIMATE [90] stands out for its grand support for analysis techniques (e.g., interpolation, abstract interpretation, predicate abstraction, bounded model checking, etc.) and unified components that are flexible for users to extend its functionalities. ULTIMATE is a program analysis framework, that uses an automata-theoretic approach to verify safety, termination, and LTL properties of the program ($C/C++$, Boogie, etc.), and it provides rich automaton operation libraries. The ULTIMATE framework consists of various plugins that are capable of complex verification tasks. The front end plugin can parse source programs into boogie programs as internal high level representation, then it transforms program expressions to SMT and builds up intermediate control

flow graphs. From the control flow graph, various automatons (Büchi automaton 2.3.3, nested word automaton) can be constructed for verification analysis.

FUNCTION [72] and T2 [32] are established tools for termination verification and temporal verification (CTL). FUNCTION infers piecewise-defined ranking functions through abstract interpretation, combining various abstract domains to balance the precision and the cost of analysis. The analyzer is implemented in Ocaml on top of the APRON [1] library. T2 is the publicly released tool from the Terminator project [52]. Over the past decade, it has been extended to support temporal-logic proving techniques, and it allows users to provide a broader class of liveness and safety properties. Internally T2 works on an intermediate representation, also called `.t2`, that can be extracted from LLVM framework, allowing support for a wide range of programming languages (e.g. C, C++, Objective C, etc.). T2 can reduce temporal specifications to *safety*, *termination* and *nontermination* analysis techniques, more details of its architecture are introduced in [33].

2.4.2 Dynamic Verification

In chapter 5 we introduce a temporal analysis technique that employs dynamic analysis, and we use dynamic tool DIG [130]. DIG can generate possible ("cadidate") program invariants at arbitrary program locations. Users can specify program locations and use DIG to run on source code. DIG performs symbolic execution on the input program and generates concrete state (concrete values of global and local variables) from program locations. It then infers numerical invariants among arbitrary variables. Its numeric relations include non-linear/linear equalities, linear inequalities, minimum/maximum equalities/inequalities, congruent relations, and array invariants. Users can also provide concrete trace data of the program to DIG for invariant inference. The project is open source and core algorithms are implemented in `Python`. We now formally introduce nonlinear programs that our work

[1] http://apron.cri.ensmp.fr/library/

operates on and define their related terms.

Syntax. We work with a simple model of programs that supports nonlinear arithmetic expressions (branch/loop conditions and assignments), the statement is presented in Boogie syntax but with a focus on branching statements, and its full syntax and semantics are shown in Section 2.1:

$$
\begin{array}{lll}
P & ::= & \mathsf{stmt} \\
\mathsf{stmt} & ::= & \mathsf{stmt}\ \texttt{;}\ \mathsf{stmt} \mid \texttt{if}\ b^{\ell}\ \texttt{then}\ \mathsf{stmt}\ \texttt{else}\ \mathsf{stmt} \mid v := e \mid \texttt{skip} \\
 & & \mid \texttt{while(true) if(}\neg b^{\ell}\texttt{) then break else}\ \mathsf{stmt} \\
\mathsf{e} & ::= & c \mid v \mid \mathsf{e} \otimes \mathsf{e} \mid \ldots \\
\mathsf{b} & ::= & \texttt{true} \mid \texttt{false} \mid \neg \mathsf{b} \mid \mathsf{b}\ \texttt{and}\ \mathsf{b} \mid \mathsf{b}\ \texttt{or}\ \mathsf{b} \mid \mathsf{e} \lessapprox \mathsf{e}
\end{array}
$$

where $\lessapprox ::= < \mid \leq \mid = \mid \ldots$ are comparisons between integer expressions, $\otimes$ are typical integer operations, and c refers to integer constants. We represent `while` loops with `if` and `break`, so that we can apply our algorithms to `if` statements only, yet still capture branching in loop headers. We label branch condition expressions b^{ℓ} and assume that each ℓ is chosen to uniquely identify the expressions (*i.e.* a program location). We also assume that the program location is stored in a variable pc for the program counter whose value is ℓ when evaluating expression b^{ℓ}.

Snapshots. As is common in dynamic analysis, instrumentation is used so that when the program is executed, states can be recorded. We introduce a family of "snapshot" keywords $\mathbf{snap}^{\ell}$ to represent statements added to the program at location ℓ that read and log the current state, along with the location ℓ where the snapshot is taken before the ℓ statement is executed. Apart from the side-effect of saving the state, the semantics of $\mathbf{snap}$ is otherwise `skip`.

Approximating sampled states. For a set of states S, we define an *over-approximation of a sampling*, denoted α_S, to be an abstraction that is guaranteed to be an abstraction only of the states in S. If $S = \Sigma$, then α_S is sound for the program. We assume a facility for learning abstractions α_S of a sample of states S. Such techniques are available from recent tools such as DIG [129], Diakon [67], etc. and provide the following function:

Definition 2.4.1. *[Learning sample approximations] For a set of states $S \subseteq \Sigma$, we assume the availability of a function* learn $: S \to \bar{\mathcal{C}}$ *such that $\forall \sigma \in S, c \in \bar{\mathcal{C}}$ we have that $[\![c]\!]\sigma$ is true.*

Static verification (notes). We assume a special location denoted `err` and the static verification problem is to determine whether `err` is reachable in a given program. If not, static validation returns a counterexample *path* cex. We assume that this counterexample includes the error location that was possibly reachable.

Chapter 3

Temporal Verification of Bitvector Programs

There is increasing interest in applying verification tools to programs that have bitvector operations. SMT solvers, which serve as a foundation for these tools, have thus increased support for bitvector reasoning through bit-blasting and linear arithmetic approximations. One common strategy employed by these SMT solvers is *bit-blasting*, which translates the input bitvector formula to an equi-satisfiable propositional formula and utilizes Boolean Satisfiability (SAT) solvers to discharge it. Another strategy is to approximate bitvector operations with integer linear arithmetic [30]. CVC4 now employs a new approach called *int-blasting* [174], which reasons about bitvector formulas via integer nonlinear arithmetic.

Inspired by these SMT strategies, in this chapter we show that similar linear arithmetic approximation of bitvector operations can be done at the source level through transformations. Specifically, we present *bitwise branching* that introduces new paths to over-approximate bitvector operations with linear conditions/constraints, increasing branching but allowing us to better exploit the well-developed integer reasoning and interpolation of verification tools. *Bitwise branching*(BWB) can be combined with various tools, making it an appealing general strategy. We implement *bitwise branching* as a source translation, in the ULTIMATE verifier. We at first chose to implement bitwise branching within ULTIMATE source code (during the C-to-Boogie [24] translation) so that we could compare against unmodified ULTIMATE, which is already one of the more effective Termination/LTL verifiers. Furthermore, to our knowledge other tools don't allow one to flip a switch to enable their own bit-precise analysis (i.e., CBMC's Bitblasting or CPACHECKER's FixedSizeBitVectors theory) or disable that analysis, abstracting with integers. We needed such a switch to evaluate bitwise branching. Other tools that employ non-bitprecise techniques simply

report "Unknown" as soon as they encounter bit operations.

We evaluate BWB on improving verifiers of reachability, terminatio, and LTL. SV-COMP (software verification competition) has a collection of benchmarks for various verification tasks, however, most SV-COMP benchmarks require little or no bitvector reasoning and the majority of benchmarks have no bitvector operations in them whatsoever. Others have bitvector operations, but those operations are not relevant to the property and existing tools abstract them away. Since the SV-COMP benchmarks do not include examples targeted to bitvector termination or bitvector LTL, we created new benchmarks by extending examples from the SV-COMP termination category [9]. In our experiment, we show that, for reachability of bitvector programs, increased branching incurs negligible overhead yet, when combined with integer interpolation optimizations, enables more programs to be verified. We further show this exploitation of integer interpolation in the common case also enables competitive termination verification of bitvector programs and leads to the first effective technique for LTL verification of bitvector programs. We implement *bitwise branching* as a source translation, in the ULTIMATE verifier.

3.1 Motivating Examples

(1) Reachability	**(2) Termination**	**(3) LTL** $\varphi = \mathsf{G}(\mathsf{F}(n < 0))$

`int r, s, x;`
`while (x>0){`
`  s = x >> 31;`
`  x--;`
`  r = x + (s&(1-s));`
`  if (r<0) error();`
`}`
	`a = *; assume(a>0);`
	`while (x>0){`
	`  a--;`
	`  x = x & a;`
	`}`
		`while(1) {`
		`  n = *; x = *; y = x-1;`
		`  while (x>0 && n>0) {`
		`  n++;`
		`  y = x | n;`
		`  x = x - y;`
		`  }`
		`  n = -1;`
		`}`

and_reach1.c	and-01.c	or_loop3.c

We refer to the above bitvector programs throughout the chapter. To prove `error` unreachable in the Example **(1)**, a verifier must be able to reason about the bitvector `>>` and `&` operations. Specifically, it must be able to conclude that expression `s&(1-s)` is always positive (so `r` cannot be negative) which also depends on the earlier `x>>31` expression. We will use this example to explain our work in Sec. 3.2, and compare the performance of ULTIMATE using state-of-the-art SMT solvers, with and without bitwise branching. In Sec. 3.3, we describe our implementation inside ULTIMATE (with MATHSAT) and show that our technique allows us to prove this example in 0.387s as opposed to 0.740s. Similar speedups are found when using CVC4 or Z3.

The key benefits of bitwise branching arise when concerned with termination and LTL. Example **(2)** involves a simple loop, in which `a` is decremented, but the loop condition is on variable `x`, whose value is a bitvector expression over `a`. Today's tools for Termination of bitvector programs struggle with this example: APROVE, CPACHECKER and ULTIMATE report unknown and KITTEL and 2LS timeout after 900s . Critical to verifying termination of this program are (1) proving the invariant $\mathcal{I}$: `x` $> 0 \wedge$ `a` > 0 on Line 3 within the body of the loop and (2) synthesizing a rank function. To prove the invariant, tools must show that it holds after a step of the loop's transition relation $\mathcal{T} = x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x \,\&\, a'$, which requires reasoning about the bitwise-`&` operation because if we simply treat the `&` as an uninterpreted function, $\mathcal{I} \wedge \mathcal{T} \wedge x'{>}0 \implies \mathcal{I}'$.

```
1  a = *; assume(a > 0);
2  while (x > 0) {
3   { x > 0 ∧ a > 0 }
4   a--;
5   if (x >= 0 && a >= 0)
6   then { x = *; assume(x <= a); }
7   else {x = x & a;}
8  }
```

Figure 3.1: Transformed Version for Example 2.

The bitwise branching strategy we describe in this chapter helps the verifier infer these invariants (and later synthesize rank functions) by transforming the bitvector assignment to `x` into linear constraint `x<=a`, but only under the condition that `x>=0` and `a>=0`. That is, bitwise branching translates the loop in Example **(2)** to the version as depicted in the gray box of Figure 3.1. This transformation changes the transition relation of the loop body from $\mathcal{T}$ (the original program) to $\mathcal{T}'$:

$$\mathcal{T}' = x{>}0 \wedge a'{=}a{-}1 \wedge ((x{\geq}0 \wedge a'{\geq}0 \wedge x'{\leq}a') \vee (\neg(x{\geq}0 \wedge a'{\geq}0) \wedge x'{=}x \,\&\, a'))$$

Importantly, when $\mathcal{I}$ holds, the else branch with the bitwise `&` is infeasible, and thus we can treat the bitwise `&` as an uninterpreted function and yet still prove that $\mathcal{I} \wedge \mathcal{T}' \wedge x'{>}0 \implies \mathcal{I}'$. With the proof of $\mathcal{I}$ a tool can then move to the next step and synthesizing a ranking function $\mathcal{R}(x, a)$ that satisfies $\mathcal{I} \wedge \mathcal{T}' \implies \mathcal{R}(x, a){\geq}0 \wedge \mathcal{R}(x, a){>}\mathcal{R}(x', a')$, namely, $\mathcal{R}(x, a) = a$.

Bitwise branching also enables LTL verification of bitvector programs. We examine the behavior of programs such as Example **(3)** above, with LTL property $\mathsf{G}(\mathsf{F}(n < 0))$. The state-of-the-art program verifier for LTL is ULTIMATE, but ULTIMATE cannot verify this program due to the bitvector operations. (ULTIMATE's internal overapproximation is too imprecise so it returns Unknown.) In Sec. 3.4 we show that with bitwise branching, our implementation can prove this property of this program in 8.04s.

3.2 Bitwise-branching

We now describe our main technique. We build our *bitwise-branching* technique on the known strategy of transforming bitvector operations into integer approximations [30, 174] but explore a new direction: source-level transformations to introduce new conditional paths that approximate many (but not all) behaviors of a bitvector program. These new paths through the program have linear input conditions and linear output constraints and

frequently cover all of the program's behavior (with respect to the goal property), but otherwise fall back on the original bitvector behavior when none of the input conditions hold. We provide two sets of bitwise-branching rules:

$$
\begin{array}{rlll}
e_1 = 0 & \vdash_E \; e_1 \& e_2 & \rightsquigarrow 0 & \text{[R-AND-0]} \\
(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1 & \vdash_E \; e_1 \& e_2 & \rightsquigarrow e_1 & \text{[R-AND-1]} \\
(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1) & \vdash_E \; e_1 \& e_2 & \rightsquigarrow e_1 \&\& e_2 & \text{[R-AND-LOG]} \\
(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1) & \vdash_E \; (e_1 \mid e_2) == 0 & \rightsquigarrow e_1 == 0 \&\& e_2 == 0 & \text{[R-OR-LOG]} \\
e_1 \geq 0 \wedge e_2 = 1 & \vdash_E \; e_1 \& e_2 & \rightsquigarrow e_1 \% 2 & \text{[R-AND-LBS]} \\
e_2 = 0 & \vdash_E \; e_1 \mid e_2 & \rightsquigarrow e_1 & \text{[R-OR-0]} \\
(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1 & \vdash_E \; e_1 \mid e_2 & \rightsquigarrow 1 & \text{[R-OR-1]} \\
e_2 = 0 & \vdash_E \; e_1 \hat{} e_2 & \rightsquigarrow e_1 & \text{[R-XOR-0]} \\
e_1 = e_2 = 0 \vee e_1 = e_2 = 1 & \vdash_E \; e_1 \hat{} e_2 & \rightsquigarrow 0 & \text{[R-XOR-EQ]} \\
(e_1 = 1 \wedge e_2 = 0) \vee (e_1 = 0 \wedge e_2 = 1) & \vdash_E \; e_1 \hat{} e_2 & \rightsquigarrow 1 & \text{[R-XOR-NEQ]} \\
e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT} \star \texttt{sizeof}(e_1) - 1 & \vdash_E \; e_1 \gg e_2 & \rightsquigarrow 0 & \text{[R-RSHIFT-POS]} \\
e_1 < 0 \wedge e_2 = \texttt{CHAR\_BIT} \star \texttt{sizeof}(e_1) - 1 & \vdash_E \; e_1 \gg e_2 & \rightsquigarrow -1 & \text{[R-RSHIFT-NEG]}
\end{array}
$$

Figure 3.2: Rewriting rules for arithmetic expressions.

1. Rewriting rules of the form $\mathcal{C} \vdash_E e_{bv} \rightsquigarrow e_{int}$ in Fig. 3.2. These rules are applied to bitwise arithmetic expressions e_{bv} and specify a condition $\mathcal{C}$ for which one can use integer approximate behavior e_{int} of e_{bv}. In other words, rewriting rule $\mathcal{C} \vdash_E e_{bv} \rightsquigarrow e_{int}$ can be applied only when $\mathcal{C}$ holds and a bitwise arithmetic expression e in the program structurally matches its e_{bv} with a substitution δ. Then, e will be transformed into a conditional approximation: $\mathcal{C}\delta \; ? \; e_{int}\delta \; : \; e_{bv}$. Note that, although modulo-2 is computationally more expensive, it is often more amenable to integer reasoning strategies. Note that there is a rewriting rule we use modulo constant 2, which in general is more expensive than bitwise operation, while our goal is to transform common bitwise operations to integer arithmetic, which opens more efficient verification tasks in integer domain. For conciseness, we omitted condition variants that arise from commutative re-ordering of the rules (in both Figs. 3.2 and 3.3).

For example, consider the bitvector arithmetic expression `s&(1-s)` in Example **(1)** of Sec. 3.1. If we apply the rewriting rule $e_1 \geq 0 \wedge e_2 = 1 \vdash_E e_1 \& e_2 \rightsquigarrow e_1 \% 2$ with the substitution $\texttt{s}/e_1, \texttt{1-s}/e_2$ then the expression is transformed into `s>=0 && (1-s)==1`

`? s%2 : (s&(1-s))`. Since `s` reflects the sign bit of the positive variable `x`, it is always 0, and the `if` condition is feasible. In general, all the rules can be applied one at a time, and we can further replace the remaining bitwise operation in the `else` expression with other applicable rules. There may still be executions that fall into the final catch-all case where the bitwise operation is performed. However, as we see later, these case splits are nonetheless practically significant because often the final `else` is infeasible.

$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	$r\ \mathtt{op}_{le}\ e_1\mathtt{\&}e_2$	$\rightsquigarrow r\mathtt{<=}e_1\ \mathtt{\&\&}\ r\mathtt{<=}e_2$	[W-AND-POS]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{le}\ e_1\mathtt{\&}e_2$	$\rightsquigarrow r\mathtt{<=}e_1\ \mathtt{\&\&}\ r\mathtt{<=}e_2\ \mathtt{\&\&}\ r\mathtt{<}0$	[W-AND-NEG]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{eq}\ e_1\mathtt{\&}e_2$	$\rightsquigarrow 0\mathtt{<=}r\ \mathtt{\&\&}\ r\mathtt{<=}e_1$	[W-AND-MIX]
$e_1 \geq 0 \wedge \mathtt{is\_const}(e_2)$	$\vdash_S$	$r\ \mathtt{op}_{ge}\ e_1\mathtt{\|}e_2$	$\rightsquigarrow r\mathtt{>=}e_2$	[W-OR-CONST]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	$r\ \mathtt{op}_{ge}\ e_1\mathtt{\|}e_2$	$\rightsquigarrow r\mathtt{>=}e_1\ \mathtt{\&\&}\ r\mathtt{>=}e_2$	[W-OR-POS]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{eq}\ e_1\mathtt{\|}e_2$	$\rightsquigarrow r\mathtt{>=}e_1\ \mathtt{\&\&}\ r\mathtt{>=}e_2\ \mathtt{\&\&}\ r\mathtt{<}0$	[W-OR-NEG]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{eq}\ e_1\mathtt{\|}e_2$	$\rightsquigarrow e_2\mathtt{<=}r\ \mathtt{\&\&}\ r\mathtt{<}0$	[W-OR-MIX]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	$r\ \mathtt{op}_{ge}\ e_1\mathtt{\^}e_2$	$\rightsquigarrow r\mathtt{>=}0$	[W-XOR-POS]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{ge}\ e_1\mathtt{\^}e_2$	$\rightsquigarrow r\mathtt{>=}0$	[W-XOR-NEG]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	$r\ \mathtt{op}_{le}\ e_1\mathtt{\^}e_2$	$\rightsquigarrow r\mathtt{<}0$	[W-XOR-MIX]
$e_1 \geq 0$	$\vdash_S$	$r\ \mathtt{op}_{eq}\ \mathtt{\sim}e_1$	$\rightsquigarrow r\mathtt{<}0$	[W-CPL-POS]
$e_1 < 0$	$\vdash_S$	$r\ \mathtt{op}_{eq}\ \mathtt{\sim}e_1$	$\rightsquigarrow r\mathtt{>=}0$	[W-CPL-NEG]

Figure 3.3: Weakening rules for relational expressions and assignments. $\mathtt{op}_{le} \in \{\mathtt{<}, \mathtt{<=}, \mathtt{==}, \mathtt{:=}\}$, $\mathtt{op}_{ge} \in \{\mathtt{>}, \mathtt{>=}, \mathtt{==}, \mathtt{:=}\}$, and $\mathtt{op}_{eq} \in \{\mathtt{==}, \mathtt{:=}\}$

2. Weakening rules of the form $\mathcal{C} \vdash_S s_{bv} \rightsquigarrow s_{int}$ are in Fig. 3.3. These rules are applied to relational condition expressions (*e.g.* from assumptions) and assignment statements s_{bv}, specifying an integer condition $\mathcal{C}$ and over-approximation transition constraint s_{int}. When the rule is applied to a statement (as opposed to a conditional), replacement s_{int} can be implemented as `assume(`s_{int}`)` statement followed by a `havoc()` statement. When a weakening rule $\mathcal{C} \vdash_S s_{bv} \rightsquigarrow s_{int}$ is applied to an assignment s with substitution δ, the transformed statement is `if` $\mathcal{C}\delta$ `assume(`$s_{int}\delta$`) else` s_{bv}. In addition, when s_{bv} of a weakening rule can be matched to the condition `c` in an `assume(c)` of the original program via a substitution δ, then the `assume(c)` statement is transformed to:
`if` $\mathcal{C}\delta$ `then assume(`$s_{int}\delta$`) else assume(c).`

This rule can be applied to Example **(2)** of Sec. 3.1. In the view of CFA, consider

the set of statements of a given program as an alphabet set in CFA, Fig. 3.4 shows this rule application for $x = x\&a$ in automaton level, the top is the original automaton, the bottom is the transformed automaton after bitwise branching rule applied.

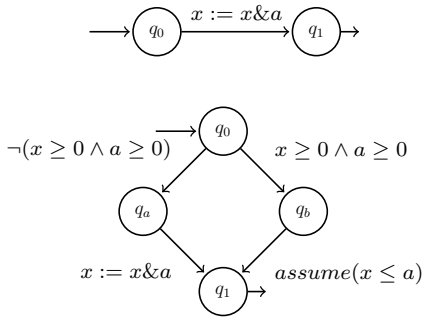


Figure 3.4: Weakening rules application in CFA (simplified for demonstration).

For all the branching rules, we introduce the following lemma that assists our transformation soundness proof.

Lemma 3.2.1 (Rule correctness)**.** *For every rule* $\mathcal{C} \vdash_E e \rightsquigarrow e'$, $\forall\sigma.\, \mathcal{C}(\sigma) \Rightarrow [\![e]\!]\sigma = [\![e']\!]\sigma$. *For every* $\mathcal{C} \vdash_S s \rightsquigarrow s'$, $\forall\sigma.\, \mathcal{C}(\sigma) \Rightarrow [\![s]\!]\sigma \subseteq [\![s']\!]\sigma$.

We encode each rule's correctness with SMT solver script (we use Z3 with 32-bit size), proof details are in Appendix A.1. The choice of rules in Fig. 3.2 and Fig. 3.3 was developed empirically, from the reachability/termination/LTL benchmarks in the next sections and, especially, based on patterns found in decompiled binaries (Sec. 4.2). We then generalized these rules to expand coverage. As mentioned before, bitwise branching rules can help abstract away bitwise operations and transform bitvector program into an over-approximated program, which can be reasoned in integer arithmetic. Particularly, for verification tasks of termination and temporal properties, instead of synthesizing ranking function over bitvector directly, our branching rules which are motivated by our empirical study and generalized to have broad coverage, can help verification tools prove these properties more effectively in the integer domain.

Translation algorithm. Our translation acts on the AST of the program, with one method T_E `: exp -> exp` to translate expressions and another method T_S `: stmt -> stmt` to translate assignment statements, each according to the set of available rules, detail algorithms of T_E and T_S are shown in Figure 3.5.

```
type rule_exp = (exp -> exp -> exp) * (
    exp -> exp -> exp)
let rec T_E (e:exp) : exp =
  match e with
  | BinOp(⊗,e1,e2) ->
    let e1' = T_E e1 in
    let e2' = T_E e2 in
    let rules = Rules.find_exp(⊗) in
    fold_left (fun acc (cond,repl) ->
      ITE(cond e1' e2',repl e1' e2',acc)
    )  (BinOp(⊗,e1, e2)) rules
  | _ -> e
```

```
type rule_stmt = (exp -> exp -> exp) * (lhs
    -> exp -> exp -> stmt)
let T_s (s:stmt) : stmt =
  match s with
  | Assign(lhs,BinOp(⊗,e1,e2)) ->
    let e1' = T_E e1 in
    let e2' = T_E e2 in
    let rules = Rules.find_stmt(⊗)in
    fold_left (fun acc (cond,repl) ->
      IfElseStmt(cond e1' e2',repl lhs e1' e2
    ', acc)
    ) (Assign(l, BinOp(⊗, e1, e2) rules
   | _ -> s
```

Figure 3.5: Bitwise branching algorithm.

In brief, when we reach a node with a bitwise operator, we recursively translate the operands, match the current operator against our collection of rules, and apply all matching rules to construct nested if-then-else expressions/statements. We found that, when multiple rules matched, the order did not matter much.

Let $T_E\{e\} : e$ denote the result of applying substitutions to e, and similar for $T_S\{s\} : s$. We lift this to a translation on a Boogie program P with $T_E\{P\} : P$ and $T_S\{P\} : P$, referring to all expressions and statements in P, respectively. It is straightforward to show that these rules are correct and that the translation is sound. Our rules are given in Figures 3.2 and 3.3, and each can be thought of as a substitution δ on both the condition $\mathcal{C}$ and expression e, obtaining e'. Let $D_E : e \to \delta$ list be a transformation decision algorithm defining, for an expression of e, which rules (*i.e.* a list of substitutions) to apply. Similarly, let $D_S : s \to \delta$ list decide for a statement s which substitutions to apply. Further, let $T_E\{e\} : e$ denote the result of applying substitutions to e as decided by D_E, and similar for $T_S\{s\} : s$. Finally, we lift this to translations on programs P with $T_E\{P\} : P$ and

$T_S\{P\} : P$, referring to all expressions and statements in P, respectively. We define translations for expressions T_E and for assignment statements T_S, over a Boogie program P, with semantics $[\![P]\!]$. We then show the following:

Theorem 3.2.2 (Soundness)**.** *For every* P, T_E, T_S, $[\![P]\!] \subseteq [\![T_S\{T_E\{P\}\}]\!]$.

Proof. Induction on traces, showing equality on expression translation T_E via induction on expressions/statements and then inclusion on statement translations T_S. First show that T_E preserves trace equivalence. Structural induction on e, with base cases being constants, variables, etc. In the inductive case, for a bitvector operation $e_1 \otimes e_2$, assume e_1, e_2 has been (potentially) transformed to e_1', e_2' (resp.) and that Lemma 3.2.1 holds for each $i \in \{1, 2\}$: $\forall\sigma.[\![e_i]\!]\sigma = [\![e_i']\!]\sigma$. Since $\otimes$ is deterministic, $[\![e_1' \otimes e_2']\!]\sigma = [\![e_1 \otimes e_2]\!]\sigma$. Finally, applying the transformation to $\otimes$, we show that $[\![T_E\{e_1' \otimes e_2'\}]\!] = [\![e_1' \otimes e_2']\!]$ again by Lemma 3.2.1. Next, for each statement s or relational condition c step, we prove T_S preserves trace inclusion: that $[\![s]\!] \subseteq [\![T_S\{s\}]\!]$ or that $[\![c]\!] \subseteq [\![T_S\{c\}]\!]$. We do not recursively weaken conditional boolean expressions, which would require alternating strengthening/weakening. Thus, inclusion holds directly from Lemma 3.2.1. □

3.3 Reachability of Bitvector Programs

We now evaluate the effectiveness of bitwise branching (BwB) toward reachability verification. We developed a new suite of 28 bitvector programs, including those adapted from existing code snippets like the "BitHacks" programs, which uses bitwise operations for various tasks. We implemented bitwise branching via a translation algorithm, in a fork of ULTIMATE (now merged to the main branch). We denote our version as ULTIMATEBWB, and it is publicly available [1].

We ran our experiments with BENCHEXEC [28] on a machine with an AMD Ryzen

[1] `https://github.com/cyruliu/darksea`

Table 3.1: Performance of ULTIMATE on bitvector programs, *e.g.* drawn from Sean Andersen's "Bit Hacks" repository, using various SMT solvers, with and without bitwise branching (BWB).

	Integer										Bitvector							
	BWB CVC4		BWB MS Itp		BWB MS		BWB SItp+Z3		SItp+Z3		BWB Z3		CVC4		MS Itp		MS		Z3	
Simple																				
logic_cmpl.c	✔	6.10s	✔	5.28s	✔	5.43s	✔	5.68s	?	4.90s	✔	5.91s	✔	5.90s	✔	5.65s	✔	6.86s	✔	5.67s
logic_cmpl_f.c	✗	6.97s	✗	6.64s	✗	6.74s	✗	5.09s	?	6.69s	✗	6.76s	✗	5.05s	✗	4.74s	✗	4.92s	✗	4.96s
and_loop.c	✔	7.27s	✔	5.39s	✔	6.14s	✔	5.42s	✔	5.42s	✔	5.19s	✔	5.60s	✔	6.49s	✔	5.21s	✔	5.46s
and_loop_f.c	✗	6.72s	✗	6.57s	✗	5.52s	✗	5.47s	✗	6.90s	✗	7.55s	✗	5.44s	✗	6.55s	✗	6.92s	✗	6.99s
logic_or.c	✔	6.11s	✔	5.35s	✔	7.93s	✔	7.19s	?	5.36s	✔	6.11s	✔	7.02s	✔	6.02s	✔	7.49s	✔	6.89s
logic_or_f.c	✗	5.35s	✗	4.91s	✗	5.52s	✗	4.58s	?	4.85s	✗	4.93s	✗	5.47s	✗	5.68s	✗	4.66s	✗	5.22s
logic_and.c	✔	10.91s	✔	7.27s	✔	11.66s	✔	7.30s	?	5.64s	✔	8.83s	✔	5.89s	✔	6.88s	✔	7.04s	✔	6.80s
logic_and_f.c	✗	4.94s	✗	4.93s	✗	5.01s	✗	6.42s	?	6.36s	✗	4.58s	✗	4.98s	✗	5.04s	✗	4.95s	✗	4.88s
logic_xor.c	✔	5.99s	✔	5.54s	✔	5.94s	✔	5.06s	?	6.64s	✔	5.69s	✔	5.77s	✔	5.23s	✔	5.30s	✔	5.25s
logic_xor_f.c	✗	4.90s	✗	4.93s	✗	5.19s	✗	4.64s	?	5.16s	✗	5.16s	✗	5.29s	✗	5.02s	✗	6.26s	✗	4.87s
and_reach1.c	✔	8.17s	✔	5.08s	✔	7.59s	✔	5.80s	?	5.08s	✔	10.77s	✔	7.10s	✔	5.06s	✔	5.94s	✔	5.61s
and_reach2.c	✔	6.53s	✔	5.21s	✔	6.40s	✔	5.28s	?	6.50s	✔	8.14s	✔	5.94s	✔	5.31s	✔	6.10s	✔	7.51s
BitHacks																				
parity_f.c	✗	6.24s	✗	5.72s	✗	5.64s	✗	5.67s	?	5.33s	✗	6.14s	✗	5.66s	✗	5.32s	✗	5.95s	✗	5.73s
cnt-bits-set.c	✔	8.16s	✔	7.59s	✔	8.34s	✔	7.84s	?	7.18s	✔	8.62s	✔	7.26s	✔	7.50s	✔	7.62s	✔	8.01s
cnt-bits-set_f.c	✗	5.99s	✗	5.26s	✗	5.88s	✗	5.98s	?	5.90s	✗	5.93s	✗	6.56s	✗	5.78s	✗	5.67s	✗	5.75s
display-bit.c	✔	5.92s	✔	7.46s	✔	5.68s	✔	7.28s	?	6.11s	✔	6.11s	✔	7.25s	✔	5.46s	✔	6.29s	✔	5.81s
display-bit_f.c	✗	34.16s	✗	51.33s	✗	28.61s	✗	35.44s	?	7.60s	✗	26.64s	✗	29.78s	✗	44.89s	✗	27.13s	✗	32.37s
display-bit1.c	✔	7.08s	✔	5.83s	✔	30.98s	✔	5.81s	?	5.43s	✔	6.47s	✔	7.69s	✔	5.42s	✔	22.28s	✔	5.65s
display-bit1_f.c	✗	25.34s	✗	43.19s	✗	24.94s	✗	24.63s	?	6.43s	✗	19.91s	✗	25.27s	✗	35.84s	✗	20.87s	✗	24.73s
reverse-bits1.c	✔	7.69s	✔	6.36s	✔	7.11s	✔	6.35s	?	5.01s	✔	7.34s	✔	6.36s	✔	5.77s	✔	6.21s	✔	6.36s
reverse-bits1_f.c	✗	7.67s	✗	7.05s	✗	7.12s	✗	7.22s	?	6.67s	✗	7.46s	✗	6.80s	✗	6.57s	✗	6.87s	✗	6.81s
cz-bits-trailing.c	✔	6.32s	✔	5.16s	✔	5.89s	✔	6.08s	?	6.12s	✔	5.83s	✔	5.73s	✔	6.10s	✔	5.78s	✔	6.24s
cz-bits-trailing_f.c	✗	7.11s	✗	6.72s	✗	6.68s	✗	7.22s	?	6.54s	✗	6.77s	✗	6.93s	✗	7.12s	✗	6.42s	✗	6.79s
cnt-bits-BK1.c	✔	8.99s	✔	5.48s	✔	38.14s	✔	5.44s	?	5.40s	✔	34.03s	✔	6.27s	**T**	300.81s	✔	5.01s	✔	5.80s
cnt-bits-BK1_f.c	✗	5.30s	✗	4.96s	✗	5.51s	✗	5.70s	?	5.12s	✗	5.29s	✗	5.36s	✗	5.54s	✗	4.99s	✗	5.02s
cnt-bits-BK.c	✗	5.43s	✗	4.79s	✗	4.97s	✗	4.74s	?	5.10s	✗	5.02s	✗	5.62s	✗	5.10s	✗	4.81s	✗	5.00s
cnt-bits-BK_f.c	✗	7.70s	✗	6.92s	✗	7.41s	✗	7.23s	?	6.60s	✗	7.06s	✗	7.45s	✗	7.02s	✗	6.57s	✗	7.50s
parity1.c	✔	6.55s	✔	19.52s	✔	6.76s	✔	6.11s	?	24.22s	✔	6.65s	**T**	300.97s	✔	80.68s	**T**	300.96s	**T**	300.96s
∑Time		242.14s		266.64s		285.59s		222.99s		189.62s		251.51s		517.86s		608.01s		522.57s		515.84s

3970X 32 Core CPU with 3.7GHz and 256GB RAM running Linux 5.4.65. We limited CPU time to 5 minutes, memory to 8GB, and restricted each run to two cores. We built ULTIMATE 0.2.1 from source[2] and used it as baseline.

The results are summarized in Table 3.1. Labels are ✔ for satisfied properties, ✗ for violated properties with counterexamples, and **?** for the unknown results where the tools could not decide. We also report timeouts (**T**), out-of-memory (**M**), crashes (☢), and highlight false positive (↯✗) and false negative (↯✔) results in gray, if any. ULTIMATE has two modes: *integer* and *bitvector*, each specialized to the corresponding kind of programs. In ULTIMATE's integer mode, overflow/underflow is accounted for with `assume` statements. In its bitvector mode, ULTIMATE can utilize a variety of back-end SMT solvers with internal bitvector reasoning strategies, such as CVC4, Z3, and MATHSAT (MS). By contrast,

[2] `github.com/ultimate-pa/ultimate`, `b4afca67`, dev

ULTIMATEBWB does not use bitvector mode but instead transforms bitvector programs (through bitwise branching) and verifies them in ULTIMATE's integer mode using the same set of back-end SMT solvers. Table 3.1 shows that the performance of the integer verification with bitwise branching is comparable to the bitvector verification, despite the fact that the bitwise branching transformation may introduce many new paths.

Because ULTIMATE's verification algorithms heavily utilize interpolation for optimizations, we also ran the experiment with interpolation enabled when possible, using MATHSAT's interpolation (MS Itp, in both modes) and SMTINTERPOL (SItp, only in the integer mode because SMTINTERPOL does not support bitvectors). Notably, without bitwise branching, ULTIMATE can only verify 2 of 28 programs using the default setting (SItp+Z3) in its integer mode while ULTIMATEBWB can verify all 28 programs in the same settings. Moreover, while interpolation is less effective in the bitvector mode (see MS Itp vs. MS), when combined with bitwise branching in the integer mode, it improves over those solvers (about 1.2x speedup, the total time for all benchmarks at the bottom row of Table 3.1, e.g. see total time, BwB MS Itp vs BwB SItp+Z3) and has the best results (BwB SItp+Z3 column).

3.4 Termination and LTL of Bitvector Programs

We now evaluate bitwise branching on the main target: liveness properties of bitvector programs. There are few comparable tools that support bitvector reasoning and these properties; the most comparable (and mature) tools are listed in Table 3.2, along with their limitations.

[2] `github.com/aprove-developers/aprove-releases/releases`, `master_2019_09_03`

[3] `github.com/s-falke/kittel-koat`, `c00d21f`, master

[4] `github.com/sosy-lab/cpachecker`,`c2f1d8cce6`, master

[5] `github.com/diffblue/2ls`, `d35ccf73`, master

Table 3.2: Static verification tools.

Tool	BitVec.	Term.	LTL
ULTIMATE	Limited	Yes	Yes
APROVE[3] [77]	Yes	Yes	No
KITTEL[4] [69]	Yes	Yes	No
CPACHECKER[5]	Limited	Yes	No
2LS[6] [39]	Yes	Yes	No
ULTIMATEBWB	Yes	Yes	Yes

Termination. We compare bitwise branching with the termination provers in the Table 3.2. that support bitvector arithmetic, i.e., CPACHECKER, KITTEL, APROVE, and 2LS. We used the latest release of APROVE. We build CPACHECKER 2.0.1 from source and also built 2LS from source. We applied these tools to two benchmarks suites: **(i)** We first used 18 bitvector terminating programs selected from APROVE's bitvector benchmarks [94]. Notably, those benchmarks were designed with general bitvector arithmetic in mind so that there is only a small portion of bitvector programs (with bitwise operations) in it (i.e. 18/118 or 15%). **(ii)** We therefore built a second set of 31 termination benchmarks, including 18 terminating programs (✔) and 13 non-terminating programs (✗), called TermBitBenchwith bitvector operations including bitwise |, &, ^, <<, >>, ~.

Results. Table 3.3 summarizes our results. In our tables here, rows are labeled with the number of verified properties (✔), provided counterexamples (✗), timeouts (**T**), crashes (☢), and the number of instances where the tool could not decide (**?**). We also report false positive (↯✗) and false negative (↯✔) results, if any, and highlight them in gray. For the APROVE benchmarks, our tool can correctly prove the termination or non-termination of 2 programs, which is less than the number of programs that can be proved by CPACHECKER (3), KITTEL (3), and 2LS (14). However, for TermBitBench, while ULTIMATEBWB can prove *all* 31 programs, CPACHECKER, KITTEL, and 2LS can only prove at most 16 programs. Moreover, while our tool was built on top of ULTIMATE, it outperforms ULTI-

Table 3.3: Termination results.

	(ii) TermBitBench						(i) AproveBench					
	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB
✔	5	1	7	8	2	18	1	3	3	14	2	2
↯✔	1	-	-	-	-	-	-	-	-	-	-	-
✗	6	10	-	8	-	13	-	-	-	-	-	-
↯✗	2	7	-	3	-	-	-	10	-	-	2	6
?	14	13	-	-	29	-	10	3	-	1	14	8
T	3	-	19	12	-	-	7	-	10	2	-	1
M	-	-	-	-	-	-	-	-	-	1	-	1
☢	-	-	5	-	-	-	-	2	5	-	-	-

MATE in proving termination and non-termination of bitwise programs. (details shown in Table 3.4 and Table 3.5). This is because ULTIMATE's algorithms for synthesizing termination [92] and non-termination proofs [113] are not applicable to SMT formulas containing bitvectors. As a consequence, ULTIMATE relies on integer-based encodings of source programs together with overapproximations of bitwise operations. These results confirm that bitwise branching provides an effective means for termination of bitvector programs. Note that there are 6 false results in AproveBench for termination, they are spurious counterexamples that arise due to Ultimate's overapproximation for unsigned integers, they do not involve branches created by our bitwise branching strategy.

Linear temporal logic. We compared our tool against ULTIMATE, which is the state-of-the-art LTL prover and the only mature LTL verifier that supports some bitvector programs. To our knowledge, there are no available bitwise benchmarks with LTL properties so we create new benchmarks for this purpose: (iii) New hand-crafted benchmarks

Table 3.4: Details for APROVE termination benchmarks.

		APROVE		CPACHECKER		KITTEL		2LS		ULTIMATE		ULTIMATEBWB	
Benchmark	Expected	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result
signed/wdk-signed-overflow/eeprom2.c	✔	2.03s	?	5.68s	✗	0.08s	☢	0.17s	✔	22.91s	?	130.32s	M
signed/wdk-signed-overflow/common.c	✔	2.12s	?	5.49s	✔	0.04s	✔	0.25s	✔	24.38s	?	900.44s	T
unsigned/juggernaut-paper/a.c	✔	2.37s	?	3.76s	✗	900.25s	T	0.15s	✔	5.50s	?	9.72s	✗
unsigned/pointer/p03.c	✔	2.05s	?	4.24s	?	0.04s	☢	0.13s	?	7.16s	?	5.60s	?
unsigned/wdk-no-signed-overflow/gsm6102.c	✔	900.26s	T	3.84s	✗	900.25s	T	900.38s	T	6.92s	✔	5.96s	✔
unsigned/wdk-no-signed-overflow/hw_ccmp.c	✔	4.30s	?	3.59s	☢	0.04s	✔	0.13s	✔	22.03s	?	14.77s	?
unsigned/wdk-no-signed-overflow/comm.c	✔	3.88s	✔	5.13s	✔	0.06s	☢	0.20s	✔	19.42s	?	22.91s	?
unsigned/wdk-no-signed-overflow/gsm6103.c	✔	900.26s	T	4.76s	✗	900.25s	T	900.37s	T	6.62s	✔	8.73s	✔
unsigned/wdk-no-signed-overflow/miniport.c	✔	900.26s	T	5.90s	?	900.25s	T	0.25s	✔	7.12s	✗	8.30s	✗
unsigned/wdk-no-signed-overflow/namesup2.c	✔	900.26s	T	3.76s	✗	0.03s	☢	0.59s	✔	7.67s	✗	4.94s	✗
unsigned/wdk-no-signed-overflow/eeprom.c	✔	6.71s	?	9.65s	✔	900.25s	T	0.13s	✔	16.79s	?	16.54s	?
unsigned/wdk-no-signed-overflow/intrface.c	✔	900.25s	T	3.77s	✗	900.15s	T	0.17s	✔	6.61s	?	5.23s	✗
unsigned/wdk-no-signed-overflow/init.c	✔	5.56s	?	3.69s	✗	900.14s	T	0.10s	✔	5.25s	?	13.11s	✗
unsigned/wdk-no-signed-overflow/namesup.c	✔	2.39s	?	2.01s	☢	0.05s	✔	185.56s	M	28.44s	?	27.67s	?
unsigned/wdk-no-signed-overflow/image.c	✔	900.26s	T	5.06s	✗	796.47s	T	1.24s	✔	14.35s	?	18.52s	?
unsigned/wdk-no-signed-overflow/mp_util.c	✔	3.82s	?	3.80s	?	900.27s	T	0.95s	✔	30.24s	?	26.87s	?
unsigned/wdk-no-signed-overflow/allocsup1.c	✔	900.26s	T	3.78s	✗	0.07s	☢	0.17s	✔	6.73s	?	5.62s	?
unsigned/juggernaut/loop6.c	✔	2.33s	?	3.91s	✗	900.25s	T	0.15s	✔	5.76s	?	7.55s	✗

called LTLBitBenchof 42 C programs with LTL properties, in which bitwise operations are heavily used in assignments, loop conditions, and branching conditions. There are 22 programs in which the provided LTL properties are satisfied (✔) and 20 programs in which the LTL properties are violated (✗). (iv) Benchmarks adapted from the "BitHacks" programs, consisting of 26 programs with LTL properties (18 satisfied and 8 violated).

ULTIMATE's configuration for LTL software model checking uses the configuration introduced in [61], with two differences. We disabled small block encoding, because our rules can introduce a large number of disjunctions which prevented verification with small block encoding in three instances. Because small block encoding also prevents the creation of `goto` edges in ULTIMATE's control-flow graph, and because the Büchi program product construction cannot deal with these edges, we had to enable their explicit removal.

The Table 3.6 summarizes the result of applying ULTIMATE and ULTIMATEBWB on these two bitvector benchmarks (see Table 3.7 and Table 3.8 for details). Besides the successful verification results for satisfied (✔) and violated (✗) LTL properties, we also report unknown (**?**) results, as well as timeout (**T**), out-of-memory (**M**), and crashes (☢). ULTIMATEBWB outperforms ULTIMATE: ULTIMATEBWB can successfully verify 41 of 42 programs in LTLBitBench and 18 of 26 BitHacks programs while ULTIMATE can only

Table 3.5: Details for TermBitBench.

		APROVE		CPACHECKER		KITTEL		2LS		ULTIMATE		ULTIMATEBWB	
Benchmark	Expected	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result
xor-01-false.c	✗	18.41s	✗	5.17s	✗	900.24s	T	900.53s	T	5.54s	?	6.39s	✗
and-04-false.c	✗	185.04s	✗	3.68s	✗	900.24s	T	0.15s	✗	5.33s	?	7.78s	✗
not-04-false.c	✗	4.14s	?	3.78s	✗	0.08s	☢	0.14s	✗	6.84s	?	6.78s	✗
not-05-false.c	✗	3.03s	?	4.64s	✗	0.10s	☢	0.13s	✗	5.50s	?	8.66s	✗
and-05-false.c	✗	10.11s	?	3.79s	✗	900.32s	T	0.16s	✗	5.60s	?	8.17s	✗
and-01-false.c	✗	4.83s	?	5.21s	?	900.24s	T	900.46s	T	6.27s	?	6.35s	✗
or-02-false.c	✗	13.66s	✗	3.93s	✗	900.28s	T	900.39s	T	7.40s	?	8.28s	✗
not-03-false.c	✗	2.95s	?	3.86s	✗	0.10s	☢	0.17s	✗	5.34s	?	7.25s	✗
and-03-false.c	✗	4.65s	✗	3.67s	✗	900.24s	T	0.14s	✗	5.29s	?	5.63s	✗
not-02-false.c	✗	1.90s	✔	3.84s	✗	0.05s	☢	0.15s	✗	5.26s	?	4.86s	✗
or-05-false.c	✗	6.29s	✗	4.18s	?	900.35s	T	0.16s	✗	6.07s	?	9.34s	✗
or-01-false.c	✗	5.76s	✗	3.73s	✗	900.31s	T	900.69s	T	7.64s	?	10.22s	✗
and-02-false.c	✗	6.30s	?	5.78s	?	900.24s	T	900.39s	T	6.72s	?	5.89s	✗
not-01.c	✔	902.06s	T	4.69s	?	0.06s	✔	0.39s	✔	6.54s	?	5.52s	✔
and-02.c	✔	901.23s	T	8.00s	?	900.25s	T	900.38s	T	6.05s	?	11.63s	✔
or-01.c	✔	23.72s	✔	6.50s	?	900.29s	T	900.41s	T	7.85s	?	13.10s	✔
xor-01.c	✔	4.14s	✔	4.77s	✔	0.05s	✔	0.31s	✔	7.16s	?	8.07s	✔
and-03.c	✔	12.83s	?	3.80s	✗	900.25s	T	900.44s	T	6.36s	?	6.98s	✔
not-02.c	✔	3.11s	?	4.03s	✗	0.31s	☢	0.18s	✗	5.09s	?	9.23s	✔
and-01.c	✔	5.74s	?	6.05s	?	900.25s	T	900.46s	T	6.04s	?	8.14s	✔
or-02.c	✔	10.16s	✔	4.14s	?	900.29s	T	900.46s	T	7.42s	?	6.56s	✔
or-03.c	✔	5.27s	✔	4.18s	?	0.04s	✔	0.24s	✔	5.83s	✔	6.94s	✔
not-03.c	✔	3.95s	?	5.32s	✗	0.06s	✔	0.27s	✔	5.55s	?	6.30s	✔
and-04.c	✔	901.56s	T	3.61s	✗	900.17s	T	0.21s	✔	5.50s	?	8.56s	✔
or-06.c	✔	7.48s	✔	4.22s	?	0.04s	✔	0.31s	✔	7.11s	✔	6.22s	✔
and-05.c	✔	4.40s	?	3.61s	✗	900.37s	T	0.19s	✔	7.04s	?	11.40s	✔
not-04.c	✔	2.39s	?	4.70s	?	0.10s	✔	900.42s	T	11.49s	?	12.85s	✔
or-04.c	✔	17.17s	✗	4.18s	✗	900.30s	T	0.15s	✗	8.13s	?	5.90s	✔
or-05.c	✔	5.12s	✗	6.44s	?	900.37s	T	0.16s	✗	7.22s	?	8.02s	✔
and-06.c	✔	10.12s	?	5.78s	?	900.25s	T	900.56s	T	7.24s	?	9.42s	✔
not-05.c	✔	3.21s	?	4.30s	✗	0.06s	✔	0.24s	✔	5.55s	?	5.88s	✔

handle a few of them. Note that we have more out-of-memory results in BitHacks Benchmarks, perhaps due to memory consumption reasoning about the introduced paths, however, in most of the case it helps the verification tool prove a wider range of bitvector programs effectively.

In conclusion, bitwise branching appears to be the first effective technique for verifying LTL properties of bitvector programs.

Table 3.6: LTL benchmarks experiment overview.

	(iv)Bithacks		(iii)LTLBit Bench	
	ULTIMATE	w. BWB	ULTIMATE	w. BWB
✔	3	10	-	21
✗	-	7	-	20
?	21	5	42	-
T	1	1	-	1
M	1	3	-	-

Table 3.7: Details for LTL Bithack benchmarks.

			ULTIMATE		ULTIMATEBWB	
Benchmark	Property	Expected	Time	Result	Time	Result
counting-bits-BK1_false.c	$\Box(\Diamond y >= 0)$	✗	8.80s	**?**	10.75s	✗
consecutive-zero-bits-trailing_false.c	$\Diamond y = 1$	✗	5.91s	**?**	7.24s	**?**
counting-bits-BK_false.c	$\Box(\Diamond y <= 1)$	✗	6.93s	**?**	8.15s	✗
display-bit1_false.c	$\Diamond y > 1$	✗	24.55s	**?**	59.18s	✗
parity_false.c	$\Diamond y >= 1$	✗	23.81s	**?**	9.09s	✗
display-bit_false.c	$\Diamond y < 0$	✗	27.20s	**?**	64.63s	✗
counting-bits-set_false.c	$\Diamond y >= 1$	✗	8.88s	**?**	7.77s	✗
reverse-bits1_false.c	$\Diamond n < 0$	✗	8.11s	**?**	10.21s	✗
logbase2.c	$\Diamond y >= 1$	✔	7.60s	**?**	7.49s	**?**
base64_ltl.c	$\Box(\Diamond start = 1)$	✔	124.37s	**M**	602.61s	**M**
modulus-division.c	$\Diamond y > 1$	✔	6.67s	**?**	17.25s	**?**
consecutive-zero-bits-trailing.c	$\Diamond y >= 1$	✔	6.07s	**?**	7.51s	✔
interleave-bits.c	$\Diamond y >= 1$	✔	11.40s	**?**	300.66s	**?**
logbase2-N-bit1.c	$\Diamond y >= 1$	✔	12.39s	**?**	547.43s	**M**
reverse-N-bit.c	$\Diamond n >= 1$	✔	6.91s	✔	13.09s	✔
counting-bits-set.c	$\Diamond y >= 1$	✔	9.23s	**?**	8.52s	✔
consecutive-zero-bits.c	$\Diamond y >= 1$	✔	5.29s	✔	5.85s	✔
counting-bits-BK.c	$\Box(\Diamond y <= 1)$	✔	6.63s	**?**	8.46s	✔
dropbf_ltl.c	$\Box(A! = 1 \vee RELEASE = 0)$	✔	8.42s	✔	13.47s	✔
reverse-bits.c	$\Diamond y >= 1$	✔	7.21s	**?**	10.81s	**?**
counting-bits-lookup.c	$\Diamond y >= 1$	✔	900.41s	**T**	900.41s	**T**
display-bit.c	$\Diamond y >= 32$	✔	24.99s	**?**	705.44s	**M**
counting-bits-BK1.c	$\Box(\Diamond y >= 0)$	✔	7.51s	**?**	8.59s	✔
display-bit1.c	$\Diamond y >= 1$	✔	20.46s	**?**	7.29s	✔
reverse-bits1.c	$\Diamond n < 0$	✔	6.33s	**?**	12.55s	✔
parity.c	$\Diamond y >= 1$	✔	19.67s	**?**	6.93s	✔

Table 3.8: Details for LTLBitBench.

Benchmark	Property	Expected	ULTIMATE Time	ULTIMATE Result	ULTIMATEBWB Time	ULTIMATEBWB Result
and_guard2_false.c	$\Diamond z \geq 100$	✗	7.54s	?	7.65s	✗
xor_stem1_false.c	$\Diamond n < 0$	✗	5.78s	?	6.41s	✗
and_guard_false.c	$\Diamond y > 0$	✗	7.11s	?	5.52s	✗
and_stem_false.c	$\Diamond n < 0$	✗	6.34s	?	5.86s	✗
xor_stem_false.c	$\Diamond n < 0$	✗	5.67s	?	6.33s	✗
xor_guard_false.c	$\Diamond n > 0$	✗	6.40s	?	6.24s	✗
or_loop1_false.c	$\Box(\Diamond n < 0)$	✗	6.98s	?	6.76s	✗
and_loop_false.c	$\Diamond y \geq 1$	✗	5.32s	?	9.28s	✗
and_stem1_false.c	$\Diamond n < 0$	✗	7.63s	?	5.75s	✗
xor_loop_false.c	$\Diamond n < 0$	✗	6.18s	?	8.01s	✗
or_guard_false.c	$\Diamond n < 0$	✗	5.82s	?	5.96s	✗
and_guard1_false.c	$\Diamond n > 0$	✗	7.82s	?	6.04s	✗
com_loop_false.c	$\Diamond y < 0$	✗	5.70s	?	6.81s	✗
or_stem_false.c	$\Diamond n < 0$	✗	7.11s	?	5.81s	✗
and_guard4_false.c	$\Diamond n > 0$	✗	8.62s	?	5.55s	✗
and_stem2_false.c	$\Diamond n > 0$	✗	7.69s	?	6.35s	✗
or_loop2_false.c	$\Diamond n > 0$	✗	6.47s	?	8.89s	✗
and_loop1_false.c	$\Diamond z < 0$	✗	9.22s	?	7.60s	✗
com_stem_false.c	$\Box(\Diamond y < 0)$	✗	8.89s	?	9.52s	✗
or_loop_false.c	$\Box(\Diamond n < 0)$	✗	7.60s	?	7.82s	✗
and_stem2.c	$\Diamond n > 0$	✔	5.81s	?	5.36s	✔
xor_stem1.c	$\Diamond n < 0$	✔	7.05s	?	5.76s	✔
xor_guard.c	$\Diamond n < 0$	✔	6.41s	?	5.67s	✔
and_guard4.c	$\Diamond n > 0$	✔	10.25s	?	6.00s	✔
or_stem.c	$\Diamond n < 0$	✔	5.38s	?	7.38s	✔
com_stem.c	$\Box(\Diamond y = 1)$	✔	6.50s	?	5.59s	✔
xor_stem.c	$\Diamond n < 0$	✔	5.40s	?	5.41s	✔
xor_loop.c	$\Diamond n < 0$	✔	6.14s	?	7.97s	✔
and_stem1.c	$\Diamond n < 0$	✔	7.05s	?	5.32s	✔
and_guard.c	$\Diamond n < 0$	✔	6.43s	?	901.21s	**T**
and_loop1.c	$\Diamond z < 0$	✔	7.63s	?	5.46s	✔
com_loop.c	$\Diamond y < 0$	✔	6.44s	?	5.39s	✔
or_loop.c	$\Box(\Diamond n < 0)$	✔	7.55s	?	10.05s	✔
and_guard2.c	$\Diamond z \geq 100$	✔	9.58s	?	17.04s	✔
or_loop2.c	$\Diamond n < 0$	✔	6.17s	?	6.33s	✔
and_stem.c	$\Diamond n < 0$	✔	5.49s	?	5.57s	✔
or_0s_int.c	$\Diamond p = 1$	✔	67.11s	?	9.38s	✔
and_loop.c	$\Diamond y \geq 1$	✔	6.15s	?	5.92s	✔
and_guard1.c	$\Diamond n > 0$	✔	7.81s	?	8.04s	✔
or_loop1.c	$\Box(\Diamond n < 0)$	✔	8.44s	?	6.58s	✔
or_loop3.c	$\Box(\Diamond n < 0)$	✔	7.47s	?	6.11s	✔
or_guard.c	$\Diamond n < 0$	✔	5.92s	?	4.52s	✔

Chapter 4

Temporal Verification of Decompiled Binaries

We have discussed that despite challenges in binary lifting, verifying decompiled program involves heavy reasoning in bitvector operations, in which our bitwise branching can be helpful within the scope of existing verification techniques. There are many types of representations for the lifted code (e.g. LLVM IR), our approach to verify binary programs is to decompile the binary and lift it to a much higher level representation, in our experiment we lift the assembly code to a `C` syntax program that can be parsed by standard libraries. Besides nested structures and other common syntax in standard `C`, a simplified statement syntax of our decompiled program is shown as following, $AStmt$ is a statement in our decompiled program that simulates the instruction from LLVM IR.

$$
\begin{array}{rcl}
Stmt & ::= & \texttt{do } \{\ Stmt\ \} \texttt{ while(1);} \\
& | & \texttt{if(}Expr\texttt{)}\ AStmt \texttt{ goto } Label \texttt{ else } AStmt \texttt{ goto } Label\texttt{;} \\
& | & Lhs\texttt{:=}Expr\texttt{;} \\
& | & \texttt{return;} \\
& | & Label\texttt{:}\ Stmt \\
& | & \texttt{goto } Label\texttt{;} \\
& | & Stmt\texttt{;}\ Stmt \\
AStmt & ::= & Lhs\texttt{:=}Expr\texttt{;} \ |\ AStmt\texttt{;}\ AStmt
\end{array}
$$

In this chapter, we first provide an in-depth case study of decompiled ("lifted") binary programs, which emulate X86 execution through frequent use of bitvector operations, we then summarize challenges in lifting binaries for verification, and we present transfor-

mations that are needed to re-target today's lifting tools from re-compilation to make them suitable for verification tools. With the implementation of our bitwise branching rules, we develop a new tool DARKSEA, the first tool capable of verifying reachability, termination, and LTL of lifted binaries, DARKSEA also provides complete binary decompilation from end to end.

4.1 Overview

In recent years many tools have been developed for decompiling (or "lifting") binaries into a source code format [34, 122, 14, 63, 168]. The resulting code, however, has lost the original source abstractions and instead emulates the hardware, making frequent use of bitvector operations. These challenging programs are beyond the capabilities of existing tools for LTL verification, making them an interesting case study and an important application for binary verification.

For example, consider the (source) program shown in Figure 4.1. This program, which does not contain any bitvector operations, is taken from the ULTIMATE repository[1]. Some existing techniques and tools (*e.g.* [49, 10]) can prove that the LTL property $\mathsf{G}(\mathtt{x} > 0 \Rightarrow \mathsf{F}(\mathtt{y} = 0))$ holds. However, after the program is compiled (with `gcc`) and then disassembled and lifted (with IDPro and MCSEMA), the resulting code has many bitvector

```
while(1) {
  y = 1; x = *;
  while (x>0) {
    x--;
    if (x <= 1)
    y = 0;
  }
}
```

Figure 4.1: An LTL example from ULTIMATE repository.

[1] http://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/LTL/simple/PotentialMinimizeSEVPABug.c

operations. The resulting lifted code is quite non-trivial (full version in Apx. A.2) and required substantial engineering efforts just to parse and analyze with existing verifiers (see Sec. 4.2). Let's first focus on the bitvector complexities; here is a fragment of the lifted IR (in C for readability):

```
1  while(true) {
2    tmp_x = load i32, i32* bitcast (%x_type* @x to i32*)
3    ...
4    if ( ((tmp_x >> 31) == 0) & ((tmp_x == 0) ^ true) )  {
5      tmp_40 = add i32 tmp_x, -1
6      store i32 tmp_40, i32* bitcast (%x_type* @x to i32*)
7      tmp_xp = load i32, i32* bitcast (%x_type* @x to i32*)
8      tmp_42 = tmp_xp + -1;  tmp_45 = tmp_42 >> 31;
9      tmp_43 = tmp_xp + -2;  tmp_44 = tmp_43 >> 31;
10     if (((((((tmp_42 != 0u)&1)) & ((((((tmp_44 == 0u)&1)) ^ ((((((tmp_44 ^ tmp_45) +
   tmp_45)) == 2u)&1)))&1)))&1))) {
11         store i32 0, i32* bitcast (%y_type* @y to i32*)
12     }
13   } else { break; }
14 }
```

Line 4 corresponds to the `x>0` comparison, and Line 10 corresponds to the `x<=1` comparison. These bitvector operations, introduced to emulate the behavior of the binary, make it challenging for existing verification tools. Our bitwise branching theory introduced in Chapter 3 can help overcome this challenge. We develop a new tool DARKSEA that uses bitwise branching in the context of a decompilation toolchain involving IDA PRO (used as disassembler), MCSEMA and ULTIMATE with our bitwise branching implemented. The lifting performed by tools like MCSEMA is geared toward *re*compilation rather than verification, thus foiling existing tools.

In Sec. 4.2, we introduce some background about binary de-compilation and demonstrate verification-oriented transformations in binary de-compilation with a detailed example. In Sec. 4.4, we summarize our translation strategy FABE in DARKSEA. We introduce

our binary verification tool DARKSEA design in Sec. 4.5, and show experimental results for the evaluation of DARKSEA.

4.2 LTL Verification of Decompiled Binaries

Decompiled binary executables are rife with bitvector operations, making them an interesting domain for the application of bitwise branching. Many tools [63, 11, 148, 73, 74, 101, 59] have been developed for decompilation. Similar to compilation, the decompilation process consists of multiple phases, beginning with disassembly. Some techniques have emerged for verifying low-level aspects of decompiled binaries such as architectural semantics [147, 57, 17], decompilation into logic [120, 121, 122, 168], and translation validation [56] (discussed in Chapter 1). Further along the decompilation process, other tools aim to represent a binary at a higher level of abstraction through a process called *lifting*. A lifted binary can be represented in IR or source code, but includes only some of the source-level abstractions of the original program. Instead, a lifted "program" emulates the machine itself, with data structures that mimic the hardware (*e.g.* registers, flags, stack, heap, etc.) and control that mimics the behavior of the binary. To a large extent today's automated verification techniques have relied on source abstractions (*e.g.* invariants and rank functions over loop variables, structured control flow, procedure boundaries, etc.). A principal challenge in verifying lifted binaries is that the target is an emulated machine, therefore the decompiled "program" involves complications such as irrelevant code for emulating the environment, data structures for emulating the architecture, and lifted binaries frequently use bitvector operations *e.g.* to reflect signed/unsigned comparison of variables whose type was lost in the compilation.

As we show in Sec. 4.5, lifted programs are beyond the capabilities of termination verification tools (ULTIMATE [89], CPACHECKER [152], APROVE [77] or KITTEL [68]).

Even though some lifting tools emit programs in familiar languages (LLVM IR or C), those programs employ many bitwise operations and complex (often irrelevant) data and control that place them beyond, using available verification tools can provide a wide range of verification techniques that are already implemented and tested, therefore we can shift our focus to the defects of these techniques, i.e. the verification target has bitvector operations and other complications.

Returning to previous our previous example in Section 4.1, while the source code for the inner loop of the program in Fig. 4.1 is straight-forward (decrementing `x` and assigning 0 to `y` if `x <= 1`), the corresponding expressions in the lifted binaries involve multiple bitvector operations:

```
(((tmp_42 != 0u)&1) & ((((tmp_44 == 0u)&1 ^
(((((tmp_44 ^ tmp_45) + tmp_45) == 2u)&1)))&1)))&1
```

This expression simulates branch comparisons that the machine would perform on values whose type was discarded during compilation. The source code variable `x` is a signed integer, but the compilation has stripped its type. During the decompilation, to approximate the origin code from binary, lifting procedures consider these `tmp` variables (and all integer variables) to be unsigned. Meanwhile, in the binary, the condition `x<=0` is compiled to be a *signed* comparison. Therefore, lifting recreates a signed comparison using the unsigned `tmp` variables.

In this scenario, lifted binaries are good candidates for bitwise branching. For the above example, we can use three rules: R-RShift-Pos, R-And-1, R-And-Log. The variable `tmp_x` (*i.e.*`x` loaded from memory) is a signed integer but, corresponds to the condition "x > 0" in the original source code, where "x" is a "signed integer"; when the original source code is translated into binary code, type of `tmp_x` is stripped and thus, after decompilation,

the type of `tmp_x` is unknown. To approximate, lifting procedures consider `tmp_x` (and all integers) to be unsigned (McSema considers every integer unsigned). Meanwhile, in the binary, the *condition* `x>0` is compiled to be a *signed* comparison. Therefore, binary lifting will create a signed comparison using the unsigned version of `tmp_x`. The decompiled code of the inner loop of this program as shown in Section 4.1, notice that the comparison `x>0` is instead represented on Line 4 as:

```
((tmp_x >> 31) == 0) & ((tmp_x == 0) ^ true)
```

Above, the `if` statement on Line 4 is a bitwise calculation equivalent to `x>0`.

The variable `tmp_x` (*i.e.*`x` loaded from memory) is a signed integer but, corresponds to the condition "x > 0" in the original source code, where "x" is a "signed integer"; when the original source code is translated into binary code, the type of `tmp_x` is stripped and thus, after decompilation, the type of `tmp_x` is unknown. To approximate, lifting procedures consider `tmp_x` (and all integers) to be unsigned (McSema considers every integer unsigned). Meanwhile, in the binary, the *condition* `x>0` is compiled to be a *signed* comparison. Therefore, binary lifting will create a signed comparison using the unsigned version of `tmp_x`. Roughly, the lifting process proceeds as:

```
(int)tmp_x > 0  ⇝ tmp_x != 0 & tmp_x <= 0x7fffffff
                ⇝ tmp_x != 0 & (tmp_x>>31) == 0
                ⇝ tmp_x != 0 & ( ((tmp_x>>31) == 0) ^ 1)
```

In lifted programs, the operands of bitwise expressions often involve subexpressions whose values are 0, 1, -1, MAXINT, etc. Moreover, the results of bitwise expressions are also often simple (compared to the structure of source code). Notice that in the above example, `tmp_44` is the 31. Consequently, we can often avoid bitwise comparisons altogether by case

splitting on values or ranges of values for the operands and constraining the corresponding bitwise result accordingly.

4.3 Verification Oriented Translations for Decompiled Binaries

We here explain via previous example the need for the translations, introduced in Sec. 4.5 for evaluation. Figure 4.2 shows an (condensed) example of the result of lifting a GCC-compiled binary version of program in Figure 4.1 (using McSema). In the following, we describe, in detail, how decompilation tools (*e.g.* McSema) target *re-compilation* and the challenges this poses for existing verification techniques and tools. We then describe translations performed by DarkSea to make lifted binaries more amenable to verification. As it will be discussed in Sec. 4.4, the translations below were implemented as passes on the lifted LLVM IR.

Run-time environment. Binary lifting de-compiles into a program that mimics the binary behavior. To ensure that a new *re*-compiled binary would run correctly, lifting yields code that switches the contexts between the run-time environments and the simulated code, somewhat akin to how a loader first moves environment variables onto the stack. This context-switch code wraps around the simulated program and is indispensable for execution of re-compiled code. Context-switch code can be fairly complex. For example, it frequently uses assembly code to move values between the physical environment (*e.g.* environment variables like `$PATH`) and the simulated constructs (*e.g.* registers and the stack). The initial state of the registers is also loaded from the runtime, as seen by the instructions in callout ❶ in Fig. 4.2. Tracing back the origins of these values, which stem from the runtime code, poses a nearly impossible, yet unnecessary task for verification tools. For most verification tasks, it does not matter where the initial values of registers or environment variables such as `$PATH` come from; we can just treat them as nondeterministic input

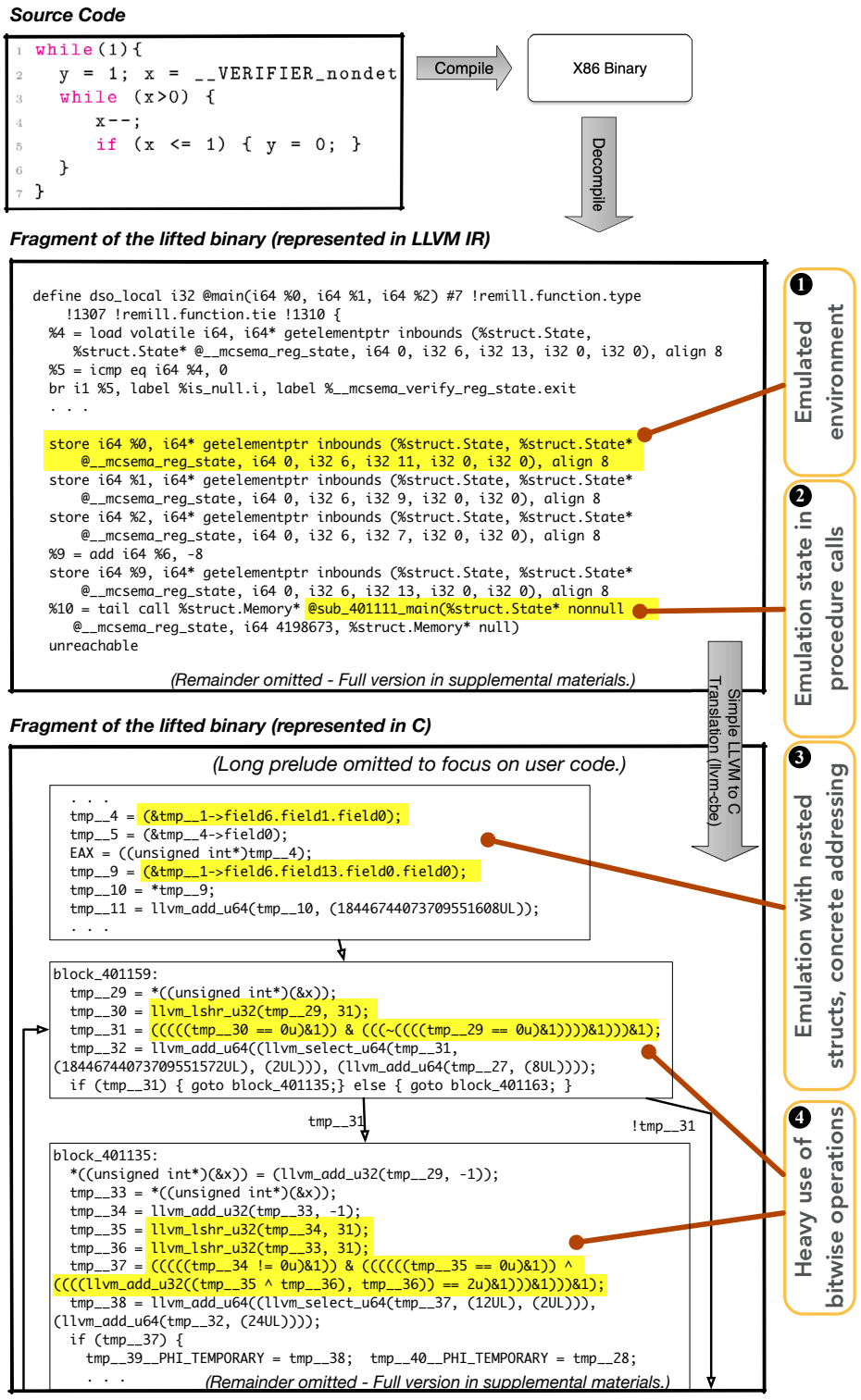


Figure 4.2: Challenges involved in reasoning about the lifted binary of the program in Fig. 4.1.

values on the stack.

Translation: Removing verification-unrelated code and data. We implemented a pass to analyze lifted output and decouple context-switch code from the code that simulates the original program. We first locate the original `main` function in the simulated code and then follow the control flow to statically extract and trim code that can reach `main`. Further, the context-switch code also includes program-dependent functions that are registered to be executed before the `main` or after the `exit`. To avoid missing such functions in verification, we allocate calls to them at the beginning or the end of the `main` function, following the order these functions are registered to run in the original binary.

Passing emulation state through procedures. Binary lifting for recompilation generates programs in which function calls are used to pass the emulation state. This can be seen, for example, in callout ❷ in Fig. 4.2, where `struct.State` is passed as the first argument to `sub_401111_main`. These arguments are *not* part of the original program. Rather, lifting introduces these additional arguments to simulate the possibility of context-switches within function calls. Further, when the lifted code is recompiled, code simulating the callee can access the contexts through these `struct.State` arguments, these arguments (corresponding to source code arguments) are not pointers pointing to different fields of the global `struct.state`, which, is in running time typically, more efficient than directly accessing the global data structure.

Interprocedural reasoning is known to make verification more challenging and requires more sophisticated algorithms such as procedure summaries [172, 62] and nested interpolants [91], especially when context sensitivity is required. When machine emulation involves expanding the use of arguments, we found this complicates analysis and hampers verification.

Translation: Simplifying function arguments. Adding arguments to user procedures (as done by McSema) complicates verification. Fortunately, a translation is possible: the

arguments in a McSema-generated function point to the same global data structure. As such, we eliminate these arguments from every function call. We then create a pointer pointing to the global data struct and replace all *uses* of the first argument in the function body with uses of our new pointer.

Nested structures for emulation. Lifted binaries encode complicated structures that simulate hardware features such as registers, arithmetic flags, FPU status flags, the stack, and instruction pointers. These are represented as nested structures, *e.g.*`state->general_registers.register13.union.uint64cell`. This can be seen, for example, in callout ❸ in Fig. 4.2, where the field `tmp__1->field6.field13.field0.field0` is accessed. The use of nesting in these structures provides efficiency: constructs that are commonly used together (*e.g.* general purpose registers) can be artificially grouped to the same cache line, avoiding cache evictions. These nested structures mirror the computer organization and also simplify the management of constructs during lifting. However, reasoning about these nested data structures is difficult because verification tools cannot make any assumptions about where these data-structures come from, how they are used, and most importantly, how they may be aliased. Consequently, verification tools have to carefully track heap references to infer non-aliasing, even though the lifting process ensures that they will not.

Translation: Flattening the emulation state. Many of the data structures for the emulated state are functionally independent and hence the complex nesting is not necessary to maintain the original semantics. We implemented a pass to flatten the data structures. We create individual variables for all the innermost and separable fields. We then translate accesses to these nested structures, with use-define reasoning to identify all the accesses to a flattened field. For the aforementioned `state->general_registers.register13.union.uint64cell`, we allocate a new global variable `register13` with the same type of `uint64cell` and re-locate

all the original accesses to `register13`.

Other challenges. Finally, lifted binaries pose other quirks that needed to be addressed before they can be verified. We encountered many of them and needed to perform additional analyses, translations, slicing, etc, that we will discuss in Section 4.4. For example, some of the lifted code is irrelevant to the property of interest. Slicing, which is already useful in verifying source code, is even more essential for lifted binaries. Another case is that, lifted binaries often involve type-casting short-cuts to enable efficient re-compilation, such as the following:

```
1 struct OC_a{uint64_t f0;}
2 int func(){
3   struct OC_a tmp;
4   uint64_t* t_ptr;
5   t_ptr=(uint64_t*)&tmp;
6 }
7
```

```
1 struct OC_a{uint64_t f0;}
2 int func() {
3  struct OC_a tmp;
4  uint64_t* t_ptr;
5  t_ptr=(uint64_t*)(&tmp.f0);
6 }
7
```

Here, `t_ptr` is supposed to point to the first field in `tmp`. Lifting can take a short-cut to make `t_ptr` directly point to `tmp` because the first field in `tmp` and `tmp` itself have identical memory addresses. If not correctly used, type casting may return unsafe and incorrectly casted values, violating origin source code intended pointer semantics, and also increasing pointer analysis in verification. Our fixing eliminates such short-cut to ensure safety of type-casting, these types of short-cut can typically ensure the correctness but reduce certain operations. We found that verification tools often cannot treat the code on the left and lose information.

Another example is shown below in Figure 4.3, shows a lifted program before and after we replace `tmp1->field6` with `g_ptr->field6`. We implemented a pass to

Before Simplifying

```
1  struct OC_State g_state;
2  int main(){
3   struct OC_State *tmp = &g_state
4   foo(tmp, 0, 0); //arg passing
5   return 0;
6  }
7  void foo(OC_State* tmp1, uint64_t tmp2,
       void* tmp3){
8   struct OC_anon* tmp4;
9   tmp4 = (&tmp1->field6);
10  error(tmp1,tmp2,tmp3);
11 }
```

After Simplifying

```
1  struct OC_State g_state;
2  struct OC_State* g_ptr // ptr
3   = &g_state;
4  int main(){
5   foo();  // args. removed
6   return 0;
7  }
8  void foo(){
9   struct OC_anon* tmp4;
10  tmp4 = (g_ptr->field6); //global
11  error();  // args. removed
12 }
```

Figure 4.3: Example showing our argument removal. In the `main` function before our simplification, `tmp`, which points to a global data structure `g_state`, is passed to the `foo` function and its alias `tmp1` is further passed to `error`. After our simplification, all the arguments are removed, and the accesses to `tmp` and `tmp1` are fixed.

revoke the type-casting shortcuts described above, making the dereference more explicit. McSema also often brings redundant type-casting. For instance, it can create operations like `*((int*)(&p))` even if `p` has the type of `int`. DarkSea strips redundant type-casting, in a way such as changing `*((int*)(&p))` to `*(&p)` and then `p`.

4.4 DarkSea: A Toolchain for Temporal Verification of Lifted Binaries

Bitvector operations are not the only issue: lifted binaries have several other wrinkles (as we discussed in previous sections) that preclude them from being verified with today's tools. We address them in a new toolchain called DarkSea, that is capable of verifying reachability, termination, and LTL properties of lifted binaries. DarkSea is comprised of several components, depicted in the following diagram:

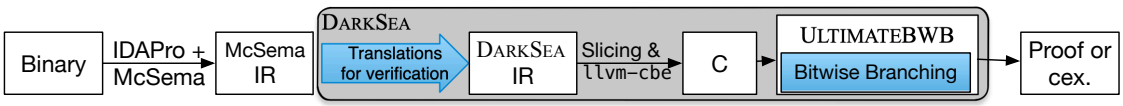


Figure 4.4: The work-flow of our de-compilation.

DARKSEA takes as input a lifted binary, internally call a disassembler IDA PRO for disassembly, and calls MCSEMA as a lifting tool works on disassembly, with various code recovery and control flow reconstruction techniques shipped with these tools, disassembly code is lifted and mapped into LLVM IR format. DARKSEA implements various transformation passes (in Sec. 4.3) processing the decompiled IR, and we have our IR that is friendly for verification, with property-driven program slicing, we can further slice down the transformed IR (DARKSEA IR), which finally can be converted to `C` via `llvm-cbe`. The final step in DARKSEA is calling verification tool, ULTIMATEBWB with our bitwise branching, performing verification tasks, and reporting the verification results.

4.4.1 FABE in DARKSEA

Lifting tools like MCSEMA [14, 63] are often designed with the goal of *re-compilation* rather than verification. Consequently, the MCSEMA IR, even if converted to C, cannot be analyzed by existing tools (see Sec. 4.5) which either crash, timeout, out of memory, or fail during parsing. We therefore perform a series of translations discussed below to re-target the lifted binaries into a format more amenable to verification, which we then input to ULTIMATEBWB. We now introduce our techniques learned from previous section to enable automated termination and temporal verification of lifted binaries. To this end, we first aim to marry the realistic capabilities of the lifting process with the requirements needed for verifying temporal properties. We propose *flat abstract binary emulation* (FABE), a lifting format in which the emulation of the machine is more amenable to verification. FABE places requirements on how emulation data structures must be organized, how procedures can be used, and how the environment (*i.e.* non-user code) must be treated. We then describe translations from the output of an existing lifting tool [63] into FABE format. FABE differs from existing lifted output and is also distinct from compilation-side IRs. For example, FABE excludes lifted output that involves aspects of the machine emulation

that are needed for *re*-compilation but are not part of the original program. Additionally, unlike compilation-side IRs, procedure calls are not permitted, even if they were present in the original program source, this mitigates the burden of inter-procedure analysis especially having pointer arguments in it. FABE differs from compilation-chain IRs for a few reasons, notably that procedure calls are not permitted. Intuitively FABE thus allows multiple benefits in the community of safety/termination/temporal verification tools. We show that FABE enjoys practical benefits akin to traditional IRs such as serving as a common target format for lifting (*e.g.* from x86 or amd64), common source format for verification (*e.g.* ULTIMATE, CPACHECKER, APROVE, KITTEL), and a unified format for analysis.

Furthermore, since it is impossible in general to re-create the original source, lifting techniques instead generate a program that simulates execution of binary code in the target machine architecture. These tools thus generate data structures that represent components such as registers (*e.g.* RBP, RSP, IP), arithmetic flags, the stack (via an array), the heap, etc. In our experience, we found four impediments to applying today's verification tools on lifted binaries. In the following, we summarize the impediment and describe translations performed by DARKSEA to address the issue (as details discussed in Sec. 4.3). FABE translations summarized below work with LLVM-8.0 and consist of around 500 lines of C++ and 200 lines of `bash`. We also identified and fixed several defects in MCSEMA [3, 2, 5].

1. *Run-time environment.* For *re*-compilation, lifting yields code that switches context between the run-time environments and the simulated code, akin to how a loader moves environment variables onto the stack. The first pass of DARKSEA analyzes lifted output to discover the original program's `main`, decouples the surrounding context-switch code, and removes it from the code that simulates the original program, starting from `main`.

2. *Passing emulation state through procedures.* McSema generates lifted programs in which function arguments pass emulation state that is used for re-compilation. When the lifted code is recompiled, code simulating the callee can access the contexts through these `struct.State` arguments, which is typically more efficient than directly accessing the global data structure. As this burdens verification analysis on the lifted programs, another pass implemented in DarkSea eliminates these arguments from every function call.

3. *Nested structures.* Lifted binaries simulate hardware features (*e.g.* registers, arithmetic flags, FPU status flags) and, for cache efficiency, represent them as nested structures. DarkSea flattens these nested data structures, creating individual variables for all the innermost and separable fields, and then translates accesses to these nested structures.

4. *Property-directed slicing.* Not all the instructions are relevant to the properties we aim to verify, so we further slice the program to keep only property-dependent code, using DG [38] (an open source library for LLVM program analysis) in termination-sensitive mode. For LTL properties, we use the atomic propositions' variables to seed our slicing criteria. Given slicing criteria (*e.g.* a use of a global variable), DG identifies both control-flow dependent code and data-flow dependent code, using the termination-sensitive mode to ensure the soundness of slicing (*i.e.* all code that can effect execution at the slicing criteria are identified).

4.5 Evaluation

We evaluated whether our translations (Sec. 4.2) and bitwise branching (Sec. 3.2) enabled tools to verify termination and LTL properties of decompiled binaries. The `example*` programs are taken from real bugs in GCC optimizations [6, 7, 144] (see Apx. A.3).

Table 4.1: Termination of Lifted Binaries, with and without DARKSEA translations.

	Raw MCSEMA						DARKSEA **transl.**					
	APROVE	CPACHECKER	KITTEL	2LS	ULTIMATE	ULTIMATEBWB	APROVE	CPACHECKER	KITTEL	2LS	ULTIMATE	ULTIMATEBWB
✔ (Satisfied)	-	-	-	-	-	-	-	-	-	-	18	18
☢ (Crashed)	-	18	-	-	3	-	-	-	-	-	-	-
M (Out of Memory)	-	-	-	-	-	3	-	-	-	-	-	-
T (Timeout)	-	-	18	-	15	15	-	18	18	-	-	-
?(Unknown)	18	-	-	18	-	-	18	-	-	18	-	-

4.5.1 Termination of lifted binaries

As discussed in Sec. 4.2, there are several termination provers that support bitvector programs. We thus applied those termination provers to today's lifting results on both the raw output of MCSEMA and then on the output of our translation. We used a standard termination benchmark (*i.e.* 18 small, but challenging programs in literature selected from the SV-COMP `termination-crafted` benchmark). As discussed in Sec. 4.2, lifted code is more complicated than its corresponding source (*e.g.* >10k vs 533 LOC in total). Although today's termination provers can verify the source of these programs, they struggle to analyze the corresponding code lifted from the programs' binaries, as seen in the **Raw MCSEMA** columns in Table 4.1.

We devoted genuine effort to overcome small hurdles but, fundamentally, without the DARKSEA translations, tools struggled for the following reasons:

- APROVE: Errors in conversion from LLVM IR to the internal representation.

Table 4.2: Details for termination verification of vanilla McSema binary lifting.

		AProVE		CPAchecker		KITTeL		2LS		Ultimate		DarkSea	
Benchmark	Expected	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result
Singapore-2_gccO0.mcsema.cbe.c	✔	8.28s	?	2.04s	☢	0.06s	☢	0.17s	?	900.43s	T	900.36s	T
aaron2-2_gccO0.mcsema.cbe.c	✔	5.48s	?	2.13s	☢	0.06s	☢	0.17s	?	900.37s	T	900.43s	T
Singapore_plus_gccO0.mcsema.cbe.c	✔	4.13s	?	2.58s	☢	0.06s	☢	0.17s	?	900.45s	T	900.56s	T
Mysore-1_gccO0.mcsema.cbe.c	✔	3.77s	?	1.98s	☢	0.05s	☢	0.16s	?	900.35s	T	900.39s	T
Parallel_gccO0.mcsema.cbe.c	✔	3.01s	?	2.12s	☢	0.06s	☢	0.15s	?	900.38s	T	900.33s	T
Pure2Phase-1_gccO0.mcsema.cbe.c	✔	8.60s	?	2.50s	☢	0.05s	☢	0.16s	?	900.38s	T	900.44s	T
Thun-1_gccO0.mcsema.cbe.c	✔	3.55s	?	2.41s	☢	0.05s	☢	0.15s	?	900.44s	T	900.42s	T
easy2-2_gccO0.mcsema.cbe.c	✔	2.72s	?	2.22s	☢	0.05s	☢	0.16s	?	589.64s	☢	760.59s	M
aaron3-2_gccO0.mcsema.cbe.c	✔	9.82s	?	2.44s	☢	0.06s	☢	0.20s	?	900.41s	T	900.47s	T
Pure3Phase-2_gccO0.mcsema.cbe.c	✔	5.85s	?	2.54s	☢	0.06s	☢	0.22s	?	900.40s	T	900.43s	T
easy_debug_gccO0.mcsema.cbe.c	✔	3.69s	?	2.28s	☢	0.06s	☢	0.19s	?	900.40s	T	902.27s	T
Mysore-2_gccO0.mcsema.cbe.c	✔	7.70s	?	2.03s	☢	0.06s	☢	0.17s	?	900.43s	T	900.49s	T
easy1_gccO0.mcsema.cbe.c	✔	2.92s	?	2.24s	☢	0.05s	☢	0.16s	?	785.05s	☢	802.15s	M
aaron2-1_gccO0.mcsema.cbe.c	✔	9.27s	?	2.15s	☢	0.05s	☢	0.16s	?	900.31s	T	900.37s	T
easy2-1_gccO0.mcsema.cbe.c	✔	2.60s	?	1.83s	☢	0.02s	☢	0.11s	?	566.99s	☢	870.64s	M
Thun-2_gccO0.mcsema.cbe.c	✔	5.91s	?	2.41s	☢	0.03s	☢	0.16s	?	900.36s	T	900.45s	T
Pure2Phase-2_gccO0.mcsema.cbe.c	✔	4.57s	?	1.90s	☢	0.02s	☢	0.13s	?	900.38s	T	900.35s	T
aaron3-1_gccO0.mcsema.cbe.c	✔	46.87s	?	1.91s	☢	0.03s	☢	0.13s	?	900.42s	T	900.44s	T

- KITTeL: Parsing (from C to KITTeL's format via LLVM bitcode with LLVM2KITTeL) succeeded, but then KITTeL silently hung until timeout.
- CPAchecker: Crashes on all benchmarks, while parsing system headers.
- Ultimate: Crashes on 3 benchmarks, due to inconsistent type exceptions.

Table 4.1 also shows the verification results of those termination provers when applied to DarkSea's translated output (second set of columns).

Table 4.2 and Table 4.3 show the details of applying termination verifiers to McSema output and the output after DarkSea's translations, respectively.

In sum, the results show that our translations benefit both CPAchecker and Ultimate (which already have sophisticated parsers), reducing crashes in analyzing lifted code. As highlighted in green, DarkSea translations enabled Ultimate to prove termination on all of the 18 lifted programs, as compared to Ultimate timing out on 15 of the programs without DarkSea's translations.

Table 4.3: Details for termination verification of DARKSEA translated lifted binaries.

		APROVE		CPACHECKER		KITTEL		2LS		ULTIMATE		DARKSEA	
Benchmark	Expected	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result	Time	Result
Parallel_gccO0.simplify.cbe.c.instr.c	✔	1.85s	?	900.51s	T	0.01s	☢	0.12s	?	8.14s	✔	7.27s	✔
Thun-1_gccO0.simplify.cbe.c.instr.c	✔	1.63s	?	900.55s	T	0.01s	☢	0.15s	?	9.68s	✔	9.95s	✔
aaron2-1_gccO0.simplify.cbe.c.instr.c	✔	1.43s	?	900.48s	T	0.02s	☢	0.12s	?	7.91s	✔	10.32s	✔
Pure3Phase-2_gccO0.simplify.cbe.c.instr.c	✔	1.86s	?	900.43s	T	0.02s	☢	0.12s	?	8.34s	✔	9.44s	✔
Mysore-2_gccO0.simplify.cbe.c.instr.c	✔	1.40s	?	900.38s	T	0.02s	☢	0.15s	?	7.03s	✔	10.25s	✔
easy1_gccO0.simplify.cbe.c.instr.c	✔	1.92s	?	900.29s	T	0.02s	☢	0.11s	?	6.47s	✔	7.79s	✔
Pure2Phase-1_gccO0.simplify.cbe.c.instr.c	✔	1.58s	?	900.35s	T	0.02s	☢	0.12s	?	7.19s	✔	8.00s	✔
aaron3-2_gccO0.simplify.cbe.c.instr.c	✔	1.88s	?	900.40s	T	0.02s	☢	0.12s	?	7.22s	✔	7.54s	✔
easy2-1_gccO0.simplify.cbe.c.instr.c	✔	1.63s	?	900.44s	T	0.01s	☢	0.12s	?	6.73s	✔	6.81s	✔
aaron3-1_gccO0.simplify.cbe.c.instr.c	✔	1.46s	?	900.45s	T	0.02s	☢	0.14s	?	8.09s	✔	12.30s	✔
Pure2Phase-2_gccO0.simplify.cbe.c.instr.c	✔	1.87s	?	900.48s	T	0.02s	☢	0.09s	?	8.09s	✔	6.96s	✔
easy2-2_gccO0.simplify.cbe.c.instr.c	✔	1.79s	?	900.47s	T	0.02s	☢	0.09s	?	8.36s	✔	6.40s	✔
Mysore-1_gccO0.simplify.cbe.c.instr.c	✔	1.95s	?	900.41s	T	0.01s	☢	0.10s	?	6.55s	✔	8.63s	✔
aaron2-2_gccO0.simplify.cbe.c.instr.c	✔	1.48s	?	900.55s	T	0.01s	☢	0.11s	?	8.19s	✔	8.87s	✔
easy_debug_gccO0.simplify.cbe.c.instr.c	✔	1.52s	?	900.54s	T	0.02s	☢	0.10s	?	8.16s	✔	9.27s	✔
Thun-2_gccO0.simplify.cbe.c.instr.c	✔	1.88s	?	900.55s	T	0.02s	☢	0.10s	?	7.02s	✔	7.71s	✔
Singapore_plus_gccO0.simplify.cbe.c.instr.c	✔	1.83s	?	900.56s	T	0.01s	☢	0.15s	?	6.96s	✔	6.95s	✔
Singapore-2_gccO0.simplify.cbe.c.instr.c	✔	1.45s	?	900.51s	T	0.02s	☢	0.10s	?	7.05s	✔	7.68s	✔

4.5.2 LTL of lifted binaries

We finally evaluate the effectiveness of DARKSEA in proving LTL properties of 8 lifted binaries. Each benchmark has two corresponding lifted programs: the raw output of MCSEMA and its simplified version by our FABE translation. In Table 4.4 we report the LTL property and expected verification result of each benchmark, as well as the verification time and result of ULTIMATE and DARKSEA on them . Some examples are termination properties (*i.e.* $\mathsf{F}p$), some are reachability properties (*i.e.* $\mathsf{G}p$) and some are more expressive LTL properties such as $\mathsf{G}(p \Rightarrow \mathsf{F}q)$. For each benchmark, we lifted both correct and incorrect binary, for which the LTL property $\square(error = 0)$ (indicating that the error is unreachable) holds and does not hold, respectively. Green cells use slightly different settings (enabled SBE, there are various setting strategies in ULTIMATE framework). DARKSEA's translations eliminate unsoundness/crashed results that come from applying ULTIMATE directly to MCSEMA IR.

A more detailed version is shown in Table 4.5, comparing the performance of ULTIMATE versus DARKSEA, when applied first to vanilla MCSEMA IR, and then applied to DARKSEA's translated IR. The experimental result shows that our translations signif-

Table 4.4: ULTIMATE vs. DARKSEA on lifted programs with LTL properties.

Benchmark	Property	Exp.	ULTIMATE Time	ULTIMATE Result	DARKSEA Time	DARKSEA Result
01-exsec2.s.c	$\Diamond(\Box x = 1)$	✔	4.45s	☢	11.23s	✔
01-exsec2.s.f.c.c	$\Diamond(\Box x \neq 1)$	✗	6.31s	☢	10.36s	✗
SEVPA_gccO0.s.c	$\Box(x > 0 \Rightarrow \Diamond y = 0)$	✔	6.31s	☢	22.92s	✔
SEVPA_gccO0.s.f.c	$\Box(x > 0 \Rightarrow \Diamond y = 2)$	✗	5.16s	?	14.92s	✗
acqrel.simplify.s.c	$\Box(x = 0 \Rightarrow \Diamond y = 0)$	✔	5.17s	☢	9.00s	✔
acqrel.simplify.s.f.c.c	$\Box(x = 0 \Rightarrow \Diamond y = 1)$	✗	6.06s	☢	17.60s	✗
exsec2.simplify.s.c	$\Box\Diamond x = 1$	✔	4.92s	☢	5.60s	✔
exsec2.simplify.s.f.c.c	$\Box\Diamond x \neq 1$	✗	4.55s	☢	6.28s	✗

icantly eliminate all crashes and possibly unsound results occurring in the verification of the original lifted code. The unsoundness came from the fact that ULTIMATE detected possible memory errors in the code snippet setting the run-time environment up in those programs, thus considering the programs to be infeasible with respect to the LTL properties and assuming that they always hold. DARKSEA inherits such behavior from ULTIMATE. Moreover, these results on the simplified lifted code highlight the effectiveness of our bit-wise branching technique, which helps DARKSEA to prove the LTL properties of all 17 benchmarks correctly while ULTIMATE can only prove 6 of them. The possible memory errors were eliminated when our translation flattened the emulation state.

In summary, we have shown that DARKSEA can verify reachability, termination, and LTL properties of lifted binaries. To our knowledge, DARKSEA is the first to do so.

Table 4.5: Details for LTL lifted binary benchmarks, using vanilla McSema versus DarkSea's translated IR and vanilla Ultimate versus DarkSea's bitwise-branching (Section 3.2). Gray cells are unsound, green cells use slightly different settings (enabled SBE).

			Vanilla McSema IR				DarkSea's translated IR			
			Ultimate		DarkSea		Ultimate		DarkSea	
Benchmark	Property	Exp.	Time	Result	Time	Result	Time	Result	Time	Result
01-exsec2.s.c	$\Diamond(\Box x = 1)$	✔	4.45s	☢	4.56s	☢	11.01s	✔	11.23s	✔
01-exsec2.s.f.c.c	$\Diamond(\Box x \neq 1)$	✗	6.31s	☢	5.79s	☢	9.21s	?	10.36s	✗
SEVPA_gccO0.s.c	$\Box(x > 0 \Rightarrow \Diamond y = 0)$	✔	6.31s	☢	5.93s	☢	11.17s	?	22.92s	✔
SEVPA_gccO0.s.f.c	$\Box(x > 0 \Rightarrow \Diamond y = 2)$	✗	5.16s	?	5.25s	?	35.46s	?	14.92s	✗
acqrel.simplify.s.c	$\Box(x = 0 \Rightarrow \Diamond y = 0)$	✔	5.17s	☢	5.38s	☢	10.66s	✔	9.00s	✔
acqrel.simplify.s.f.c.c	$\Box(x = 0 \Rightarrow \Diamond y = 1)$	✗	6.06s	☢	5.48s	☢	21.12s	?	17.60s	✗
example1_fea.s.c	$\Box error = 0$	✗	13.06s	✔	13.44s	✔	6.96s	✗	6.25s	✗
example1_sea.s.c	$\Box error = 0$	✔	11.22s	✔	11.11s	✔	12.02s	✔	8.51s	✔
example2_fea.s.c	$\Box error = 0$	✗	12.26s	✔	13.54s	✔	10.99s	?	12.66s	✗
example2_sea.s.c	$\Box error = 0$	✔	14.24s	✔	14.15s	✔	11.75s	?	8.94s	✔
example3_fea.s.c	$\Box error = 0$	✗	10.47s	✔	12.02s	✔	8.51s	?	8.34s	✗
example3_sea.s.c	$\Box error = 0$	✔	11.11s	✔	11.36s	✔	8.70s	?	6.67s	✔
exsec2.simplify.s.c	$\Box\Diamond x = 1$	✔	4.92s	☢	4.96s	☢	6.38s	✔	5.60s	✔
exsec2.simplify.s.f.c.c	$\Box\Diamond x \neq 1$	✗	4.55s	☢	5.22s	☢	7.85s	✗	6.28s	✗
nondet_gccO0.s.c	$\Box x > 0$	✗	4.57s	☢	4.92s	☢	10.99s	?	10.59s	✗
simple3_gccO0.s.c	$\Diamond p = 1$	✔	4.97s	☢	5.82s	☢	91.22s	?	15.87s	✔
simple3_gccO0.s.f.c	$\Diamond p = 2$	✗	4.96s	☢	4.87s	☢	80.77s	?	20.85s	✗

Chapter 5

Temporal Verification of Polynomial Programs

In this Chapter we now focus on temporal verification of polynomial programs, another class of non-linear programs and, specifically, focus on verifying so-called branching-time temporal properties. As we will see, the techniques we develop have a bigger impact on branch-time than linear-time temporal properties.

Branching-time logics such as Computation Tree Logic (CTL) [44] are well-known in the literature and are of practical interest for expressing a variety of temporal program properties. CTL mixes the ability to express what must happen across *all paths* in the future, with what must happen for *some path* in the future with a focus on choices made along the way. The ability to express the existence of paths is useful, *e.g.*, for verifying the ability of a system to take actions to reach a good state and, increasingly, in verifying autonomous agents [173, 64]. This ability to express branching makes CTL verification of infinite-state programs, in some ways, more challenging than LTL. Linear temporal logic specifies program properties over a single program path one at a time, and it quantifies over all program paths. Branching time logic describes properties over multiple program branches at the same time, therefore the typical LTL strategy of over-approximating one path is not always helpful, because we need to consider its effects for other paths. We thus need strategies that tackle both the positive and negative branches at the same time.

In recent years, numerous techniques [53, 50, 25, 48, 156, 163] and tools such as T2 [47] and FUNCTION [72] have been developed for statically verifying CTL properties of programs. Such state-of-the-art CTL verification tools perform well on programs and properties involving linear integer arithmetics (LIA), e.g., updating variables with LIA expressions, branch conditions in LIA, and proving LIA properties. While having a strong

performance for LIA programs and properties, these tools have limited support for programs with non-linear arithmetics (NLA). In our experiments, the CTL verification tool FUNCTION, for example, would return `unknowns` on programs with NLA properties, even simple ones such as $x^2 > 49$. On some cases, when facing NLA programs or properties, the T2 CTL verification tool becomes unsound and reports proofs even for incorrect programs, making it difficult to trust its results.

NLA properties are difficult to analyze, partially because existing analyses often rely on SAT/SMT solvers, which still have limited support for NLA, or they rely on imprecise convex abstraction domains to represent the non-convex semantics of NLA (e.g., the octagon abstract domain would likely overapproximate, $x^2 > 49$, which involves two distinct regions $x > 7 \vee x < -7$, as $-7 \leq x \leq 7$). However, despite their difficulties, NLA properties are important and arise in many scientific, engineering, and safety- and security-critical applications.

In general, very few works, even non-CTL program analysis, have good treatment for NLA properties (e.g., Ultimate [159], the popular verification suite and winner of several recent annual software verification competitions (SV-COMP) would time out when analyzing programs containing simple NLA branch conditions such as $x^2 > 49$). While the benchmarks used in SV-COMP are very comprehensive, none of them focus on NLA until the recent submission of the NLA-Digbench benchmarks in 2020 containing 27 programs with nonlinear properties, which to the best of our knowledge, cannot be thoroughly analyzed by any existing verification tools.

In this chapter, we present a *rewriting* approach to transform existing programs with NLA properties and expressions into equivalent programs containing only linear expressions. We developed a tool, DRNLA, which takes as input a program containing arithmetics that cannot be analyzed by an existing CTL verification tool and returns another that can be. Thus, our work allows us to apply existing LIA analyses to effectively reason about

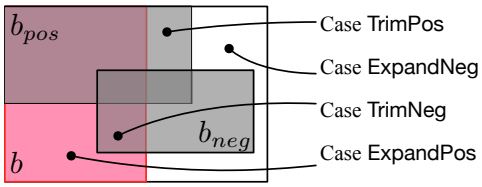


Figure 5.1: Cases layout for b_{pos} of b and b_{neg} of $\neg b$.

NLA programs and properties. We focus on polynomials but the approach also generalizes to other complex numerical arithmetics. Specifically, we propose a new technique for automatically discovering LIA alternatives to NLA expressions found in *boolean conditions*. Thus, our technique can be used as a preprocessing phase to enable CTL tools to handle programs with NLA expressions, when previously they were limited to LIA programs.

Our overall approach begins with a new way to integrate static and dynamic analysis, which we call *dual rewriting*. We analyze a given boolean NLA expression b in a branch condition or loop guard and iteratively synthesize a boolean combination of LIA expressions that are equivalent replacements for the NLA expression in that program context. The key idea is to simultaneously synthesize LIA candidates for both the positive side of b and the negative side of b at the same time, constructing b_{pos} and b_{neg} respectively. We use dynamic analysis to infer candidate guesses for these condition pairs, and then static validation to determine if they are correct. If not, there are four possible cases depending on how b_{pos} overlaps b and how b_{neg} overlaps $\neg b$ depicted in Figure 5.1.

In those cases, b_{pos} may need to be expanded or trimmed, and same for b_{neg}, we then describe a way for static validation to generate a counterexample that can then be used to conjunctively or disjunctively (as the case may be) refine b_{pos} or b_{neg}.

While dual rewriting is sound (thanks to the static validation), it normally would not converge quickly because validation only emits a single concrete counterexample as a snapshot at the error location at a time. We next introduce DyGeneralize, a static/dynamic method for generalizing a single counterexample. We employ an SMT solver to generate

many states at the counterexample's error location that share the same error path condition with the counterexample. We then use dynamic analysis to learn linear conditions over these snapshots to refine b_{pos} or b_{neg}.

We next discuss how our static validation is performed via reachability. We introduce a transformation that takes a program and a candidate pair (b_{pos}, b_{neg}) and, at the location of b in the original program, constructs a four-way conditional and four possible error states, one for each way in which the candidate pair could be too strong/weak. Consequently, if a safety/reachability verifier (eg Ultimate [159] or CPAchecker [152]) discovers a path to an error, that error path will indicate (i) which of the four cases holds, (ii) the input conditions that lead to a state witnessing the case. Note that this transformation preserves the program context, which is important to effectively generate LIA pairs (b_{pos}, b_{neg}) that may exploit context-specific invariants.

We implemented the rewriting technique in a new tool DRNLA that analyzes programs and emits a mapping from the NLA expressions in a program to equivalent LIA expression replacements. DRNLA is written in Python and OCaml and built on top of CIL, Z3, DIG [129] (for dynamic learning) and Ultimate [159] (for static validation). We evaluate DRNLA using three benchmarks consisting of 92 NLA programs with CTL properties. Our results show that the existing tools Function and T2 perform poorly with these benchmarks: Function returns unknown for every program while T2 proves every program (including incorrect ones). However, with the help of DRNLA, these tools (especially T2) perform much better and were able to (dis)prove up to 50% more programs. Moreover, the run time of DRNLA is negligible, making it an ideal add-on to existing CTL analyses.
Source code of DRNLA *is publicly available on GitHub.* [1]

[1] https://github.com/cyruliu/drnla

5.1 Overview Through A Motivating Example

We now dissect our approach through an example, considering two non-linear programs in Figure 5.2, the non-linear loop conditions in these programs are adapted from the non-linear loop invariants in `cohencu.c`, a linear program found in the SV-COMP non-linear benchmarks[2]. We have also adapted the programs for CTL verification by adding atomic propositions over the variable `p` and a CTL property $\mathsf{EF}(p = 0) \wedge \mathsf{EF}(p = 1)$. The CTL property states that from the start state there is at least one execution that leads to a state where $\mathtt{p} = 0$ holds, and also from the start state there is at least one execution that leads to a state where $\mathtt{p} = 1$ holds. Assuming precondition `c <= k` holds initially, the CTL property holds for the left program. The temporal verification requires non-linear reasoning to determine that the loop terminates, thus leading to the $\mathtt{p} = 0$ state, as well as to determine the feasibility of at least one loop iteration in order to lead to the $\mathtt{p} = 1$ state. On the other hand, the CTL property does not hold for the right program. In order to validate or invalidate the property, non-linear reasoning is required to discover the loop invariant `z*z - 12*y - 6*z + 12 == 0` (thus the loop condition is equivalent to `c <= k`) and determines that the assignment of `c - k` into `p` at Line 8 is reachable only when `c > k` (i.e., after the loop terminates). As a result, `p` is always greater than 0, thus the conjunct $\mathsf{EF}(\mathtt{p} = 0)$ is not valid from the initial state and the overall property does not hold.

The loop conditions in the two programs are non-linear polynomials, which is problematic for typical static temporal verification tools that have largely focused on linear conditions. For example, when applying the T2 [33] verifier to this example, it is unsound and incorrectly claims that the property is valid for the second example. Meanwhile, the FuncTion abstract interpreter [72] promptly reports "unknown" for both examples.

[2] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/nla-digbench/cohencu.c`

```
int y=1, z=6, c=0, p=2;
int k=*;
while (z*z - 12y - 6z + 12 + c
     <= k):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

Valid: $\mathsf{EF}(p = 0) \wedge \mathsf{EF}(p = 1)$

```
int y=1, z=6, c=0, p=2;
int k=*;
while (z*z - 12y - 6z + 12 + c
     <= k):
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = c-k;
return 0;
```

Invalid: $\mathsf{EF}(p = 0) \wedge \mathsf{EF}(p = 1)$

Figure 5.2: Nonlinear programs with valid and invalid CTL properties.

Our goal is to enable such existing CTL verification tools to reason about such non-linear programs. To that end, our strategy is to try to discover equivalent *linear* expressions that can be used to replace NLA expressions as a pre-processing step, enabling one to apply existing tools to this broader class of problems.

Learning linear conditions. As discussed earlier, static verification tools struggle with NLA reasoning and, thus, cannot be directly used to analyze them. However, a variety of works in recent years have shown that dynamic analysis can learn non-linear program behaviors by analyzing concrete executions and inferring correlations. This has been done for invariants [129, 132, 128, 171], termination [108], separation logic [111], etc. Many such works combine dynamic learning to find candidate invariants/rank-functions/etc., with static validation to check sufficiency. Although this hybrid strategy means that static tools are used to do some nonlinear reasoning, it is only for the purpose of validation, not for the harder problem of inference/search. We exploit this general strategy in this chapter, but aim now at the specific problem of synthesizing equivalent LIA alternatives to NLA conditions, which would be beneficial for further inferring ranking functions in existing verification tools.

The key idea of our procedure is to identify NLA boolean conditions (and chal-

lenging linear conditions) and for each such condition b, to simultaneously synthesize two linear conditions b_{pos} and b_{neg}, the former reflecting the conditions under which b holds and the latter reflecting the conditions under which $\neg b$ holds. Our algorithm (Sec. 5.2) aims for these conditions to be *exact* (*i.e.*, these conditions will neither be an over-approximation nor an under-approximation), since branching-time verification requires that we reason precisely about branching. Accumulating these conditions *together* enables us to explore from both directions until we have exactly captured both truth values of b. In the above example, b is the loop guard `z*z - 12y - 6z + 12 + c <= k` and naturally $\neg b$ is its negation.

Step 1. Initial guess for b_{pos} and b_{neg}. This can be done through dynamic analysis by simply executing the program on random inputs and instrumenting the program to capture the values of variables `y,z,c,k` inside the loop and, separately, capture the values of those variables after the loop. The first set is examples where b holds and the second where $\neg b$ holds.

b_{pos}; k; y; z; c; p

```
294; 18487; 474; 78; 1
271; 9919;  348; 57; 1
26;  217;   54;  8;  1
296; 8587;  324; 53; 1
...
```

b_{neg}; k; y; z; c; p

```
22; 1657; 144; 23; 1
11; 469;  78;  12; 1
21; 1519; 138; 22; 1
0;  7;    12;  1;  1
...
```

Figure 5.3: Random input snapshots for b_{pos} and b_{neg}.

Using off-the-shelf learning procedures [129, 132] we can infer candidate invariants for these two program locations which will be our initial guesses for b_{pos} and b_{neg}, with the caveat that they are only sound for the random inputs considered in Figure 5.3. After instrumenting the program as described above, the DIG tool [129, 132] learns many possible

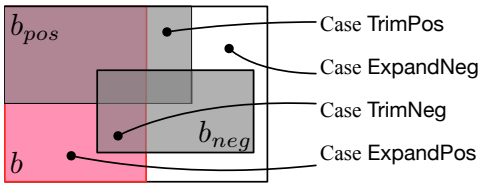
candidate invariants as follows:

$$\text{For } b_{pos}: \quad \{2 \geq p, -p \leq -1, 0 = -6 * c + z - 6, -p - z \leq -8, 0 \geq -c, 0 \geq c - k\}$$
$$\text{For } b_{neg}: \quad \{0 \geq -c + p, 0 \geq -k, 0 = -6 * k + z - 12, 0 = p - 1, 0 = c - k - 1\}$$

Due to the nature of dynamic analysis which derives conditions that hold at the program locations of interest w.r.t just the snapshots at those locations, the above guesses for b_{pos} and b_{neg} contain many irrelevant conditions to the expected condition b and $\neg b$. For example, they have loop invariants at the locations (such as $0 = -6 * c + z - 6$ in b_{pos}) or conditions specific to the given snapshots (such as $0 = -6 * k + z - 12$ in b_{neg}). However, b_{pos} and b_{neg} also have conditions like $0 \geq c - k$ and $0 = c - k - 1$ which are close to the desired result $b \equiv c \leq k$ and $\neg b \equiv c > k$, respectively. Since the condition b and its negation $\neg b$ always contradict each other, we should only keep conditions in the guesses b_{pos} and b_{neg} which also contradict the conditions in the other group. Such conditions can be found via the unsatisfiable core of $\bigwedge(b_{pos} \cup b_{neg})$, which are $\{0 \geq c - k, 0 = c - k - 1\}$ in the form of a set of conjuncts. We then just consider conditions in b_{pos} and b_{neg} which are also in the unsatisfiable core. This leads us to the first guess pair:

$$b_{pos} \equiv 0 \geq c - k \qquad\qquad b_{neg} \equiv 0 = c - k - 1$$

Step 2. Validating the pair (b_{pos}, b_{neg}). The next step is to validate whether $b_{pos} \Leftrightarrow b$ and $b_{neg} \Leftrightarrow \neg b$. If we got lucky and it holds we are done. Otherwise, there are four possible cases to consider as depicted in the following diagram:



In the diagram, the pink area reflects the goal condition to match. Our guess for the positive

side (grey box b_{pos}) could be incorrect in two possible ways: In case TrimPos it includes executions where $\neg b$ holds and must be trimmed down, and in case ExpandPos it does not include all executions where b does hold. Dually, our guess for the negative side (grey box b_{neg}) could be incorrect in two possible ways: In case TrimNeg it includes executions where b holds and must be trimmed down, and in case ExpandNeg it does not include all executions where $\neg b$ does hold.

In Section 5.3 we describe a method for static validation through a program transformation that maintains the context in which b occurs and emits counterexamples that indicate which trim/expand positive/negative case needs to be refined next. When the validation returns "safe", b_{neg} is the negation of b_{pos}. When attempting to validate our first guess b_{pos}, b_{neg} for the running example, we may find a counterexample consisting of the following valuation of variables $c = 0, k = -2, p = 2, y = 1, z = 6$. For these valuations, the original negated condition $\neg b \equiv$ `z*z - 12y - 6z + 12 + c > k` is true because $36 - 12 - 36 + 12 + 0 > -2$, yet our approximation of this negated condition $b_{neg} \equiv$ `0 = c - k - 1` is false because $0 \neq 1$. This case is ExpandNeg: we must increase the size of b_{neg} so that it is true for this state. The counterexamples returned by Section 5.3 are next used to refine b_{pos} and b_{neg}, as we will discuss in a moment. The overall algorithm is depicted in the diagram in Fig. 5.4. So far we have discussed how the input program progresses to the initial guess (in light blue) and how static validation yields a state σ that is one of four possible counterexamples (in dark blue). Below we will discuss the next gray box, used to generate a new expression b_{cex} that can be used to conjunctively/disjunctively amend b_{pos} or b_{neg}, as the case may be.

Step 3. Dynamic Generalization of Counterexamples. A single-state counterexample is not enough to enable a refinement procedure to be tractable. In Section 5.4 we discuss a dynamic analysis procedure that goes beyond a single-state model of a counterexample and instead generates many models and then learns an expression b_{cex} from them. In this

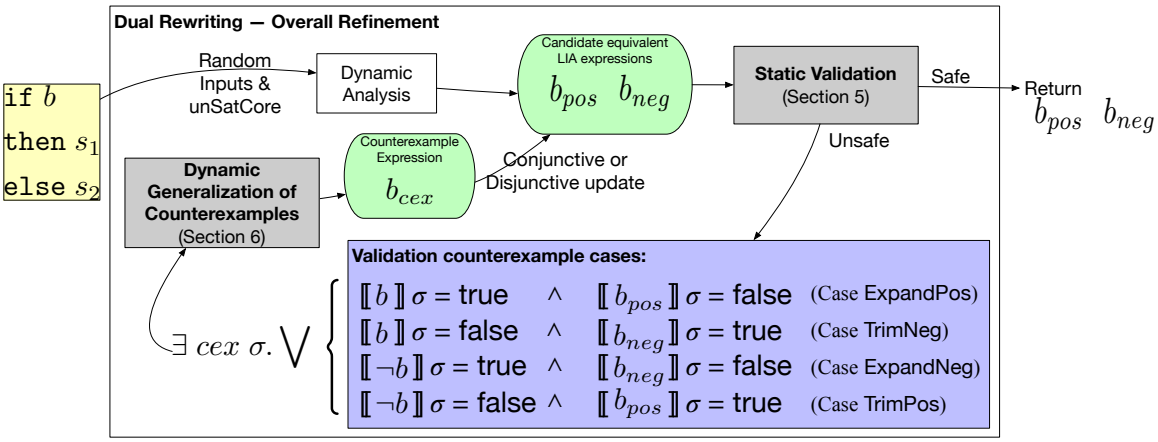


Figure 5.4: Overall flow of the Dual Rewriting algorithm.

way, b_{cex} captures more of the way in which b_{pos} or b_{neg} must be amended than would be by a single state. For example, the validation for the program above failed and found that $b_{neg} \equiv$ `0=c-k-1`. While this single condition could be helpful it is rather a restrictive case being an equality. Our procedure instead generates many models of the counterexample and then uses DIG to learn an expression capturing those models. After some filtration via the UNSATcore, we obtain $b_{cex} \equiv 0 \geq p + k \wedge 0 = p - 2 \wedge 0 = c$.

We next need to disjunctively combine this b_{cex} with b_{neg} (because we were in the ExpandNeg case). However, a direct logical disjunction $b_{neg} \vee b_{cex}$ does not always work well because b_{cex} is still a bit too concrete and specific to this counterexample. By using the convex hull, however, we can generalize the current b_{neg} to be weaker enough to also include b_{cex}. We therefore attempt to compute the convex hull [21] of the disjunction which, in this case, yields $0 + k - c \leq -1$. We now have our second guess:

$$b_{pos} \equiv 0 \geq c - k, \qquad b_{neg} \equiv k - c \leq -1$$

So we return to the validation phase and, at this point, validation does not find any counterexamples so these conditions are precise and are returned. Since these conditions are

equivalent to the original polynomial condition in the program, replacing the polynomial with simply $c - k \leq 0$ leads to a linear program with exactly the same behavior as the original program.

Verifying CTL Properties with the DRNLA **Tool.** We implemented the above technique in a new tool called DRNLA. We discuss its implementation in Sec. 5.6. The upshot is that DRNLA can be used as a pre-processing step to transform an NLA program into an LIA program. We next evaluated whether (i) DRNLA was able to generate LIA expressions for NLA expressions and (ii) whether doing so enabled existing tools to verify CTL properties of NLA programs. Since no NLA CTL benchmarks exist we adapted the DynamiTe NLA termination/nontermination benchmarks to have CTL properties (CTLNLABench-DYNAMITE) and we adapted other known CTL benchmarks [50] to include NLA expressions. Below is an example of such program:

```
1 int a=0, s=1, t=1, k=*, c=0, p=0, x=5;
2 while (t*t - 4*s + 2*t + 1 + c <= k):
3     a = a + 1;
4     t = t + 2;
5     s = s + t;
6     c = c + 1;
7 while (x >= 0):
8     if (*) x--;
9 p = 1;
```

Property: $\mathsf{AF}(\mathsf{EF}(\mathrm{p} > 0))$

This program involves an NLA loop on Line 2. The CTL property says that across all paths, eventually a state is reached, from which point, there is at least one path to reach a state where p is positive.

We report our experimental results in Sec. 5.7.

5.2 Dual Refinement

We now describe the details of our dual refinement procedure. For a given NLA branch expression b at some location ℓ of the program, in a statement of the form $\texttt{if}\ (b^{\ell})\ \texttt{then}\ s_1\ \texttt{else}\ s_2$, dual refinement attempts to synthesize an equivalent boolean LIA expression, using a combination of static and dynamic analysis.

A key enabling insight is that, rather than synthesizing a single expression alternative to b, we can better explore both the positive and negative sides of the state space (*i.e.*, where b holds and where it does not hold) by synthesize two expressions b_{pos} and b_{neg}, respectively. To specify program locations for each expression b, we use snapshots in the following program transformation, wherein for each b^{ℓ}, one snapshot $\mathbf{snap}^{\ell,pos}$ is added immediately inside the positive 'then' branch and $\mathbf{snap}^{\ell,neg}$ immediately inside the negative 'else' branch:

Definition 5.2.1. *[Instrumentation for Snapshots] For a given program P, instrumentation is the following transformation:*

$$P^{snap}[\ell] \hat{=} \left[\texttt{if}\ b^{\ell}\ \texttt{then}\ s\ \texttt{else}\ s' \rightsquigarrow \left(\begin{array}{l} \texttt{if}\ b^{\ell}\ \texttt{then}\ \mathbf{snap}^{\ell,pos}; s \\ \texttt{else}\ \mathbf{snap}^{\ell,neg}; s' \end{array} \right) \right]$$

After this transformation, dynamic analysis tool DIG can be used to synthesize both branches from executing traces. With this in mind, the refinement strategy is described in the algorithm in Fig. 5.5.

More specifically, we aim to such that for any state σ_i reachable at this location L, $[\![b_{pos}]\!]\sigma_i \Leftrightarrow [\![b]\!]\sigma_i$ and $[\![b_{neg}]\!]\sigma_i \Leftrightarrow [\![\neg b]\!]\sigma_i$. We update b_{pos} and b_{neg} with the newly dynamically learned invariants that are presented in $formula(\sigma)$, as following. (s_1 are statements in b branch, s_2 are statements in $\neg b$ branch.)

```
procedure DYREFINE(b, b_pos, b_neg):
  loop
    case TrimPos:    ∃σ. [[b_pos]]σ ∧ ¬[[b]]σ  →  b_pos  := b_pos ∧ ¬formula(σ)
    case ExpandPos:  ∃σ. ¬[[b_pos]]σ ∧ [[b]]σ → b_pos := b_pos ∨ formula(σ)
    case TrimNeg:    ∃σ. [[b_neg]]σ ∧ [[b]]σ → b_neg := b_neg ∧ ¬formula(σ)
    case ExpandNeg:  ∃σ. ¬[[b_neg]]σ ∧ ¬[[b]]σ→  b_neg  := b_neg ∨ formula(σ)
  else → return (b_pos, b_neg)
```

Figure 5.5: Algorithm DYREFINE: Overall strategy synthesize an alternative to boolean condition b by refining a pair of conditions b_{pos}, b_{neg}, so that b_{pos} captures the conditions where b holds and b_{neg} captures the conditions where $\neg b$ holds.

- b_{pos} := learn(execute("assume b; s_1"));
- b_{neg} := learn(execute("assume $\neg b$; s_2"));

As we are iteratively refining a pair of conditions b_{pos}, b_{neg}, the algorithm iteratively checks to see whether one of four possible cases still holds. The cases and amendments needed for each are also represented pictorially in Fig. 5.6.

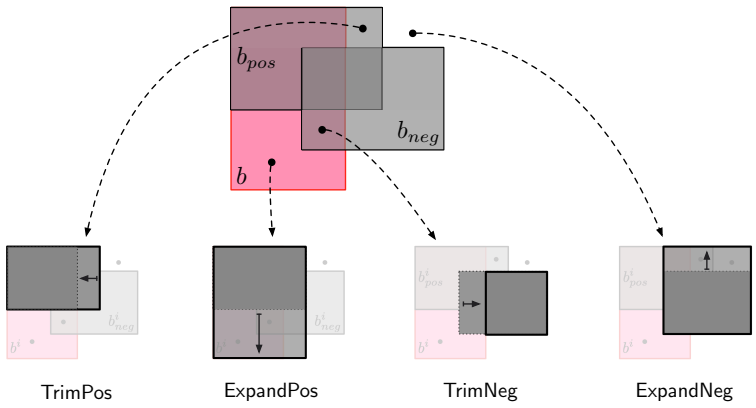


Figure 5.6: Depictions of how candidate LIA conditions b_{pos} and b_{neg} align with states where b holds (in pink) and what actions are needed to remedy.

- TrimPos: In this case b_{pos} captures some states for which b does not hold. For example, if $b \equiv x^2 > 4$ and $b_{pos} \equiv x > 1$ there is at least one state such as $x = 2$, where b_{pos} holds, but b does not. In this case, we need to reduce the size of b_{pos}, as depicted in the first case of Fig. 5.6, so that it does not go beyond the pink b area.

This will be accomplished by discovering some new boolean formula subexpression (Sec. 5.4 we introduce a method for this called DYGENERALIZE) and conjunctively add it to b_{pos}. For example, if DYGENERALIZE returns $0 \leq x \leq 1$, then the conjunction is $b_{pos} \equiv 0 \leq x \leq 1 \wedge x > 1$, which can be simplified to $b_{pos} \equiv 0 \leq x$.

- ExpandPos: In this case b_{pos} does not capture all of the states for which b holds. For example, if $b \equiv x > 0$ and $b_{pos} \equiv x > 5$, then there is a state such as $x = 2$ where b holds, but b_{pos} does not. In this case we need to refine by "expanding" b_{pos} to include the $x = 2$ (and possibly other) states. This is depicted with the gray b_{pos} box expanding to cover more of the pink b area. In order to expand we again find a formula that represents σ (and ideally other similar states) and add it disjunctively to b_{pos}.

- ExpandNeg: In this case b_{neg} does not capture all of the states for which $\neg b$ holds. As depicted in the diagram, we need to enlarge the gray b_{neg} box so that it covers states that are outside the pink box.

- TrimNeg: In this case b_{neg} captures some states for which $\neg b$ does not hold. Here we need to shrink the gray b_{neg} box to reduce its overlap with the pink b box.

Since ideally we would like to have an exact linear mapping for b, which means that the b_{pos} for b and b_{neg} for $\neg b$ should be complemented to each other. One simple optimization of this procedure is, while trimming the positive, use the same σ to expand the negative (and similar for the expanding the positive while trimming the negative), as this could potentially reduce the refinement steps (in the case that it terminates, we will discuss divergence and termination of the algorithm in Sec. 5.5).

DYREFINE takes as argument initial guesses for b_{pos}, b_{neg}. Any initial guess is sound, but we use dynamic analysis to learn conditions that are close to b and $\neg b$, respectively. To

this end we use the instrumentation in Definition 5.2.1: for a given b^ℓ in the program, we apply the instrumentation $P^{snap}[\ell]$, execute the program under random inputs, collect the sets of states $\mathbf{snap}^{\ell,pos}$ and $\mathbf{snap}^{\ell,neg}$, and then employ a learning routine learn (Definition 2.4.1) on each set to obtain conditions b_{pos}, b_{neg}.

While the above algorithm is simple, critical to the practical success of the algorithm are the strategies needed for many subcomponents of the algorithm. First, the semantics of condition b depend on its context in the program, what invariants must hold of other variables, etc. To that end, in Sec. 5.3 we discuss how counterexamples (*i.e.* σ) can be obtained that account for the program context, through a static transformation that reduces the problem to reachability and returns a counterexample from which σ can be derived. Second, individual counterexamples do not allow the refinement algorithm to make much progress on each iteration. Therefore in Sec. 5.4 we describe a method of generalizing counterexamples using dynamic analysis. There are also other details that arise at the implementation level, such as how to exploit the *convex hull* when forming the disjunction in the refinement algorithm. We discuss these issues in the next sections.

Classes of input NLAs and synthesized LIA expressions. On the input side, our implementation does not enforce any specific class of non-linear expressions, apart from what is expressible in the C input programming language, and what the dynamic and static tools can support. As discussed in [129], DIG uses dynamic analysis to analyze programs with polynomial expressions and infer nonlinear equalities and octagonal inequalities. Meanwhile, Ultimate [159] parses input C programs into Boogie programs, and has some support for reachability of polynomials, a fact that we exploit in our static validation in Section 5.3.

On the output side, as one can see, the DYREFINE algorithm synthesizes Boolean combinations of LIA equalities/inequalities or, formally:

$$\mathit{blia} \;::=\; \mathit{blia} \wedge \mathit{blia}' \mid \mathit{blia} \vee \mathit{blia}' \mid \neg \mathit{blia} \mid f_{LIA}(x_1, \ldots, x_n) \leq 0$$

where variables $x_1, \ldots, x_n \in \mathsf{Vars}$ and f_{LIA} is a linear integer arithmetic expression over those variables. The soundness of DYREFINE depends on static validation, so we return to it at the end of the next section. Another natural question is termination of refinement. This partially depends on validation (Sec. 5.3) as well as counterexample generalization (Sec. 5.4), so we return to a discussion of divergence and termination in Sec. 5.5.

5.3 Static Validation Through Reachability

Due to the unsoundness of dynamic analysis, the candidate linear conditions b_{pos} and b_{neg} of the non-linear conditions b and $\neg b$, resp., must be validated to be exact w.r.t. their non-linear versions in the *same* program context where they occur. Note that b and b_{pos} as well as $\neg b$ and b_{neg} are not necessarily equivalent in general (where all models of their variables are considered), but instead it is only necessary for them to be equivalent in the reachable program states in which b and $\neg b$ are evaluated. For example, the condition `z*z - 12*y - 6*z + 12 + c <= k` in the running examples is not equivalent to its linear version `c <= k` in general (e.g., when $\mathtt{y} = 0, \mathtt{z} = 0$) but they are equivalent in the program context where the loop invariant $\mathtt{z*z - 12*y - 6*z + 12} = 0$ holds. In this section we describe how this validation can be done through a program transformation that reduces the problem to reachability, and determines the refinement for b_{pos} and b_{neg} when the validation fails.

We define an *approximation mapping* $m : \mathbb{N} \to \mathsf{b} \times \mathsf{b}$ to map a program location i of a condition b^i to a pair of conditions (b^i_{pos}, b^i_{neg}) that, respectively, approximate the conditions b^i and $\neg b^i$. We now transform the original program P according to the mapping m (iterate m) into a program $P_{\mathsf{valid}}[m]$ by introducing some error locations in it for the following reason: if the program $P_{\mathsf{valid}}[m]$ is safe (*i.e.* no errors in $P_{\mathsf{valid}}[m]$ are reachable) then all approximating positive and negative conditions (b^i_{pos}, b^i_{neg}) in m are exact. The

program transformation is defined as follows:

$$P_{\mathsf{valid}}[m] \;\hat{=}\; \left\{ \begin{array}{l} \forall i \mapsto (b^i_{pos}, b^i_{neg}) \in m. \\ \texttt{if}\ b^i\ \texttt{then}\ s\ \texttt{else}\ s' \rightsquigarrow \\ \quad \left\{ \begin{array}{llll} \texttt{if} & b^i \texttt{and}\ \neg b^i_{pos} & \texttt{then goto error}^i_{\mathsf{ExpandPos}} \\ \texttt{elseif} & b^i \texttt{and}\ b^i_{neg} & \texttt{then goto error}^i_{\mathsf{TrimNeg}} \\ \texttt{elseif} & \neg b^i \texttt{and}\ b^i_{pos} & \texttt{then goto error}^i_{\mathsf{TrimPos}} \\ \texttt{elseif} & \neg b^i \texttt{and}\ \neg b^i_{neg} & \texttt{then goto error}^i_{\mathsf{ExpandNeg}} \\ \texttt{elseif} & b^i & \texttt{then}\ s\ \texttt{else}\ s' \end{array} \right. \end{array} \right\}$$

Intuitively, the above transformation replaces each occurrence of an $\texttt{if}\ b^i\ \texttt{then}\ s\ \texttt{else}\ s'$ with 5-way branching. The final branch involves the normal control-flow of the program, allowing executions of the original program to also exist in the transformed program. However, the first four branches test all of the ways in which b^i can be inconsistent with b^i_{pos} and b^i_{neg}. If it is possible for an execution of the original program P to reach location ℓ in a state where such an inconsistency holds, then there will be an execution of $P_{\mathsf{valid}}[m]$ that can reach the corresponding error label for that form of inconsistency.

Figure 5.7 shows an example of instrumentation for static validation, for nonlinear expression in line 12, five cases for static validation are introduced as nest branches showed on the right side, which maps the polynomial on line 12 to the current candidate b_{pos}, b_{neg}. There will be an execution of this transformed program that enters the branch on Line 4 if ever there is a reachable program state where the polynomial inequality holds, but b_{pos} does not. In such a circumstance a reachability analysis will emit a counterexample that reaches $\texttt{error}_{\mathsf{ExpandPos}}$, reflecting that b_{pos} needs to be expanded. The next 3 branches are similar. If an execution does not fall into one of the first four branches, this does not (yet)

```
int y=1, z=6, c=0, p=2;
int k=*;
while (true):
                                    
                                    
                                    
                                    
                                    
                                    
                                    
  if(z*z-12y-6z+12+c>k):
    break;
  else:
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

```
int y=1, z=6, c=0, p=2;
int k=*;
while(true):
  if(z*z-12y-6z+12+c>k && ¬b_pos):
    error_ExpandPos // b_pos too small
  elsif (z*z-12y-6z+12+c>k && b_neg):
    error_TrimNeg // b_neg too big
  elsif (¬(z*z-12y-6z+12+c>k) && b_pos):
    error_TrimPos // b_pos too big
  elsif (¬(z*z-12y-6z+12+c>k) && ¬b_neg):
    error_ExpandNeg // b_neg too small
  if(z*z-12y-6z+12+c>k):
    break
  else:
    y = y + z;
    z = z + 6;
    c = c + 1;
    p = 1;
p = 0;
return 0;
```

Figure 5.7: Demonstration of instrumentation for static validation.

mean that b_{pos}, b_{neg} are valid: it could, *e.g.* be that a state at a later loop iteration witnesses a shortcoming of b_{pos} or b_{neg}. Thus, the fifth branch allows executions to fall through the check and continue to later program states. If no execution can ever reach any of the error labels, then b_{pos} and b_{neg} must be accurate, *i.e.* the soundness condition discussed in Lemma 5.3.1.

Let us now look at a counterexample to the validity of $P_{\mathsf{valid}}[m]$ and see that it indicates how b_{pos} or b_{neg} needs to be amended. Recall that the counterexample is in the form of a feasible sequence of program statements that lead to the error location, such as

the following (for the instrumented program on the right in Fig. 5.7):

$$cex_1 \equiv \begin{cases} \ell\,1: & \texttt{int y=1, z=6, c=0, p=2;} \\ \ell\,2: & \texttt{int k=*;} \\ \ell\,3: & \texttt{assume(true);} \\ \ell\,8: & \texttt{assume(!(z*z - 12y - 6z + 12 + c <= k));} \\ \ell\,8: & \texttt{assume(!(0 >= c-k));} \\ \ell\,10: & \texttt{assume(!(1 = c-k);} \\ \ell\,11: & \texttt{error}_{\mathsf{ExpandNeg}} \end{cases} \tag{5.1}$$

This is a feasible program path. Consider, for example, the case where $\texttt{k} = -2, c = 0$ initially and, in that case both b and b_{neg} ($c - k - 1 = 0$) are `false`. Therefore branch condition ($\neg b$ `&&` $\neg b_{neg}$) holds so error location $\texttt{error}_{\mathsf{ExpandNeg}}$ can be reached and the above program path cex_1 is output by a reachability verifier, with the error location indicating that b_{neg} must be expanded.

Effectiveness of static validation. We began from the premise that static validation techniques do not cope well with polynomials, yet we now find ourselves using them for exactly that purpose in $P_{\mathsf{valid}}[m]$. There are a few reasons why this is not a sleight of hand. First, we do not ask a static tool to discover a linear replacement for a polynomial but instead validate one that was obtained through dynamic learning. Second, we do not need to reason perfectly about the polynomials that occur inside the Boolean b^i's in $P_{\mathsf{valid}}[m]$ above but instead only need to reason about the Boolean properties of the polynomials. Finally, as mentioned above, we also do not need to reason perfectly about the polynomials on all inputs but instead only in the program context (reachable states) where they occur. In Section 5.7, we will see that tools such as Ultimate [159] are indeed able to validate $P_{\mathsf{valid}}[m]$, despite these instances of polynomial Boolean expressions. In formal, if the program is

shown to be safe we have the following guarantees.

Lemma 5.3.1 (Transformation Correctness)**.** *If all errors in* $P^{check}[m]$ *are unreachable then*

$$\forall i \mapsto (b^i_{pos}, b^i_{neg}) \in m. \forall \sigma \in \textit{preds}(b^i). [\![b^i_{pos}]\!]\sigma = [\![b^i]\!]\sigma \wedge [\![b^i_{neg}]\!]\sigma = [\![\neg b^i]\!]\sigma$$

Proof. Consider a mapping $i \mapsto (b^i_{pos}, b^i_{neg}) \in m$. Because $\texttt{error}^i_{\mathsf{ExpandPos}}$, $\texttt{error}^i_{\mathsf{TrimPos}}$, $\texttt{error}^i_{\mathsf{ExpandNeg}}$, $\texttt{error}^i_{\mathsf{TrimNeg}}$ are unreachable, their corresponding conditional conditions are unsatisfiable under the program context in which b^i could be evaluated (to either `true` or `false`). Therefore,

$$\begin{aligned}
& \forall \sigma \in \mathsf{preds}(b^i). \neg([\![b^i]\!]\sigma \wedge \neg[\![b^i_{pos}]\!]\sigma) \wedge \neg(\neg[\![b^i]\!]\sigma \wedge [\![b^i_{pos}]\!]\sigma) \\
\Leftrightarrow\ & \forall \sigma \in \mathsf{preds}(b^i). (\neg[\![b^i]\!]\sigma \vee [\![b^i_{pos}]\!]\sigma) \wedge ([\![b^i]\!]\sigma \vee \neg[\![b^i_{pos}]\!]\sigma) \\
\Leftrightarrow\ & \forall \sigma \in \mathsf{preds}(b^i). ([\![b^i]\!]\sigma \implies [\![b^i_{pos}]\!]\sigma) \wedge ([\![b^i_{pos}]\!]\sigma \implies [\![b^i]\!]\sigma) \\
\Leftrightarrow\ & \forall \sigma \in \mathsf{preds}(b^i). [\![b^i]\!]\sigma = [\![b^i_{pos}]\!]\sigma
\end{aligned}$$

Similarly, we can prove that $\forall \sigma \in \mathsf{preds}(b^i). [\![\neg b^i]\!]\sigma = [\![b^i_{neg}]\!]\sigma$. □

5.4 Dynamic Generalization of Counterexamples

A counterexample to the static validation ($P_{\mathsf{valid}}[m]$ in the previous section), including concrete witness values for the variables (a model), demonstrates the insufficiency of the current candidate linear approximating conditions b_{pos} and b_{neg}. As seen above, this counterexample from $P_{\mathsf{valid}}[m]$ corresponds to a path in the original program P leading to a place where the truth value of some b^i is inconsistent with b^i_{pos} or b^i_{neg}. Already the counterexample model could be used to assist in refining b_{pos} and b_{neg}. Unfortunately, a single such model would only refine b^i_{pos} or b^i_{neg} by a single data point, which would not lead to a

tractable overall algorithm. In infinite domains, the refinement process may diverge if we refine with only concrete error snapshots.

In this section we introduce DYGENERALIZE (shown in Fig. 5.8 and discussed below), a technique to employ dynamic learning to discover a broader *condition* denoted b_{cex} that is learned from many counterexample models, and allows our overall DYREFINE to take more significant steps toward completion. DYGENERALIZE is called with the current b_{pos}/b_{neg} needed amendment, the current counterexample path cex and the `Expand/Trim` direction of amendment. There are then three steps:

Step 1. DYGENERALIZE first generates many concrete snapshots (*e.g.* a parameter value of 1,000) at the error location from the input counterexample. This can be seen on Line 2 in Fig. 5.8. Our procedure DYGENERALIZE transforms a counterexample path into an SSA logical formula using standard techniques, denoted `path2formula`(cex). From such a formula, we can harvest distinct error snapshots by iteratively invoking an SMT solver to get a model at the error location and then adding constraints to prevent the solver from generating the same model in future iterations. For example, the table on the right shows some error snapshots extracted from counterexample cex_1 (Eqn. 5.1).

y	z	c	k	p
1	6	0	-2	2
1	6	0	-3	2
1	6	0	-4	2
1	6	0	-5	2
1	6	0	-6	2
		...		

Step 2. Next, DYGENERALIZE employs dynamic analysis to learn an error condition b_{cex} from the collected error snapshots (Line 3 in Fig. 5.8). From the data points in the above right table, dynamic analysis can learn the condition $b_{cex} \equiv 0 \geq p + k \wedge 0 = p - 2 \wedge 0 = c$.

Step 3. Depending on the refinement action determined from the counterexample, a simpler and more useful condition can be extracted from the error condition. In particular, if the counterexample indicates that an expansion refinement is needed, the current approximating condition b_{cur} should be expanded with only new conditions which cover program

```
1 procedure DYGENERALIZE(b_cur, cex, direction):
2   S := getModels(path2formula(cex), iters=1000);
3   b_cex := learn(S);
4   if (direction == Expand):
5     match  UnsatCorePair(b_cur, b_cex) with
6     | Some(b_usc) ⇒ return b_usc
7     | None ⇒ return b_cex
8   else:
9     return b_cex
```

Figure 5.8: Algorithm DYGENERALIZE: Generalizing a single counterexample cex beyond a single model, to a formula that captures many states that could reach the same counterexample location.

states not covered by b_{cur}. Because such new conditions in b_{cex} contradict b_{cur}, they can be identified from the unsatisfiable core of $b_{cur} \wedge b_{cex}$, as seen on Line 5. The exact way the unsatisfiable core is utilized is described below.

Definition 5.4.1 (Unsatisfiable core pairs)**.** *Given φ, φ_1 and φ_2 in CNF as a set of clauses, and assuming a* `UnsatCore`(φ) *that returns an unsatisfiable subset of the set of clauses in φ, if one exists, then* `UnsatCorePair`*, given φ_1 and φ_2, is defined as the set of clauses in* `UnsatCore`$(\varphi_1 \cup \varphi_2) \cap \varphi_2$.

```
1 procedure ∨̂(b, b′):
2    match convexHull(b ∨ b′) with
3    | Some(b″)   → return b″
4    | None   → return b ∨ b′
5
6 procedure DYREFINE′(b, b_pos, b_neg):
7 loop
8   case TrimPos:    ∃cex. b_pos ∧ ¬b → b_pos := b_pos ∧ DYGENERALIZE(b_pos, cex, Trim)
9   case ExpandPos: ∃cex. ¬b_pos ∧ b → b_pos := b_pos ∨̂ DYGENERALIZE(b_pos, cex, Expand)
10  case TrimNeg:    ∃cex. b_neg ∧ b → b_neg := b_neg ∧ ¬DYGENERALIZE(b_neg, cex, Trim)
11  case ExpandNeg: ∃cex. ¬b_neg ∧ ¬b → b_neg := b_neg ∨̂ DYGENERALIZE(b_neg, cex, Expand)
12  else ⇒ return (b_pos, b_neg)
```

Figure 5.9: Algorithm DYREFINE′: A revised version of DYREFINE from Fig. 5.5 that now employs dynamic counterexample generalization, and uses the convex hull for disjunction.

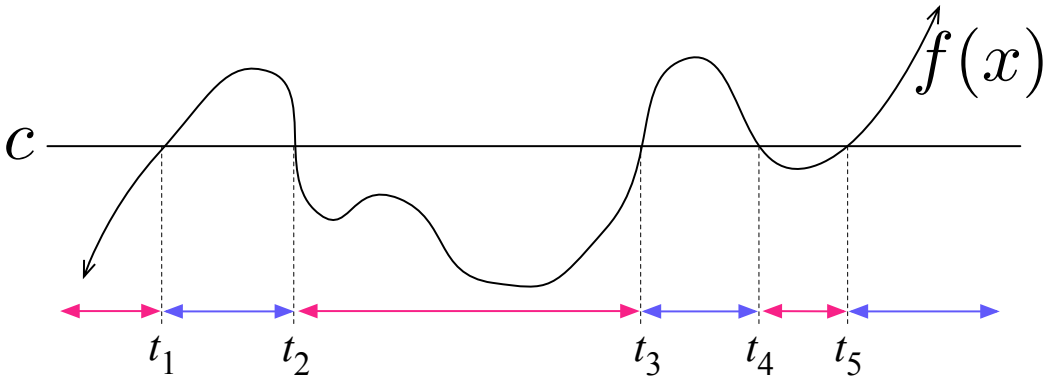
Putting it all together. We now describe a revised DYREFINE′, shown in Fig. 5.9

that employs this DYGENERALIZE. First, we use a counterexample path cex (rather than a single model σ in Fig. 5.5), which is passed to DYGENERALIZE, along with the current condition to be modified, and the appropriate `Expand/Trim` directive. In the `Expand` case, the condition returned by DYGENERALIZE is used to form a disjunction with the current condition b_{cur}. Although it could be used immediately, we can further approximate that disjunction with its convex hull, denoted $\hat{\vee}$ and defined in Fig. 5.9, for a faster converging refinement. For instance, in the running example, consider the refinement when the counterexample determines that the current condition $b_{neg} \equiv c - k = 1$ can be expanded with the condition $b_{cex} \equiv 0 \geq p + k \wedge 0 = p - 2 \wedge 0 = c$. It contains the variable p which is irrelevant to the result. Fortunately, the convex hull of this disjunction is exactly $c > k$ so no further refinement steps are needed (otherwise we need more refinement steps to reason about p).

5.5 Convergence and Termination of DRNLA

The output of our algorithm is a Boolean combination of linear integer arithmetic equalities/inequalities ($blia$ as defined in Sec. 5.2). However, expressing arbitrary polynomial equalities as a $blia$ is not always feasible, much less whether our algorithm would always discover one. Nonetheless, as seen in Sec. 5.7, our algorithm frequently does discover such $blia$ conditions, and we now discuss why, identify fragments where termination is guaranteed, and identify sources of divergence.

Termination due to interval analysis. Perhaps surprisingly, we often do discover a $blia$ equivalent of the original NLA. The main reason is that we only need to capture the Boolean aspects of the polynomial, which amounts to knowing when the polynomial will be above or below a certain bound. Consider the diagram to the right. In this single-variable example of some polynomial $f(x)$ and Boolean property $f(x) < c$, an



equivalent $blia$ need only capture when $f(x)$ is above or below c. That is, the $blia$ simply needs to distinguish the blue values of x (when $f(x) \geq c$) from the red values of x (when $f(x) < c$), and this can be achieved with the expressions:

$$b_{pos} \equiv (t_1 < x \wedge x < t_2) \vee (t_3 < x \wedge x < t_4) \vee (t_5 < x),$$
$$b_{neg} \equiv (x \leq t_1) \vee (t_2 \leq x \wedge x \leq t_3) \vee (t_4 \leq x \wedge x \leq t_5).$$

These $blia$ expressions are essentially interval constraints [42]. Note that there is a potential application here toward compiler optimization, if the calculation of such intervals is more efficient than calculating the polynomial, though we leave this to future work. A simple example of the interval phenomenon is an inequality such as $x^2 > 49$, where an equivalent $blia$ is a simple disjunction of intervals. Beyond a single variable, the two-variable inequality $x^2 + y^2 < 4$ has a $blia$ alternative of $(-2 < y \wedge y < 2) \wedge (-2 < x \wedge x < 2)$.

Termination in other special cases. In some program contexts, polynomial expressions may always evaluate to a constant amount. For example, if an NLA such as $a^2 - b^2 + y + 5 > 0$ occurs inside a loop and an invariant of the loop is that $y = 0 \wedge a = b$, then the NLA will always be equivalent to $5 > 0$. Slightly more generally, in some program contexts, polynomial expressions may always be directly equivalent to an LIA equality/inequality. For example, in a loop where $a = -b$ is an invariant, the polynomial inequality $(a + b)^2 + x > 0$ is exactly equal to $x > 0$. An example is a program in Sec. 5.1,

where a portion of the polynomial is equivalent to 0 due to loop invariants.

Divergence. Finally, there are cases where divergence is inevitable. First, as noted, there may be polynomials that simply cannot be expressed as a $blia$. Second, practically speaking, our dynamic learning in DIG may not be able to learn a sufficiently precise LIA expression for us to use as a building block. Finally, in some cases the dynamic generalization may cause us to "over-shoot," generalizing a counterexample to create a b_{cex} that encompasses an important interval stopping point, *e.g.* we may generalize data points $x = 25$ and $x = 50$ to $b_{cex} = x > 20$, despite there being an important interval at $x = 51$. This could be a future work, which aims to detect those circumstances and attempt a binary search strategy to iteratively reduce the generalization ranges. Practically, to avoid divergence, our implementation currently forces termination after a fixed number of steps (18 iterations) and returns the partially synthesized b_{pos} and b_{neg}.

5.6 DRNLA Implementation

We implemented dual rewriting refinement in a new tool called DRNLA, written in a combination of Python and OCaml. DRNLA uses DIG [129] for dynamic learning and Ultimate [159] as a black-box reachability verifier for static validation. The algorithm takes an input program P with NLA expressions and emits a mapping $m : \mathcal{L} \rightarrow (e, e')$ mapping expression locations in P to a pair of LIA conditions b_{pos}, b_{neg} that can replace those NLAs.

An overview of DRNLA is shown in Fig. 5.10. DRNLA's main driver is implemented in Python. During the refinement loop, DRNLA calls DIG and Ultimate iteratively, and DRNLA uses OCaml/CIL to construct the P^{snap} instrumentation for DIG and P^{check} (for validation) instrumentation for Ultimate. Ultimate returns counterexample paths, which we parse and convert to SSA, so that we can query Z3 to generate many models. We then use DIG to learn from those models a new expression b_{exp} which we

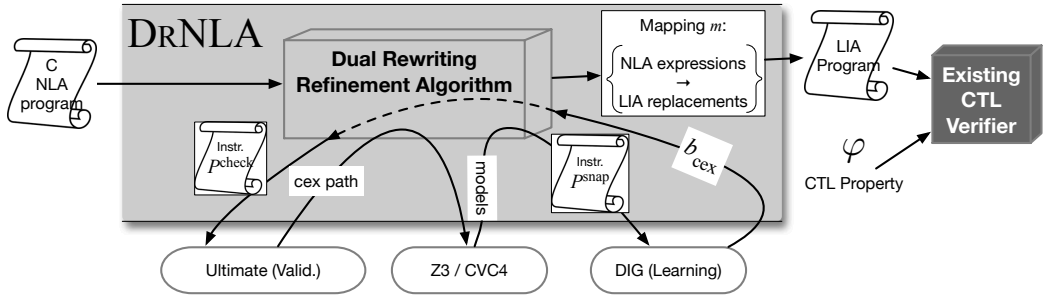


Figure 5.10: DRNLA implementation overview.

parse, add to b_{pos}/b_{neg}, and reincorporate into the current map m to be validated again. The refinement loop can be bounded and if iteration reaches the bound before obtaining an exact solution, DRNLA terminates and return approximated result.

Our transformations are implemented with OCaml/CIL. For static validation with P^{check}, given a candidate mapping m, we transform source program into four cases with nested `if-else` statements (as mentioned in precious section 5.3), and inject `error` labels in each case. For dynamic learning P^{snap} identifies NLA expressions b and creates labels so that learning can discriminate states where b held (following into the `then` branch) from states where $\neg b$ held (following into the `else` branch).

The procedure provided by DIG is used in two circumstances: first to infer initial guess for b_{pos}/b_{neg} and second during the generalization of counterexamples. The learning procedure actually returns a *list* of candidate invariants and not all of them are useful. In fact, we have two lists: one for b_{pos} and one for b_{neg}. We start by removing identical invariants which are useless in discriminating b_{pos} from b_{neg}. We then prune away irrelevant invariants using an UNSATcore procedure to return the minimal sets of candidate invariants. Our dynamic analysis instrumentation also has to take care that the programs do not run forever, so we also instrument loop bound, when the refinement steps exceed the bound, DRNLA terminates with current results.

5.7 Evaluation

We evaluated whether DRNLA (i) was able to synthesize equivalent LIA Boolean conditions from NLA conditions and (ii) whether these synthesized LIA expressions, when replaced in the program, enable existing tools to verify the CTL properties of these previously unsolvable problems.

All experiments described below were run on an Amazon AWS cloud instance with 4 virtual 2.4GHz CPUs, 16GB of memory, and Ubuntu Linux. DRNLA and all external tools (e.g., Ultimate, T2, Function) use a 900s timeout. We run T2 and FUNCTION with default parameters (though we did try to use various parameters such as abstract domains in T2 but did not find ones that improve the performance of the tools).

5.7.1 Nonlinear CTL Benchmarks

To our knowledge, there are no existing benchmark suites consisting of CTL properties for programs with NLA expressions. We thus sought to create such benchmarks using two complementary strategies: extending NLA benchmarks to include CTL properties and extending CTL benchmarks to include NLAs. We then also crafted another set of benchmarks with NLAs that trigger more refinement steps in DRNLA.

CTLNLABench-DYNAMITE We first use the NLA benchmarks in the Dynamite termination work [108]. These programs implement mathematical functions such as `intdiv`, `gcd`, `lcm`, and `power sum`. They contain NLA loop invariants and nontrivial structures such as nested loops. We instrument these programs to include CTL properties. For each program with n NLA loop invariants, we create n variants of the program, where each has an NLA loop invariant and a new variable p that is updated in existing loops in such a way that the variable is either 0 or 1 when the program exits. Thus all programs have the CTL

property $EF(p = 0) \wedge EF(p = 1)$. Next, we create another n variants of the program with the same NLA loop invariants but the variable p is updated in such a way that the CTL property does not hold. Tab. 5.1 shows the 56 created programs: the CTL property is valid for programs with suffix-T and invalid for those with suffix-F.

CTLNLABench-PLDI13 We also created 26 benchmarks with CTL properties from [50] and shown in the top part of Tab. 5.2. For each program, we insert a simple terminating loop that contains NLA expressions. The goal of this instrumentation is to force the CTL tool to reason about the behaviors of these NLA expressions to determine the termination of the loop. In addition, we select 5 random programs from these benchmarks (shown in the bottom part of Tab. 5.2. For these programs, we insert several NLA loops (taken from CTLNLABench-DYNAMITE) to increase the difficulty and also diversity of NLA expressions, we would like to use these programs to test the robustness of DRNLAregarding different types of NLA expressions presented for their CTL verification.

CTLNLABench-Handcraft We also create a set of 10 more challenging benchmarks for DRNLA, shown in Tab. 5.3. These are created with combinations of NLA expressions, which result in higher degrees and are used as conditional or loop guards. The LIA equivalent expressions of these NLA expressions are also more complex and involve disjunctions of linear constraints.

In total, we have 92 CTL benchmarks with NLA and CTL properties (56 from CTLNLABench-DYNAMITE, 26 from CTLNLABench-PLDI13, and 10 CTLNLABench-Handcraft). Our modifications to the original benchmarks are relatively simple (e.g., simple CTL property used in CTLNLABench-DYNAMITE), yet sufficiently strong to illustrate the limitations of existing CTL work in dealing with NLA programs as shown in Section 5.7.2.

5.7.2 DRNLA Synthesizing Results

Tables 5.1, 5.2 and 5.3 show the results of applying DRNLA to CTLNLABench-DYNAMITE, CTLNLABench-PLDI13 and CTLNLABench-Handcraft benchmarks, respectively. We also report programs that DrNLA cannot handle (marked as ?, e.g., because they use double/unsigned variable types or because they may contain NLA expressions in assignments). We report whether the final result was validated by Ultimate (✔) or else needed manual validation (≈). Typically refinement is completed in around 10 minutes, with most time spent in Ultimate. We also report the number of refinement iterations that were needed (**It.**).

As can be seen, for most programs, DRNLA was able to synthesize correct LIA replacements and verified by Ultimate. Even in the cases that need manual validation, the synthesized LIA expressions are also correct, and as will be shown in Section 5.7.3, can help existing CTL tools. Moreover, for CTLNLABench-DYNAMITE and CTLNLABench-PLDI13, DRNLA requires few refinement iterations to synthesize LIA expressions. However, DRNLA requires more refinement iterations for CTLNLABench-Handcraft benchmarks as the LIA expressions for these programs are much more complicated (shown next).

The following are a few representative examples of the benchmark programs' NLA expressions and our generated LIA expressions (Tab. 5.4 and Tab. 5.5). The complete output is given in given in Apx. A.4, Apx. A.5 and Apx. A.6.

In the case of `bresenham1-T.c`, Ultimate was able to validate the final b_{pos} /b_{neg} pair. In the other cases, Ultimate timed out, so we verified them manually (like all ≈ examples).

As is easy to see, the output LIA expressions are far simpler than the input NLA expressions. The following is an example of one of our handcrafted benchmarks

Table 5.1: DRNLA's rewrite results for CTLNLABench-DYNAMITE

Benchmark	Res	T(s)	It.	Benchmark	Res	T(s)	It.
bresenham1-F.c	✔	210.1	2	fermat1-F.c	≈	**TO**	0
bresenham1-T.c	✔	204.2	2	fermat1-T.c	≈	**TO**	0
cohencu2-F.c	≈	517.5	1	geo1-F.c	✔	131.0	1
cohencu2-T.c	≈	473.7	1	geo1-T.c	✔	130.6	1
cohencu3-F.c	≈	621.8	1	geo2-F.c	✔	122.5	1
cohencu3-T.c	≈	621.4	1	geo2-T.c	✔	134.7	1
cohencu4-F.c	≈	780.3	2	geo3-F.c	✔	171.1	1
cohencu4-T.c	≈	773.9	2	geo3-T.c	✔	163.5	1
cohencu5-F.c	✔	179.4	2	hard-F.c	**?**	7.9	0
cohencu5-T.c	✔	177.4	2	hard-T.c	**?**	7.4	0
cohencu7-F.c	≈	194.9	2	hard2-F.c	**?**	0.9	0
cohencu7-T.c	≈	667.5	2	hard2-T.c	**?**	0.8	0
dijkstra2-F.c	≈	687.5	2	prod4br-F.c	≈	239.0	5
dijkstra2-T.c	≈	686.0	2	prod4br-T.c	≈	159.3	4
dijkstra3-F.c	≈	628.6	1	prodbin-F.c	≈	654.5	2
dijkstra3-T.c	≈	627.5	1	prodbin-T.c	≈	619.6	1
dijkstra4-F.c	≈	629.7	1	ps2-F.c	✔	29.0	1
dijkstra4-T.c	≈	629.1	1	ps2-T.c	✔	53.4	2
dijkstra5-F.c	≈	630.8	1	ps3-F.c	✔	62.8	2
dijkstra5-T.c	≈	613.9	1	ps3-T.c	✔	44.9	1
divbin1-F.c	**?**	1.2	0	ps4-F.c	✔	59.8	2
divbin1-T.c	**?**	1.3	0	ps4-T.c	✔	58.1	2
egcd-F.c	≈	**TO**	0	ps5-F.c	✔	30.4	1
egcd-T.c	≈	**TO**	0	ps5-T.c	✔	41.3	1
egcd2-F.c	≈	696.0	1	ps6-F.c	✔	60.3	2
egcd2-T.c	≈	681.8	1	ps6-T.c	✔	68.6	2
egcd3-F.c	≈	**TO**	8	sqrt1-F.c	✔	117.0	2
egcd3-T.c	≈	**TO**	8	sqrt1-T.c	✔	116.3	2

if-cubic-F.c. The source contained $\ell_6 : 8 = x^3$ and we synthesized the following:

$$
\begin{aligned}
b_{pos}: \quad & (((4 \geq p + x) \wedge (0 \geq p - x) \wedge (0 \geq -p + x) \\
& \wedge(-p - x \leq -4)) \vee (2 \geq p \wedge (0 \geq -p + x) \wedge (-p - x \leq -4))) \\
b_{neg}: \quad & ((0 == p - 2 \wedge 1 \geq p - x \wedge !((2 \geq p \\
& \wedge(0 \geq -p + x) \wedge (-p - x \leq -4)))) \vee (0 \geq x))
\end{aligned}
$$

Table 5.2: DRNLA's rewrite results for CTLNLABench-PLDI13

Benchmark	Res	T(s)	It.	Benchmark	Res	T(s)	It.
afagp-F.c	✔	103.9	2	afp-F.c	✔	163.8	2
afagp-T.c	✔	74.8	1	afp-T.c	✔	166.8	2
afefp-T.c	✔	179.8	2	agafp-F.c	✔	65.7	1
afegp-F.c	✔	239.7	2	agafp-T.c	✔	36.3	1
afegp-T.c	✔	177.8	2	efafp-T.c	✔	231.1	2
neg-afagp-F.c	✔	289.1	2	neg-afp-F.c	✔	164.4	2
neg-afagp-T.c	✔	287.2	2	neg-afp-T.c	✔	160.9	2
neg-afefp-F.c	✔	225.2	2	neg-efafp-F.c	✔	243.0	2
neg-afegp-F.c	✔	226.0	2	neg-egafp-F.c	✔	282.2	2
neg-afegp-T.c	✔	217.4	2	neg-egafp-T.c	✔	313.3	2
neg-egimpafp-T.c	✔	251.4	2				
afagp-T.c	✔	160.3	1	neg-afefp-F.c	✔	233.2	1
afefp-T.c	✔	312.1	2	neg-afp-F.c	✔	68.2	2
agafp-T.c	✔	24.4	1				

Table 5.3: DRNLA's rewrite results for handcrafted benchmarks

Benchmark	Res	T(s)	It.	Benchmark	Res	T(s)	It.
if-cubic-F.c	✔	51.5	3	square-loop-F.c	✔	383.6	12
if-cubic-T.c	✔	51.3	3	square-loop-T.c	✔	233.3	7
if-F.c	✔	78.8	6	while-cubic-F.c	✔	87.0	7
if-T.c	✔	76.5	6	while-cubic-T.c	✔	84.4	7
				while-F.c	✔	190.4	6
				while-T.c	✔	189.6	6

Table 5.4: Example output of DRNLA on CTLNLABench-DYNAMITE

Benchmark	Source NLA	Output b_{pos}	Output b_{neg}
bresenham1-T.c	$\ell_{36} : 2Yx - 2X^2y + 2Y - v + c \leq k$	$0 \geq c - k,$	$k - c \leq -1$
cohencu2-T.c	$\ell_{32} : 3n^2 + 3n + 1 \leq k$	$0 \geq y - k,$	$k - y \leq -1$
egcd2-T.c	$\ell_{33} : c \geq xq + ys$	$0 \geq b - c,$	$-b + c \leq -1$

In short, DRNLA is indeed able to synthesize LIA alternatives for NLA expressions across a variety of benchmarks.

Table 5.5: Example output of DRNLA on CTLNLABench-PLDI13

Benchmark	Source NLA	Output b_{pos}	Output b_{neg}
`afefp-T.c`	$\ell_{19} : t^2 - 4s + 2t + 1 + c \leq k$	$0 \geq a - k$	$k - a \leq -1$
`afagp-T.c`	$\ell_{21} : \neg(xz - x - y + 1 + c < k)$	$0 \geq -c + k$	$c - k \leq -1$
`neg-afp-F.c`	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k$	$0 \geq n - k$	$k - n \leq -1$

5.7.3 Enabling CTL Verification of NLA Programs

Tabs. 5.6, 5.7, and 5.8 show how DRNLA improves the two state of the art CTL analyzers T2 [47] and Function [72]. In these tables, in the Improve T2 (Function) group, columns **Res** under T2 (Function) and DRNLA show the result of running T2 (Function) on the original program and on the DRNLA's generated program, respectively. The symbol ✓means proved (expected for benchmarks with T-suffix), **X** means disproved (expected for benchmarks with F-suffix), those with ↯ means incorrect result (e.g., ↯✓ is unsound as it proves an invalid property), ☢ means crash, **?** means unknown, - means DRNLA was not able to rewrite NLA expressions. Results with green background indicate that with the help of DRNLA, the CTL drnla was able to analyze the program correctly.

CTLNLABench-DYNAMITE As can be seen from Tab. 5.6, T2 does not support programs with NLA, and worse, it appears to be unsound by proving everything, including invalid properties. However, with DRNLA's help, T2 was able correctly to analyze 26/56 programs (green background). For the other 30/56 programs that DRNLA failed to help: DRNLA was not able to rewrite 14 programs (due to NLAs in assignments, Double/Unsigned variable types, etc.); T2 crashed on 3 of DRNLA's generated programs (arguably still better than the unsound results that T2 had for these programs); and T2 still gives incorrect results for 13 programs. Function failed to run for all but two programs, `cohencu5-T` and `sqrt1-F`, in which it did run but gave unsound results by proving invalid properties. With DRNLA's help, Function was able to correctly analyze 12/56 programs. The running

Table 5.6: DRNLA's improvements for CTLNLABench-DYNAMITE

Benchmark	Improve T2 T2 Res	T2 T(s)	DRNLA Res	DRNLA T(s)	Improve FUNCTION FT Res	FT T(s)	DRNLA Res	DRNLA T(s)	Benchmark	Improve T2 T2 Res	T2 T(s)	DRNLA Res	DRNLA T(s)	Improve FUNCTION FT Res	FT T(s)	DRNLA Res	DRNLA T(s)
bresenham1-F.c	↯✓	0.7	☢	1.7	?	5.9	?	10.9	fermat1-F.c	↯✓	2.1	-	-	?	1.3	-	-
bresenham1-T.c	✓	0.7	✓	0.7	?	0.6	?	2.7	fermat1-T.c	✓	1.9	-	-	?	0.4	-	-
cohencu2-F.c	↯✓	0.7	**X**	0.8	?	4.4	?	0.2	geo1-F.c	↯✓	1.3	**X**	1.4	?	0.7	?	1.8
cohencu2-T.c	✓	0.7	✓	0.8	?	0.1	?	0.7	geo1-T.c	✓	1.4	✓	1.6	?	0.1	✓	0.6
cohencu3-F.c	↯✓	0.7	**X**	0.8	?	0.1	?	0.7	geo2-F.c	↯✓	1.5	**X**	1.6	?	0.7	?	0.5
cohencu3-T.c	✓	0.7	✓	0.8	?	0.1	?	0.7	geo2-T.c	✓	1.5	✓	1.4	?	0.1	✓	0.6
cohencu4-F.c	↯✓	0.7	**X**	0.7	?	0.0	?	0.0	geo3-F.c	↯✓	1.5	**X**	1.4	?	0.8	?	4.6
cohencu4-T.c	✓	0.7	✓	0.7	?	0.2	✓	0.7	geo3-T.c	✓	1.5	✓	1.4	?	0.1	✓	0.7
cohencu5-F.c	↯✓	0.7	**X**	0.7	↯✓	1.1	?	0.5	hard-F.c	↯✓	1.5	-	-	?	1.6	-	-
cohencu5-T.c	✓	0.7	✓	0.7	?	0.2	✓	1.0	hard-T.c	✓	1.6	-	-	?	5.3	-	-
cohencu7-F.c	↯✓	0.7	☢	1.7	?	1.5	?	0.5	hard2-F.c	↯✓	1.4	-	-	?	0.0	-	-
cohencu7-T.c	✓	0.8	✓	0.7	?	0.4	✓	1.3	hard2-T.c	✓	1.5	-	-	?	0.0	-	-
dijkstra2-F.c	↯✓	0.8	↯✓	0.8	?	0.0	?	0.0	prod4br-F.c	↯✓	2.0	-	-	?	2.0	-	-
dijkstra2-T.c	✓	0.8	✓	0.7	?	670.1	?	**TO**	prod4br-T.c	☢	2.4	-	-	?	0.5	-	-
dijkstra3-F.c	↯✓	0.8	↯✓	0.8	?	**TO**	?	**TO**	prodbin-F.c	↯✓	1.3	-	-	?	0.5	-	-
dijkstra3-T.c	☢	1.7	☢	1.7	?	642.9	?	**TO**	prodbin-T.c	✓	1.1	-	-	?	0.2	-	-
dijkstra4-F.c	↯✓	0.8	☢	1.8	?	**TO**	?	**TO**	ps2-F.c	↯✓	1.0	**X**	1.0	?	0.6	?	1.5
dijkstra4-T.c	✓	0.8	✓	0.7	?	603.8	?	**TO**	ps2-T.c	✓	0.6	✓	0.7	?	0.1	✓	0.5
dijkstra5-F.c	↯✓	0.8	↯✓	0.8	?	**TO**	?	**TO**	ps3-F.c	↯✓	0.7	**X**	0.7	?	0.6	?	1.5
dijkstra5-T.c	✓	0.8	✓	0.7	?	738.5	?	**TO**	ps3-T.c	✓	0.7	✓	1.1	?	0.1	✓	0.5
divbin1-F.c	?	0.2	-	-	?	0.0	-	-	ps4-F.c	↯✓	0.7	**X**	0.7	?	0.6	?	1.5
divbin1-T.c	?	0.2	-	-	?	0.0	-	-	ps4-T.c	✓	0.7	✓	0.7	?	0.1	✓	0.5
egcd-F.c	↯✓	0.8	-	-	?	0.2	-	-	ps5-F.c	↯✓	0.7	**X**	0.7	?	0.6	?	1.5
egcd-T.c	☢	1.8	-	-	?	0.2	-	-	ps5-T.c	✓	0.9	✓	0.9	?	0.1	✓	0.5
egcd2-F.c	↯✓	1.9	↯✓	1.4	?	3.1	?	6.3	ps6-F.c	↯✓	1.0	**X**	0.9	?	0.6	?	1.5
egcd2-T.c	✓	1.3	✓	1.6	?	1.7	?	11.9	ps6-T.c	✓	1.0	✓	1.0	?	0.1	✓	0.5
egcd3-F.c	↯✓	1.5	-	-	?	0.0	-	-	sqrt1-F.c	↯✓	0.8	**X**	0.8	↯✓	0.8	?	2.3
egcd3-T.c	✓	1.8	-	-	?	18.9	-	-	sqrt1-T.c	✓	0.7	✓	0.6	?	0.1	✓	0.8

times of T2 and Function with the original programs and the DRNLA's generated programs are similar (less than a second in most cases). This is expected as DRNLA's generated programs are simpler than the original ones, and thus should not cost extra analysis time.

Table 5.7: DRNLA's improvements for CTLNLABench-PLDI13

Benchmark	Improve T2 T2 Res	T2 T(s)	DRNLA Res	DRNLA T(s)	Improve FUNCTION FT Res	FT T(s)	DRNLA Res	DRNLA T(s)	Benchmark	Improve T2 T2 Res	T2 T(s)	DRNLA Res	DRNLA T(s)	Improve FUNCTION FT Res	FT T(s)	DRNLA Res	DRNLA T(s)
afagp-F.c	**X**	1.6	**X**	1.9	?	0.0	?	0.0	afp-F.c	**X**	1.0	**X**	1.1	?	0.0	↯✓	0.1
afagp-T.c	↯**X**	1.2	✓	1.2	?	0.0	?	0.2	afp-T.c	✓	0.9	✓	1.2	✓	0.0	✓	0.1
afefp-T.c	↯**X**	16.6	✓	25.9	?	0.0	?	0.0	agafp-F.c	**X**	2.2	**X**	1.6	?	0.1	?	0.6
afegp-F.c	**X**	1.8	**X**	1.8	?	0.1	?	0.0	agafp-T.c	↯**X**	1.3	✓	1.0	?	0.0	?	1.0
afegp-T.c	✓	1.7	✓	1.8	?	0.1	?	0.2	efafp-T.c	✓	3.4	✓	2.1	?	0.2	?	1.5
neg-afagp-F.c	**X**	1.4	**X**	1.5	↯✓	0.1	?	0.2	neg-afp-F.c	↯✓	1.1	**X**	1.0	?	0.0	?	0.1
neg-afagp-T.c	✓	2.1	✓	2.1	✓	0.1	?	0.2	neg-afp-T.c	✓	1.1	✓	1.1	?	0.0	?	0.1
neg-afefp-F.c	↯✓	13.3	**X**	57.1	?	0.2	?	0.7	neg-efafp-F.c	**X**	3.8	**X**	2.4	?	0.1	?	0.4
neg-afegp-F.c	↯✓	1.7	↯✓	1.7	↯✓	0.1	?	0.2	neg-egafp-F.c	**X**	2.1	**X**	1.4	?	0.0	?	0.2
neg-afegp-T.c	↯**X**	1.6	↯**X**	1.7	✓	0.1	?	0.2	neg-egafp-T.c	↯**X**	2.3	↯**X**	1.6	?	0.0	?	0.2
neg-egimpafp-T.c	↯**X**	2.4	↯**X**	1.7	?	0.1	?	0.6									
afagp-T.c	↯**X**	1.3	↯**X**	1.2	?	0.0	?	0.2	neg-afefp-F.c	↯✓	13.2	**X**	22.3	?	0.1	?	0.8
afefp-T.c	↯**X**	12.9	✓	25.7	?	0.7	?	2.4	neg-afp-F.c	↯✓	1.0	**X**	0.9	?	0.0	?	0.1
agafp-T.c	↯**X**	1.2	✓	0.9	?	0.0	?	0.5									

CTLNLABench-PLDI13 From Tab. 5.7, T2 also has difficulties in analyzing the instrumented NLA code and returns unsound results. Similarly, Function does not run on most programs, and in the few cases that it does, it likely gives unsound results. DRNLA was able to help T2 with a 10/26 improvement but does not appear to help Function. The differences in running times are also negligible.

Table 5.8: DRNLA's improvements for handcrafted benchmarks

Benchmark	Improve T2				Improve FUNCTION				Benchmark	Improve T2				Improve FUNCTION			
	T2		DRNLA		FT		DRNLA			T2		DRNLA		FT		DRNLA	
	Res	T(s)	Res	T(s)	Res	T(s)	Res	T(s)		Res	T(s)	Res	T(s)	Res	T(s)	Res	T(s)
if-cubic-F.c	↯✓	0.9	**X**	0.7	**?**	0.1	**?**	0.1	square-loop-F.c	**X**	0.8	↯✓	0.8	**?**	0.0	**?**	0.0
if-cubic-T.c	✓	0.6	✓	0.7	**?**	0.0	**?**	0.0	square-loop-T.c	✓	0.8	↯**X**	0.9	**?**	0.1	**?**	3.1
if-F.c	↯✓	0.7	**X**	0.7	**?**	0.0	**?**	0.1	while-cubic-F.c	↯✓	0.7	**X**	0.8	**?**	0.0	**?**	0.2
if-T.c	✓	0.7	✓	0.7	**?**	0.0	**?**	0.1	while-cubic-T.c	✓	0.7	✓	0.7	**?**	0.0	**?**	0.5
									while-F.c	**X**	0.9	**X**	0.9	**?**	0.1	**?**	0.4
									while-T.c	✓	0.7	✓	0.8	**?**	0.1	**?**	3.2

CTLNLABench-Handcraft From Tab. 5.8, as expected, both T2 and Function have difficulties and return incorrect results for some programs. Here, DRNLA was not able to help Function but was able to help T2 with a 60% (6/10) of benchmarks that now can be verified. However, it appears that DRNLA's generated `square-loop` programs with complex linear constraints confused T2 and caused it to return incorrect results.

In summary, for the NLA benchmarks, T2 and Functions just simply do not support these NLA programs: T2 does not support NLA and Function fails to run, and in many cases, they provide unsound results. With the help of DRNLA, T2 can now analyze 26 programs and Function correctly analyzes 12 programs of the 56 SVCOMP benchmarks. Similarly, for the CTLNLABench-PLDI13 benchmarks, we observe a 10/26 for T2, but 0/26 FUNCTION (though instead of returning unsound, FUNCTION now returns unknowns). Moreover, DRNLA's rewrite does not decrease the quality of the CTL tools, i.e., never a case when the CTL tool runs worse with DRNLA's program compared to the original program. Finally, these improvements come with almost no additional runtime

cost (in most cases the analyses took less than a second).

Chapter 6
Conclusions

We have arrived at the end of this dissertation. In this chapter, we recap the highlights of this dissertation and sketch possible extensions in the future.

6.1 Summary

This dissertation presents rewriting theories and tools for temporal verification of nonlinear programs. Instead of developing analysis techniques that directly reason about nonlinear program behaviors like bitwise operations and higher order polynomial computations, we present rewriting methods to approximate these nonlinear behaviors with linear ones, shifting the reasoning burden of existing verification tools back to integer and linear domain, a space where mature static analysis thrives. We have, respectively, implemented our theories and have developed our toolchain DARKSEA for linear temporal verification of decompiled binaries, and DRNLA which combines static and dynamic analysis for branching-time verification of nonlinear programs. The core contributions of this dissertation can be concluded as following parts.

BWB **Chapter 3.** We first present our bitwise branching theory BWB, a term rewriting theory of bitwise operations for linear temporal verification. Two sets of rules *rewriting* and *weakening* rules subsume BWB, covering a wide range of bitwise operations like *logic and, or, bit shifting* etc., each rule has certain conditions that operands should satisfy, under these conditions, a bitvector program can be soundly transformed to a simpler version that is more friendly (can be effectively verified against temporal properties, note that reachability and termination can also be specified in temporal formula) for verification tools.

We have provided soundness proof of rules in BWB with Z3 SMT scripts and implemented our bitwise branching in program analysis framework ULTIMATE, and it has been merged to its main repository, at the same time we developed a rich set of bitvector benchmarks ranging from reachability, termination and LTL verification for better evaluation of our approach as well as other verification tools, the benchmark sets have also been submitted to SV-COMP, a software verification competition repository. In the end, we performed various experiments on our BWB implementation, comparing our results with different SMT solvers (Z3, CVC4, MATHSAT, etc.) encompassed in ULTIMATE and other verification tools, our experimental results show that our BWB theory is effective in helping existing verification tools verify temporal properties of bitvector programs.

DARKSEA **Chapter 4.** With bitwise branching implemented in ULTIMATE as a verification subroutine, we developed DARKSEA, a complete binary end-to-end binary verification toolchain with formal methods, tailored to temporal verification. The majority of binary analysis with formal methods research work focuses on fine-grained verification like correctness of instruction recovery, control flow reconstruction, etc., our tool DARKSEA works on high-level decompiled binaries, verifying coarse properties like termination and LTL. We first studied challenges in binary disassembly and lifting, and identified certain issues like verification of unrelated metadata, nested struct simulating registers, and stack, that existing verification tools face working on decompiled binaries.

There are no standard representations for decompiled binaries, leaving abundant research space in discussion about what is a good representation of decompiled binary, that is tied to verification instead of recompilation as most existing works do, we start our work with binary analysis framework MCSEMA, which lifts disassembly to LLVM intermediate representation. DARKSEA uses IDA PRO and MCSEMA disassembles and lifts binaries into LLVM bytecode, then performs a series of simplification passes on lifted LLVM byte-

code, the transformation passes target challenges we identified earlier for binary verification of decompiled code. With the transformed decompiled IR (which can be disassembled from LLVM byte code), DARKSEA can map it back to C syntax program, which in the end can be handled with our bitwise branching strategy. Finally, we compile benchmarks from SV-COMP and reported GCC bug benchmarks into binaries, our experiment results with these compiled binaries show that DARKSEA is at the time of writing the first effective tool in verifying termination and LTL properties of decompiled binaries.

DRNLA **Chapter 5.** Branching-time properties specify behaviors of programs over time including the ability to express choices from a given state such as, *e.g.*, whether there is some choice or whether all choices lead to some intended outcome. Consequently, reasoning about conditional branching is crucial to branching-time verification techniques and when those conditions are beyond the comfort of linear expressions, branching-time tools report unknown or, worse, unsound results. We introduce a new method of converting programs with non-linear arithmetic (NLA) into equivalent programs with linear arithmetic (LIA) via a technique we call *dual rewriting*. Dual rewriting discovers a linear replacement for an NLA boolean expression b (*e.g.*, as found in conditional branching) through a combination of dynamic learning and static validation of counterexamples to iteratively explore and construct linear expressions for both the positive and negative sides of b. The replacement is a boolean combination of linear expressions that exploits the program context and trades off boolean complexity for NLA complexity but, in doing so, puts more static verification tools within reach.

We implemented our work in a new tool DRNLA that performs the analysis to replace NLA conditional expressions with equivalent LIA conditions. DRNLA takes as input a program containing NLA that cannot be analyzed by an existing verification tool (e.g., FUNCTION or T2) and returns a transformed equivalent version (in the same program

context) that can be reasoned by verification tools. Thus, DRNLA allows us to transform programs, and apply existing LIA analyses to effectively reason about NLA programs and properties. At its core, DRNLA relies on a dual analysis that performs both dynamic and static analysis and learns properties from both positive and negative concrete traces. It uses dynamic analysis to infer a logical LIA formula representing a convex region which approximates the potential non-convex region representing the complex property (NLA expressions). This dynamically inferred LIA formula is likely inaccurate and thus DRNLA refines it by analyzing whether over- or under-approximation occurs using a static analysis tool.

We have shown that CTL properties can indeed be verified in programs with non-linear expressions. Our dual rewriting technique implemented in DRNLA iteratively synthesizes boolean combinations of linear expressions that are equivalent and can be replaced in the program as a pre-processing step before CTL verification. Although static verification tools struggle with many aspects of reasoning about NLA programs, interestingly we found that they were often useful at *validating* whether NLA expressions are equivalent to provided LIA alternatives.

We built the first suite of CTL benchmarks for programs with non-linear expressions, based upon augmenting an existing suite of NLA programs [108] with new CTL properties and augmenting a known existing suite of linear CTL benchmarks [50] with NLA conditions. In the end, We use 92 CTL programs with NLA to evaluate whether these written programs can be more easily verified, showing that our transformed programs enable tools such as FUNCTION and T2 to verify branching-time (CTL) properties of more programs that they could not previously verify.

6.2 Future Research

Our works involve a broad range of research domains from low-level systems research to high-level program transformation, abstraction, and temporal verification, etc., there are a huge number of problems remained unanswered! In the following, we provide a few interesting directions for possible future work.

Bitwise branching extension. The bitwise branching theory we present in this dissertation covers the majority bitwise operations, the size of bit matters due to different types of architectures, in our work bitwise branching is implemented with the assumption of 32-bit processor, a potential extension is to take care of all possible bit size such as 64-bit, 32-bit, 16-bit, etc. In low-level code like assembly and llvm byte code, variables often have smaller size of bit defined in one program, resulting in different upper bounds and lower bounds for the variables, this requires our bitwise branching to make a corresponding adaption to the variable size, in order to have a more precise approximation for bitvector operations. On the other hand, we implemented BWB on the source code level (we decompile and lift binaries to C like high-level programs), instead of implementing bitwise branching on high level code, perhaps another potential direction worth exploring is to apply bitwise branching in a low-level code like LLVM bytecode, with more bit size extension and more coverage on different types of bitvector operations. More approximation rules can be found and applied, which can potentially transform bitvector programs more precisely while still maintaining soundness (e.g. if we know a variable is defined with 8-bit size, then its upper bounds would be 2^8, we would have a stronger condition than the case we interpret it with 32-bit size that lifted code in high level would be).

Property driven decompilation. As we have discussed in Chapter 4, there are lots of noises in decompiled code that are irrelevant to verification, we have the final pass to per-

form program slicing on our lifted code based on LTL formula, in the future, this characteristic can be exploited further. An intuitive approach is that, we can track property related information (like path, and variables) that is identified in disassembly, and lifted IR in lifting, which can help reduce the burden of control flow reconstruction, as we could end up with sub-graph of the program paths that are only related to our target property being verified. However, with this approach, instead of slicing the whole lifted program, reconstructing property-related control flow graph raises a completeness question that how do we know we have traversed all possible paths? One answer would be finding a proof of all unions of sub-graphs are a superset of the original control flow graph. On the other side, decompilation is a process of code discovery in binaries, different disassemblers applying different algorithms have different accuracy in disassembly, similarly, there are discrepancies in lifting tools [37] that construct IRs from disassembly. What was previously built is not focused on verification, instead, we could build our own disassembler with open source tools (e.g. Ramblr) and lifter to construct decompiled IR, in a way that it can be oriented to a format that is formally verified and effective in proof with theorem provers (e.g. Isabelle/HOL [147, 122]). Combining these two directions, with property-driven control flow recovery and formally verified lifted IR, we can build an efficient temporal verification tool on top of them.

Disjunctive invariants. Our work on DRNLA uses dynamic analysis to infer invariants from concrete program snapshots, it works well when these concrete data are scattered in a convex geometry region, which means the dynamic results often are in the form of conjunctions, this is indeed most of the cases that we saw in our benchmarks discussed in Section 5.7. However, if the snapshots we take at a program location which in fact is a condition in the form of disjunction, dynamic analysis would fail to find effective invariants for this set of data. Although our dual rewriting algorithm updates (expand/trim)

the approximated condition with disjunctive results from dynamic analysis, it can only resolve a smaller case of disjunctions, and it can hardly scale. Therefore a more interesting direction is to design an algorithm that is able to infer/learn disjunctive invariants in general. The geometry shapes of disjunctive program snapshots are mostly non-convex, a possible solution is to find a way to decompose them into small groups of data sets, within each group they form a convex region, which we can use our conventional dynamic analysis to infer its invariants, and on top it we can find a way to compose all the sub-group results. With the extension of finding disjunctive invariants in general, DRNLA would be able to verify a broader range of nonlinear programs comparing what we have achieved.

Temporal verification of distributed ledger. Recently, distributed computing systems have more practical applications driven by blockchain technology, bringing smart contracts into the distributed ledger, at the same time raising huge concerns about security. A smart contract itself is a program managing funds automatically on a distributed ledger, any unintended bug would put any asset in a potential capital loss. Temporal actions are common in smart contracts e.g. transferring funds to address B from address A in two days, or sending out asset A as long as address B received funds, etc. Making sure these temporal behaviors perform as intended is vital before smart contracts are released live onchain, since any transaction onchain is immutable, assets loss would be permanent [134]. Works on smart contract specification [157] have been published recently, as well as work on verification tool for smart contracts [154], our works on temporal verification for more practical programs can be extended to smart contracts written on distributed systems too.

Bibliography

[1] Distributed LTL Model Checking with Hash Compaction | Elsevier Enhanced Reader.

[2] External variable problem. https://github.com/lifting-bits/mcsema/issues/566.

[3] Glitch in jump table identification. https://github.com/lifting-bits/mcsema/issues/558.

[4] Hex-rays decompiler: Overview. https://www.hex-rays.com/products/decompiler/.

[5] Miss data cross reference due to resetting ida's analysis flag. https://github.com/lifting-bits/mcsema/issues/561.

[6] Possible wrong code bug. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=43438.

[7] Possible wrong code bug. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=42952.

[8] Sv-comp collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks.

[9] Sv-comp termination benchmarks. https://github.com/sosy-lab/sv-benchmarks/tree/master/c/termination-crafted.

[10] Ultimate's ltl benchmarks. https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL/.

[11] National Security Agency. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.

[12] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination Analysis of Java Bytecode. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[13] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. 2:117–126, September 1987.

[14] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. Binrec: dynamic binary lifting and recompilation. In *EuroSys*, pages 36:1–36:16. ACM, 2020.

[15] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

[16] AProVE. AProVE: Automated program verification environment, 2020. `http://aprove.informatik.rwth-aachen.de/`.

[17] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. Isa semantics for armv8-a, risc-v, and cheri-mips. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[18] Domagoj Babić, Alan J. Hu, Zvonimir Rakamaric, and Byron Cook. Proving termination by divergence. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10-14 September 2007, London, England, UK*, pages 93–102. IEEE Computer Society, 2007.

[19] Peter Backeman, Philipp Rummer, and Aleksandar Zeljic. Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction. pages 1–10, Austin, TX, October 2018. IEEE.

[20] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[21] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[22] Jiři Barnat, Luboš Brim, and Petr Ročkai. Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs. In *NASA Formal Methods*, volume 7226, pages 252–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[23] 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 *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[24] 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 *International Symposium on Formal Methods for Components and Objects*, pages 364–387, 2005.

[25] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified Horn clauses. volume 8044, pages 869–882, 2013.

[26] Dirk Beyer. Advances in automatic software verification: SV-COMP 2020. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 347–367. Springer, 2020.

[27] Dirk Beyer and M Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*, pages 184–190. Springer, 2011.

[28] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.

[29] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, 2003.

[30] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL Constructs for MathSAT: a Preliminary Report. *Electron. Notes Theor. Comput. Sci.*, 144(2):3–14, 2006.

[31] Aaron R Bradley, Zohar Manna, and Henny B Sipma. Linear ranking with reachability. In *International Conference on Computer Aided Verification*, pages 491–504. Springer, 2005.

[32] Marc Brockschmidt. T2: Temporal logic prover, 2020. `https://github.com/mmjb/T2`.

[33] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: Temporal property verification, 2015.

[34] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, 2011.

[35] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424, pages 358–372. Springer Berlin Heidelberg, 2007.

[36] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473. IEEE, 2009.

[37] Chris Casinghino, JT Paasch, Cody Roux, John Altidor, Michael Dixon, and Dustin Jamner. Using binary analysis frameworks: The case for bap and angr. In *NASA Formal Methods Symposium*, pages 123–129. Springer, 2019.

[38] Marek Chalupa. mchalupa/dg, January 2021.

[39] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Synthesising interprocedural bit-precise termination proofs (T). In *ASE*, pages 53–64. IEEE Computer Society, 2015.

[40] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Synthesising Interprocedural Bit-Precise Termination Proofs (extended version). *arXiv:1505.04581 [cs]*, May 2015.

[41] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Bit-Precise Procedure-Modular Termination Analysis. *ACM Transactions on Programming Languages and Systems*, 40:1–38, January 2018.

[42] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. An abstract domain to discover interval linear equalities. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 112–128. Springer, 2010.

[43] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[44] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[45] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 328–340. Springer, 2008.

[46] Byron Cook, Heidy Khlaaf, and Nir Piterman. Faster temporal reasoning for infinite-state programs. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 75–82, Lausanne, Switzerland, October 2014. IEEE.

[47] Byron Cook, Heidy Khlaaf, and Nir Piterman. On automation of ctl* verification for infinite-state systems. In *International Conference on Computer Aided Verification*, pages 13–29. Springer, 2015.

[48] Byron Cook, Heidy Khlaaf, and Nir Piterman. Verifying increasingly expressive temporal logics for infinite-state systems. *Journal of the ACM*, 64(2):15:1–15:39, April 2017.

[49] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 399–410, New York, NY, USA, 2011. Association for Computing Machinery.

[50] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 219–230. ACM, 2013.

[51] Byron Cook, Daniel Kroening, Philipp Rummer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[52] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 415–426. ACM, 2006.

[53] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.

[54] Patrick Cousot and Radhia Cousot. An abstract interpretation framework for termination. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 245–258. ACM, 2012.

[55] Anusha Damodaran, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H. Austin, and Mark Stamp. A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection. *J Comput Virol Hack Tech*, 13(1):1–12, February 2017. arXiv: 2203.09938.

[56] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 655–671, London UK, June 2020. ACM.

[57] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. page 16, 2019.

[58] Cristina David, Daniel Kroening, and Matt Lewis. Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032, pages 183–204. Springer Berlin Heidelberg, 2015.

[59] Yegor Derevenets. Snowman. `https://derevenets.com/`.

[60] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[61] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. Fairness modulo theory: A new approach to LTL software model checking. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 49–66. Springer, 2015.

[62] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. *ACM SIGPLAN Notices*, 46(6):567–577, 2011.

[63] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[64] Nagat Drawel, Jamal Bentahar, Mohamed El-Menshawy, and Amine Laarej. Verifying temporal trust logic using ctl model checking. In *TRUST@ AAMAS*, pages 62–74, 2018.

[65] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[66] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[67] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[68] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany*, 2011.

[69] Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In *Verified Software: Theories, Tools, Experiments*, volume 7152, pages 261–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[70] Jérôme Feret. Static analysis of digital filters. In *European Symposium on Programming*, pages 33–48. Springer, 2004.

[71] Justin Ferguson and Dan Kaminsky. *Reverse engineering code with IDA Pro*. Syngress, 2008.

[72] FuncTion. FuncTion: An abstract domain functor for termination, 2020. `https://www.di.ens.fr/~urban/FuncTion.html`.

[73] Inc. Galois. Macaw. `https://github.com/GaloisInc/macaw`.

[74] Inc. Galois. Reopt vcg. `https://github.com/GaloisInc/reopt-vcg`.

[75] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[76] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reason.*, 58(1):3–31, 2017.

[77] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reason.*, 58(1):3–31, 2017.

[78] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with aprove. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.

[79] Patrice Godefroid. Between Testing and Verification: Dynamic Software Model Checking. page 18.

[80] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*, pages 174–186. ACM Press, 1997.

[81] Patrice Godefroid. Between testing and verification: Dynamic software model checking. In *Dependable Software Systems Engineering*, 2016.

[82] Alex Groce and Rajeev Joshi. Extending Model Checking with Dynamic Analysis. In *Verification, Model Checking, and Abstract Interpretation*, volume 4905, pages 142–156. Springer Berlin Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[83] Bhargav S Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *International Conference on Computer Aided Verification*, pages 370–384. Springer, 2008.

[84] Sumit Gulwani. Speed: Symbolic complexity bound analysis. In *International Conference on Computer Aided Verification*, pages 51–62. Springer, 2009.

[85] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices*, 44(1):127–139, 2009.

[86] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[87] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for termination. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 304–319. Springer, 2010.

[88] Shaobo He and Zvonimir Rakamarić. Counterexample-Guided Bit-Precision Selection. In *Programming Languages and Systems*, volume 10695, pages 534–553. Springer International Publishing, 2017.

[89] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2018.

[90] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Jochen Hoenicke, Markus Lindenmann, Betim Musa, Christian Schilling, Stefan Wissert, and Andreas Podelski. Ultimate Automizer with Unsatisfiable Cores. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, pages 418–420. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[91] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. *ACM Sigplan Notices*, 45(1):471–482, 2010.

[92] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, volume 8559, pages 797–813. Springer International Publishing, 2014.

[93] Joe Hendrix, Guannan Wei, and Simon Winwood. Towards Verified Binary Raising. page 4.

[94] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. In *SEFM*, volume 9763 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2016.

[95] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution. In *Software Engineering and Formal Methods*, volume 9763, pages 234–252. Springer International Publishing, 2016.

[96] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 526–538, 2002.

[97] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 357–370. ACM, 2011.

[98] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In Kazunori Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2010.

[99] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[100] Hong Jin Kang and David Lo. Adversarial Specification Mining. *ACM Trans. Softw. Eng. Methodol.*, 30(2):1–40, March 2021. arXiv: 2103.15350.

[101] Johannes Kinder. Jakstab: The static analysis platform for binaries. `http://www.jakstab.org/`.

[102] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification*, pages 423–427. Springer, 2008.

[103] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer Aided Design*, pages 43–50. IEEE, 2010.

[104] Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, October 2014.

[105] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Complexity of Fixed-Size Bit-Vector Logics. *Theory of Computing Systems*, 59:323–376, August 2016.

[106] Daniel Kroening and Natasha Sharygina. Approximating Predicate Images for Bit-Vector Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920, pages 242–256. Springer Berlin Heidelberg, 2006.

[107] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. Dynamite: Dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[108] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. Dynamite: Dynamic termination and non-termination proofs, 2020.

[109] Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A resource-based logic for termination and non-termination proofs. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2014.

[110] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 489–498. ACM, 2015.

[111] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 788–801. ACM, 2019.

[112] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001.

[113] Jan Leike and Matthias Heizmann. Geometric nontermination arguments. In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 266–283, 2018.

[114] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General LTL Specification Mining. page 12.

[115] Yuandong Cyrus Liu, Ton-Chanh Le, and Eric Koskinen. Source-level bitwise branching for temporal verification, 2021.

[116] Yuandong Cyrus Liu, Chengbin Pang, Daniel Dietsch, Eric Koskinen, Ton-Chanh Le, Georgios Portokalidis, and Jun Xu. Proving ltl properties of bitvector programs and decompiled binaries. In *Programming Languages and Systems*, pages 285–304. Springer International Publishing, 2021.

[117] Sven Mattsen, Arne Wichmann, and Sibylle Schupp. A non-convex abstract domain for the value analysis of binaries. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 271–280, Montreal, QC, Canada, 2015. IEEE.

[118] Kenneth L McMillan. Applications of craig interpolants in model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12. Springer, 2005.

[119] Roberto Metere, Andreas Lindner, and Roberto Guanciale. Sound Transpilation from Binary to Machine-Independent Code. *arXiv:1807.10664 [cs]*, 10623:197–214, 2017.

[120] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference,*

TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, pages 568–582, 2007.

[121] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Machine-code verification for multiple architectures - an application of decompilation into logic. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8, 2008.

[122] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 78–81, 2012.

[123] Alexander Nadel. Solving MaxSAT with Bit-Vector Optimization. In *Theory and Applications of Satisfiability Testing – SAT 2018*, volume 10929, pages 54–72. Springer International Publishing, 2018. Series Title: Lecture Notes in Computer Science.

[124] Daniel Neider and Ivan Gavran. Learning Linear Temporal Properties. *arXiv:1806.03953 [cs]*, September 2018. arXiv: 1806.03953.

[125] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 845–859. ACM, 2019.

[126] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 605–615, 2017.

[127] ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 804–814. IEEE, 2017.

[128] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 683–693. IEEE, 2012.

[129] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Transactions on Software Engineering and Methodology, to appear*, 2014.

[130] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Trans. Softw. Eng. Methodol.*, 23(4), sep 2014.

[131] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*, pages 608–619, 2014.

[132] Thanhvu Nguyen, Kim Hao Nguyen, and Matthew Dwyer. Using symbolic states to infer numerical invariants. *IEEE Transactions on Software Engineering*, 2021.

[133] Aina Niemetz, Mathias Preiner, Andrew Reynolds, Yoni Zohar, Clark Barrett, and Cesare Tinelli. Towards Bit-Width-Independent Proofs in SMT Solvers. *arXiv:1905.10434 [cs]*, June 2019.

[134] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *arXiv:1802.06038 [cs]*, March 2018. arXiv: 1802.06038.

[135] Aditya V Nori and Rahul Sharma. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 246–256, 2013.

[136] NSA. Ghidra. `https://www.nsa.gov/resources/everyone/ghidra/`.

[137] Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.

[138] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-Driven Precondition Inference with Learned Features. page 15.

[139] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.

[140] Julian Parsert, Mirco Giacobbe, and Daniel Kroening. Neural termination analysis. In *ESEC/FSE*, 2022. To appear.

[141] Etienne Payet, Fred Mesnard, and Fausto Spoto. Non-Termination Analysis of Java Bytecode. *arXiv:1401.5292 [cs]*, January 2014.

[142] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004.

[143] Zvonimir Rakamaric and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.

[144] John Regehr. 42721 – possible integer wrong code bug. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=42721`.

[145] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[146] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

[147] Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 181–195, 2019.

[148] Hex-Rays SA. Ida pro. `https://www.hex-rays.com/products/ida/`.

[149] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *European Symposium on Programming*, pages 574–592. Springer, 2013.

[150] Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver. In *Computer Aided Verification*, volume 12760, pages 149–171. Springer International Publishing, 2021. Series Title: Lecture Notes in Computer Science.

[151] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war:

Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[152] SoSy-Lab. Cpachecker: The configurable software-verification platform, 2020. `https://cpachecker.sosy-lab.org/`.

[153] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. page 17.

[154] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. page 17.

[155] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving Termination and Memory Safety for Programs with Pointer Arithmetic. In *Automated Reasoning*, volume 8562, pages 208–223. Springer International Publishing, Cham, 2014.

[156] Gadi Tellez and James Brotherston. Automatically verifying temporal properties of pointer programs with cyclic proof. pages 491–508, 2017.

[157] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A Survey of Smart Contract Formal Specification and Verification. *arXiv:2008.02712 [cs]*, April 2021. arXiv: 2008.02712.

[158] Vu Xuan Tung. raSAT : an SMT Solver for Polynomial Constraints. page 9.

[159] Ultimate. Ultimate automizer, 2020. `https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=ltl_automizer`.

[160] Caterina Urban. Piecewise-defined ranking functions. In *13th International Workshop on Termination (WST 2013)*, page 69, 2013.

[161] Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. Synthesizing Ranking Functions from Bits and Pieces. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636, pages 54–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[162] Caterina Urban and Antoine Miné. An abstract domain to infer ordinal-valued ranking functions. In *European Symposium on Programming Languages and Systems*, pages 412–431. Springer, 2014.

[163] Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract interpretation of CTL properties. volume 11002, pages 402–422, 2018.

[164] Moshe Y. Vardi. *Alternating automata and program verification*, pages 471–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[165] Moshe Y. Vardi. *An automata-theoretic approach to linear temporal logic*, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[166] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 332–344. IEEE Computer Society, 1986.

[167] Freek Verbeek, Joshua Bockenek, Zhoulai Fu, and Binoy Ravindran. Formally verified lifting of c-compiled x86-64 binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 934–949, New York, NY, USA, 2022. Association for Computing Machinery.

[168] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound C Code Decompilation for a Subset of x86-64 Binaries. In *Software Engineering and Formal Methods*, volume 12310, pages 247–264. Springer International Publishing, 2020. Series Title: Lecture Notes in Computer Science.

[169] Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[170] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42:3–23, February 2013.

[171] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 106–120. ACM, 2020.

[172] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–234, 2008.

[173] Benjamin Zarrieß and Jens Claßen. Verifying ctl* properties of golog programs over local-effect actions. In *ECAI 2014*, pages 939–944. IOS Press, 2014.

[174] Yoni Zohar, Ahmed Irfan, Makai Mann, Aina Niemetz, Andres Notzli, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Bit-Precise Reasoning via Int-Blasting, 2021.

[175] •. Sail architecture definition language. `https://github.com/rems-project/sail`.

Appendix A

Appendix for All Chapters

A.1 Proofs of Bitwise Branching Rules (Sec. 3.2)

```
from z3 import *

def prove(r, f):
    s = Solver()
    s.add(Not(f))
    if s.check() == unsat:
        print ("proved rule: " + r)
    else:
        print ("failed to prove rule: " + r)
        print(s.model())

def vec2bool(v):
    if v!=0: return True
    else: return False

def bool2vec(b):
    if (b==True): return BitVecVal(1,1)
    else: return BitVecVal(0,1)

def lt(e1, e2):
    return e1 < e2

def leq(e1, e2):
    return e1 <= e2

def gt(e1, e2):
    return e1 > e2

def geq(e1, e2):
    return e1 >= e2

```

```
32 def eq(e1, e2):
33     return e1 == e2
34 
35 op_le = [lt, leq, eq]
36 op_ge = [gt, geq, eq]
37 
38 # Although we define all variables in the 32 bit sizes,
39 # all rules are in general and size independent,
40 # except R_RSHIFT_POS and R_RSHIFT_NEG.
41 # e1 and e2 are commutative in binary operations,
42 # therefore we prove them in one direction here.
43 
44 e1, e2, r = BitVecs('e1 e2 r', 32)
45 e1bit, e2bit = BitVecs('e1bit e2bit', 1)
46 
47 # Rewriting rules
48 # R. rules for &
49 rule = f"R_AND_0 e1 == 0 |- e1 & e2 <==> 0"
50 pre = And (e1 == 0)
51 post = (e1& e2 == 0)
52 prove(rule, Implies(pre, post))
53 
54 rule = f"R_AND_1 (e1 == 0 \/ e1 == 1) /\ e2 = 1 |- e1 & e2 <==> e1"
55 pre = And (Or(e1==0, e1==1), e2 == 1)
56 post = (e1 & e2 == e1)
57 prove(rule, Implies(pre, post))
58 
59 # we split it into two cases proofs, e1 & e2 == 1, e1 & e2 == 0
60 rule = f"R_AND_LOG (case 1) (e1 == 0 \/ e1 == 1) /\ (e2 = 1 \/ e2 = 0) |- e1 & e2 <==>
       e1 && e2"
61 pre = And (e1bit & e2bit == 1)
62 post = And(e1bit == 1, e2bit == 1)
63 prove(rule, Implies(pre, post))
64 
65 rule = f"R_AND_LOG (case 2) (e1 == 0 \/ e1 == 1) /\ (e2 = 1 \/ e2 = 0) |- e1 & e2 <==>
       e1 && e2"
66 pre = And (e1bit & e2bit == 0)
67 post = Not(And(e1bit == 1, e2bit == 1))
68 prove(rule, Implies(pre, post))
```

```
69
70 rule = f"R-AND-LBS e1 >= 0 && e2 == 1 |- e1 & e2 <==> e1%2"
71 pre = And (e1 >= 0, e2 == 1)
72 post = (e1 & e2 == e1 % 2)
73 prove(rule, Implies(pre, post))
74
75 # R. rules for |
76 rule = f"R-OR-LOG (e1 == 0 \/ e1 == 1) /\ (e2 = 1 \/ e2 = 0) |- e1 | e2 == 0 <==> e1 &&
       e2"
77 pre = And (e1bit | e2bit == 0)
78 post = And(e1bit == 0, e2bit == 0)
79 prove(rule, Implies(pre, post))
80
81 rule = f"R-OR-0 e2 == 0 |- e1 | e2 <==> e1"
82 pre = And (e2 == 0)
83 post = (e1 | e2 == e1)
84 prove(rule, Implies(pre, post))
85
86 rule = f"R-OR-1 (e1 == 0 \/ e1 == 1) && e2 == 1 |- e1 | e2 <==> 1"
87 pre = And (Or (e1 == 0, e1 == 0), e2 == 1)
88 post = (e1 | e2 == 1)
89 prove(rule, Implies(pre, post))
90
91 # R. rules for ^
92 rule = f"R-XOR-0 e2 == 0 |- e1 ^ e2 <==> e1"
93 pre = And (e2 == 0)
94 post = (e1 ^ e2 == e1)
95 prove(rule, Implies(pre, post))
96
97 rule = f"R-XOR-EQ e1=e2=0 \/ e1=e2=1 |- e1 ^ e2 <==> 0"
98 pre = Or (And(e1==0, e2==0), And(e1==1, e2 == 1))
99 post = (e1 ^ e2 == 0)
100 prove(rule, Implies(pre, post))
101
102 rule = f"R-XOR-NEQ (e1=0 /\ e2=1) \/ (e1=1 /\ e2=0) |- e1 ^ e2 <==> 1"
103 pre = Or (And(e1==0, e2==1), And(e1==0, e2 == 1))
104 post = (e1 ^ e2 == 1)
105 prove(rule, Implies(pre, post))
106
```

```
# R. rules for >>, depends on the maximum bit size, here we use 32 as maximum
# 32 bit 2's complement
rule = f"R-RSHIFT-POS (e1 >= 0) && (e2=31) |- e1 >> e2 <==> 0"
pre = And(e1>=0, e2==31)
post = (e1 >> e2 == 0)
prove(rule, Implies(pre, post))

# In C right/left shit with negative number is undefined, therefore implementation-
    dependent.
rule = f"R-RSHIFT-NEG (e1 < 0) && (e2=31) |- e1 >> e2 <==> -1"
pre = And(e1<0, e2==31)
post = (e1 >> e2 == -1)
prove(rule, Implies(pre, post))

# Weakening rules, in this proof, we omit assignment ":=", as it can be treated as "=="
# W. rules for bitwise operator &
for op in op_le:
    rule = f"W-AND-POS e1 >= 0 && e2 >= 0 |- r {op.__name__} e1 & e2 ==> r <= e1 && r<=
     e2"
    pre = And (e1 >= 0, e2 >= 0, op(r, e1 & e2))
    post = And (r <= e1, r <= e2)
    prove(rule, Implies(pre, post))

for op in op_le:
    rule = f"W-AND-NEG e1 < 0 && e2 < 0 |- r {op.__name__} e1 & e2 ==> r <= e1 && r<=
    e2 && r < 0"
    pre = And (e1 < 0, e2 < 0, op(r, e1 & e2))
    post = And (r <= e1, r <= e2, r < 0)
    prove(rule, Implies(pre, post))

rule = f"W-AND-Mix e1 >= 0 && e2 < 0 |- r == e1 & e2 ==> 0 <= r && r<= e1"
pre = And (e1 >= 0, e2 < 0, eq(r, e1 & e2))
post = And (0 <= r, r <= e1)
prove(rule, Implies(pre, post))

# W. rules for |
for op in op_ge:
    rule = f"W-OR-CONST e1 > 0 && is_const(e2) |- r {op.__name__} e1 | e2 ==> r >= e2"
    pre = And (e1 > 0, op(r, e1 | e2))
```

```
143     post = And (r >= e2)
144     prove(rule, Implies(pre, post))
145 
146 for op in op_ge:
147     rule = f"W-OR-POS e1 >= 0 && e2 >= 0 |- r {op.__name__} e1 | e2 ==> r >= e1 && r >=
         e2"
148     pre = And (e1 >= 0, e2 >= 0,  op(r, e1 | e2))
149     post = And (r >= e1, r >= e2)
150     prove(rule, Implies(pre, post))
151 
152 rule = f"W-OR-NEG e1 < 0 && e2 < 0 |- r == e1 | e2 ==> r >= e1 && r>= e2 && r < 0"
153 pre = And (e1 < 0, e2 < 0, eq(r, e1 | e2))
154 post = And (r >= e1, r >= e2, r < 0)
155 prove(rule, Implies(pre, post))
156 
157 rule = f"W-OR-MIX e1 >= 0 && e2 < 0 |- r == e1 | e2 ==> r>= e2 && r < 0"
158 pre = And (e1 >= 0, e2 < 0, eq(r, e1 | e2))
159 post = And (r >= e2, r < 0)
160 prove(rule, Implies(pre, post))
161 
162 # W. rules for ^
163 for op in op_ge:
164     rule = f"W-XOR-POS e1 >= 0 && e2 >= 0 |- r {op.__name__} e1 ^ e2 ==> r >= 0"
165     pre = And (e1 >= 0, e2 >= 0,  op(r, e1 ^ e2))
166     post = And (r >= 0)
167     prove(rule, Implies(pre, post))
168 
169 for op in op_ge:
170     rule = f"W-XOR-NEG e1 < 0 && e2 < 0 |- r {op.__name__} e1 ^ e2 ==> r >= 0"
171     pre = And (e1 < 0, e2 < 0,  op(r, e1 ^ e2))
172     post = And (r >= 0)
173     prove(rule, Implies(pre, post))
174 
175 for op in op_le:
176     rule = f"W-XOR-MIX e1 >= 0 && e2 < 0 |- r {op.__name__} e1 ^ e2 ==> r < 0"
177     pre = And (e1 >= 0, e2 < 0,  op(r, e1 ^ e2))
178     post = And (r < 0)
179     prove(rule, Implies(pre, post))
180 
```

```
# W. rules for ~
rule = f"W-CPL-POS e1 >= 0 |- r == ~e1 ==> r < 0"
pre = And (e1 >= 0,  eq(r, ~e1))
post = And (r < 0)
prove(rule, Implies(pre, post))

rule = f"W-CPL-NEG e1 < 0 |- r == ~e1 ==> r >= 0"
pre = And (e1 < 0,  eq(r, ~e1))
post = And (r >= 0)
prove(rule, Implies(pre, post))
```

A.2 Full Lifted Code for `PotentialMinimizeSEVPABug` (Chapter. 4)

```
//@ ltl invariant positive: ([] ( AP(x > 0) ==> <>AP(y==0)));

/* Provide Declarations */
#include <stdarg.h>
#include <setjmp.h>
#include <limits.h>
#include <stdint.h>
#include <math.h>

/* Global Declarations */

/* Types Declarations */
struct l_struct_x_type;
struct l_struct_y_type;
struct l_struct_union_OC_anon;
struct l_struct_struct_OC_ArchState;
struct l_struct_struct_OC_uint64v8_t;
struct l_struct_union_OC_vec512_t;
struct l_struct_union_OC_VectorReg;
struct l_struct_struct_OC_ArithFlags;
struct l_struct_union_OC_SegmentSelector;
struct l_struct_struct_OC_Segments;
struct l_struct_struct_OC_Reg;
struct l_struct_struct_OC_AddressSpace;
struct l_struct_struct_OC_GPR;
struct l_struct_struct_OC_anon_OC_3;
struct l_struct_struct_OC_X87Stack;
struct l_struct_struct_OC_uint64v1_t;
struct l_struct_union_OC_vec64_t;
struct l_struct_struct_OC_anon_OC_4;
struct l_struct_struct_OC_MMX;
struct l_struct_struct_OC_FPUStatusFlags;
struct l_struct_union_OC_FPUAbridgedTagWord;
struct l_struct_union_OC_FPUControlStatus;
struct l_struct_struct_OC_float80_t;
struct l_struct_union_OC_anon_OC_11;
struct l_struct_struct_OC_FPUStackElem;
```

```
struct l_struct_struct_OC_uint128v1_t;
struct l_struct_union_OC_vec128_t;
struct l_struct_struct_OC_FpuFXSAVE;

/* Types Definitions */
struct l_array_4_ureplace_u8int {
   int array[4];
};
struct l_struct_x_type {
  struct l_array_4_ureplace_u8int field0;
} __attribute__ ((packed));
struct l_array_8_ureplace_u8int {
   int array[8];
};
struct l_struct_y_type {
  struct l_array_8_ureplace_u8int field0;
} __attribute__ ((packed));
struct l_struct_union_OC_anon {
   int field0;
};
struct l_struct_struct_OC_ArchState {
   int field0;
   int field1;
  struct l_struct_union_OC_anon field2;
};
struct l_array_8_ureplace_u64int {
   int array[8];
};
struct l_struct_struct_OC_uint64v8_t {
  struct l_array_8_ureplace_u64int field0;
};
struct l_struct_union_OC_vec512_t {
  struct l_struct_struct_OC_uint64v8_t field0;
};
struct l_struct_union_OC_VectorReg {
  struct l_struct_union_OC_vec512_t field0;
};
struct l_array_32_struct_AC_l_struct_union_OC_VectorReg {
  struct l_struct_union_OC_VectorReg array[32];
```

```
};
struct l_struct_struct_OC_ArithFlags {
   int field0;
   int field1;
   int field2;
   int field3;
   int field4;
   int field5;
   int field6;
   int field7;
   int field8;
   int field9;
   int field10;
   int field11;
   int field12;
   int field13;
   int field14;
   int field15;
};
struct l_struct_union_OC_SegmentSelector {
   short field0;
};
struct l_struct_struct_OC_Segments {
   short field0;
  struct l_struct_union_OC_SegmentSelector field1;
   short field2;
  struct l_struct_union_OC_SegmentSelector field3;
   short field4;
  struct l_struct_union_OC_SegmentSelector field5;
   short field6;
  struct l_struct_union_OC_SegmentSelector field7;
   short field8;
  struct l_struct_union_OC_SegmentSelector field9;
   short field10;
  struct l_struct_union_OC_SegmentSelector field11;
};
struct l_struct_struct_OC_Reg {
  struct l_struct_union_OC_anon field0;
};
```

```
116 struct l_struct_struct_OC_AddressSpace {
117   int field0;
118  struct l_struct_struct_OC_Reg field1;
119   int field2;
120  struct l_struct_struct_OC_Reg field3;
121   int field4;
122  struct l_struct_struct_OC_Reg field5;
123   int field6;
124  struct l_struct_struct_OC_Reg field7;
125   int field8;
126  struct l_struct_struct_OC_Reg field9;
127   int field10;
128  struct l_struct_struct_OC_Reg field11;
129 };
130 struct l_struct_struct_OC_GPR {
131   int field0;
132  struct l_struct_struct_OC_Reg field1;
133   int field2;
134  struct l_struct_struct_OC_Reg field3;
135   int field4;
136  struct l_struct_struct_OC_Reg field5;
137   int field6;
138  struct l_struct_struct_OC_Reg field7;
139   int field8;
140  struct l_struct_struct_OC_Reg field9;
141   int field10;
142  struct l_struct_struct_OC_Reg field11;
143   int field12;
144  struct l_struct_struct_OC_Reg field13;
145   int field14;
146  struct l_struct_struct_OC_Reg field15;
147   int field16;
148  struct l_struct_struct_OC_Reg field17;
149   int field18;
150  struct l_struct_struct_OC_Reg field19;
151   int field20;
152  struct l_struct_struct_OC_Reg field21;
153   int field22;
154  struct l_struct_struct_OC_Reg field23;
```

```
155   int field24;
156   struct l_struct_struct_OC_Reg field25;
157   int field26;
158   struct l_struct_struct_OC_Reg field27;
159   int field28;
160   struct l_struct_struct_OC_Reg field29;
161   int field30;
162   struct l_struct_struct_OC_Reg field31;
163   int field32;
164   struct l_struct_struct_OC_Reg field33;
165 };
166 struct l_struct_struct_OC_anon_OC_3 {
167   int field0;
168   int field1;
169 };
170 struct l_array_8_struct_AC_l_struct_struct_OC_anon_OC_3 {
171   struct l_struct_struct_OC_anon_OC_3 array[8];
172 };
173 struct l_struct_struct_OC_X87Stack {
174   struct l_array_8_struct_AC_l_struct_struct_OC_anon_OC_3 field0;
175 };
176 struct l_array_1_ureplace_u64int {
177   int array[1];
178 };
179 struct l_struct_struct_OC_uint64v1_t {
180   struct l_array_1_ureplace_u64int field0;
181 };
182 struct l_struct_union_OC_vec64_t {
183   struct l_struct_struct_OC_uint64v1_t field0;
184 };
185 struct l_struct_struct_OC_anon_OC_4 {
186   int field0;
187   struct l_struct_union_OC_vec64_t field1;
188 };
189 struct l_array_8_struct_AC_l_struct_struct_OC_anon_OC_4 {
190   struct l_struct_struct_OC_anon_OC_4 array[8];
191 };
192 struct l_struct_struct_OC_MMX {
193   struct l_array_8_struct_AC_l_struct_struct_OC_anon_OC_4 field0;
```

```
};
struct l_struct_struct_OC_FPUStatusFlags {
   int field0;
   int field1;
   int field2;
   int field3;
   int field4;
   int field5;
   int field6;
   int field7;
   int field8;
   int field9;
   int field10;
   int field11;
   int field12;
   int field13;
   int field14;
   int field15;
   int field16;
   int field17;
   int field18;
   int field19;
  struct l_array_4_ureplace_u8int field20;
};
struct l_struct_union_OC_FPUAbridgedTagWord {
   int field0;
};
struct l_struct_union_OC_FPUControlStatus {
   int field0;
};
struct l_array_10_ureplace_u8int {
   int array[10];
};
struct l_struct_struct_OC_float80_t {
  struct l_array_10_ureplace_u8int field0;
};
struct l_struct_union_OC_anon_OC_11 {
  struct l_struct_struct_OC_float80_t field0;
};
```

```
struct l_array_6_ureplace_u8int {
   int array[6];
};
struct l_struct_struct_OC_FPUStackElem {
  struct l_struct_union_OC_anon_OC_11 field0;
  struct l_array_6_ureplace_u8int field1;
};
struct l_array_8_struct_AC_l_struct_struct_OC_FPUStackElem {
  struct l_struct_struct_OC_FPUStackElem array[8];
};
struct l_array_96_ureplace_u8int {
   int array[96];
};
struct l_struct_struct_OC_SegmentShadow {
  struct l_struct_union_OC_anon field0;
   int field1;
   int field2;
};
struct l_struct_struct_OC_SegmentCaches {
  struct l_struct_struct_OC_SegmentShadow field0;
  struct l_struct_struct_OC_SegmentShadow field1;
  struct l_struct_struct_OC_SegmentShadow field2;
  struct l_struct_struct_OC_SegmentShadow field3;
  struct l_struct_struct_OC_SegmentShadow field4;
  struct l_struct_struct_OC_SegmentShadow field5;
};
struct l_struct_struct_OC_State {
  struct l_struct_struct_OC_ArchState field0;
  struct l_array_32_struct_AC_l_struct_union_OC_VectorReg field1;
  struct l_struct_struct_OC_ArithFlags field2;
  struct l_struct_union_OC_anon field3;
  struct l_struct_struct_OC_Segments field4;
  struct l_struct_struct_OC_AddressSpace field5;
  struct l_struct_struct_OC_GPR field6;
  struct l_struct_struct_OC_X87Stack field7;
  struct l_struct_struct_OC_MMX field8;
  struct l_struct_struct_OC_FPUStatusFlags field9;
  struct l_struct_union_OC_anon field10;
  struct l_struct_struct_OC_SegmentCaches field12;
```

```
272 };
273
274 /* External Global Variable Declarations */
275 extern struct l_struct_struct_OC_State* globalState;
276
277 /* Function Declarations */
278 sub_401106_foo(struct l_struct_struct_OC_State* tmp__14, int tmp__15, void* tmp__16);
279 void* sub_401106___VERIFIER_nondet_int(struct l_struct_struct_OC_State* tmp__39, int
        tmp__40, void* tmp__41);
280 extern void __VERIFIER_nondet_unsigned() __attribute__ ((__));
281
282 extern void __VERIFIER_assume() __attribute__ ((__noreturn__));
283
284 /* Global Variable Definitions and Initialization */
285 int x;
286 int y;
287 int STATE_REG_RAX ;
288
289
290 /* LLVM Intrinsic Builtin Function Bodies */
291 static   int llvm_add_u32( int a,  int b) {
292   int r = a + b;
293  return r;
294 }
295 static   int llvm_lshr_u32( int a,  int b) {
296   int r = a >> b;
297  return r;
298 }
299 static   int llvm_and_u8( int a,  int b) {
300   int r = a & b;
301  return r;
302 }
303 static   int llvm_xor_u8( int a,  int b) {
304   int r = a ^ b;
305  return r;
306 }
307
308
309 /* Function Bodies */
```

```
310
311 void* main(struct l_struct_struct_OC_State* tmp__1,  int tmp__2, void* tmp__3) {
312  struct l_struct_struct_OC_State* tmp__4;
313   int* tmp__5;
314   int* tmp__6;
315   int* tmp__7;
316   int* tmp__8;
317   int* tmp__9;
318   int* tmp__10;
319  void* tmp__11;
320   int tmp__12;
321   int tmp__13;
322   int tmp__14;
323   int tmp__14__PHI_TEMPORARY;
324   int tmp__15;
325   int tmp__16;
326   int tmp__17;
327   int tmp__18;
328   int _2e_lcssa3;
329   int _2e_lcssa3__PHI_TEMPORARY;
330  int _2e_lcssa2;
331  int _2e_lcssa2__PHI_TEMPORARY;
332   int _2e_lcssa1;
333   int _2e_lcssa1__PHI_TEMPORARY;
334   int tmp__19;
335
336  tmp__4 = globalState;
337  tmp__5 = (&tmp__4->field2.field1);
338  tmp__6 = (&tmp__4->field2.field3);
339  tmp__7 = (&tmp__4->field2.field7);
340  tmp__8 = (&tmp__4->field2.field9);
341  tmp__9 = (&tmp__4->field2.field13);
342  tmp__10 = (&tmp__4->field2.field5);
343  goto block_401119;
344
345  do {     /* Syntactic loop 'block_401119' to make GCC happy */
346 block_401119:
347     STATE_REG_RAX = __VERIFIER_nondet_int();
348
```

```
349     tmp__11 =  /*tail*/ sub_401106___VERIFIER_nondet_int(/*UNDEF*/((struct
        l_struct_struct_OC_State*)/*NULL*/0), /*UNDEF*/(0UL), tmp__3);
350     tmp__12 = STATE_REG_RAX;
351     x = tmp__12;
352     y = 1;
353     tmp__13 = tmp__12 >> 31;
354
355
356   /* if (((((((tmp__13 == 0u)&1)) & (((~((((tmp__12 == 0u)&1))))&1)))&1))) { */
357     if (((((tmp__13 == 0u)&1)) && (tmp__12 != 0u))&1) {
358
359     tmp__14__PHI_TEMPORARY = tmp__12;   /* for PHI node */
360     goto block_401135;
361   } else {
362     _2e_lcssa3__PHI_TEMPORARY = tmp__12;   /* for PHI node */
363     _2e_lcssa2__PHI_TEMPORARY = (((tmp__12 == 0u)&1));   /* for PHI node */
364     _2e_lcssa1__PHI_TEMPORARY = tmp__13;   /* for PHI node */
365     goto block_401163;
366   }
367
368   do {     /* Syntactic loop 'block_401135' to make GCC happy */
369 block_401135:
370   tmp__14 = tmp__14__PHI_TEMPORARY;
371   tmp__15 = tmp__14-1;
372   tmp__16 = tmp__14-2;
373   tmp__17 = tmp__16>>31;
374   tmp__18 = tmp__15>>31;
375
376   /* if (((((((tmp__16 != 0u)&1)) & ((((((tmp__17 == 0u)&1)) ^ ((((llvm_add_u32((
        tmp__17 ^ tmp__18), tmp__18)) == 2u)&1)))&1)))&1))) { */
377   if ( ((tmp__16 != 0u)&1) &&(tmp__17 == 0u)) {
378     goto block_401159_2e_backedge;
379   } else {
380     goto block_40114f;
381   }
382
383 block_40114f:
384   y = 0;
385   goto block_401159_2e_backedge;
```

```
386
387 block_401159_2e_backedge:
388   /* if (((((((tmp__18 == 0u)&1)) & (((~((((tmp__15 == 0u)&1))))&1)))&1))) { */
389
390     if (((((((tmp__18 == 0u)&1)) && ( tmp__15 != 0u) )&1))) {
391     tmp__14__PHI_TEMPORARY = tmp__15;   /* for PHI node */
392     goto block_401135;
393   } else {
394     goto block_401159_2e_block_401163_crit_edge;
395   }
396
397   } while (1); /* end of syntactic loop 'block_401135' */
398 block_401159_2e_block_401163_crit_edge:
399    x = tmp__15;
400   _2e_lcssa3__PHI_TEMPORARY = tmp__15;   /* for PHI node */
401   _2e_lcssa2__PHI_TEMPORARY = (((tmp__15 == 0u)&1));   /* for PHI node */
402   _2e_lcssa1__PHI_TEMPORARY = tmp__18;   /* for PHI node */
403   goto block_401163;
404
405 block_401163:
406   _2e_lcssa3 = _2e_lcssa3__PHI_TEMPORARY;
407   _2e_lcssa2 = ((_2e_lcssa2__PHI_TEMPORARY)&1);
408   _2e_lcssa1 = _2e_lcssa1__PHI_TEMPORARY;
409   tmp__19 =  /*tail*/ llvm_OC_ctpop_OC_i32((_2e_lcssa3 & 255));
410   *tmp__5 = 0;
411   *tmp__6 =  ( ((((int)tmp__19))& 1))^ 1 ;
412   *tmp__7 = ((( int)(int)_2e_lcssa2));
413   *tmp__8 = ((( int)_2e_lcssa1));
414   *tmp__9 = 0;
415   *tmp__10 = 0;
416   goto block_401119;
417
418   } while (1); /* end of syntactic loop 'block_401119' */
419 }
```

A.3 Bug in GCC

Example 1 (A reported bug in `gcc` [144])**.**

```
1  static uint64_t div(uint64_t ui1, uint64_t ui2){
2    return (ui2 == 0) ? ui1 : (ui1 / ui2); }
3  static int8_t mod(int8_t si1, int8_t si2){
4    return (si2==0) || ((si1==-128) && (si2==-1)) ?  si1 : (si1 % si2); }
5  static int32_t g_5=0, g_11=0;
6  int main (){
7    uint64_t l_7 = 0x509CB0BEFCDF11BBLL;
8    g_11 ^= l_7 && ((div((mod(g_5, 0)), -1L)) != 1L);
9    if (!g_11) return __VERIFIER_error();
10   return 0;
11 }
```

From the source semantics of this example, variable `g_11` would have value 1 at Line 9 and `__VERIFIER_error` is un-reachable. However, a defect [7] in `gcc` folds `((div((mod(g_5, 0)), -1L)) != 1L)` to 0. This makes `g_11` become 0 at Line 9, introducing a new error behavior.

A.4 DRNLA on CTLNLABench-DYNAMITE benchmarks

Benchmark	Res	Time	It.	Ref. Stages	Output
bresenham1-F.c	✔	210.1	2	v,en,v,v	$\ell_{36} : 2Yx-2X^2y+2Y-v+c \leq k \mapsto 0 \geq c-k, k-x \leq -1$
bresenham1-T.c	✔	204.2	2	v,en,v,v	$\ell_{36} : 2Yx-2X^2y+2Y-v+c \leq k \mapsto 0 \geq c-k, k-c \leq -1$
cohencu2-F.c	≈	517.5	1	v,v	$\ell_{28} : 3n^2+3n+1 \leq k \mapsto 0 \geq y-k, k-y \leq -1$
cohencu2-T.c	≈	473.7	1	v,v	$\ell_{32} : 3n^2+3n+1 \leq k \mapsto 0 \geq y-k, k-y \leq -1$
cohencu3-F.c	≈	621.8	1	v,v	$\ell_{31} : n^3 \leq k \mapsto 0 \geq c-k, ((k-x) <= -(1))$
cohencu3-T.c	≈	621.4	1	v,v	$\ell_{31} : n^3 \leq k \mapsto 0 \geq c-k, ((k-x) <= -(1))$
cohencu4-F.c	≈	780.3	2	v,en,v,v	$\ell_{31} : yz-18x-12y+2z-6+c \leq k \mapsto 0 \geq c-k, k-c \leq -1$
cohencu4-T.c	≈	773.9	2	v,en,v,v	$\ell_{31} : yz-18x-12y+2z-6+c \leq k \mapsto 0 \geq c-k, k-c \leq -1$
cohencu5-F.c	✔	179.4	2	v,en,v,v	$\ell_{35} : z^2-12y-6z+12+c \leq k \mapsto 0 \geq c-k, k-c \leq -1$
cohencu5-T.c	✔	177.4	2	v,en,v,v	$\ell_{35} : z^2-12y-6z+12+c \leq k \mapsto 0 \geq n-k, (((0+(k1))+(n-1)) <= -1)$
cohencu7-F.c	≈	194.9	2	v,en,v,v	$\ell_{30} : ((x+y) <= (((a+1)(a+1))(a+1))) \mapsto (0 >= (-(a)+n)), (((0+(n-1))+(a1)) <= -1)$
cohencu7-T.c	≈	667.5	2	v,en,v,v	$\ell_{28} : ((x+y) <= (((a+1)(a+1))(a+1))) \mapsto (0 >= (-(a)+n)), (((0+(n-1))+(a1)) <= -1)$
dijkstra2-F.c	≈	687.5	2	v,en,v,v	$\ell_{46} : (((((xpxp)+(rq))-(nq))+c) <= k) \mapsto 0 \geq c-k, k-c \leq -1$
dijkstra2-T.c	≈	686.0	2	v,en,v,v	$\ell_{48} : (((((xpxp)+(rq))-(nq))+c) <= k) \mapsto 0 \geq c-k, k-c \leq -1$
dijkstra3-F.c	≈	628.6	1	v,v	$\ell_{45} : h^3-12hnq+16nx'-hq^2-4x'q^2+12hqr-6x'qr+c \leq k \mapsto 0 \geq c-k, ((-(c)+k) <= -(1))$
dijkstra3-T.c	≈	627.5	1	v,v	$\ell_{45} : h^3-12hnq+16nx'-hq^2-4x'q^2+12hqr-6x'qr+c \leq k \mapsto 0 \geq c-k, ((-(c)+k) <= -(1))$
dijkstra4-F.c	≈	629.7	1	v,v	$\ell_{45} : ((((((((((((hh)n)-(((4h)n)xp))+((4(nn))q))-((nq)q))-((hh)r))+(((4h)xp)r))-(((8n)q)r))+((qq)r))+(((4q)r)r))+c) <= k) \mapsto 0 \geq c-k, ((-(c)+k) <= -(1))$
dijkstra4-T.c	≈	629.1	1	v,v	$\ell_{45} : ((((((((((((hh)n)-(((4h)n)xp))+((4(nn))q))-((nq)q))-((hh)r))+(((4h)xp)r))-(((8n)q)r))+((qq)r))+(((4q)r)r))+c) <= k) \mapsto 0 \geq c-k, ((-(c)+k) <= -(1))$
dijkstra5-F.c	≈	630.8	1	v,v	$\ell_{46} : (((((((((hh)xp)-(((4h)n)q))+(((4n)xp)q))-((xpq)q))+(((4h)q)r))-(((4xp)q)r))+c) <= k) \mapsto 0 \geq c-k, ((-(c)+k) <= -(1))$

dijkstra5-T.c	≈	613.9	1	v,v	$\ell_{46} : ((((((((((hh)xp) - (((4h)n)q)) + (((4n)xp)q)) - ((xpq)q)) + (((4h)q)r)) - (((4xp)q)r)) + c) <= k) \mapsto 0 \geq c - k, ((-(c) + k) <= -(1))$
divbin1-F.c	?	1.2	0		
divbin1-T.c	?	1.3	0		
egcd-F.c	≈	**TO**	0		
egcd-T.c	≈	**TO**	0		
egcd2-F.c	≈	696.0	1	v,v	$\ell_{33} : (c >= ((xq) + (ys))) \mapsto (0 >= (b - c)), ((-(b) + c) <= -(1))$
egcd2-T.c	≈	681.8	1	v,v	$\ell_{33} : (c >= ((xq) + (ys))) \mapsto (0 >= (b - c)), ((-(b) + c) <= -(1))$
egcd3-F.c	≈	**TO**	8	v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v	
egcd3-T.c	≈	**TO**	8	v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v,tn,v,tp, v	
fermat1-F.c	≈	**TO**	0		
fermat1-T.c	≈	**TO**	0		
geo1-F.c	✔	131.0	1	v,v	$\ell_{24} :!(((((((xz) - x) - y) + 1) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
geo1-T.c	✔	130.6	1	v,v	$\ell_{24} :!(((((((xz) - x) - y) + 1) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
geo2-F.c	✔	122.5	1	v,v	$\ell_{24} :!((((((1 + (xz)) - x) - (zy)) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
geo2-T.c	✔	134.7	1	v,v	$\ell_{25} :!((((((1 + (xz)) - x) - (zy)) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
geo3-F.c	✔	171.1	1	v,v	$\ell_{25} :!(((((((zx) - x) + a) - ((az)y)) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
geo3-T.c	✔	163.5	1	v,v	$\ell_{25} :!(((((((zx) - x) + a) - ((az)y)) + c) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
hard-F.c	?	7.9	0		
hard-T.c	?	7.4	0		
hard2-F.c	?	0.9	0		
hard2-T.c	?	0.8	0		

prod4br-F.c	≈	239.0	5	v,ep,v,en, v,tn,v,en, v,tn,v	
prod4br-T.c	≈	159.3	4	v,ep,v,en, v,tn,v,en, v	
prodbin-F.c	≈	654.5	2	v,en,v,v	ℓ_{26} :!((0! = (((y + z) + (xy)) − (ab)))) ↦ (0 >= y), ((0+(y−1)) <= −1)&&(((0+(p−1))+(y−1)) <= −2)&&((0+(p−1)) <= −1)&&(((0+(p1))+(y−1)) <= 1)
prodbin-T.c	≈	619.6	1	v,v	ℓ_{26} :!((0! = (((y + z) + (xy)) − (ab)))) ↦ (0 >= y), (−(y) <= −(1))
ps2-F.c	✔	29.0	1	v,v	ℓ_{20} :!(((((c + (yy)) − (2x)) + y) < k)) ↦ (0 >= (−(c) + k)), ((c − k) <= −(1))
ps2-T.c	✔	53.4	2	v,ep,v,v	ℓ_{20} :!(((((c + (yy)) − (2x)) + y) < k)) ↦ (((0 + (k1)) + (y − 1)) <= 0), ((−(k) + y) <= −(1))
ps3-F.c	✔	62.8	2	v,ep,v,v	ℓ_{20} :!(((((((c + (6x)) − (((2y)y)y)) − ((3y)y)) − y) < k)) ↦ (((0 + (k1)) + (c − 1)) <= 0), ((c − k) <= −(1))
ps3-T.c	✔	44.9	1	v,v	ℓ_{20} :!(((((((c + (6x)) − (((2y)y)y)) − ((3y)y)) − y) < k)) ↦ (0 >= (k − y)), ((−(k) + y) <= −(1))
ps4-F.c	✔	59.8	2	v,ep,v,v	ℓ_{20} :!(((((((c+(4x))−(((yy)y)y))−(((2y)y)y))−(yy)) < k)) ↦ (((0+(k1))+(y−1)) <= 0), ((−(k)+y) <= −(1))
ps4-T.c	✔	58.1	2	v,ep,v,v	ℓ_{20} :!(((((((c+(4x))−(((yy)y)y))−(((2y)y)y))−(yy)) < k)) ↦ (((0+(k1))+(y−1)) <= 0), ((−(k)+y) <= −(1))
ps5-F.c	✔	30.4	1	v,v	ℓ_{19} :!(((((((c + (((((6y)y)y)y)y)) + ((((15y)y)y)y)) + (((10y)y)y)) − (30x)) − y) < k)) ↦ (0 >= (−(c) + k)), ((c − k) <= −(1))
ps5-T.c	✔	41.3	1	v,v	ℓ_{19} :!(((((((c + (((((6y)y)y)y)y)) + ((((15y)y)y)y)) + (((10y)y)y)) − (30x)) − y) < k)) ↦ (0 >= (k − y)), ((−(k) + y) <= −(1))
ps6-F.c	✔	60.3	2	v,ep,v,v	ℓ_{19} : ¬(c − 2y^6 − 6y^5 − 5y^4 + y^2 + 12x ≤ k) ↦ (((0 + (k1)) + (c − 1)) <= 0), ((c − k) <= −(1))
ps6-T.c	✔	68.6	2	v,ep,v,v	ℓ_{19} : ¬(c − 2y^6 − 6y^5 − 5y^4 + y^2 + 12x ≤ k) ↦ (((0 + (k1)) + (y − 1)) <= 0), ((−(k) + y) <= −(1))
sqrt1-F.c	✔	117.0	2	v,en,v,v	ℓ_{21} : $t^2 - 4s + 2t + 1 + c \leq k \mapsto 0 \geq a - k, k - a \leq -1$
sqrt1-T.c	✔	116.3	2	v,en,v,v	ℓ_{21} : $t^2 - 4s + 2t + 1 + c \leq k \mapsto 0 \geq a - k, k - a \leq -1$

A.5 DRNLA on CTLNLABench-PLDI13 benchmarks

Benchmark	Res	Time	It.	Ref. Stages	Output
neg-afagp-F.c	✔	289.1	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
neg-afagp-T.c	✔	287.2	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
neg-afefp-F.c	✔	225.2	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (n - 1)) + (k1)) <= -1)$
neg-afegp-F.c	✔	226.0	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
neg-afegp-T.c	✔	217.4	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
neg-afp-F.c	✔	164.4	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
neg-afp-T.c	✔	160.9	2	v,en,v,v	$\ell_{18} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
neg-efafp-F.c	✔	243.0	2	v,en,v,v	$\ell_{31} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
neg-egafp-F.c	✔	282.2	2	v,en,v,v	$\ell_{19} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
neg-egafp-T.c	✔	313.3	2	v,en,v,v	$\ell_{19} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
neg-egimpafp-T.c	✔	251.4	2	v,en,v,v	$\ell_{20} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
afagp-F.c	✔	103.9	2	v,ep,v,v	$\ell_{18} :!(((((c + (yy)) - (2x)) + y) < k)) \mapsto (((0 + (k1)) + (c - 1)) <= 0), ((c - k) <= -(1))$
afagp-T.c	✔	74.8	1	v,v	$\ell_{18} :!(((((c + (yy)) - (2x)) + y) < k)) \mapsto (0 >= (-(c) + k)), ((c - k) <= -(1))$
afefp-T.c	✔	179.8	2	v,en,v,v	$\ell_{19} : t^2 - 4s + 2t + 1 + c \leq k \mapsto 0 \geq a - k, k - a \leq -1$
afegp-F.c	✔	239.7	2	v,en,v,v	$\ell_{18} : t^2 - 4s + 2t + 1 + c \leq k \mapsto 0 \geq a - k, k - a \leq -1$
afegp-T.c	✔	177.8	2	v,en,v,v	$\ell_{19} : t^2 - 4s + 2t + 1 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
afp-F.c	✔	163.8	2	v,en,v,v	$\ell_{19} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
afp-T.c	✔	166.8	2	v,en,v,v	$\ell_{19} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, k - c \leq -1$
agafp-F.c	✔	65.7	1	v,v	$\ell_{23} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq (c - k)\&\&(0 == (c - n)), ((-(n) + k) <= -(1))$
agafp-T.c	✔	36.3	1	v,v	$\ell_{23} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq c - k, ((-(c) + k) <= -(1))$

efafp-T.c	✔	231.1	2	v,en,v,v	$\ell_{29} : z^2 - 12y - 6z + 12 + c \leq k \mapsto 0 \geq n - k, (((0 + (k1)) + (n - 1)) <= -1)$
afagp-T.c	✔	160.3	1	v,v	ℓ_{21} :!(((((((xz) − x) − y) + 1) + c) < k)) ↦ (0 >= (−(c) + k)), ((c − k) <= −(1))
afefp-T.c	✔	312.1	2	v,en,v,v	$\ell_{23} : (((((((((2Y)xp)-((2X)y))-X)+(2Y))-v)+c) <= k) \mapsto 0 \geq c - k, k - c \leq -1$
agafp-T.c	✔	24.4	1	v,v	$\ell_{22} : (((((c+(4x))-(((yy)y)y))-(((2y)y)y))-(yy)) < k) \mapsto ((c - k) <= -(1)), (0 >= (-(c) + k))$
neg-afefp-F.c	✔	233.2	1	v,v	$\ell_{17} : (((((1+(xz))-x)-(zy))+c) < k) \mapsto ((c-k) <= -(1)), (0 >= (-(c) + k))$
neg-afp-F.c	✔	68.2	2	v,en,v,v	$\ell_{17} : (((((c+(6x))-(((2y)y)y))-((3y)y))-y) < k) \mapsto ((c - k) <= -(1)), (((0 + (k1)) + (c - 1)) <= 0)$

A.6 DRNLA on Handcrafted benchmarks

Benchmark	Res	Time	It.	Stages	Output
`if-cubic-F.c`	✔	51.5	3	v, tn, v, en, v, v	$\ell_6 : (8 == ((xx)x)) \mapsto (((4 >= (p + x))\&\&(0 >= (p - x))\&\&(0 >= (-(p) + x))\&\&((-(p) - x) <= -(4)))\|\|((2 >= p)\&\&(0 >= (-(p) + x))\&\&((-(p) - x) <= -(4)))), (((0 == (p - 2))\&\&(1 >= (p - x))\&\&!(((2 >= p)\&\&(0 >= (-(p) + x))\&\&((-(p) - x) <= -(4)))))\|\|(0 >= x))$
`if-cubic-T.c`	✔	51.3	3	v, tn, v, en, v, v	$\ell_6 : (8 == ((xx)x)) \mapsto (((2 >= p)\&\&(0 >= (-(p) + x))\&\&((-(p) - x) <= -(4)))\|\|((-(x) <= -(2))\&\&(4 >= (p + x))\&\&(0 >= (-(p) + x)))), (((0 == (p - 2))\&\&(0 >= -(x))\&\&!(((-(x) <= -(2))\&\&(4 >= (p+x))\&\&(0 >= (-(p)+x)))))\|\|(1 >= (p + x)))$
`if-F.c`	✔	78.8	6	v, tn, v, ep, v, tp, v, tn, v, tp, v, v	$\ell_6 : (36 == (xx)) \mapsto ((((0 + (x1)) <= 6)\&\&!(((0 == (p - 2))\&\&((-(p) + x) <= -(1)))))\|\|((x <= -(6))\&\&(-(p) <= -(2))\&\&(8 >= (p - x))))\&\&!(((0 == (p - 2))\&\&(0 >= (p - x))\&\&(3 >= (-(p) + x)))), (((((0 == (p - 2))\&\&(0 >= -(x))\&\&!(((2 >= p)\&\&(-(x) <= -(6))\&\&(4 >= (-(p) + x)))))\|\|((0 == (p - 2))\&\&((-(p) + x) <= -(1))))\&\&!(((x <= -(6))\&\&(-(p) <= -(2))\&\&(8 >= (p - x)))))\|\|((0 == (p - 2))\&\&(0 >= (p - x))\&\&(3 >= (-(p) + x))))$
`if-T.c`	✔	76.5	6	v, tn, v, ep, v, tp, v, tn, v, tp, v, v	$\ell_7 : (36 == (xx)) \mapsto (((((0 + (x1)) + (p1)) <= 8)\&\&!(((0 == (p - 2))\&\&(3 >= x))))\|\|((-(p) <= -(2))\&\&(8 >= (p - x))\&\&((p + x) <= -(4))))\&\&!(((2 >= p)\&\&(5 >= x)\&\&(-(p) <= -(2))\&\&((-(p) - x) <= -(6)))), (((((0 == (p - 2))\&\&(1 >= (p - x))\&\&!(((6 >= x)\&\&(-(p) <= -(2))\&\&((p - x) <= -(4)))))\|\|((0 == (p - 2))\&\&(3 >= x)))\&\&!(((-(p) <= -(2))\&\&(8 >= (p - x))\&\&((p + x) <= -(4)))))\|\|((2 >= p)\&\&(5 >= x)\&\&(-(p) <= -(2))\&\&((-(p) - x) <= -(6))))$

square-loop-F.c	✔	383.6	12	v, tn, v, tn, v, tn, v, tn, v, tn, v, tn, v, tn, v, v	$\ell_8 : (49 < (xx)) \mapsto ((((((((((((-(x) <= -(8))\|\|((0 == (p-2))\&\&(3 >= y)\&\&((-(p)-y) <= -(3))\&\&((-(p)+x) <= -(10))))\|\|((0 == (p-2))\&\&(7 >= (p+y))\&\&((x+y) <= -(4))\&\&((-(p)-y) <= -(6))))\|\|((0 == (p-2))\&\&(10 >= y)\&\&((-(p)-y) <= -(7))\&\&((-(p)+x) <= -(10))))\|\|((0 == (p-2))\&\&(x <= -(8))\&\&(10 >= (-(p)+y))\&\&((-(p)-y) <= -(13))))\|\|((0 == (p-2))\&\&(x <= -(8))\&\&(16 >= (-(p)+y))\&\&((-(p)-y) <= -(15))))\|\|((0 == (p-2))\&\&((p+x) <= -(6))\&\&((p-y) <= -(17))\&\&(19 >= (-(p)+y))))\|\|((0 == (p-2))\&\&(17 >= (x+y))\&\&((p-y) <= -(20))\&\&((-(p)+x) <= -(10))))\|\|((0 == (p-2))\&\&((p+x) <= -(6))\&\&((-(x)-y) <= -(18))))\|\|((0 == (p-1))\&\&(14 >= y)\&\&(5 >= (x+y))\&\&(0 >= (p-y))\&\&((-(p)+x) <= -(9))))\|\|((0 == (p-1))\&\&(8 >= (x+y))\&\&((p-y) <= -(13))\&\&((-(p)+x) <= -(9))))\|\|((0 == (p-1))\&\&((-(x)-y) <= -(9))\&\&((-(p)+x) <= -(9)))), (7 >= x)\&\&!(((0 == (p-2))\&\&(3 >= y)\&\&((-(p)-y) <= -(3))\&\&((-(p)+x) <= -(10))))\&\&!(((0 == (p-2))\&\&(7 >= (p+y))\&\&((x+y) <= -(4))\&\&((-(p)-y) <= -(6))))\&\&!(((0 == (p-2))\&\&(10 >= y)\&\&((-(p)-y) <= -(7))\&\&((-(p)+x) <= -(10))))\&\&!(((0 == (p-2))\&\&(x <= -(8))\&\&(10 >= (-(p)+y))\&\&((-(p)-y) <= -(13))))\&\&!(((0 == (p-2))\&\&(x <= -(8))\&\&(16 >= (-(p)+y))\&\&((-(p)-y) <= -(15))))\&\&!(((0 == (p-2))\&\&((p+x) <= -(6))\&\&((p-y) <= -(17))\&\&(19 >= (-(p)+y))))\&\&!(((0 == (p-2))\&\&(17 >= (x+y))\&\&((p-y) <= -(20))\&\&((-(p)+x) <= -(10))))\&\&!(((0 == (p-2))\&\&((p+x) <= -(6))\&\&((-(x)-y) <= -(18))))\&\&!(((0 == (p-1))\&\&(14 >= y)\&\&(5 >= (x+y))\&\&(0 >= (p-y))\&\&((-(p)+x) <= -(9))))\&\&!(((0 == (p-1))\&\&(8 >= (x+y))\&\&((p-y) <= -(13))\&\&((-(p)+x) <= -(9))))\&\&!(((0 == (p-1))\&\&((-(x)-y) <= -(9))\&\&((-(p)+x) <= -(9))))$

square-loop-T.c	✔	233.3	7	v, tn, v, tn, v, tn, v, tn, v, tn, v, v	$\ell_8 : (49 < (xx)) \mapsto (((((((-(x) <= -(8))\|\|((0 == (p - 2))\&\&(1 >= (p - y))\&\&(5 >= (p + y))\&\&((p + x) <= -(6))))\|\|((0 == (p - 2))\&\&(5 >= y)\&\&(-(y) <= -(4))\&\&((x + y) <= -(4))))\|\|((0 == (p - 2))\&\&((p + x) <= -(6))\&\&((x + y) <= -(1))\&\&((-(p) - y) <= -(7))))\|\|((0 == (p - 2))\&\&(0 >= (-(x) - y))\&\&((-(p) + x) <= -(10))))\|\|((0 == (p - 1))\&\&(0 >= (p - y))\&\&((x + y) <= -(6))\&\&((-(p) + x) <= -(9))))\|\|((0 == (p - 1))\&\&(x <= -(8))\&\&(5 >= (-(x) - y)))), (7 >= x)\&\&!(((0 == (p - 2))\&\&(1 >= (p - y))\&\&(5 >= (p + y))\&\&((p + x) <= -(6))))\&\&!(((0 == (p - 2))\&\&(5 >= y)\&\&(-(y) <= -(4))\&\&((x + y) <= -(4))))\&\&!(((0 == (p - 2))\&\&((p + x) <= -(6))\&\&((x + y) <= -(1))\&\&((-(p) - y) <= -(7))))\&\&!(((0 == (p - 2))\&\&(0 >= (-(x) - y))\&\&((-(p) + x) <= -(10))))\&\&!(((0 == (p - 1))\&\&(0 >= (p - y))\&\&((x + y) <= -(6))\&\&((-(p) + x) <= -(9))))\&\&!(((0 == (p - 1))\&\&(x <= -(8))\&\&(5 >= (-(x) - y))))$
while-cubic-F.c	✔	87.0	7	v, tn, v, ep, v, tp, v, tn, v, tp, v, tn, v, v	$\ell_6 : ((64 >= ((yy)y))\&\&(1 <= (yy))) \mapsto (((((((0 + (p1)) + (y1)) <= 6)\&\&!(((0 >= y)\&\&(2 >= (p - y))\&\&((-(p) - y) <= -(2)))))\|\|((0 == (p - 1))\&\&(5 >= (p + y))\&\&((p - y) <= -(1))))\&\&!(((1 >= p)\&\&(0 >= -(y))\&\&(-(p) <= -(1))\&\&(4 >= (-(p) + y)))))\|\|((0 == (p - 1))\&\&(4 >= y)\&\&((p - y) <= -(1)))), (((((-(p) <= -(1))\&\&(0 >= -(y))\&\&((-(p) - y) <= -(2))\&\&!(((0 == (p - 2))\&\&(4 >= y)\&\&((-(p) - y) <= -(3)))))\|\|((0 >= y)\&\&(2 >= (p - y))\&\&((-(p) - y) <= -(2))))\&\&!(((0 == (p - 1))\&\&(5 >= (p + y))\&\&((p - y) <= -(1)))))\|\|((1 >= p)\&\&(0 >= -(y))\&\&(-(p) <= -(1))\&\&(4 >= (-(p) + y))))\&\&!(((0 == (p - 1))\&\&(4 >= y)\&\&((p - y) <= -(1))))$

`while-cubic-T.c`	✔	84.4	7	v, tn, v, ep, v, tp, v, tn, v, tp, v, tn, v, v	ℓ_6 : $((64 >= ((yy)y))\&\&(1 <= (yy))) \mapsto (((((((0 + (p1)) + (y1)) <= 6)\&\&!(((0 >= -(y))\&\&(-(p) <= -(2))\&\&(2 >= (p + y)))))\|\|((0 == (p - 1))\&\&(4 >= y)\&\&(-(y) <= -(2))))\&\&!(((1 >= p)\&\&(5 >= y)\&\&(0 >= -(y))\&\&(-(p) <= -(1)))))\|\|((0 == (p - 1))\&\&((p - y) <= -(1))\&\&(3 >= (-(p) + y)))), (((((0 >= -(y))\&\&((-(p) - y) <= -(2))\&\&!(((0 == (p - 2))\&\&(4 >= y)\&\&(-(y) <= -(1)))))\|\|((0 >= -(y))\&\&(-(p) <= -(2))\&\&(2 >= (p + y))))\&\&!(((0 == (p - 1))\&\&(4 >= y)\&\&(-(y) <= -(2)))))\|\|((1 >= p)\&\&(5 >= y)\&\&(0 >= -(y))\&\&(-(p) <= -(1))))\&\&!(((0 == (p - 1))\&\&((p - y) <= -(1))\&\&(3 >= (-(p) + y))))$
`while-F.c`	✔	190.4	6	v, tn, v, tn, v, tn, v, tn, v, tn, v, v	ℓ_9 : $(63 <= ((xx) - (2x))) \mapsto (((((((p - x) <= -(7))\&\&((-(p) - x) <= -(10)))\|\|((0 == (p - 2))\&\&(1 >= (p - y))\&\&(0 >= (-(p) + y))\&\&((x + y) <= -(6))))\|\|((0 == (p - 2))\&\&(0 >= (p - y))\&\&((x + y) <= -(5))))\|\|((0 == (p - 2))\&\&(4 >= (-(x) - y))\&\&((-(p) + x) <= -(9))))\|\|((0 == (p - 1))\&\&(4 >= (p + y))\&\&(0 >= (p - y))\&\&((x + y) <= -(8))))\|\|((0 == (p - 1))\&\&(0 >= (p - y))\&\&((-(p) + x) <= -(8)))), (8 >= x)\&\&!(((0 == (p - 2))\&\&(1 >= (p - y))\&\&(0 >= (-(p) + y))\&\&((x + y) <= -(6))))\&\&!(((0 == (p - 2))\&\&(0 >= (p - y))\&\&((x + y) <= -(5))))\&\&!(((0 == (p - 2))\&\&(4 >= (-(x) - y))\&\&((-(p) + x) <= -(9))))\&\&!(((0 == (p - 1))\&\&(4 >= (p + y))\&\&(0 >= (p - y))\&\&((x + y) <= -(8))))\&\&!(((0 == (p - 1))\&\&(0 >= (p - y))\&\&((-(p) + x) <= -(8))))$

`while-T.c`	✔	189.6	6	v, tn, v, tn, v, tn, v, tn, v, tn, v, v	$\ell_9 : (63 <= ((xx) - (2x))) \mapsto ((((((-(x) <= -(9))\|\|((0 == (p-2))\&\&(1 >= (p-y))\&\&((x+y) <= -(6))\&\&(0 >= (-(p)+y))))\|\|((0 == (p-2))\&\&(x <= -(7))\&\&(0 >= (p-y))))\|\|((0 == (p-1))\&\&(0 >= (p-y))\&\&((x-y) <= -(9))\&\&(2 >= (-(p)+y))\&\&((x+y) <= -(5))))\|\|((0 == (p-1))\&\&(x <= -(7))\&\&(0 >= (p-y))\&\&((x+y) <= -(2))))\|\|((0 == (p-1))\&\&(1 >= (-(x)-y))\&\&((-(p)+x) <= -(8)))), (8 >= x)\&\&!(((0 == (p-2))\&\&(1 >= (p-y))\&\&((x+y) <= -(6))\&\&(0 >= (-(p)+y))))\&\&!(((0 == (p-2))\&\&(x <= -(7))\&\&(0 >= (p-y))))\&\&!(((0 == (p-1))\&\&(0 >= (p-y))\&\&((x-y) <= -(9))\&\&(2 >= (-(p)+y))\&\&((x+y) <= -(5))))\&\&!(((0 == (p-1))\&\&(x <= -(7))\&\&(0 >= (p-y))\&\&((x+y) <= -(2))))\&\&!(((0 == (p-1))\&\&(1 >= (-(x)-y))\&\&((-(p)+x) <= -(8))))$

Vita

Yuandong Cyrus Liu

Address 700 Grand Street, Hoboken, NJ 07030

Education

Stevens Institute of Technology, Hoboken, NJ
Ph.D. in Computer Science, Dec. 2022

Beijing University of Posts and Telecommunications, Beijing, China
M.S. in Information Security, March 2017

North China University of Science and Technology, Hebei, China
B.S. in Computer Science, June 2014

Publications

Yuandong Cyrus Liu, Chengbin Pang, Daniel Dietsch, Eric Koskinen, Ton-Chanh Le, Georgios Portokalidis, and Jun Xu.
Proving LTL Properties of Bitvector Programs and Decompiled Binaries, APLAS 2021

Y. Cyrus Liu, T. C. Le, E. Koskinen.
Source-Level Bitwise Branching for Temporal Verification,
arXiv e-prints, arXiv: 2111.02938

Talks

Seton Hall University seminar talk on DrNLA and Cybersecurity, Nov. 2022.
Research work on DrNLA, NJPLS, Oct. 2022.
Paper presentation on DarkSea, APLAS, Oct. 2021
Student forum, poster presentation on FMCAD, Oct. 2021.
Participant talk on LTL, OPLSS, July 2018.

Community Service

Paper Review, APLAS 2022, July 2022.
Artifact Evaluation Committee, CAV 2022, May 2022.
Artifact Evaluation Committee, TAP 2021, CGO 2021, Oct. 2020.
Student Volunteer, PLDI 2018, June 2018.