Proving LTL Properties of Bitvector Programs and Decompiled Binaries

Yuandong Cyrus Liu[1] Chengbin Pang[1], Daniel Dietsch[2], Eric Koskinen[1],
Ton-Chanh Le[1], Georgios Portokalidis[1], and Jun Xu[1]

[1] Stevens Institute of Technology
[2] University of Freiburg

Abstract. 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.
In this paper we show that similar linear arithmetic approximation of bitvector operations can be done at the source level through transformations. Specifically, we introduce new paths that 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. 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 linear temporal logic (LTL) verification of bitvector programs. Finally, we provide an in-depth case study of decompiled ("lifted") binary programs, which emulate X86 execution through frequent use of bitvector operations. We present a new tool DarkSea, the first tool capable of verifying reachability, termination and LTL of lifted binaries.

1 Introduction

There is increasing interest in using today's verification tools in domains where bitvector operations are commonplace. Toward this end, there has been a variety of efforts to enable bitvector reasoning in Satisfiability Modulo Theory (SMT) solvers, which serve as a foundation for program analysis tools. 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 [14]. CVC4 now employs a new approach called *int-blasting* [53], which reasons about bitvector formulas via integer nonlinear arithmetic.

Inspired by these SMT strategies, this paper explores the use of linear approximations of bitvector operations through source-level transformations, toward enabling Termination/LTL verification of bitvector programs. Our *bitwise branching* introduces new conditional, linear arithmetic paths that over-approximate many but not all bitvector behaviors. These paths cover the common cases and, in the remaining cases, other paths fall back on the exact bitvector behavior. As a result, in the common case, the reasoning burden is shifted to linear arithmetic conditions/constraints, a domain more suitable to today's automated termination/LTL techniques. We created source-translation rewriting rules for expressions as well as assignment statements and implemented them as a transformation on Boogie programs, within the Ultimate verifier [31].

We first examine the impact of bitwise branching on reachability and experimentally demonstrate that the translation imposes negligible overhead (from introducing additional paths), yet allows existing tools to verifying more bitvector programs. There are limited SV-COMP bitvector benchmarks (existing benchmarks require little or no real bitvector reasoning) so we first prepared 26 new bitvector reachability benchmarks, including examples drawn from Sean Anderson's "BitHacks" repository[1], which use bitvector operations for various purposes. Without bitwise branching, ULTIMATE's default setting (Z3 and SMT-Interpol) is only able to verify 2 of the 26 benchmarks. We show that bitwise branching allows us to verify these benchmarks with comparable performance with existing tools across a variety of back-end SMT solvers (MATHSAT, Z3, CVC4, SMTInterpol). We also show that bitwise branching is comparable in performance (both time and problems solved) with Z3.

The ability to use integer interpolation in the common case has far-reaching consequences, which we explore in the remainder of the paper. In Sec. 6 we show that, for bitwise termination benchmarks, bitwise branching improves ULTIMATE and is competitive with other tools that support termination of bitvector programs (*e.g.,* APROVE, KITTEL, CPACHECKER). Again SV-COMP does not have sufficient benchmarks for termination of bitvector programs, so we created new benchmarks by extending examples from the SV-COMP termination category [6], as well as the APROVE bitvector benchmarks [1].

More notably, our work leads to one of the first tools for verifying temporal logic (LTL) properties of bitvector programs. To our knowledge, the only existing tool is ULTIMATE, and we show that bitwise branching improves ULTIMATE's ability to verify LTL from merely 3 examples to a total of 59 new LTL benchmarks (out of a total of 67 benchmarks), adapted from ULTIMATE's LTL repository [7] and the BitHacks repository.

Case study: Temporal verification of lifted binaries. In Sec. 7 we explore how bitwise branching can be used as part of a novel strategy for verifying decompiled ("lifted") binaries. Lifted binaries have lost their source data-types and instead emulate the behavior of the architecture with extensive use of bitvector operations. We developed a new tool called DARKSEA, built on top of our ULTIMATE-based bitwise branching, as well as IDA PRO [48] and MCSEMA [25]. Although these decompilation tools generate IR/C programs and today's verification tools do parse C programs, we also describe some critical translations that were needed to make the output of MCSEMA suitable for verification (rather than re-compilation).

[1] https://graphics.stanford.edu/~seander/bithacks.html

We experimentally validated our work and show that DarkSea is the first tool for verifying temporal properties of lifted binaries. DarkSea is able to prove or disprove LTL properties of 8 lifted binaries. The most comparable alternative is Ultimate, which cannot prove any of them without DarkSea's translations, and can only verify 6 of them without bitwise branching.

Contributions. In summary, our contributions are:

- (Sec. 4) Bitwise branching, introducing paths with linear approximations.
- (Sec. 5) An evaluation showing that it allows one to prove reachability of more bitvector programs, with negligible overhead.
- (Sec. 6) An evaluation showing competitive performance on termination, and the first effective technique for LTL of bitvector programs.
- (Sec. 7) A case study and new tool called DarkSea, the first temporal verification technique for decompiled (lifted) binaries.
- New suites of bitvector benchmarks for reachability (23), termination (31), LTL (41) and lifted binaries (8).

We conclude with related work (Sec. 8). All code, proofs and benchmarks are available online[2]. Our benchmarks have also been submitted to SV-COMP.

2 Motivating Examples

Ex. 1. Reachability	**Ex. 2. Termination**	**Ex. 3. LTL** $\varphi = \Box(\Diamond(n < 0))$

`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 will refer to the above bitvector programs throughout the paper. To prove `error` unreachable in the **Ex. 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. 4, and compare performance of Ultimate using state-of-the-art SMT solvers, with and without bitwise branching.

We will see that the key benefits of bitwise branching arise when concerned with termination and LTL. **Ex. 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 (details in the Appendix of the extended

[2] `github.com/cyruliu/darksea`

version [40]). Critical to verifying termination of this program are (1) proving the invariant $\texttt{x} > 0 \wedge \texttt{a} > 0$ on Line 3 within the body of the loop and (2) synthesizing a rank function. To prove the invariant $\mathcal{I}$, tools must show that it holds after a step of the loop's transition relation $T = x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\texttt{\&}a'$, which requires reasoning about the bitwise-`&` operation because if we simply treat the `&` as an uninterpreted function, $\mathcal{I} \wedge T \wedge x'{>}0 \;\not\Longrightarrow\; \mathcal{I}'$.

```
a = *; assume(a > 0);
while (x > 0) {
  { x > 0 ∧ a > 0 }
  a--;
  if (x >= 0 && a >= 0)
  then { x = *; assume(x <= a); }
  else { x = x & a; }
}
```

The bitwise branching strategy we describe in this paper 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 **Ex. 2** as depicted in the gray boxes to the right. This changes the transition relation of the loop body from T (the original program) to T':

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

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

Bitwise branching also enables LTL verification of bitvector programs. We examine the behavior of programs such as **Ex. 3** above, with LTL property $\Box(\Diamond(n < 0))$. 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. 6 we show that with bitwise branching, our implementation can prove this property of this program in 8.04s.

Case study: Decompiled binary programs. In recent years many tools have been developed for decompiling (or "lifting") binaries into a source code format [15,45,9,25,51]. The resulting code, however, has long lost the original source abstractions and instead emulates the hardware. These programs are an interesting case study because their frequent use of bitvector operations places them beyond the capabilities of existing tools for LTL verification.

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

Consider the (source) program shown to the right. This program, which does not contain any bitvector operations, is taken from the Ultimate repository[3]. Some existing techniques and tools [20,7] can prove that the LTL property $\Box(\texttt{x} > 0 \Rightarrow \Diamond(\texttt{y} = 0))$ holds. However, after the program is compiled (with `gcc`) and then disassem-

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

bled and lifted (with IDPro and McSema), the resulting code has many bitvector operations. The resulting lifted code is quite non-trivial. (The full version is given in the extended version [40].)It required substantial engineering efforts just to parse and analyze the lifted code with existing verifiers (see Sec. 7). Let us 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   }
```

Roughly, 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 the program challenging for existing verifiers.

We describe a new tool DarkSea that uses bitwise branching in the context of a decompilation toolchain involving IDA Pro, McSema and Ultimate. The lifting performed by tools like McSema is geared toward *re*compilation rather than verification, thus foiling existing tools. In Sec. 7.2 we describe translations performed by DarkSea to tailor lifted binaries for verification. In Sec. 7.3, our experimental results show that DarkSea is the first tool capable of proving reachability, termination and LTL of lifted binaries.

3 Preliminaries

Our formalization is based on Boogie programs [12], denoted P. Our implementations parse input source C programs (or binaries decompiled to C) that may have bitvector operations. These programs are then translated into Boogie programs, in which bitvector operations are represented as uninterpreted functions. Figure 1 includes the standard syntax of a statement *Stmt* in a Boogie program P. For bitvector programs, we assume the following abbreviated expression *Expr* syntax, which includes bitvector operations:

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

```
       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;
```

Fig. 1: Boogie statement syntax in Ultimate framework.

We assume conditional branching has been transformed to non-deterministic branching: `if * then {assume(`b`);`s_1`} else {assume(!`b`);`s_2`}`. As discussed later, Ultimate (used in our implementation) has two modes: “bitvector mode,” in which these uninterpreted expressions are translated into SMT bitvector sorts and “integer mode,” in which they remain uninterpreted.

For the semantics, we assume a state space $\Sigma : Var \to Val$, mapping variables to values. We let $[\![e]\!] : \Sigma \to Val$ and $[\![s]\!] : \Sigma \to \mathcal{P}(\Sigma)$ be the semantics of expressions and statements, respectively, and $[\![P]\!]$ denotes traces of P.

4 Bitwise-branching

We build our *bitwise-branching* technique on the known strategy of transforming bitvector operations into integer approximations [14,53] 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:

1. Rewriting rules of the form $\mathcal{C} \vdash_E e_{bv} \rightsquigarrow e_{int}$ in Fig. 2a. 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. For conciseness, we omitted variants that arise from commutative re-ordering of the rules (in both Figs. 2a and 2b).

For example, consider the bitvector arithmetic expression `s&(1-s)` in **Ex. 1** of Sec. 2. 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 `s`$/e_1$, `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

$$
\begin{aligned}
e_1 = 0 \vdash_E e_1\texttt{\&}e_2 &\rightsquigarrow 0\\
(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1 \vdash_E e_1\texttt{\&}e_2 &\rightsquigarrow e_1\\
(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1) \vdash_E e_1\texttt{\&}e_2 &\rightsquigarrow e_1\texttt{\&\&}e_2\\
e_1 \geq 0 \wedge e_2 = 1 \vdash_E e_1\texttt{\&}e_2 &\rightsquigarrow e_1\texttt{\%}2\\
e_2 = 0 \vdash_E e_1\,\texttt{|}\,e_2 &\rightsquigarrow e_1\\
(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1 \vdash_E e_1\,\texttt{|}\,e_2 &\rightsquigarrow 1\\
e_2 = 0 \vdash_E e_1\texttt{\^{}}e_2 &\rightsquigarrow e_1\\
e_1 = e_2 = 0 \vee e_1 = e_2 = 1 \vdash_E e_1\texttt{\^{}}e_2 &\rightsquigarrow 0\\
(e_1 = 1 \wedge e_2 = 0) \vee (e_1 = 0 \wedge e_2 = 1) \vdash_E e_1\texttt{\^{}}e_2 &\rightsquigarrow 1\\
e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT * sizeof}(e_1) - 1 \vdash_E e_1\texttt{>>}e_2 &\rightsquigarrow 0\\
e_1 < 0 \wedge e_2 = \texttt{CHAR\_BIT * sizeof}(e_1) - 1 \vdash_E e_1\texttt{>>}e_2 &\rightsquigarrow -1
\end{aligned}
$$

(a) Rewriting rules for arithmetic expressions.

$$
\begin{aligned}
e_1 \geq 0 \wedge e_2 \geq 0 \vdash_S r\ \texttt{op}_{le}\ e_1\texttt{\&}e_2 &\rightsquigarrow r\texttt{<=}e_1\ \texttt{\&\&}\ r\texttt{<=}e_2\\
e_1 < 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{le}\ e_1\texttt{\&}e_2 &\rightsquigarrow r\texttt{<=}e_1\ \texttt{\&\&}\ r\texttt{<=}e_2\ \texttt{\&\&}\ r\texttt{<0}\\
e_1 \geq 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{eq}\ e_1\texttt{\&}e_2 &\rightsquigarrow \texttt{0<=}r\ \texttt{\&\&}\ r\texttt{<=}e_1\\
(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1) \vdash_S (e_1\,\texttt{|}\,e_2)\texttt{==0} &\rightsquigarrow e_1\texttt{==0}\ \ \texttt{\&\&}\ \ e_2\texttt{==0}\\
e_1 \geq 0 \wedge \texttt{is\_const}(e_2) \vdash_S r\ \texttt{op}_{ge}\ e_1\,\texttt{|}\,e_2 &\rightsquigarrow r\texttt{>=}e_2\\
e_1 \geq 0 \wedge e_2 \geq 0 \vdash_S r\ \texttt{op}_{ge}\ e_1\,\texttt{|}\,e_2 &\rightsquigarrow r\texttt{>=}e_1\ \ \texttt{\&\&}\ \ r\texttt{>=}e_2\\
e_1 < 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{eq}\ e_1\,\texttt{|}\,e_2 &\rightsquigarrow r\texttt{>=}e_1\ \ \texttt{\&\&}\ \ r\texttt{>=}e_2\ \ \texttt{\&\&}\ \ r\texttt{<0}\\
e_1 \geq 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{eq}\ e_1\,\texttt{|}\,e_2 &\rightsquigarrow e_2\texttt{<=}r\ \ \texttt{\&\&}\ \ r\texttt{<0}\\
e_1 \geq 0 \wedge e_2 \geq 0 \vdash_S r\ \texttt{op}_{ge}\ e_1\texttt{\^{}}e_2 &\rightsquigarrow r\texttt{>=0}\\
e_1 < 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{ge}\ e_1\texttt{\^{}}e_2 &\rightsquigarrow r\texttt{>=0}\\
e_1 \geq 0 \wedge e_2 < 0 \vdash_S r\ \texttt{op}_{le}\ e_1\texttt{\^{}}e_2 &\rightsquigarrow r\texttt{<0}\\
e_1 \geq 0 \vdash_S r\ \texttt{op}_{le}\ {\sim}e_1 &\rightsquigarrow r\texttt{<0}\\
e_1 < 0 \vdash_S r\ \texttt{op}_{ge}\ {\sim}e_1 &\rightsquigarrow r\texttt{>=0}
\end{aligned}
$$

(b) Weakening rules for relational expressions and assignments. $\texttt{op}_{le} \in \{\texttt{<,<=,==,:=}\}$, $\texttt{op}_{ge} \in \{\texttt{>,>=,==,:=}\}$, and $\texttt{op}_{eq} \in \{\texttt{==,:=}\}$

Fig. 2: Rewriting rules. Commutative closures omitted for brevity.

is always 0 and the `if` condition is feasible. In general, 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 in the subsequent sections of this paper, these case splits are nonetheless practically significant because often the final `else` is infeasible.

2. Weakening rules of the form $\mathcal{C} \vdash_S s_{bv} \rightsquigarrow s_{int}$ are in Fig. 2b. 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 $\texttt{assume(}s_{int}\texttt{)}$. 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 $\texttt{if}\ \mathcal{C}\delta\ \texttt{assume(}s_{int}\delta\texttt{)}\ \texttt{else}\ s_{bv}$. 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 $\texttt{if}\ \mathcal{C}\delta\ \texttt{then}\ \texttt{assume(}s_{int}\delta\texttt{)}\ \texttt{else}\ \texttt{assume(c)}$. The assignment operator in Figs. 2a and 2b, denoted :=, is included in three group of operators (op_{le}, op_{ge}, op_{eq}).

Proofs for each rule were done with Z3. Details are in the extended version [40].The rules in Fig. 2a and Fig. 2b were developed empirically, from the reachability/termination/LTL benchmarks in the next sections and, especially, based on patterns found in decompiled binaries (Sec. 7). We then generalized these rules to expand coverage.

Translation algorithm. We implemented bitwise branching via a translation algorithm, on top of Ultimate, denoted UltimateBwB. 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 (algorithms of T_E and T_S are given in the extended version [40]). 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.

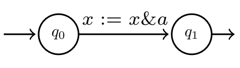
Lemma 1 (Rule correctness). *For every rule* $\mathcal{C} \vdash_E e \rightsquigarrow e'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow [\![e]\!]\sigma = [\![e']\!]\sigma$. *For every* $\mathcal{C} \vdash_S s \rightsquigarrow s'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow [\![s]\!]\sigma \subseteq [\![s']\!]\sigma$.

Theorem 1 (Soundness). *For every* P, T_E, T_S, $[\![P]\!] \subseteq [\![T_S\{T_E\{P\}\}]\!]$.

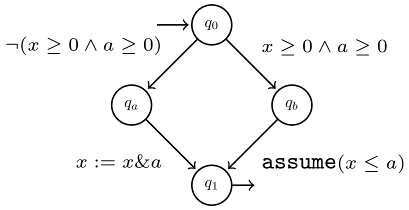
Proof. See Apx. A.

Control-flow automata. We have formalized *bitwise branching* via ASTs for readability but it can also be represented as a transformation on a program represented as a control-flow automaton. A (deterministic) *control flow automaton* (CFA) [35] is a tuple $\mathcal{A} = \langle Q, q_0, X, s, \longrightarrow\!\!\!\!\rightarrow \rangle$ 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 $\longrightarrow\!\!\!\!\rightarrow \subseteq Q \times s \times Q$ is a finite set of labeled edges.

Continuing with **Ex. 2**, an edge of the CFA labeled with statement `x = x &a` is shown to the right. Next shown is the result after applying the first weakening rule in Fig. 2b. Conditional edges are introduced (*e.g.,* $x \geq 0 \wedge q \geq 0$ to q_b) along with linear constraints (*e.g.,* `assume`$(x \leq a)$) and bitvector operations remain in the fallback case.



translated into:



5 Reachability of Bitvector Programs

We now evaluate the effectiveness of bitwise branching (BwB), as implemented in our UltimateBwB, toward reachability verification. Existing SV-COMP benchmarks require little or no bitvector reasoning; even when bitvector operations are present, they are often irrelevant to the property and can be abstracted away. We therefore created a new suite of 28 bitvector programs, including 12 simple programs (ReachBit) and 16 programs adapted from the existing code snippets "BitHacks" [10], which use bitwise operations for various tasks.

Ultimate can verify bitvector programs in two modes: *integer* and *bitvector*. In the integer mode, bitvector operations are overapproximated to nondeterminism and overflow/underflow is accounted for with `assume` statements. In the bitvector mode, Ultimate utilizes a variety of back-end SMT solvers with internal bitvector reasoning strategies, such as CVC4, Z3 and MathSAT (MS). Our implementation of bitwise branching, embodied in 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.

We ran our experiments with BenchExec [13] on a Linux 5.4.65 machine with an AMD Ryzen 3970X 32-core 3.7GHz CPU and 256GB RAM. We limited CPU time to 5 minutes, memory to 8GB, and restricted each run to two cores.

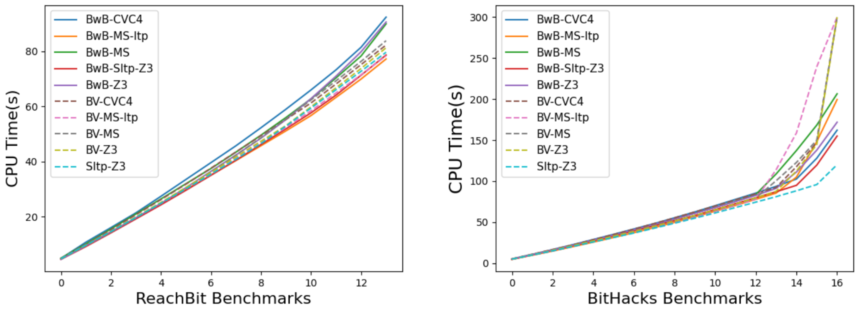


Fig. 3: Performance of UltimateBwB with bitwise branching "BwB" in *integer mode* (solid lines) versus Ultimate (dashed lines, "BV" indicating *bitvector mode*) on bitvector programs, using various SMT solvers.

Fig. 3 plots the number of ReachBit and BitHacks benchmarks solved versus the cumulative time between UltimateBwB with bitwise branching (solid lines) and Ultimate (dashed lines). These results show that the performance of UltimateBwB is comparable to Ultimate's bitvector mode, despite the fact that the bitwise branching transformation introduces 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 with the default setting (integer mode SItp-Z3 in Fig 3) returns *Unknown* for 10/12 "ReachBit" and 16/16 "BitHacks" benchmarks, despite the fact that it has a good trend in terms of runtime, while UltimateBwB can verify all 28 programs in the same settings. Moreover, while interpolation is less effective in the bitvector mode (see BV-MS-Itp vs. BV-MS), when combined with bitwise branching in the integer mode, it improves over those solvers and has the best results (BwB-SItp-Z3). The detailed result can be found in the extended version [40].

6 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 to the right, along with their limitations.

Tool	**BitVec.**	**Term.**	**LTL**
Ultimate	Limited	Yes	Yes
AProVE [29]	Yes	Yes	No
KITTeL [26]	Yes	Yes	No
CPAchecker [50]	Limited	Yes	No
2LS [18]	Yes	Yes	No
UltimateBwB	Yes	Yes	Yes

Termination. We compare bitwise branching with these termination provers in the table. We applied these tools to two benchmarks suites: **(i)** We first used 18 bitvector terminating programs selected from AProVE's bitvector benchmarks [34]. Notably, those benchmarks were designed with general bitvector arithmetic in mind so that there is only a small portion of bitvector programs 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 TermBitBench with bitvector operations including bitwise |, `&`, `^`, `<<`, `>>`, `~`.

	(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	-	-	-

Results. To the right is a table summarizing our results (details in [40]). 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 Ultimate in proving termination and non-termination of bitwise programs. This is because Ultimate's algorithms for synthesizing termination [32] and non-termination proofs [39] are not applicable to SMT formulas containing bitvectors, as discussed in Sec. 2. As a consequence, Ultimate relies on integer-based encodings of source programs together with overapproximations of bitwise operations. The 6 false results in AproveBench are spurious counterexamples that arise due to Ultimate's overapproximation of unsigned integers. Our results here confirm that bitwise branching provides an effective means for termination of bitvector programs.

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 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 called LTLBitBench of 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).

The table to the right summarizes the result of applying Ultimate and UltimateBwB on these two bitvector benchmarks (see [40] for details). UltimateBwB outperforms Ultimate: UltimateBwB can successfully verify 41 of 42 programs in LTLBitBench and 18 of 26 BitHacks programs while Ultimate can only 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. In conclusion, bitwise branching appears to be the first effective technique for verifying LTL properties of bitvector programs.

	(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	-	-

Bitwise-branching can be combined with other tools beyond Ultimate, making it an appealing general strategy. In this paper, we implemented bitwise branching within Ultimate [31] source code (during the C-to-Boogie 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 do not 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.

7 Case Study: LTL of Decompiled Binaries

Decompiled binary executables are rife with bitvector operations, making them an interesting domain for a case study. Many tools [25,8,48,27,28,36,24] 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 [47,23,11], decompilation into logic [43,44,45,51], and translation validation [22] (discussed in Sec. 8).

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.

While some of the above mentioned works involve manual or semi-automated proofs of safety properties, we have not yet seen many automated techniques for verifying reachability, termination and temporal properties of those lifted binaries. 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.).

7.1 Bitvector operations in lifted binaries

Lifted binaries frequently use bitvector operations *e.g.,* to reflect signed/unsigned comparison of variables whose type was lost in compilation. As we show in Sec. 7.3, lifted programs are beyond the capabilities of termination verification tools such as Ultimate, CPAchecker, AProVE or KITTeL.

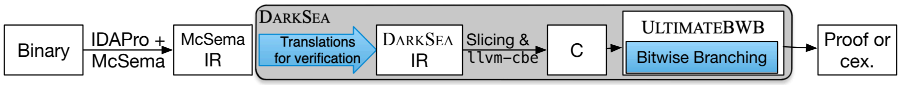
While the source code for the inner loop of `PotentialMinimizeSEVPABug.c` in **Ex. 3** is straight-forward (decrement `x`; assign 0 to `y` if `x <= 1`) the corresponding expressions in the lifted binaries involve multiple bitvector operations:

```
(((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 compilation has stripped its type. During decompilation, to approximate, 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. Lifted binaries are good candidates for bitwise branching; in this example 3 rules can be applied.

7.2 DarkSea: A toolchain for temporal verification of lifted binaries

Bitvector operations are not the only issue: lifted binaries have several other wrinkles that preclude them from being verified with today's tools. We briefly discuss these issues and how we address them in a new toolchain called DarkSea, the first tool capable of verifying reachability, termination and LTL properties of lifted binaries. DarkSea is comprised of several components:



DarkSea takes as input a lifted binary (obtained from IDA Pro and McSema) in LLVM IR format, which then can be converted to C via `llvm-cbe`.

Lifting tools like McSema [9,25] 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. 7.3) which either crash, timeout, memout, 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. The translations 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 [3,5,4].

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. A first pass of DarkSea analyzes lifted output to discover the original program's `main`, decouples the surrounding context-switch code, and removes it.
2. *Passing emulation state through procedures.* McSema generates lifted programs in which function arguments pass emulation state that is used for re-compilation. We found this to make it difficult for verifiers to track state. We thus eliminate these arguments from every function call, creating a single global pointer to the emulation state struct and replacing all uses of the first argument in the function body with a use of our new pointer.
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, *e.g.,* `state->general_registers.register13.union .uint64cell`. 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 [17] in termination-sensitive mode. For LTL properties, we use the atomic propositions' variables to seed our slicing criteria.

A longer discussion of these translations can be found in [40].

7.3 Experiments

We evaluated whether our translations (Sec. 7.2) and bitwise branching (Sec. 4) enabled tools to verify termination and LTL properties of decompiled binaries.

Termination of lifted binaries. As discussed in Sec. 6, 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. 7.2, 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 1 (details in [40]).

Table 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
✔	-	-	-	-	-	-	-	-	-	-	18	18
☢	-	18	-	-	3	-	-	-	-	-	-	-
M	-	-	-	-	-	3	-	-	-	-	-	-
T	-	-	18	-	15	15	-	18	18	-	-	-
?	18	-	-	18	-	-	18	-	-	18	-	-

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 internal representation.
- 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 1 also shows the verification results of those termination provers when applied to DarkSea's translated output (second set of columns).

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.

LTL of lifted binaries. We finally evaluate the effectiveness of DarkSea on LTL properties of 8 lifted binaries. In Table 2 we report the LTL property and expected verification result of each, as well as the verification time and result of Ultimate and DarkSea on them. Green cells use slightly different settings for single block encoding. DarkSea's translations eliminate unsoundness results that come from applying Ultimate directly to McSema IR.

Table 2: Ultimate vs. DarkSea on lifted programs with LTL properties.

			Ultimate		DarkSea	
Benchmark	Property	Exp.	Time	Result	Time	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	✗

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.

8 Related work

Bitvector reasoning. Many works support bitvector reasoning in SMT solvers (*e.g.,* [52]). Kroening *et al.* [38] perform predicate image over-approximation. Niemetz *et al.* [46] propose a translation from bitvector formulas with parametric bit-width to formulas in a logic supported by SMT solvers, making SMT-based procedures available for variant-size bitvector formulas.

He and Rakamarić [30] build on spurious counterexamples from overapproximations of bitvector operations. Mattsen *et al.* [41] use a BDD-based abstract domain for indirect jump reasoning. Bryant *et al.* [16] iterative construct an abstraction of a bit vector formula.

Other works have targeted reasoning about *termination* of bitvector programs. Cook *et al.* [21] use Presburger arithmetic for representing rank functions. Chen *et al.* [19] employ lexicographic rank function synthesis for bit precision and rely on the bit-precision of an underlying SMT solver. Falke *et al.* [26] propose an approach, implemented in KITTeL, which derives linear approximations of bitvector operations using some rules similar to our bitwise-branching rules for expressions. However, Falke *et al.* create a large disjunction of cases which puts a large burden on the solver. By contrast, our bitwise-branching creates multiple verification paths, but solver queries for most of them can be avoided through integer interpolation. As we show in Sec. 6, our UltimateBwB was able to solve 33/49 benchmarks, where as KITTeL solved only 10. Moreover, KITTeL does not support LTL properties and crashes on lifted binaries.

Tools for disassembly and decompilation. Jakstab [37] focuses on accurate control flow reconstruction in the *disassembly* process. BAP [15] performs static disassembly of stripped binaries. Angr [49] includes symbolic execution and value-set analysis used especially for control flow reconstruction. IDA Pro [48] (used in DarkSea) demonstrated high accuracy and uses value-set-analysis.

Hex-Rays Decompiler [2], Ghidra [8], and Snowman [24] further de-compile disassembled output to higher level representations such as LLVM IR or C code.

Verifying binaries. Some works focus on the low-level aspects of the binary and aim at precise de-compilation. Roessle *et al.* [47] de-compile x86-64 into a big step semantics. Earlier, others performed "decompilation-into-logic" (DiL) [43,44,45], translating assembly code into logic. While DiL provides a rich environment for precise reasoning about fine-grained instruction-level details, it incurs high complexity for reasoning about more coarse-grained properties such as reachability, termination, and temporal logic. In more recent work, Verbeek *et al.* [51] use the semantics of Roessle *et al.* [47] and describe techniques to decompile into re-compilable code.

Others focus on verifying the decompilation/lifting process itself. Dasgupta *et al.* [22] describe a translation validation on x86-64 instructions that employs their semantics for x86-64 (Dasgupta *et al.* [23]). Metere *et al.* [42] use HOL4 to verify a translation from ARMv8 to BAP. Hendrix *et al.* [33] discuss their ongoing work on verifying the translation performed by their lifting tool `reopt`. Numerous other works (*e.g.,* Sail [11]) provide formal semantics of ISAs.

9 Conclusion

We have shown that a source-level translation to approximate bitvector operations leads to tools that are competitive to the state-of-the-art in reachability and termination of bitvector programs. We show that bitwise branching incurs negligible overhead, yet enables more programs to be verified. Notably, we showed that this approach leads to the first effective technique for verifying LTL of bitvector programs and, to our knowledge, the first technique for verifying reachability, termination and LTL of lifted binary programs.

Acknowledgments. We thank the anonymous reviewers for their helpful feedback. This work is supported by ONR Grant #N00014-17-1-2787.

References

1. AProVE. `aprove.informatik.rwth-aachen.de/eval/Bitvectors/`
2. Hex-rays decompiler. `www.hex-rays.com/products/decompiler/`
3. McSema jump table bug. `github.com/lifting-bits/mcsema/issues/558`
4. McSema bug, missing data cross reference due to resetting ida's analysis flag. `github.com/lifting-bits/mcsema/issues/561`
5. McSema var. bug. `github.com/lifting-bits/mcsema/issues/566`
6. SV-COMP Termination Benchmarks. `github.com/sosy-lab/sv-benchmarks/tree/master/c/termination-crafted`
7. Ultimate's LTL benchmarks. `github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL/`
8. Agency, N.S.: Ghidra. `www.nsa.gov/resources/everyone/ghidra/`
9. Altinay, A., Nash, J., Kroes, T., Rajasekaran, P., Zhou, D., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Giuffrida, C., Bos, H., Franz, M.: Binrec: dynamic binary lifting and recompilation. In: EuroSys. pp. 36:1–36:16 (2020)

10. Anderson, S.: Bit twiddling hacks, `graphics.stanford.edu/~seander/bithacks.html`
11. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019)
12. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects. pp. 364–387 (2005)
13. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019)
14. Bozzano, M., Bruttomesso, R., Cimatti, A., Franzén, A., Hanna, Z., Khasidashvili, Z., Palti, A., Sebastiani, R.: Encoding RTL Constructs for MathSAT: a Preliminary Report. Electron. Notes Theor. Comput. Sci. **144**(2), 3–14 (2006)
15. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. pp. 463–469 (2011)
16. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding Bit-Vector Arithmetic with Abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems, vol. 4424, pp. 358–372 (2007)
17. Chalupa, M.: mchalupa/dg. `github.com/mchalupa/dg` (Jan 2021)
18. Chen, H., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs (T). In: ASE. pp. 53–64 (2015)
19. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-Precise Procedure-Modular Termination Analysis. ACM Transactions on Programming Languages and Systems **40**, 1–38 (Jan 2018)
20. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 399–410. POPL '11 (2011)
21. Cook, B., Kroening, D., Rummer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 236–250 (2010)
22. Dasgupta, S., Dinesh, S., Venkatesh, D., Adve, V.S., Fletcher, C.W.: Scalable validation of binary lifters. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 655–671 (Jun 2020)
23. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture p. 16 (2019)
24. Derevenets, Y.: Snowman. `derevenets.com/`
25. Dinaburg, A., Ruef, A.: Mcsema: Static translation of x86 instructions to llvm. In: ReCon 2014 Conference, Montreal, Canada (2014)
26. Falke, S., Kapur, D., Sinz, C.: Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In: Verified Software: Theories, Tools, Experiments, vol. 7152, pp. 261–277 (2012)
27. Galois, I.: Macaw, `github.com/GaloisInc/macaw`
28. Galois, I.: Reopt vcg, `github.com/GaloisInc/reopt-vcg`
29. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with aprove. J. Autom. Reason. **58**(1), 3–31 (2017)
30. He, S., Rakamarić, Z.: Counterexample-Guided Bit-Precision Selection. In: Programming Languages and Systems, vol. 10695, pp. 534–553 (2017)

31. Heizmann, M., Christ, J., Dietsch, D., Hoenicke, J., Lindenmann, M., Musa, B., Schilling, C., Wissert, S., Podelski, A.: Ultimate program analysis framework p. 1
32. Heizmann, M., Hoenicke, J., Podelski, A.: Termination Analysis by Learning Terminating Programs. In: Computer Aided Verification, vol. 8559, pp. 797–813 (2014)
33. Hendrix, J., Wei, G., Winwood, S.: Towards Verified Binary Raising p. 4
34. Hensel, J., Giesl, J., Frohn, F., Ströder, T.: Proving termination of programs with bitvector arithmetic by symbolic execution. In: SEFM. vol. 9763 (2016)
35. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. pp. 526–538 (2002)
36. Kinder, J.: Jakstab, http://www.jakstab.org/
37. Kinder, J., Veith, H.: Precise static analysis of untrusted driver binaries. In: Formal Methods in Computer Aided Design. pp. 43–50. IEEE (2010)
38. Kroening, D., Sharygina, N.: Approximating Predicate Images for Bit-Vector Logic. In: TACAS. Lecture Notes in Computer Science, vol. 3920, pp. 242–256 (2006)
39. Leike, J., Heizmann, M.: Geometric nontermination arguments. In: TACAS (2). Lecture Notes in Computer Science, vol. 10806, pp. 266–283 (2018)
40. Liu, Y.C., Pang, C., Dietsch, D., Koskinen, E., Le, T., Portokalidis, G., Xu, J.: Proving ltl properties of bitvector programs and decompiled binaries (extended). CoRR **abs/2105.05159** (2021), https://arxiv.org/abs/2105.05159
41. Mattsen, S., Wichmann, A., Schupp, S.: A non-convex abstract domain for the value analysis of binaries. In: SANER. pp. 271–280 (2015)
42. Metere, R., Lindner, A., Guanciale, R.: Sound Transpilation from Binary to Machine-Independent Code. arXiv:1807.10664 [cs] **10623**, 197–214 (2017)
43. Myreen, M.O., Gordon, M.J.C.: Hoare logic for realistically modelled machine code. In: TACAS. Lecture Notes in Computer Science, vol. 4424, pp. 568–582 (2007)
44. Myreen, M.O., Gordon, M.J.C., Slind, K.: Machine-code verification for multiple architectures - an application of decompilation into logic. In: Formal Methods in Computer-Aided Design, FMCAD 2008. pp. 1–8 (2008)
45. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - improved. In: Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012. pp. 78–81 (2012)
46. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C., Tinelli, C.: Towards bit-width-independent proofs in smt solvers (2019)
47. Roessle, I., Verbeek, F., Ravindran, B.: Formally verified big step semantics out of x86-64 binaries. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (2019)
48. SA, H.R.: Ida pro, `www.hex-rays.com/products/ida/`
49. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on S&P (2016)
50. SoSy-Lab: CPAchecker, `cpachecker.sosy-lab.org/`
51. Verbeek, F., Olivier, P., Ravindran, B.: Sound C Code Decompilation for a Subset of x86-64 Binaries. In: Software Engineering and Formal Methods, vol. 12310, pp. 247–264 (2020), series Title: Lecture Notes in Computer Science
52. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bitvector formulas. Formal Methods in System Design **42**, 3–23 (Feb 2013)
53. Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Notzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-Precise Reasoning via Int-Blasting (2021)

A Proof of Thm. 1

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 traces 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 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 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 1.