Lean Certified Bitvector Solving without Bitblasting

Jan Onderka
University of Freiburg
Freiburg, Germany
onderka@cs.uni-freiburg.de

Armin Biere
University of Freiburg
Freiburg, Germany
biere@cs.uni-freiburg.de

Mathias Fleury
University of Freiburg
Freiburg, Germany
fleury@cs.uni-freiburg.de

***Abstract*—Satisfiability Modulo Theories (SMT) solvers are the main working horse in many applications of automated reasoning. Unfortunately, they are frequently exposed to produce incorrect results. Trust in their results can be regained by producing proof certificates that are proof-checked within a trusted codebase. However, certificate size as well as proof-checking time of existing SMT proof checking approaches can be excessive. We introduce a new scheme of twinning an untrusted solver and a trusted proof checker using the same solving procedure, in which proof certificates produced by the solver act as an oracle for the proof checker. We have implemented our approach in two new tools, Roole (the untrusted solver, written in Rust) and Roolean (the trusted proof checker, written in Lean), using simplified Three-Valued Abstraction Refinement (TVAR) instead of bitblasting. Our approach yields small certificates and produces trusted results in reasonable time for the majority of quantifier-free bitvector benchmarks in SMT-LIB that are claimed to be unsatisfiable.**

***Index Terms*—SMT solving, interactive theorem provers, certifying oracles, three-valued bitvector domain**

I. Introduction

Boolean satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are ubiquitously used in critical workflows such as software or hardware verification. If a model satisfying the given formula is found, it can be checked easily. If the solver determines that no such model exists, i.e. the formula is *unsatisfiable*, checking it is more problematic. In applications such as model-checking safety properties, unsatisfiability means the system is bug-free, so high confidence is crucial. Despite this, many bugs have been found in traditional SMT solvers, especially using fuzzing [1]–[10].

Proof checkers directly implemented in Interactive Theorem Provers (ITPs) allow reducing trust to the ITP itself. However, as ITP programs are orders-of-magnitude slower than standard code, the proof produced by the solver must be detailed enough to speed up checking but not impractically large.

For SAT solving, proof checking [11], [12] is in widespread use, and mandatory in the annual SAT competition since 2016. Common proof formats include DRAT [13] and LRAT [14]. For SMT solving, the situation is more open, and also depends on the used theory. In this paper, we focus on QF_BV, the theory of quantifier-free bitvectors in SMT-LIB [15]. Implementing a performant proof checker for QF_BV is complex due to the large number of operations and corner cases.

The two major "theory-universal" approaches for proof-checking based on rule rewriting [16], [17] do not produce trusted results for QF_BV yet despite work in progress [18]. Another approach is bitblasting QF_BV to a SAT problem within the ITP and proof-checking the proof certificate produced by the untrusted SAT solver [19], [20]. However, bitblasting incurs ITP slowdown and the produced certificate can be impractically large, especially for larger bitvector widths.

A. Our contribution

We propose to use a pair of an untrusted SMT solver and a trusted proof checker as twins using the same evaluation procedure. The solver produces a proof that the proof checker uses as an oracle when making decisions, compensating for the relatively slow trusted code by entirely avoiding bad decisions.

As a proof of concept, we implemented a bitvector solver **Roole**, written in the programming language **Rust**, and a proof checker **Roolean**, written in the ITP language **Lean**. To avoid bitblasting, **Roole** and **Roolean** rely on the three-valued abstraction refinement (TVAR) technique [21] adapted from model-checking to SMT. We further formally proved the soundness of proof-checking within **Lean**.

Even though our approach only uses three-valued bitvector abstraction [22] without any rewriting rules or similar simplifications and does not use any heavy-duty tools such as SAT solvers, it produces trusted results for the majority of the unsatisfiable QF_BV benchmarks from SMT-LIB 2024 benchmarks [23] under reasonable time and memory constraints.

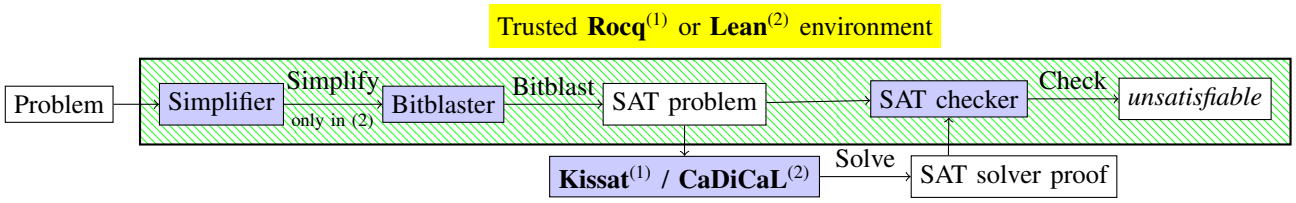
Our proofs tend to be orders-of-magnitude smaller than in a competing bitblasting approach, and proof checking usually takes less time than solving. This demonstrates the viability of our proposal even in this preliminary form.

II. Preliminaries

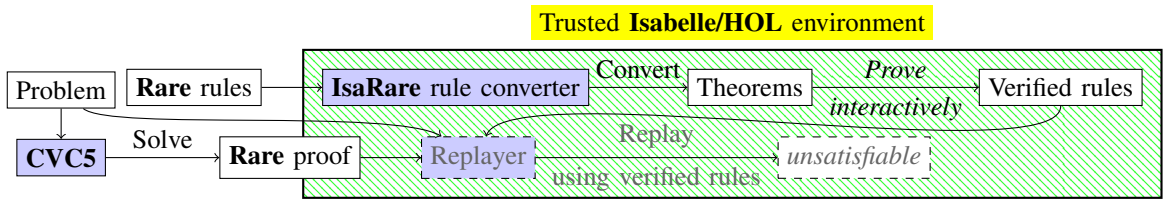
State-of-the-art [25] QF_BV SMT solvers **Bitwuzla** [26], **Yices2** [27], and **cvc5** [28] are written in the programming language **C** or **C++**. Bugs can be inadvertently introduced while improving the solvers, so their results are untrusted unless proof-checked using a trusted proof checker.

Solvers or proof-checkers written using Interactive Theorem Provers (ITPs) can be formally proven to produce correct results, allowing a greater degree of trust in the result. The major ITPs used for writing proof checkers are **Rocq** (formerly **Coq**), **Isabelle/HOL**, and **Lean**. We will discuss the most relevant trusted solving / proof-checking tools for QF_BV and explain the dataflow between their untrusted and trusted parts.

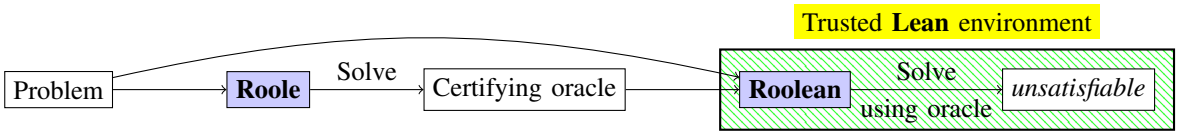
The SMT solver **CoqQFBV** [19] is written in **Coq**. As visualized in Figure 1a, it reduces the problem to SAT by



(a) **CoqQFBV**(1) [19] / **bv_decide**(2) [20]: Trusted SMT solving using bitblasting.



(b) **Rare** [24] with **IsaRare** [18]: A alternative planned scheme for trusted SMT solving using rule rewriting. The "Replayer" is still under development for QF_BV at the time of writing, so trusted results cannot be obtained yet.



(c) **Roole** & **Roolean**: Untrusted solving followed by trusted proof-checking with the proof certificate acting as an oracle.

Fig. 1: Dataflow of styles of trusted SMT solving for problems solved as *unsatisfiable*.

trusted bitblasting, calls the untrusted SAT solver **Kissat** [29], and checks the produced certificate in the trusted environment.

The automated proving tactic **bv_decide** [20] is built into **Lean**. It translates **Lean** theorems into bitvector problems, simplifies them, performs bitblasting within **Lean**, uses the untrusted SAT solver **CaDiCaL** [30] to produce an **LRAT** proof [14], and checks the proof from the trusted **Lean** codebase, as shown in Figure 1a. For checking SMT-LIB problems, the problem is translated to a **Lean** theorem first.

The domain-specific language **Rare** [24] defines rules for the proof format **Alethe**. The tool **IsaRare** [18] can be used to check that these rules themselves are correct by translating the rules into **Isabelle/HOL** lemmas, which then can be formally proven by the user. Replaying proofs (checking the **Alethe** proof using the verified **Rare** rules) as shown in Figure 1b is ongoing work [18, p. 327], not available yet.

III. Solver and Proof-Checker as Twins

Proof checkers written in traditional programming languages such as **C** or **Rust** must be much simpler than the solver to avoid bugs. This does not apply to ITP-based proof checkers: they can be arbitrarily complicated as long as their soundness is proven. We make use of the trade-off that more complicated proof checkers admit smaller proof certificates.

In the extreme case, a proof checker taking zero-sized certificates is a solver. Ideally, we would implement a fast trusted solver in an ITP, but this tends to be too slow in practice. Instead, we propose building the solver (implemented in a fast programming language) and the proof checker (implemented in an ITP) as *twins that use the same solving procedure*. The solver twin produces a proof certificate that serves as a decision oracle to the proof checker (*certifying oracle*).

As a proof of concept, we implemented the solver **Roole** in **Rust** and the proof checker **Roolean** in **Lean**. As shown in Figure 1c, they only interact through the certifying oracle. When **Roolean** is making a decision, it takes the choice **Roole** ultimately considered the best.

Certifying oracles are size-efficient compared to writing down rule applications, since every action except for choices is implicit. The drawback is code duplication but there is also a benefit: since proof-checking is guaranteed-correct and very similar to solving, any proof-checking error uncovers an inconsistency (a bug). Furthermore, in our case, the duplication was well-manageable, in part due to the similarity of **Rust** and **Lean** and in part due to our solving and evaluation strategy.

IV. Solving & Proof Checking

The implementation of **Roole** and **Roolean** is based on our previous research on fast arithmetic operations for three-valued bitvectors [22] and on Input-based Three-Valued Abstraction Refinement (IB-TVAR) [21] from model checking, similar to other approaches that use ternary abstraction [31]–[37].

As SMT solving can be interpreted as model checking limited to a single step, checking the property **EX**$[a]$ (there exists a path where a holds in the next state), the requirements for soundness are greatly simplified. We will give a brief overview on how we instantiate the simplified IB-TVAR for solving and checking SMT quantifier-free bitvector formulas.

A. Bitvector Domains

Consider the Boolean domain $\mathbb{B} = \{0, 1\}$. A bitvector of (fixed) width n is a value in the domain $\mathbb{B}^n$. In **Roole** and **Roolean**, we use three-valued bitvector (TVBV) abstraction. A three-valued bit is an element of the set

$$\mathbb{X} = \{\{0\}, \{1\}, \{0, 1\}\} \tag{1}$$

We name the three elements '0' = $\{0\}$, '1' = $\{1\}$, 'X' = $\{0, 1\}$. Intuitively, 'X' stands for either 0 or 1 (it is *unknown* which).

An abstract n-bit bitvector $\hat{b}$ is a Cartesian product of three-valued bits $\mathbb{X}^n$. The *concretization function* $\gamma : \mathbb{X}^n \to 2^{\mathbb{B}^n}$ relates concrete and abstract bitvectors where

$$\gamma(\hat{b}) = \{b \mid \forall i < n \,.\, b_i \in \hat{b}_i\}. \tag{2}$$

We identify bitvectors with their binary number representation and for ternary values in TVBV similarly, but within double-quotes. In our implementation, we represent TVBV in a *zeros-ones encoding* [22, p. 4] (a form of dual rail encoding), using two bitvectors acting as flags for the possibilites of *zero* and *one*, respectively, at a given bit. In each bit position, at least one of the flag bits must be set to 1.

Example 1: $\gamma(\text{“0X1X”}) = \{0010, 0011, 0110, 0111\}$. Internally, “0X1X” is represented as two bitvectors (1101,0111). The first (*zeros*) has each bit set for abstract bit values '0' and 'X', the second (*ones*) for abstract bit values '1' and 'X'.

B. Bitvector Operations

SMT-LIB defines many bitvector operations $f : \mathbb{B}^m \to \mathbb{B}^n$. For evaluation within the abstraction, we replace f with the abstract operation $\hat{f} : \mathbb{X}^m \to \mathbb{X}^n$. To maintain soundness of abstract evaluation, it suffices to ensure:

$$\forall \hat{v} \in \mathbb{X}^m \,.\, \forall v \in \gamma(\hat{v}) \,.\, f(v) \in \gamma(\hat{f}(\hat{v})). \tag{3}$$

Below, we discuss how TVBV operations were implemented to both satisfy this condition and obtain fast evaluation performance. Concrete examples are listed in Table II.

*1) Bitwise Operations (**bvnot**, **bvand**, **bvor**, **bvxor**):* Trivial implementation using three-valued logic.

*2) Arithmetic Operations (**bvneg**, **bvadd**, **bvsub**, **bvmul**, **bvudiv**, **bvurem**, **bvsdiv**, **bvsrem**, **bvsmod**):* We implement bitvector addition, subtraction, and multiplication by our “modular-extreme-finding” algorithm [22, p. 247-250] and negation as subtraction from zero. We implement **bvudiv** by considering the bitvectors as unsigned intervals and converting the result back to three-valued abstraction, and **bvurem** similarly as long as the division result contains no 'X', otherwise we return an all-'X' result. For **bvsdiv** and **bvsrem**, we separately join results for existing sign-bit quadrants. For the more complicated and rarely used **bvsmod**, we return an all-'X' result if there is at least one 'X' in the operands.

*3) Equality and Comparison (**=**, **bvule**, **bvult**, **bvslt**, **bvsle**):* The result of equality can be zero exactly if at least one bit can differ, and can be one exactly if all bits can be the same. For comparisons, we convert to unsigned/signed intervals and consider their positions on the number line.

TABLE II: Examples of evaluating bitvector operations in the three-valued bitvector (TVBV) abstraction domain.

Operation	Operand 1	Operand 2	Result
bvnot	“0X1X”	-	“1X0X”
bvand	“0X1X”	“1011”	“001X”
bvor	“0X1X”	“1011”	“1X11”
bvxor	“0X1X”	“1011”	“1X0X”
bvadd	“00X0”	“0011”	“0XX1”
bvsub	“00X0”	“0011”	“11X1”
bvmul	“001X”	“0010”	“01X0”
bvudiv	“110X”	“1010”	“0001”
bvurem	“110X”	“1010”	“001X”
bvsdiv	“110X”	“1010”	“000X”’
bvsrem	“110X”	“1010”	“XXXX”
bvsmod	“110X”	“1010”	“XXXX”
=	“0000”	“00X0”	“X”
bvult	“00X0”	“010X”	“1”
zero_extend	2	“X1”	“00X1”
sign_extend	2	“X1”	“XXX1”
bvshl	“0010”	“000X”	“0XX0”
bvlshr	“0010”	“000X”	“00XX”
bvashr	“1010”	“X000”	“1X1X”

*4) Bit Extension (**zero_extend**, **sign_extend**):* For zero-extension, we perform bit extension of the zeros-ones flags. and then set the newly added bits in the zero flag. For sign-extension, it suffices to sign-extend both flags.

*5) Bit Shift (**bvshl**, **bvlshr**, **bvashr**):* As bit-shifting by a larger number than the number of bits results in zero result, we handle the overflow case separately and then join the results of shifts for all other shift amounts.

*6) Other Operations (**concat**, **extract**, ...):* We do not implement other operations directly on the abstract domain, but compose them from the aforementioned ones. For **extract**, this is possible because we internally implement bit-extension operations as arbitrary resizing.

C. Evaluation Functions

A QF_BV problem is given as a set of assertions[1]. Their conjunction produces a tree of operations on bitvectors[2]. Given an assignment of free variables $v \in V$, the tree evaluates to a single-bit bitvector that determines whether the assignment is a model of the formula given by the assertion set.

The evaluation function e is similar to operations except that its input is an assignment $v \in V$. Naming the abstract evaluation function $\hat{e}$, its soundness requirement is as follows:

$$\forall \hat{v} \in V \,.\, \forall v \in \gamma(\hat{v}) \,.\, e(v) \in \gamma(\hat{e}(\hat{v})). \tag{4}$$

Internally, e and $\hat{e}$ evaluate in postfix order, and each operation can use results of its predecessors. Abstract evaluation soundness follows inductively from operation soundness.

D. High-level Soundness

The formula can be solved by evaluating (by brute force)

$$\exists v \in V \,.\, e(v) = 1. \tag{5}$$

[1]We do not discuss features such as function definitions for conciseness.
[2]We identify Booleans with single-bit bitvectors for simplicity.

For sound abstract interpretation, we evaluate $\hat{e}$ on a set of abstract values $\hat{V} = (\hat{v}_0, \hat{v}_1, \dots)$, requiring no spurious abstract values ($\forall \hat{v} \in \hat{V} \,.\, \gamma(\hat{v}) \neq \varnothing$) and that all concrete values are considered ($\bigcup_{\hat{v} \in \hat{V}} \gamma(\hat{v}) = V$). Proof-checking the latter with general $\hat{V}$ is NP-complete for TVBV[3]. Thus a proof is a *splitting tree* starting with an all-'X' assignment and non-leaf nodes split a single bit to '0' and '1'.

The only major difference between **Roole** and **Roolean** is how the splitting tree is obtained. In **Roole**, we use a search algorithm we will describe in the next subsection. In **Roolean**, we obtain the splitting tree from the certificate, evaluate the leaves and combine the three-valued results.

Our main **Roolean** theorem states that if the proof-checking procedure returns a non-'X' result with value $r \in \mathbb{B}$, then

$$r = 1 \iff \exists v \in V \,.\, e(v) = 1. \tag{6}$$

We proved the theorem by induction on proof-checking the splitting tree, with evaluation soundness used in nodes.

As **Lean** already provides a bitvector type implementing bitvector logic with operations typically following SMT-LIB semantics, it is only additionally needed to trust that our SMT-LIB parser and evaluator produce the correct e. This is a necessary limitation as SMT-LIB and QF_BV are largely only informally defined. However, the surface area for deviations in concrete (brute-force) evaluation is compact and not impacted by addition of abstract domains or improvements to them.

E. Solving Approach

In **Roole**, we build the splitting tree while solving. If we find an abstract assignment of variables that satisfies the formula, we discard it and construct a new splitting tree where only the leaf corresponding to the assignment is marked as relevant. Otherwise, we conclude that the problem is unsatisfiable as the formula cannot be satisfied through any leaf.

Our a search algorithm is based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure with additional basic machinery for backjumping and a simple form of learning of TVBV constraints, inspired by Conflict-Driven Clause Learning (CDCL)[4]. A simplified overview is shown in Alg. 3.

We do not yet use any simplification techniques (rewriting, preprocessing) considered crucial for performance of conventional SAT/SMT solvers. We plan to enable such simplification by adding more powerful domains in the future.

Example 2: Consider whether there exists an x such that

$$x \neq 000_2 \wedge ((x \wedge 100_2) = 000_2) \wedge x + x = 000_2. \tag{7}$$

After encoding this formula into SMT-LIB, **Roole** solves the problem as follows. Starting with decision order $(0, 1, 2)$, the result of evaluating "XXX" is 'X'. **Roole** thus splits at bit 0 (the least significant bit) to "XX0" and "XX1". Then, it evaluates "XX0" and continues splitting until it reaches "000",

[3]Consider each bit as a Boolean variable. A TVBV becomes a disjunction of literals where the value is not 'X'. We can receive any disjunctive-normal-form formula and must prove it is a tautology, a dual of the SAT problem.

[4]We also experimented using **CaDiCaL** as an underlying CDCL(T) solver, but in our preliminary testing, this approach did not improve performance.

```
SEARCH (assignment â)
1  if exists b̂ ∈ Λ with γ(â) ⊆ γ(b̂) return unsatisfiable
2  let v̂ ← ê(â)
3  if v̂ ='1' return satisfiable
4  if v̂ ='0'
5      reduce â to "relevant" values
6      add reduced â to Λ
7      return unsatisfiable
8  pick input bit i with â_i ='X'
9  for d ∈ {0, 1}
10     let â^d ← â updated with â_i^d ↦ d
11     if SEARCH(â^d) = satisfiable return satisfiable
12     if i is not "relevant" return unsatisfiable
13 add â to Λ
14 return unsatisfiable
```

Algorithm 3: Simplified algorithm to search for a satisfying assignment of an abstract evaluation function $\hat{e}$ using a global assignment cache Λ of falsifying assignments. The actual **Rust** implementation of SEARCH in **Roole** is non-recursive. Furthermore, in order to not descend into nodes removed by "relevancy analysis" at line 12, **Roole** reorders the assignment tree on-the-fly (implementing *backjumping*).

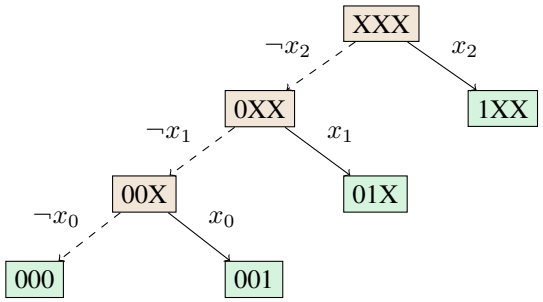


Fig. 4: The final splitting tree produced by **Roole** for the considered Example 2. The certificate oracle provided to **Roolean** is composed of the decisions present in the tree.

which evaluates to '0' and is closed and learned. Next, "100" evaluates to '0'. From here, *backjumping* is possible, as "1X0" and "1XX" also evaluate to '0'.

As a consequence, **Roole** backtracks to the root of the splitting tree, swaps 0 and 2 in the current decision order to $(2, 1, 0)$, and immediately closes "1XX". Descending through "0XX", it then closes "000" (already learned), "001", and "01X", concluding that the formula is *unsatisfiable*.

Roolean proof-checks unsatisfiability by evaluating the formula in the leaves of the splitting tree shown in Figure 4, essentially avoiding the bad initial decision order $(0, 1, 2)$.

Note that while the decision order in Figure 4 is fixed, i.e. the tree forms an Ordered Binary Decision Diagram (OBDD), this is not the case in general as the decision order is allowed to vary between tree branches after backtracking.

V. Evaluation

We evaluated **Roole** and **Roolean** in the spirit of the previous evaluation [20, p. 20-21] of **bv_decide** against **Bitwuzla** and **CoqQFBV** on 2024 QF_BV benchmarks [23]. As **CoqQFBV** previously performed similarly to **bv_decide**, we did not evaluate it. We also did not evaluate **IsaRare** as it currently cannot produce a trusted result. As **Roole** and **Roolean** only interact through certificates, we ran them separately.

Our experiments used the compute cluster bwForCluster Helix with AMD EPYC 7513 CPUs. The entire benchmark set was run with an individual CPU time limit of 1200 seconds and 8 GB of memory per task. In our discussion, we further focus on *unsatisfiable* benchmarks as trusting unsatisfiability claims is nontrivial. For fairness, we only consider trusted results reached by **Roole** and **Roolean** if the sum of solving and proof-checking time did not exceed 1200 seconds.

We visualised solving and checking time (for **Roolean** including proof-checking certificates produced by **Roole**) in the manner of the previous evaluation [20, p. 20-21] in Figure 6.

The data on **Bitwuzla** and **bv_decide** are roughly consistent with the previous evaluation except for **Bitwuzla** producing first results immediately instead of a ramp-up. This may have been caused by differences in the experimental setup.

Our proof-of-concept tools **Roole** and **Roolean** do not use any preprocessing nor sophisticated CDCL techniques. Therefore, we consider our trusted-result performance quite encouraging in comparison to **bv_decide**. Figures 6 and 7a show that our trusted proof-checking generally keeps up with untrusted solving, confirming the viability of our scheme.

Regarding memory usage, shown in Figure 8, **Lean**-based tools have issues, notable also when comparing solving and proof-checking in Figure 7b. The most memory-intense benchmarks for **Roolean** are easily solvable using a small splitting tree but contain a large number of operations.

We further show proof sizes in Figure 9. The total size of all 17 486 *unsatisfiability* proofs generated by **Roole** is 1.64 GB. All of them are claimed *unsatisfiable*, which is 63% of all 27 758 claimed-*unsatisfiable* benchmarks. 16 169 proofs are smaller than 1 kB. 17 433 *unsatisfiability* proofs (62.8% of all 27 758 claimed-*unsatisfiable* benchmarks) were successfully proof-checked by **Roolean** within the combined time of 1200 seconds. This is despite the fact we did not employ any proof-trimming to reduce memory usage yet.

We show the number of successfully solved benchmarks claimed *unsatisfiable* grouped by the benchmark family in Table V. The QF_BV benchmark set is highly skewed, with 67% of benchmarks claimed *unsatisfiable* belonging to a single family **sage**, and 17% belonging to the related family **Sage2** [38], [39]. While **Bitwuzla** and **bv_decide** perform well across families, our tools perform disproportionately well on the **sage** family and extremely poorly on **Sage2** and **bruttomesso**. We think this is due to the used simple abstract domain and absence of preprocessing, problematic when equalities or disequalities between variables are encountered.

TABLE V: Solved benchmarks claimed *unsatisfiable*, grouped by benchmark family.

Family	Claimed	Bitwuzla	bv_decide	Roole+Roolean
sage	18530	18530	18256	16618
Sage2	4672	4670	4644	2
Noetzli	1432	1432	1399	297
bruttomesso	976	976	967	0
(other)	2148	2017	1649	541

VI. Conclusion & Future Work

We presented a scheme of "twinning" an untrusted solver and a trusted proof checker and shown its viability by implementing the twin tools **Roole** and **Roolean**, competitive with the trusted Lean tactic **bv_decide** on SMT-LIB problems with up to orders-of-magnitude smaller proof certificates.

Our abstract-evaluation approach and the ability to obtain trusted results without the limitations of a standardised proof format present many interesting avenues of future work:

- Lazy evaluation in **Roole** and extension of certificates to identify unnecessary operations to evaluate in **Roolean**, balancing certificate sizes and proof-checking time.
- Additional abstract domains for better solver strength. We have started implementing a symbolic domain to quickly resolve standard bitvector tricks [41] in **Roole**, but it requires further work and soundness proofs in **Roolean**.
- Using Directed Acyclic Graphs (DAGs) instead of trees for certificates would allow faster proof-checking by evaluating shared learned clauses instead of leaf nodes.
- Using the combination of **Roole** and **Roolean** in a **Lean** proving tactic, similarly to **bv_decide**.
- Extension to more logics including quantified bitvectors. Recently, it has been shown that using abstract over- and underapproximating Binary Decision Diagrams [42] over standard SMT solving is performant for quantified bitvectors. Our solving strategy would have the advantage of being able to recover from bad decision ordering.
- Extension to optimisation problems. Unlike the traditional bitblasting approach, there is no fundamental limitation to decision problems in our strategy.
- Extension to model checking. While our previous tool **machine-check** [43] performs model checking for the full μ-calculus [21], the intricate algorithms make improvements prone to bugs. Using our scheme, we could achieve fast trusted model checking for complicated systems.

Data availability

We provide evaluation results, measurement scripts, a setup for local evaluation and visualisation, and a short discussion of the soundness theorem in an artefact on Zenodo [44]. **Roole** and **Roolean** are available on GitHub [45], [46].

Acknowledgement

The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG. Furthermore, the work was supported by a gift from Amazon.

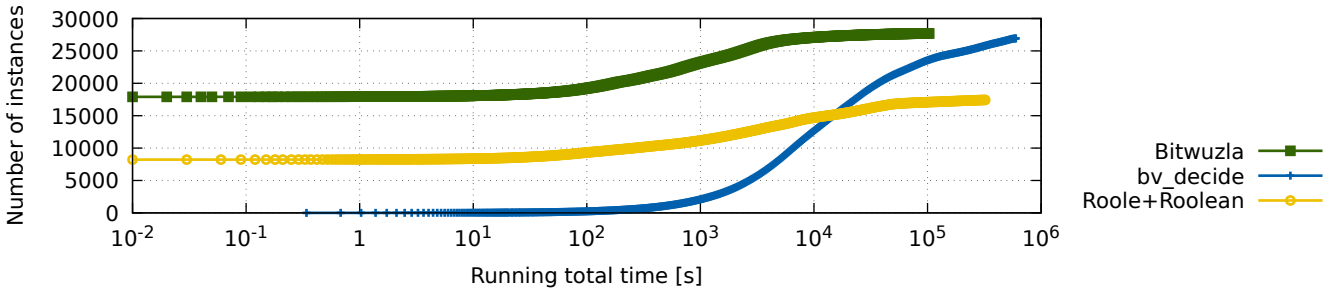
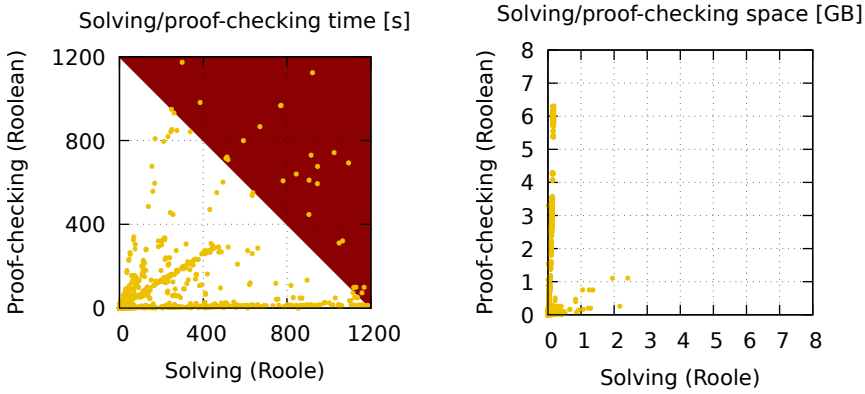


Fig. 6: Logarithmic **cumulative** total running CPU time of solved *unsatisfiable* problems in SMT-LIB 2024 QF_BV benchmarks [23] with an individual CPU time limit of 1200 seconds and a memory limit of 8 GB per task. The plot follows the presentation of the main results in the closest related paper [20, p. 18, fig. 7] and shows similar results as in [20] for **Bitwuzla** and **bv_decide** except for a stronger start by **Bitwuzla** here, presumably due to a slightly different experimental setup. While **Bitwuzla** is fast and powerful, it only produces untrusted results. Producing trusted results by **bv_decide** is slow, but almost reaches the number produced by **Bitwuzla** at the end. Solving by **Roole** and then proof-checking by **Roolean** produces a trusted result for 17 433 benchmarks claimed *unsatisfiable* (62.8% of total 27 758) within the limits despite the simple implementation. Unlike **bv_decide**, our approach produces a trusted result for more than 7500 instances nearly instantly.



(a) CPU-time comparison of solving by **Roole** against checking the proof by **Roolean**. Proof-checking usually takes less time than solving. Results tend to form clusters. Notably, many tasks take a long time to solve but are proof-checked nearly instantly.

(b) Memory usage comparison. **Roole** is usually much more memory-efficient than **Roolean**. This could be alleviated e.g. by adding information for lazy evaluation to certificates. Unfortunately, debugging **Lean** memory usage is hard [40].

Fig. 7: Comparisons between **Roole** and **Roolean** on problems that were both solved and proof-checked as *unsatisfiable*. Results where the sum of solving and proof-checking time exceeded 1200 seconds (corresponding to the red area in Figure 7a) are not counted as solved by **Roole+Roolean**.

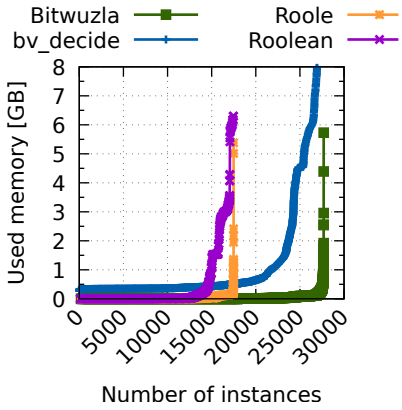


Fig. 8: Memory used for solved/proof-checked *unsatisfiable* problems, sorted ascending. **Roole** and **Bitwuzla** use small amounts of memory except for a few problems (sharp knee). **Lean**-based **Roolean** and **bv_decide** are less frugal.

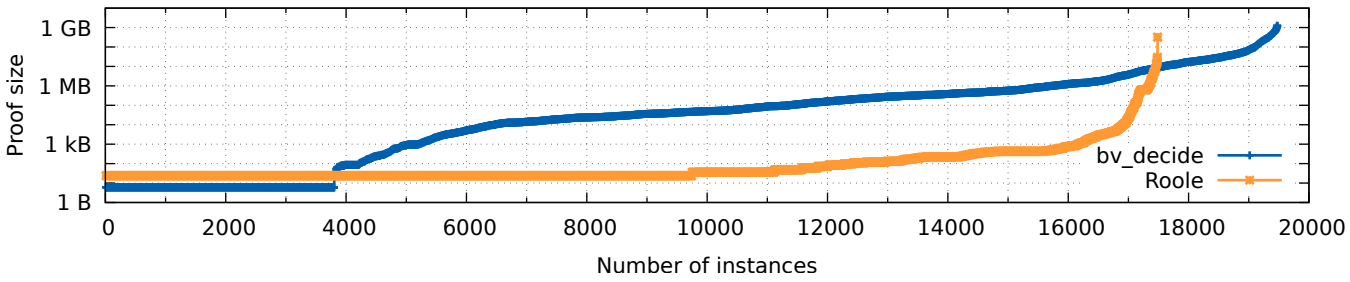


Fig. 9: Logarithmic certificate sizes of solved *unsatisfiable* problems. Despite **bv_decide** using simplification and binary **LRAT**, **Roole** proof certificates tend to be much more size-efficient, with typically orders-of-magnitude less storage space needed.

References

[1] R. Brummayer and A. Biere, "Fuzzing and delta-debugging SMT solvers," in *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ser. SMT '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–5. [Online]. Available: https://doi.org/10.1145/1670412.1670413

[2] A. Niemetz, M. Preiner, and A. Biere, "Model-based API testing for SMT solvers," in *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories affiliated with the International Conference on Computer-Aided Verification (CAV 2017), Heidelberg, Germany, July 22 - 23, 2017*, ser. CEUR Workshop Proceedings, M. Brain and L. Hadarean, Eds., vol. 1889. CEUR-WS.org, 2017, pp. 3–14. [Online]. Available: https://ceur-ws.org/Vol-1889/paper1.pdf

[3] G. Kremer, A. Niemetz, and M. Preiner, "ddSMT 2.0: Better delta debugging for the SMT-LIBv2 language and friends," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds., vol. 12760. Springer, 2021, pp. 231–242. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_11

[4] A. Niemetz, M. Preiner, and C. W. Barrett, "Murxla: A modular and highly extensible API fuzzer for SMT solvers," in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 92–106. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_5

[5] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in SMT solvers using blackbox mutational fuzzing," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 701–712. [Online]. Available: https://doi.org/10.1145/3368089.3409763

[6] D. Winterer, C. Zhang, and Z. Su, "Validating SMT solvers via semantic fusion," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[7] ——, "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 193:1–193:25, 2020. [Online]. Available: https://doi.org/10.1145/3428261

[8] M. Bringolf, D. Winterer, and Z. Su, "Finding and understanding incompleteness bugs in SMT solvers," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 43:1–43:10. [Online]. Available: https://doi.org/10.1145/3551349.3560435

[9] D. Winterer and Z. Su, "Validating SMT solvers for correctness and performance via grammar-based enumeration," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 2378–2401, 2024. [Online]. Available: https://doi.org/10.1145/3689795

[10] M. Sun, Y. Yang, and Y. Zhou, "Once4all: Skeleton-guided SMT solver fuzzing with llm-synthesized generators," in *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2026, Pittsburgh, PA, USA, March 22-26, 2026*, B. C. Lee, H. Xu, M. Silberstein, and B. Li, Eds. ACM, 2026, pp. 1316–1332. [Online]. Available: https://doi.org/10.1145/3779212.3790195

[11] M. J. H. Heule and A. Biere, "Proofs for satisfiability problems," in *All about Proofs, Proofs for All (APPA)*, ser. Math. Logic and Foundations. College Pub., 2015, vol. 55.

[12] M. J. H. Heule, "Proofs of unsatisfiability," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 635–668.

[13] ——, "The DRAT format and drat-trim checker," *CoRR*, vol. abs/1610.06229, 2016. [Online]. Available: http://arxiv.org/abs/1610.06229

[14] L. Cruz-Filipe, M. J. H. Heule, J. Hunt, Warren A., M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, ser. Lecture Notes in Computer Science, L. de Moura, Ed. Springer, 2017, pp. 220–236. [Online]. Available: https://doi.org/10.1007/978-3-319-63046-5_14

[15] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[16] H. Barbosa, J. C. Blanchette, M. Fleury, and P. Fontaine, "Scalable fine-grained proofs for formula processing," *J. Autom. Reason.*, vol. 64, no. 3, pp. 485–510, 2020. [Online]. Available: https://doi.org/10.1007/s10817-018-09502-y

[17] J. Hoenicke and T. Schindler, "A simple proof format for SMT," in *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, ser. CEUR Workshop Proceedings, D. Déharbe and A. E. J. Hyvärinen, Eds. CEUR-WS.org, 2022, pp. 54–70. [Online]. Available: https://ceur-ws.org/Vol-3185/paper9527.pdf

[18] H. Lachnitt, M. Fleury, L. Aniva, A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, and C. Tinelli, "IsaRare: Automatic verification of SMT rewrites in Isabelle/HOL," in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds. Springer, 2024, pp. 311–330. [Online]. Available: https://doi.org/10.1007/978-3-031-57246-3_17

[19] X. Shi, Y. Fu, J. Liu, M. Tsai, B. Wang, and B. Yang, "CoqQFBV: A scalable certified SMT quantifier-free bit-vector solver," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, A. Silva and K. R. M. Leino, Eds. Springer, 2021, pp. 149–171. [Online]. Available: https://doi.org/10.1007/978-3-030-81688-9_7

[20] H. Böving, S. Bhat, L. Cicolini, A. C. Keizer, L. Frénot, A. Mohamed, L. Stefanesco, H. Khan, J. Clune, C. W. Barrett, and T. Grosser, "Interactive bitvector reasoning using verified bit-blasting," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 3259–3285, 2025. [Online]. Available: https://doi.org/10.1145/3763167

[21] J. Onderka and S. Ratschan, "Input-based three-valued abstraction refinement," in *Verification, Model Checking, and Abstract Interpretation - 27th International Conference, VMCAI 2026, Rennes, France, January 12-13, 2026, Proceedings*, ser. Lecture Notes in Computer Science, Y. Chen, T. P. Jensen, and O. Lengál, Eds. Springer, 2026, pp. 237–262. [Online]. Available: https://doi.org/10.1007/978-3-032-15700-3_12

[22] ——, "Fast three-valued abstract bit-vector arithmetic," in *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, ser. Lecture Notes in Computer Science, B. Finkbeiner and T. Wies, Eds. Springer, 2022, pp. 242–262. [Online]. Available: https://doi.org/10.1007/978-3-030-94583-1_12

[23] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, and C. Tinelli, "SMT-LIB release 2024 (non-incremental benchmarks)," Apr. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11061097

[24] A. Nötzli, H. Barbosa, A. Niemetz, M. Preiner, A. Reynolds, C. W. Barrett, and C. Tinelli, "Reconstructing fine-grained proofs of rewrites using a domain-specific language," in *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, A. Griggio and N. Rungta, Eds. IEEE, 2022, pp. 65–74. [Online]. Available: https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12

[25] M. Jonáš, F. Bobot, D. Déharbe, and D. Winterer, "SMT-COMP 2025: results for the QF_Bitvec division in the single query track," https://smt-comp.github.io/2025/results/qf_bitvec-single-query/, 2025, results were generated on 2025-08-11. Accessed 2026-07-22.

[26] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds., vol. 13965. Springer, 2023, pp. 3–17. [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_1

[27] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[28] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[29] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024," in *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, ser. Department of Computer Science Report Series B, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., vol. B-2024-1. University of Helsinki, 2024, pp. 8–10.

[30] ——, "CaDiCaL 2.0," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, ser. Lecture Notes in Computer Science, A. Gurfinkel and V. Ganesh, Eds., vol. 14681. Springer, 2024, pp. 133–152.

[31] C. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, 1995.

[32] P. Bjesse and J. H. Kukula, "Automatic generalized phase abstraction for formal verification," in *2005 International Conference on Computer-Aided Design, ICCAD 2005, San Jose, CA, USA, November 6-10, 2005*. IEEE Computer Society, 2005, pp. 1076–1082.

[33] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157675

[34] M. L. Case, J. Baumgartner, H. Mony, and R. Kanzelman, "Approximate reachability with combined symbolic and ternary simulation," in *FMCAD*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 109–115.

[35] T. Melham, "Symbolic trajectory evaluation," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 831–870.

[36] A. Niemetz and M. Preiner, "Ternary propagation-based local search for more bit-precise reasoning," in *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 214–224.

[37] N. Froleyks, E. Yu, and A. Biere, "Ternary simulation as abstract interpretation (work in progress)," in *27th GMM/ITG/GI Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV'24), Kaiserslautern, Germany*, ser. ITG-Fachberichte, W. Kunz, Ed., vol. 314. VDE Verlag, 2024, pp. 148–151.

[38] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. [Online]. Available: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[39] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi, "Stochastic local search for satisfiability modulo theories," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 1136–1143. [Online]. Available: https://doi.org/10.1609/aaai.v29i1.9372

[40] "Zulip chat topic: Lean4 debugger," https://leanprover-community.github.io/archive/stream/270676-lean4/topic/Lean4.20Debugger.html, [Accessed 2026-05-04].

[41] H. S. Warren Jr., *Hacker's Delight, Second Edition*. Pearson Education, 2013. [Online]. Available: http://www.hackersdelight.org/

[42] J. Horák and M. Jonáš, "BDD-based formula approximations for quantified bit-vector satisfiability," in *Tools and Algorithms for the Construction and Analysis of Systems - 32nd International Conference, TACAS 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. Junges and G. Katz, Eds. Springer, 2026, pp. 213–232. [Online]. Available: https://doi.org/10.1007/978-3-032-22752-2_11

[43] J. Onderka, "machine-check," https://machine-check.org/, 2026.

[44] ——, "Artefact to the paper "Lean certified bitvector solving without bitblasting"," https://doi.org/10.5281/zenodo.20120022, 2026, [Accessed 2026-08-01].

[45] ——, "GitHub - Roole," https://github.com/onderjan/roole, 2026, this paper discusses the version 0.1.0. [Accessed 2026-05-11].

[46] ——, "GitHub - Roolean," https://github.com/onderjan/roolean, 2026, this paper discusses the version 0.1.0. [Accessed 2026-05-11].