# Bit-Precise Reasoning via Int-Blasting*

Yoni Zohar[1], Ahmed Irfan[1**], Makai Mann[1], Aina Niemetz[1], Andres Nötzli[1], Mathias Preiner[1], Andrew Reynolds[2], Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University, Stanford, USA
[2] The University of Iowa, Iowa City, USA

**Abstract.** The state of the art for bit-precise reasoning in the context of Satisfiability Modulo Theories (SMT) is a SAT-based technique called bit-blasting where the input formula is first simplified and then translated to an equisatisfiable propositional formula. The main limitation of this technique is scalability, especially in the presence of large bit-widths and arithmetic operators. We introduce an alternative technique, which we call *int-blasting*, based on a translation to an extension of integer arithmetic rather than propositional logic. We present several translations, discuss their differences, and evaluate them on benchmarks that arise from the verification of rewrite rule candidates for bit-vector solving, as well as benchmarks from SMT-LIB. We also provide preliminary results on 35 benchmarks that arise from smart contract verification. The evaluation shows that this technique is particularly useful for benchmarks with large bit-widths and can solve benchmarks that the state of the art cannot.

## 1 Introduction

Bit-precise reasoning is paramount for software and hardware verification. Bit-vectors directly and naturally model basic building blocks of both software and hardware, like registers, integers, memory, and more. Many applications rely on satisfiability modulo theories (SMT) for reasoning about bit-vectors, and the number of solvers and techniques for handling bit-vector formulas is large and increasing. One indication of that is the number of bit-vector benchmarks in the SMT-LIB [7] benchmark library, by far the highest among all benchmark categories in the library. The current state of the art for determining the satisfiability of fixed-size bit-vector formulas is a technique called *bit-blasting*. With this technique, the input formula is first simplified by means of satisfiability preserving transformations. Then, it is fully reduced to a propositional satisfiability (SAT) problem and handed to a SAT solver [11]. The success of this approach

* This work was supported in part by DARPA (awards N66001-18-C-4012, FA8650-18-2-7854 and FA8650-18-2-7861), ONR (award N68335-17-C-0558), the Stanford Center for Blockchain Research, Certora Inc., and by an NSF Graduate Fellowship (to Makai Mann).

** This author's contributions were made while he was a postdoc at Stanford University.

is mainly due to the fact that modern SAT solvers are able to solve complex propositional formulas with millions of variables very efficiently. Thus, problems that can be efficiently encoded as SAT instances can leverage the great progress in SAT solving. Nevertheless, bit-blasting has scalability limitations, especially with large bit-widths. In fact, even for conventional bit-widths such as 32 and 64, bit-blasting may face scalability issues, in particular for formulas containing bit-vector arithmetic operators.

The work described in this paper is part of an ongoing effort to improve the scalability of bit-precise reasoning by offering alternatives to bit-blasting that primarily use word-level reasoning and rely on bit-level reasoning only when needed. Specifically, we study a translation of bit-vector formulas to an extension of integer arithmetic; that is, we replace bit-blasting by *int-blasting*. To encode bitwise bit-vector operators, the extension introduces an operator that represents the bitwise *and* operation over integers, parameterized by bit-width. The idea of using arithmetic reasoning to solve bit-vector formulas is not new (e.g., [12, 19]). We believe, however, that recent progress in arithmetic solvers (e.g., [14]), especially for non-linear arithmetic (e.g., [17, 18, 26, 41]), make it worthwhile to revisit this approach, as these techniques can be leveraged by applying them to the int-blasted formulas.

We study two kinds of translations: an eager one and a (semi-)lazy one. In the former, the input bit-vector formula is eagerly translated to an integer formula with uninterpreted functions. In the latter, most of the formula is translated eagerly while preserving satisfiability except for bitwise operators (such as bitwise *and*), which are handled lazily using a counterexample-guided abstraction refinement (CEGAR) loop [28].

We additionally consider two alternative ways to encode bitwise bit-vector operations in integer arithmetic for the purposes of abstraction refinement: one based on a polynomial expansion and the other based on bit-level comparisons. Both alternatives require non-linear arithmetic reasoning, as recovering individual bits from an integer encoding of a bit-vector is achieved via division and modulo operations. The main difference between the two alternatives in the context of an SMT solver implementation, and our reason for considering both, is that the first further exercises the arithmetic subsolver whereas the second relies more heavily on the underlying SAT engine.

*Contributions.* We have implemented the aforementioned variants of int-blasting in the cvc5 SMT solver (the successor of CVC4 [5]) and evaluated our implementation experimentally to estimate its potential. For that, we compiled a new set of benchmarks, encoding equivalence checks of rewrite rule candidates proposed by the syntax-guided rewrite rule enumeration framework presented by Nötzli et al. [36]. We show that for those benchmarks, int-blasting significantly outperforms bit-blasting as the bit-width increases. We further evaluated our technique on the QF_BV benchmarks in the SMT-LIB benchmark library [6], as well as on 35 benchmarks that arise from smart contract verification, and observed that int-blasting is complementary to bit-blasting on those benchmarks.

| **Symbol** | **SMT-LIB Syntax** | **Arity** |
|---|---|---|
| $=$ | $=$ | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $<_{\mathrm{u}}^{\mathrm{BV}}$, $>_{\mathrm{u}}^{\mathrm{BV}}$ | bvult, bvugt | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $<_{\mathrm{s}}^{\mathrm{BV}}$, $>_{\mathrm{s}}^{\mathrm{BV}}$ | bvslt, bvsgt | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $\leq_{\mathrm{u}}^{\mathrm{BV}}$, $\geq_{\mathrm{u}}^{\mathrm{BV}}$ | bvule, bvuge | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $\leq_{\mathrm{s}}^{\mathrm{BV}}$, $\geq_{\mathrm{s}}^{\mathrm{BV}}$ | bvsle, bvsge | $\sigma_{[n]} \times \sigma_{[n]} \to \mathsf{Bool}$ |
| $\sim^{\mathrm{BV}}$, $-^{\mathrm{BV}}$ | bvnot, bvneg | $\sigma_{[n]} \to \sigma_{[n]}$ |
| $\&^{\mathrm{BV}}$, $\mid^{\mathrm{BV}}$, $\oplus^{\mathrm{BV}}$ | bvand, bvor, bvxor | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $\lll^{\mathrm{BV}}$, $\ggg^{\mathrm{BV}}$ | bvshl, bvlshr | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $+^{\mathrm{BV}}$, $-^{\mathrm{BV}}$ | bvadd, bvsub | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $\cdot^{\mathrm{BV}}$ | bvmul | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $\mathrm{mod}^{\mathrm{BV}}$, $\mathrm{div}^{\mathrm{BV}}$ | bvurem, bvudiv | $\sigma_{[n]} \times \sigma_{[n]} \to \sigma_{[n]}$ |
| $[u:l]^{\mathrm{BV}}$ | extract | $\sigma_{[n]} \to \sigma_{[u-l+1]}$ |
| $\circ^{\mathrm{BV}}$ | concatenation | $\sigma_{[n]} \times \sigma_{[m]} \to \sigma_{[n+m]}$ |

Table 1: Considered bit-vector operators with SMT-LIB 2 syntax. In $[u:l]^{\mathrm{BV}}$, $0 \leq l \leq u < n$.

*Outline.* After introducing some background and notation in Section 2, Section 3 introduces an extension of the theory of integer arithmetic, in which an operator representing bitwise *and* is added for each bit-width. We present a translation from the theory of bit-vectors to this extension, in Section 4, along with eager and lazy algorithms for solving the translated formula. We discuss an initial experimental evaluation of the various translations in Section 5 and conclude in Section 6 with some directions for further work.

## 2 Preliminaries

We review the usual notions and terminology of many-sorted first-order logic with equality (see [21, 44] for more detailed information). Let $S$ be a set of *sort symbols*. For every sort $\sigma \in S$, we assume an infinite set of variables that are pairwise disjoint across sorts. A *signature* $\Sigma$ consists of a set $\Sigma^s \subseteq S$ of sort symbols and a set $\Sigma^f$ of function symbols. Arities of function symbols are defined in the usual way, and correspond to their types, that is, they take the form $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ where $\sigma_1, \ldots, \sigma_n, \sigma$ are sorts. Constants are treated as functions with no input sorts. We assume that $\Sigma$ includes a sort $\mathsf{Bool}$, interpreted as the Boolean domain, and the $\mathsf{Bool}$ constants $\top$ and $\bot$ (respectively for *true* and *false*). Signatures do not contain separate predicate symbols and use instead function symbols with $\mathsf{Bool}$ return type.

We assume the usual definitions of well-sorted terms, literals, and formulas, and refer to them as $\Sigma$-terms, $\Sigma$-literals, and $\Sigma$-formulas, respectively. These are constructed using the symbols in $\Sigma$, variables, quantifiers and connectives, as well as the if-then-else constructor $\text{ite}(\varphi, t_1, t_2)$, where $\varphi$ is a formula and $t_1$ and $t_2$ are $\Sigma$-terms of the same sort.

A *$\Sigma$-interpretation* $\mathcal{I}$ maps: each $\sigma \in \Sigma^s$ to a distinct non-empty set of values $\sigma^{\mathcal{I}}$ (the *domain* of $\sigma$ in $\mathcal{I}$); each variable $x$ of sort $\sigma$ to an element $x^{\mathcal{I}} \in \sigma^{\mathcal{I}}$; and each $f^{\sigma_1 \cdots \sigma_n \sigma} \in \Sigma^f$ to a total function $f^{\mathcal{I}}$: $\sigma_1^{\mathcal{I}} \times ... \times \sigma_n^{\mathcal{I}} \to \sigma^{\mathcal{I}}$ if $n > 0$, and to an element in $\sigma^{\mathcal{I}}$ if $n = 0$. We use the usual notion of a satisfiability relation $\models$ between $\Sigma$-interpretations and $\Sigma$-formulas. A term of the form $\text{ite}(\varphi, t_1, t_2)$ is interpreted in an interpretation $\mathcal{I}$ as $t_1^{\mathcal{I}}$ if $\mathcal{I} \models \varphi$, and as $t_2^{\mathcal{I}}$ otherwise. For each sub-signature $\Sigma'$ of $\Sigma$, the *reduct* $\mathcal{I}^{\Sigma'}$ of $\mathcal{I}$ to $\Sigma'$ is obtained from $\mathcal{I}$ by restricting it to the sorts and symbols of $\Sigma'$.

A *$\Sigma$-theory* $T$ is a non-empty class of $\Sigma$-interpretations, such that every interpretation that only disagrees from one in $T$ on the variable assignments is also in $T$. A $\Sigma$-formula $\varphi$ is *$T$-satisfiable* (resp., *$T$-unsatisfiable*, *$T$-valid*) if it is satisfied by some (resp., no, all) interpretations in $T$.

The signature $\Sigma_{\text{BV}}$ of fixed-size bit-vectors is defined in the SMT-LIB 2 standard [7], and includes a unique sort for each positive integer $n$ (representing the bit-vector width), denoted here as $\sigma_{[n]}$. Without loss of generality, we take $\Sigma_{\text{BV}}$ to consist of a restricted set of bit-vector function symbols (or *bit-vector operators*) as listed in Table 1. The selection of operators is arbitrary but complete in the sense that it suffices to express all bit-vector operators defined in SMT-LIB 2. We further assume that $\Sigma_{\text{BV}}$ includes all *bit-vector constants* of sort $\sigma_{[n]}$ for each $n$, represented as bit-strings. To simplify the notation, we will sometimes denote them by the corresponding natural number. If a term $t$ has sort $\sigma_{[n]}$ then we denote $n$ by $\kappa(t)$. The SMT-LIB 2 standard for the $\Sigma_{\text{BV}}$-theory $T_{\text{BV}}$ defines a set of $\Sigma_{\text{BV}}$-interpretations $\mathcal{I}$, such that for each positive integer $n$, ${\sigma_{[n]}}^{\mathcal{I}}$ is the set of all bit-vectors of size $n$ and function symbols are interpreted as the corresponding word-level operations in these domains (for details, see [38, 39]). All function symbols (of non-zero arity) in $\Sigma_{\text{BV}}$ are overloaded for every $\sigma_{[n]} \in \Sigma_{\text{BV}}$. We refer to the $i$-th bit of $t$ as $t[i]$ with $0 \leq i < n$. We interpret $t[0]$ as the least significant bit (LSB), and $t[n-1]$ as the most significant bit (MSB). The unsigned interpretation of a bit-vector $v$ of width $k$ as a natural number is given by $[v]_{\mathbb{N}} = \Sigma_{i=0}^{k-1} v\,[i] \cdot 2^i$, and its signed interpretation as an integer is given by $[v]_{\mathbb{Z}} = -v\,[k-1] \cdot 2^{k-1} + [v[k-2:0]^{\text{BV}}]_{\mathbb{N}}$. Given $0 \leq n < 2^k$, the bit-vector of width $k$ with unsigned interpretation $n$ is denoted $[n]^k_{\mathbb{BV}}$. This notation is extended also for $n$ outside this bound by defining $[n]^k_{\mathbb{BV}} := \left[n \bmod 2^k\right]^k_{\mathbb{BV}}$.[3]

We consider a theory $T_{\text{IA}}$ of integer arithmetic whose signature $\Sigma_{\text{IA}}$ includes a single sort $\mathsf{Int}$, function symbols $+, -, \cdot$, div , and mod of arity $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$, the function symbol pow2 of arity $\mathsf{Int} \to \mathsf{Int}$, the predicate symbols $<$ and $\leq$ of arity $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Bool}$, and a constant symbol of sort $\mathsf{Int}$ for every integer. The pow2-free fragment of this theory is identical to the SMT-LIB 2 theory of integers [43]. Its models are all possible expansions of the models of the SMT-

[3] The result of this modulo operation is non-negative, even when the argument is negative, as specified by the SMT-LIB 2 standard.

LIB 2 theory obtained by interpreting $\mathrm{pow2}(n)$ as $2^n$ when $n$ is a non-negative constant, and interpreting $\mathrm{pow2}(n)$ arbitrarily otherwise.

## 3 Integer Arithmetic with Bitwise *and*

In this paper, we reduce $T_{\mathrm{BV}}$-satisfiability to satisfiability in a theory that extends $T_{\mathrm{IA}}$ as follows. We first extend the signature $\Sigma_{\mathrm{IA}}$ with binary function symbols $\&_k^{\mathbb{N}} : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$, one for each positive integer $k$. We define two theories for the extended signature: the first treats the new symbols $\&_k^{\mathbb{N}}$ as uninterpreted functions (UF); the second interprets them as bitwise *and* operators on integers modulo $2^k$. This is defined formally as follows:

**Definition 1.** *The signature $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$ is obtained from $\Sigma_{\mathrm{IA}}$ by adding a function symbol $\&_k^{\mathbb{N}}$ of arity $\mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ for each $k > 0$. The $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-theory $T_{\mathrm{IAUF}}$ consists of all $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-interpretations whose $\Sigma_{\mathrm{IA}}$-reduct is a $T_{\mathrm{IA}}$-interpretation. The $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-theory $T_{\mathrm{IA}(\&^{\mathbb{N}})}$ consists of all $T_{\mathrm{IAUF}}$-interpretations $\mathcal{I}$ in which $(\&_k^{\mathbb{N}})^{\mathcal{I}}(a,b) = \left[[a]_{\mathbb{BV}}^k \&^{\mathrm{BV}} [b]_{\mathbb{BV}}^k\right]_{\mathbb{N}}$.*

Following Footnote 3, notice that $\&_k^{\mathbb{N}}$ is fully interpreted, even for integers that are not between 0 and $2^k$. In the following definition, we identify a decidable fragment of $T_{\mathrm{IA}(\&^{\mathbb{N}})}$ that corresponds to formulas that originate from $\Sigma_{\mathrm{BV}}$.

**Definition 2.** *Let $n$ be a positive integer, and let $t$ be a $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-term of sort* $\mathsf{Int}$. *A $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-formula $\varphi$ is a $t$-$n$-*range constraint *if it has the form $\bot$, $\top$, or $(0 \bowtie_1 t \wedge t \bowtie_2 n)$ for $\bowtie_1, \bowtie_2 \in \{<, \leq\}$. A formula $\varphi$ is a* range constraint *if it is a $t$-$n$-range constraint for some $t$ and $n$; a formula $\varphi$ is* bounded *if there are quantifier-free formulas $\varphi_1$, $\varphi_2$, $\psi_1, \dots, \psi_m$ such that $\varphi = \varphi_1 \wedge \varphi_2$, $\varphi_2 = \bigwedge_{i=1}^m \psi_i$, each $\psi_i$ is a range constraint, and for each term $t$ that occurs in $\varphi_1$ that is either a variable or has the form $\&_k^{\mathbb{N}}(t_1, t_2)$, there exist $1 \leq i \leq m$ and a positive integer $n$ such that $\psi_i$ is a $t$-$n$-range constraint.*

*Example 1.* Let $\varphi_1$ be $(\&_3^{\mathbb{N}}(x,0) < x) \vee (\&_3^{\mathbb{N}}(x,y) < x)$ and $\varphi_2$ be $(0 \leq x \wedge x < 8) \wedge (0 \leq y \wedge y < 8) \wedge (0 \leq \&_3^{\mathbb{N}}(x,0) \wedge \&_3^{\mathbb{N}}(x,0) < 8)$. Then $\varphi_1 \wedge \varphi_2$ is not bounded, because it does not include any range constraint for $\&_3^{\mathbb{N}}(x,y)$. Consider the formula $\varphi_2'$ obtained from $\varphi_2$ by conjoining the range constraint $(0 \leq \&_3^{\mathbb{N}}(x,y) \wedge \&_3^{\mathbb{N}}(x,y) < 8)$. Then $\varphi_1 \wedge \varphi_2'$ is bounded.

A naive algorithm for deciding $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability of bounded $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-formulas can be obtained by enumerating all possible values for variables within the specified bounds, and checking if the formula evaluates to true. If it does, a full model can be constructed according to Definition 1. In fact, bounds over variables are sufficient for $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability since the semantics of $\&^{\mathbb{N}}$ in $T_{\mathrm{IA}(\&^{\mathbb{N}})}$ is fixed. A similar decision procedure can be obtained for $T_{\mathrm{IAUF}}$-satisfiability, which does require the bounds over $\&^{\mathbb{N}}$-terms. This algorithm gives us:

**Proposition 1.** *The $T_{\mathrm{IA}(\&^{\mathbb{N}})}$- and $T_{\mathrm{IAUF}}$-satisfiability of bounded formulas is decidable.*

In the next section, we show that the class of bounded formulas in $T_{\mathrm{IA}(\&^{\mathbb{N}})}$ is both useful and effective: it is expressive enough to describe bit-vector formulas and can be reduced to problems for which there are efficient solvers.

## 4 Int-Blasting

In this section, we present our integer-based approach for solving $T_{\mathrm{BV}}$-satisfiability. There are two stages in our approach. The first, described in Section 4.1 and proved correct in Section 4.2, translates $T_{\mathrm{BV}}$-formulas to $T_{\mathrm{IA}(\&^{\mathbb{N}})}$. The second, described in Sections 4.3 and 4.4, solves the resulting formulas by eager and lazy reductions to $T_{\mathrm{IAUF}}$, respectively. Although we developed our translations for the full fragment of $T_{\mathrm{BV}}$, to simplify the exposition in this paper, we will restrict ourselves to quantifier-free formulas only.

### 4.1 From $T_{\mathrm{BV}}$ to $T_{\mathrm{IA}(\&^{\mathbb{N}})}$

The first step is to translate $\Sigma_{\mathrm{BV}}$-formulas to equisatisfiable $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-formulas, so that the original formula is $T_{\mathrm{BV}}$-satisfiable if, and only if, its translation is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable. For this purpose, we define a translation function $\mathcal{T}$ as shown in Figure 1, which recursively translates $\Sigma_{\mathrm{BV}}$-formulas to $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$ (via the conversion function $\mathcal{C}$) and collects additional lemmas about the ranges of the translated variables and the introduced $\&^{\mathbb{N}}$-terms (via the function $\textsc{Lem}^{\leq}$).

*Conversion Function $\mathcal{C}$.* We use a one-to-one mapping $\chi$ from *bit-vector variables* (i.e., variables of sort $\sigma_{[k]}$ for some $k > 0$) to integer variables (i.e., variables of sort $\mathsf{Int}$). A *bit-vector constant* $c$ is translated to its integer counterpart using $[\_]_{\mathbb{N}}$ which maps $c$ to its unsigned integer interpretation. For *Boolean connectives* $\diamond \in \{\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow\}$, *equalities*, and *unsigned comparators* $\bowtie^{\mathrm{BV}}$ with $\bowtie \in \{<, \leq, >, \geq\}$, the conversion function is recursively applied to their arguments. In the latter case, $\bowtie^{\mathrm{BV}}$ is replaced by its $\Sigma_{\mathrm{IA}}$ counterpart $\bowtie$. *Signed comparators* are handled similarly, except that the arguments are processed with function $\mathsf{uts}_k(\_)$ (**u**nsigned **t**o **s**igned with bit-width $k$), also defined in Figure 1, which ensures that the semantics of signed comparison is preserved properly. For a given integer $n$ in the range $0 \leq n < 2^k$, it returns $\left[[n]^k_{\mathbb{BV}}\right]_{\mathbb{Z}}$, the signed interpretation of the bit-vector whose unsigned interpretation is $n$. Bit-vector *addition* is translated to integer addition modulo $2^k$, where $k$ is the bit-width of the arguments. Bit-vector *subtraction*, *multiplication*, and *one's* and *two's complement* are handled similarly. For *division*, the SMT-LIB 2 standard defines a default value for bit-vector division by 0, but not for integer division by 0. This is handled by wrapping the translated division term in an ite, which embeds the semantics of bit-vector division within integer arithmetic. A similar pattern is followed for *remainder*. Note that there is no need to take the result modulo $2^k$ for one's complement and unsigned division and remainder, as they are guaranteed to be within the correct bounds. *Concatenation* and *extraction* are handled as expected, using multiplication, division, and modulo. *Left/right*

$\underline{\mathcal{T}\ \varphi}$:
$\mathcal{C}\ \varphi \wedge \textsc{Lem}^{\leq}(\varphi)$

$\underline{\mathcal{C}\ e}$:
Match $e$:

$$\begin{array}{lcll}
x & \to & \chi(x) & \\
c & \to & [c]_{\mathbb{N}} & \\
t_1 = t_2 & \to & \mathcal{C}\ t_1 = \mathcal{C}\ t_2 & \\
t_1 \bowtie^{\mathrm{BV}} t_2 & \to & \mathcal{C}\ t_1 \bowtie \mathcal{C}\ t_2 & \\
t_1 \bowtie_s{}^{\mathrm{BV}} t_2 & \to & \mathsf{uts}_k(\mathcal{C}\ t_1) \bowtie \mathsf{uts}_k(\mathcal{C}\ t_2) & \bowtie \in \{<, \leq, >, \geq\} \\
\diamond(\varphi_1, \ldots, \varphi_n) & \to & \diamond(\mathcal{C}\ \varphi_1, \ldots, \mathcal{C}\ \varphi_n) & \diamond \in \{\wedge, \vee, \Rightarrow, \neg, \Leftrightarrow\} \\
 & & & \\
t_1 +^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\ t_1 + \mathcal{C}\ t_2) \bmod 2^k & \\
t_1 -^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\ t_1 - \mathcal{C}\ t_2) \bmod 2^k & \\
t_1 \cdot^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\ t_1 \cdot \mathcal{C}\ t_2) \bmod 2^k & \\
\sim^{\mathrm{BV}} t_1 & \to & 2^k - (\mathcal{C}\ t_1 + 1) & \\
-^{\mathrm{BV}} t_1 & \to & (2^k - \mathcal{C}\ t_1) \bmod 2^k & \\
 & & & \\
t_1 \operatorname{div}^{\mathrm{BV}} t_2 & \to & \mathrm{ite}(\mathcal{C}\ t_2 = 0,\ 2^k - 1,\ \mathcal{C}\ t_1 \operatorname{div} \mathcal{C}\ t_2) & \\
t_1 \operatorname{mod}^{\mathrm{BV}} t_2 & \to & \mathrm{ite}(\mathcal{C}\ t_2 = 0,\ \mathcal{C}\ t_1,\ \mathcal{C}\ t_1 \bmod \mathcal{C}\ t_2) & \\
t_1 \circ^{\mathrm{BV}} t_2 & \to & \mathcal{C}\ t_1 \cdot 2^k + \mathcal{C}\ t_2 & \\
t_1[u:l]^{\mathrm{BV}} & \to & \mathcal{C}\ t_1 \operatorname{div} 2^l \bmod 2^{u-l+1} & \\
 & & & \\
t_1 \ll^{\mathrm{BV}} t_2 & \to & (\mathcal{C}\ t_1 \cdot \mathrm{pow2}(\mathcal{C}\ t_2)) \bmod 2^k & \\
t_1 \gg^{\mathrm{BV}} t_2 & \to & \mathcal{C}\ t_1 \operatorname{div} \mathrm{pow2}(\mathcal{C}\ t_2) & \\
 & & & \\
t_1\ \&^{\mathrm{BV}} t_2 & \to & \&_k^{\mathbb{N}}(\mathcal{C}\ t_1, \mathcal{C}\ t_2) & \\
t_1\ |^{\mathrm{BV}}\ t_2 & \to & \mathcal{C}\ ((t_1 +^{\mathrm{BV}} t_2) -^{\mathrm{BV}} (t_1\ \&^{\mathrm{BV}} t_2)) & \\
t_1 \oplus^{\mathrm{BV}} t_2 & \to & \mathcal{C}\ ((t_1\ |^{\mathrm{BV}}\ t_2) -^{\mathrm{BV}} (t_1\ \&^{\mathrm{BV}} t_2)) &
\end{array}$$

$\mathsf{uts}_k(x) = 2 \cdot (x \bmod 2^{k-1}) - x$

$\underline{\textsc{Lem}^{\leq}(e)}$:
Match $e$:

$$\begin{array}{lcll}
x & \to & 0 \leq \chi(x) < 2^{\kappa(x)} & \\
c & \to & \top & \\
t_1 = t_2 & \to & \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2) & \\
f^{\mathrm{BV}}(t_1, t_2) & \to & \begin{array}{l} 0 \leq \&_k^{\mathbb{N}}(\mathcal{C}\ t_1, \mathcal{C}\ t_2) < 2^k \wedge \\ \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2) \end{array} & f^{\mathrm{BV}} \in \{\&^{\mathrm{BV}}, |^{\mathrm{BV}}, \oplus^{\mathrm{BV}}\} \\
g^{\mathrm{BV}}(t_1, \ldots, t_n) & \to & \bigwedge_{i=1}^{n} \textsc{Lem}^{\leq}(t_i) & g^{\mathrm{BV}} \in \Sigma_{\mathrm{BV}} \setminus \{\&^{\mathrm{BV}}, |^{\mathrm{BV}}, \oplus^{\mathrm{BV}}\} \\
\diamond(\varphi_1, \ldots, \varphi_n) & \to & \bigwedge_{i=1}^{n} \textsc{Lem}^{\leq}(\varphi_i) &
\end{array}$$

Fig. 1: Translation $\mathcal{T}$ from $\Sigma_{\mathrm{BV}}$ to $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$. We denote by $k$ the bit-width $\kappa(t_2)$ of the second argument, except for the cases of $-^{\mathrm{BV}}$ and $\sim^{\mathrm{BV}}$, where it denotes the bit-width $\kappa(t_1)$ of the only argument; $x$ ranges over bit-vector variables; $\chi$ is a one-to-one mapping from bit-vector variables to integer variables; $c$ ranges over bit-vector constants.

*shifts* are obtained by multiplying/dividing the first argument by 2 to the power of the second argument. Bitwise *and* is translated to $\&^{\mathbb{N}}_k$, where $k$ is determined according to the bit-width of the bit-vector arguments. Bitwise *or* ($|^{\mathrm{BV}}$) and *xor* ($\oplus^{\mathrm{BV}}$) are reduced to other operators, using the following identities that hold for all bit-vectors $x$ and $y$ [46]:

$$x \;|^{\mathrm{BV}}\, y = (x +^{\mathrm{BV}} y) -^{\mathrm{BV}} (x \,\&^{\mathrm{BV}} y) \qquad x \oplus^{\mathrm{BV}} y = (x \;|^{\mathrm{BV}}\, y) -^{\mathrm{BV}} (x \,\&^{\mathrm{BV}} y) \tag{1}$$

*Lemmas Function* $\textsc{Lem}^{\leq}$. Function $\textsc{Lem}^{\leq}$ takes a $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-formula and collects necessary range constraints for integer variables and terms of the form $\&^{\mathbb{N}}_k(t_1, t_2)$ that are introduced by $\mathcal{C}$. For variables, the range is determined by the bit-width of the original bit-vector variable. For $\&^{\mathrm{BV}}$, $|^{\mathrm{BV}}$ and $\oplus^{\mathrm{BV}}$ terms, the constraint is determined by the bit-width of the arguments. Since $|^{\mathrm{BV}}$ and $\oplus^{\mathrm{BV}}$ are eliminated, the constraint is stated in terms of $\&^{\mathbb{N}}$. Notice that the $\&^{\mathrm{BV}}$ terms introduced by Equation (1) have the same arguments as the original terms. For all other terms and formulas, $\textsc{Lem}^{\leq}$ simply collects such constraints recursively.

## 4.2 Correctness

The correctness of $\mathcal{T}$ is stated in the following theorem. It follows from the SMT-LIB 2 semantics of bit-vectors and arithmetic, and from Definition 1. Its proof is by structural induction on $\varphi$. Most cases are similar to the correctness proof of the translation by Niemetz et al. [35], from bit-vectors with parametric width to integers, with the main difference being the case of $\&^{\mathrm{BV}}$. Unlike that work, where the quantified axiomatization had to be proven correct by induction on the bit-width, here the correctness follows directly from Definition 1.

**Theorem 1.** *A $\Sigma_{\mathrm{BV}}$-formula $\varphi$ is $T_{\mathrm{BV}}$-satisfiable iff $\mathcal{T}\,\varphi$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable.*

The theorem is actually stronger than stated: from any model $\mathcal{I}$ of $\mathcal{T}\,\varphi$ one can compute a satisfying assignment for $\varphi$'s free variables, simply by assigning to each free variable $x$ of $\varphi$ the bit-vector corresponding to the (integer) value of $\chi(x)$ in $\mathcal{I}$. An analogous result holds in the opposite direction as well.

We prove this theorem in the remainder of this section, focusing on the left-to-right direction. The other direction is shown similarly. Throughout the proof, we employ the following notation:

$$\mathit{bsel}_i(x) := (x \text{ div } 2^i) \bmod 2 \tag{2}$$

The term $\mathit{bsel}_i(x)$ represents the selection of the $i$-th bit in the bit-vector representation of $x$. In particular, it is always 0 or 1.

Let $\varphi$ be a $\Sigma_{\mathrm{BV}}$-formula. We assume without loss of generality that $\varphi$ does not have any occurrence of the *ite* operator, as it can be eliminated using the Boolean operators and the introduction of fresh variables. Suppose $\varphi$ is $T_{\mathrm{BV}}$-satisfiable and let $\mathcal{A}$ be a $T_{\mathrm{BV}}$-interpretation that satisfies it. We prove that $\mathcal{T}\,\varphi$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable. Define the following $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-interpretation $\mathcal{B}$: all function symbols and constants are interpreted as defined by $T_{\mathrm{IA}(\&^{\mathbb{N}})}$; division and remainder by

0, as well as $\mathrm{pow2}(m)$ for any negative $m$ are defined arbitrarily; for every bit-vector variable $x$, the value in $\mathcal{B}$ of its translation is the unsigned interpretation of its value in $\mathcal{A}$, that is:

$$\chi(x)^{\mathcal{B}} := \left[x^{\mathcal{A}}\right]_{\mathbb{N}}.$$

This fixes $\mathcal{B}$. Also, $\mathcal{B}$ is a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation by construction.

Notice that every term of the form $t_1$ div $t_2$ or $t_1$ mod $t_2$ that occurs in the translation is guarded by an assumption that $t_2$ is not 0. Similarly, pow2 is always applied on arguments that are guaranteed to be non-negative. Therefore, the interpretation of these corner cases in $\mathcal{B}$ can indeed remain arbitrary.

We first prove the following lemma, which states the correctness of the translation for terms, that is, that the translation of each $\Sigma_{\mathrm{BV}}$-term is interpreted in $\mathcal{B}$ as the unsigned interpretation of the original term's value in $\mathcal{A}$.

**Lemma 1.** $(\mathcal{C}\ t)^{\mathcal{B}} = \left[t^{\mathcal{A}}\right]_{\mathbb{N}}$ *for every $\Sigma_{\mathrm{BV}}$-term $t$ of sort $\sigma_{[k]}$.*

*Proof.* By induction on $t$. If $t$ is a bit-vector variable then $(\mathcal{C} t)^{\mathcal{B}} = \chi(t)^{\mathcal{B}} = \left[t^{\mathcal{A}}\right]_{\mathbb{N}}$ by the definitions of $\mathcal{C}$ and $\mathcal{B}$. If $t$ is a bit-vector constant then $(\mathcal{C}\ t)^{\mathcal{B}} = [t]_{\mathbb{N}}^{\mathcal{B}} = \left[t^{\mathcal{A}}\right]_{\mathbb{N}}$ by the definition of $[\_]_{\mathbb{N}}$. If $t$ has the form $t_1 +^{\mathrm{BV}} t_2$, then by the definition of $\mathcal{C}$, $(\mathcal{C}\ t)^{\mathcal{B}} = (\mathcal{C}\ t_1 + \mathcal{C}\ t_2 \bmod 2^k)^{\mathcal{B}}$. Now, $2^k \neq 0$ and hence the interpretation in $\mathcal{B}$ is governed by $T_{\mathrm{IA}(\&^{\mathbb{N}})}$, and is equal to $(\mathcal{C}\ t_1)^{\mathcal{B}} + (\mathcal{C}\ t_2)^{\mathcal{B}} \bmod 2^k$. By the induction hypothesis, this is equal to $\left[t_1^{\mathcal{A}}\right]_{\mathbb{N}} + \left[t_2^{\mathcal{A}}\right]_{\mathbb{N}} \bmod 2^k$. By the semantics of $+^{\mathrm{BV}}$ according to the SMT-LIB 2 standard, this is the same as $\left[t^{\mathcal{A}}\right]_{\mathbb{N}}$. The other bit-vector operators are handled similarly. $|^{\mathrm{BV}}$ and $\oplus^{\mathrm{BV}}$ are eliminated by $\mathcal{C}$, and the correctness of this elimination follows from [46].

Finally suppose $t$ has the form $t_1\ \&^{\mathrm{BV}} t_2$. By the definition of $\mathcal{C}$, $(\mathcal{C}\ t)^{\mathcal{B}} = \&_k^{\mathbb{N}}(\mathcal{C}t_1, \mathcal{C}t_2)^{\mathcal{B}}$. By Definition 1, since $\mathcal{B}$ is a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation, $\&_k^{\mathbb{N}}(\mathcal{C}t_1, \mathcal{C}t_2)^{\mathcal{B}} = \left[\left[(\mathcal{C}\ t_1)^{\mathcal{B}}\right]_{\mathbb{BV}}^{k} \&^{\mathrm{BV}} \left[(\mathcal{C}\ t_2)^{\mathcal{B}}\right]_{\mathbb{BV}}^{k}\right]_{\mathbb{N}}$. By the induction hypothesis, this is the same as $\left[\left[\left[t_1^{\mathcal{A}}\right]_{\mathbb{N}}\right]_{\mathbb{BV}}^{k} \&^{\mathrm{BV}} \left[\left[t_2^{\mathcal{A}}\right]_{\mathbb{N}}\right]_{\mathbb{BV}}^{k}\right]_{\mathbb{N}}$. Now, $[\_]_{\mathbb{N}}$ and $[\_]_{\mathbb{BV}}$ cancel each other, and hence we get $\left[t_1^{\mathcal{A}}\ \&^{\mathrm{BV}} t_2^{\mathcal{A}}\right]_{\mathbb{N}}$, which is the same as $\left[t^{\mathcal{A}}\right]_{\mathbb{N}}$. □

Going back to $\varphi$, which is assumed to be satisfied by $\mathcal{A}$, we now prove that $\mathcal{B} \models \mathcal{T}\varphi$, that is $\mathcal{B} \models \mathcal{C}\varphi \wedge \mathrm{LEM}^{\leq}(\varphi)$. First, we prove that $\mathcal{B} \models \mathcal{C}\varphi$ by induction on $\varphi$. The induction step, in which $\varphi$ is recursively constructed from propositional connectives, trivially follows from the induction hypothesis, hence we focus on the induction base. In the induction base, $\varphi$ has either the form $t_1 = t_2$, $t_1 \bowtie^{\mathrm{BV}} t_2$, or $t_1 \bowtie_s{}^{\mathrm{BV}} t_2$ for some $\bowtie \in \{<, \leq, >, \geq\}$. If $\varphi$ has the form $t_1 = t_2$, then since $\mathcal{A} \models \varphi$, $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$. By Lemma 1, $(\mathcal{C}\ t_1)^{\mathcal{B}} = \left[t_1^{\mathcal{A}}\right]_{\mathbb{N}} = \left[t_2^{\mathcal{A}}\right]_{\mathbb{N}} = (\mathcal{C}\ t_2)^{\mathcal{B}}$, and therefore $\mathcal{B} \models \mathcal{C}\ \varphi$. If $\varphi$ has the form $t_1 <_{\mathrm{u}}^{\mathrm{BV}} t_2$ then since $\mathcal{A} \models \varphi$, $t_1^{\mathcal{A}} <^{\mathrm{BV}} t_2^{\mathcal{A}}$. By Lemma 1, $(\mathcal{C}\ t_1)^{\mathcal{B}} = \left[t_1^{\mathcal{A}}\right]_{\mathbb{N}}$ and $\left[t_2^{\mathcal{A}}\right]_{\mathbb{N}} = (\mathcal{C}\ t_2)^{\mathcal{B}}$. Thus we get $(\mathcal{C}\ t_1)^{\mathcal{B}} < (\mathcal{C}\ t_2)^{\mathcal{B}}$, and so $\mathcal{B} \models \mathcal{C}\ \varphi$. The case of $\leq_{\mathrm{u}}^{\mathrm{BV}}$ is shown similarly. Finally, if $\varphi$ has the form $t_1 <_{\mathrm{s}}^{\mathrm{BV}} t_2$ then since $\mathcal{A} \models \varphi$, we have $t_1^{\mathcal{A}} <_{\mathrm{s}}^{\mathrm{BV}} t_2^{\mathcal{A}}$. In turn, by the semantics of $T_{\mathrm{BV}}$ as defined in the SMT-LIB 2 standard, this means that $\left[t_1^{\mathcal{A}}\right]_{\mathbb{Z}} < \left[t_2^{\mathcal{A}}\right]_{\mathbb{Z}}$. By the definition of $\mathsf{uts}$, we get $\mathsf{uts}_k(\left[t_1^{\mathcal{A}}\right]_{\mathbb{N}}) < \mathsf{uts}_k(\left[t_2^{\mathcal{A}}\right]_{\mathbb{N}})$, with $k = \kappa(t_1)$. By Lemma 1 we have:

$\mathsf{uts}_k((\mathcal{C}\ t_1)^{\mathcal{B}}) < \mathsf{uts}_k((\mathcal{C}\ t_2)^{\mathcal{B}})$, which means $\mathcal{B} \models \mathcal{C}\ \varphi$. The case of $\leq_{\mathrm{s}}^{\mathrm{BV}}$ is shown similarly.

Next, we prove that $\mathcal{B} \models \textsc{Lem}^{\leq}(\varphi)$, also by induction on $\varphi$. Similarly to the above, the induction step follows directly from the induction hypothesis and so we focus on the induction base, in which $\varphi$ is atomic, and hence it has the form $t_1 = t_2$, $t_1 \bowtie^{\mathrm{BV}} t_2$, or $t_1 {\bowtie_s}^{\mathrm{BV}} t_2$ for some $\bowtie \in \{<, \leq, >, \geq\}$. By the definition of $\textsc{Lem}^{\leq}$, $\textsc{Lem}^{\leq}(\varphi) = \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2)$. We thus prove that $\mathcal{B} \models \textsc{Lem}^{\leq}(t)$ for any term $t$ of sort $\sigma_{[k]}$ by an inner induction on $t$. If $t$ is a bit-vector variable, $\textsc{Lem}^{\leq}(t) = 0 \leq \chi(t) < 2^k$. By Lemma 1, $\chi(t)^{\mathcal{B}} = \left[t^{\mathcal{A}}\right]_{\mathbb{N}}$, and by the definition of $[\_]_{\mathbb{N}}$, $0 \leq \chi(t)^{\mathcal{B}} < 2^k$. If $t$ is a bit-vector constant, then the condition is trivially satisfied. If $t$ has the form $f^{\mathrm{BV}}(t_1, t_2)$ with $f^{\mathrm{BV}} \in \{\&^{\mathrm{BV}}, |^{\mathrm{BV}}, \oplus^{\mathrm{BV}}\}$, then $\textsc{Lem}^{\leq}(t) = 0 \leq \&_k^{\mathbb{N}}(\mathcal{C}\ t_1, \mathcal{C}\ t_2) < 2^k \wedge \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2)$. By the induction hypothesis, $\mathcal{B} \models \textsc{Lem}^{\leq}(t_1) \wedge \textsc{Lem}^{\leq}(t_2)$. Also, $\mathcal{B} \models 0 \leq \&_k^{\mathbb{N}}(\mathcal{C}\ t_1, \mathcal{C}\ t_2) < 2^k$ by Definition 1, and the fact that it is a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation. For any other form of $t$, this follows immediately from the induction hypothesis. □

## 4.3 $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-Satisfiability: Eager Approach

Now that we have reduced $T_{\mathrm{BV}}$-satisfiability to $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability, we present eager and lazy reductions from the latter to $T_{\mathrm{IAUF}}$-satisfiability. The first approach for determining $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability is an eager reduction to $T_{\mathrm{IAUF}}$-satisfiability. The reduction is defined by the translation $\mathcal{T}_A$, which is parameterized by a mode $A \in \{\mathsf{sum}, \mathsf{bitwise}\}$, as shown in Figure 2.

The translation adds to $\varphi$ a conjunction $\textsc{Lem}_A^{\&}(\varphi)$ of lemmas that reflect the definition of $\&_k^{\mathbb{N}}$ for each relevant $k$. Function $\textsc{Lem}_A^{\&}$, when applied to a term or formula $e$, recursively collects lemmas for subterms of $e$ of the form $\&_k^{\mathbb{N}}(t_1, t_2)$.

The introduced lemma depends on the mode $A$. For $A = \mathsf{sum}$, the lemma represents the usual encoding of integers in binary notation, by summing powers of 2 with coefficients that depend on the bits. Alternatively, for $A = \mathsf{bitwise}$, the translation introduces a lemma that compares each $i$-parity of the $\&_k^{\mathbb{N}}$-term to its expected result, based on the $i$-parities of the two arguments. The lemmas use the term $\mathrm{ITE}_{and}(x, y)$ to encode each bit using the ite operator. This case splitting requires access to the $i$-th bit in the bit-vector representations of $t_1$, $t_2$, and $\&_k^{\mathbb{N}}(t_1, t_2)$. These are abbreviated by $a_i$, $b_i$, and $c_i$ in Figure 2, and are defined using the function $bsel$ from Equation (2).

The main difference between $\mathsf{bitwise}$ and $\mathsf{sum}$ is in the balance between the arithmetic solver and the Boolean solver. While both approaches heavily use mod and div terms, the $\mathsf{bitwise}$ mode only includes comparisons between such terms, thus relying mainly on the SAT solver, as well as the equality solver. In contrast, the $\mathsf{sum}$ mode incorporates them within sums and multiplications by constants, making heavy use of the arithmetic solver.

The following theorem states the correctness of the reduction described in Figure 2 from $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability to $T_{\mathrm{IAUF}}$-satisfiability. It follows from the semantics of $T_{\mathrm{BV}}$ and Definition 1, which induces the same semantics for $\&^{\mathbb{N}}$ as the one induced by the lemmas that are produced in $\textsc{Iand}_A(t_1, t_2)$.

$\underline{\mathcal{T}_A\, \varphi}$:

$\textsc{Lem}^{\&}_{A}(\varphi) \wedge \varphi$

$\underline{\textsc{Lem}^{\&}_{A}(e)}$:

Match $e$:

$$\begin{array}{ll}
x & \rightarrow \top \\
c & \rightarrow \top \\
t_1 = t_2 & \rightarrow \textsc{Lem}^{\&}_{A}(t_1) \wedge \textsc{Lem}^{\&}_{A}(t_2) \\
\diamond(\varphi_1, \ldots, \varphi_n) & \rightarrow \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_{A}(\varphi_i) \\
f(t_1, \ldots, t_n) & \rightarrow \bigwedge_{i=1}^{n} \textsc{Lem}^{\&}_{A}(t_i) \\
\&^{\mathbb{N}}_{k}(t_1, t_2) & \rightarrow \textsc{Iand}_A(t_1, t_2) \wedge \bigwedge_{i \in \{1,2\}} \textsc{Lem}^{\&}_{A}(t_i)
\end{array}$$

$\underline{\textsc{Iand}_{\mathsf{sum}}(t_1, t_2)}$:

$\&^{\mathbb{N}}_{k}(t_1, t_2) = \Sigma_{i=0}^{k-1} 2^i \cdot \mathrm{ITE}_{and}(a_i, b_i)$

$\underline{\textsc{Iand}_{\mathsf{bitwise}}(t_1, t_2)}$:

$\bigwedge_{i=0}^{k-1} c_i = \mathrm{ITE}_{and}(a_i, b_i)$

where:

$$a_i = bsel_i(t_1)$$
$$b_i = bsel_i(t_2)$$
$$c_i = bsel_i(\&^{\mathbb{N}}_{k}(t_1, t_2))$$
$$\mathrm{ITE}_{and}(x, y) = \mathrm{ite}(x = 1 \wedge y = 1,\ 1,\ 0)$$

Fig. 2: Translation $\mathcal{T}_A$ from $T_{\mathrm{IA}(\&^{\mathbb{N}})}$ to $T_{\mathrm{IAUF}}$, parameterized by $A \in \{\mathsf{sum}, \mathsf{bitwise}\}$. $x$ and $c$ range over integer variables and constants, resp.; $\diamond$ ranges over the connectives; $f$ ranges over $\Sigma_{\mathrm{IA}}$-symbols; *bsel* is from Equation (2).

**Theorem 2.** *Let $\varphi$ be a $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-formula. For all $A \in \{\mathsf{sum}, \mathsf{bitwise}\}$, $\varphi$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable iff $\mathcal{T}_A\, \varphi$ is $T_{\mathrm{IAUF}}$-satisfiable.*

*Proof.* Suppose $\varphi$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable and let $\mathcal{A}$ be a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation that satisfies it. Now, $\mathcal{A}$ is also a $T_{\mathrm{IAUF}}$-interpretation, and hence what is left to show is that $\mathcal{A} \models \textsc{Lem}^{\&}_{A}(\varphi)$, which directly follows from Definition 1 and a routine verification of the $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-validity of $\textsc{Lem}^{\&}_{A}(\varphi)$ for $A \in \{\mathsf{sum}, \mathsf{bitwise}\}$.

Now suppose $\mathcal{T}_A\, \varphi$ is $T_{\mathrm{IAUF}}$-satisfiable and let $\mathcal{A}$ be a $T_{\mathrm{IAUF}}$-interpretation that satisfies $\mathcal{T}_A\, \varphi$. We prove that $\varphi$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiable. Let $\mathcal{B}$ be the $\Sigma_{\mathrm{IA}}(\&^{\mathbb{N}})$-interpretation obtained from $\mathcal{A}$ by ignoring the interpretations of $\&^{\mathbb{N}}_{k}$ in $\mathcal{A}$, and redefining them according to Definition 1. Clearly, $\mathcal{B}$ is a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation. To show that it satisfies $\varphi$, it suffices to show that $\&^{\mathbb{N}}_{k}(t_1, t_2)^{\mathcal{A}} = \&^{\mathbb{N}}_{k}(t_1, t_2)^{\mathcal{B}}$

```
EAGER$^A_{BV}(\varphi)$:
    $P_{T_{\mathrm{IAUF}}}(\mathcal{T}_A\ (\mathcal{T}\ \varphi))$

LAZY$^A_{BV}(\varphi)$:

    $\Gamma := \{\ \mathcal{T}\ \varphi\ \}$
    $\Delta := \left\{\ \&^{\mathbb{N}}_k(t_1,t_2) \mid \&^{\mathbb{N}}_k(t_1,t_2) \text{ occurs in } \mathcal{T}\ \varphi\ \right\}$
    $\Lambda := \mathit{Prop}(\Delta) \cup \left\{\ \mathrm{IAND}_A(t_1,t_2) \mid \&^{\mathbb{N}}_k(t_1,t_2) \in \Delta\ \right\}$
    Repeat:
     1. If $P_{T_{\mathrm{IAUF}}}(\bigwedge \Gamma)$ is "unsat", then return "unsat".
     2. Otherwise, let $\mathcal{I} = P_{T_{\mathrm{IAUF}}}(\bigwedge \Gamma)$
        /* check $\mathcal{I}$ against properties of $\&^{\mathbb{N}}_k$ */
        (a) If $\mathcal{I}$ satisfies $\Lambda$, return "sat".
        (b) Otherwise:
              /* refine abstraction $\Gamma$ */
              $\Gamma := \Gamma \cup \{\ \psi \in \Lambda \mid \mathcal{I} \not\models \psi\ \}$
```

$\mathit{Prop}(\Delta) = \left\{\ \mathit{Prop}(t_1,t_2) \mid \&^{\mathbb{N}}_k(t_1,t_2) \in \Delta\ \right\}$

$\mathit{Prop}(t_1,t_2)$:

$$\begin{aligned}
&\&^{\mathbb{N}}_k(t_1,t_2) \le t_1\ \wedge \&^{\mathbb{N}}_k(t_1,t_2) \le t_2\ \wedge && \text{bounds}\\
&(t_1 = t_2 \Rightarrow \&^{\mathbb{N}}_k(t_1,t_2) = t_1)\ \wedge && \text{idempotence}\\
&\&^{\mathbb{N}}_k(t_1,t_2) = \&^{\mathbb{N}}_k(t_2,t_1)\ \wedge && \text{symmetry}\\
&\left.\begin{array}{l}
(t_1 = 0 \Rightarrow \&^{\mathbb{N}}_k(t_1,t_2) = 0)\ \wedge\\
(t_1 = 2^k - 1 \Rightarrow \&^{\mathbb{N}}_k(t_1,t_2) = t_2)\ \wedge\\
(t_2 = 0 \Rightarrow \&^{\mathbb{N}}_k(t_1,t_2) = 0)\ \wedge\\
(t_2 = 2^k - 1 \Rightarrow \&^{\mathbb{N}}_k(t_1,t_2) = t_1)
\end{array}\right\} && \text{special cases}
\end{aligned}$$

Fig. 3: Procedures for $T_{\mathrm{BV}}$-satisfiability. We assume $P_{T_{\mathrm{IAUF}}}$ is a procedure for $T_{\mathrm{IAUF}}$-satisfiability that returns a finite representation of a model for satisfiable formulas.

for any term $\&_k^{\mathbb{N}}(t_1, t_2)$ that occurs in $\varphi$. All other terms that occur in $\varphi$ are interpreted the same as in $\mathcal{A}$, by the way $\mathcal{B}$ was defined. Now suppose $\&_k^{\mathbb{N}}(t_1, t_2)$ occurs in $\varphi$. Suppose for contradiction that $\&_k^{\mathbb{N}}(t_1, t_2)^{\mathcal{A}} \neq \&_k^{\mathbb{N}}(t_1, t_2)^{\mathcal{B}}$. Since $\mathcal{B}$ is a $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation, this means that $\&_k^{\mathbb{N}}(t_1, t_2)^{\mathcal{A}} \neq \left[\left[t_1^{\mathcal{A}}\right]_{\mathbb{BV}}^k \&^{\mathrm{BV}} \left[t_2^{\mathcal{A}}\right]_{\mathbb{BV}}^k\right]_{\mathbb{N}}$. In other words, $\left[\&_k^{\mathbb{N}}(t_1^{\mathcal{A}}, t_2^{\mathcal{A}})\right]_{\mathbb{BV}}^k \neq \left[t_1^{\mathcal{A}}\right]_{\mathbb{BV}}^k \&^{\mathrm{BV}} \left[t_2^{\mathcal{A}}\right]_{\mathbb{BV}}^k$. Hence there is some $0 \leq i < k$ such that $\left[\&_k^{\mathbb{N}}(t_1^{\mathcal{A}}, t_2^{\mathcal{A}})\right]_{\mathbb{BV}}^k[i] \neq (\left[t_1^{\mathcal{A}}\right]_{\mathbb{BV}}^k \&^{\mathrm{BV}} \left[t_2^{\mathcal{A}}\right]_{\mathbb{BV}}^k)[i]$. Now, recall *bsel* from Equation (2), which equals to 0 or 1, according to the $i$-th bit in the bit-vector representation of the input integer. Using the semantics of $\&^{\mathrm{BV}}$ in SMT-LIB 2, we get that $bsel_i(\&_k^{\mathbb{N}}(t_1^{\mathcal{A}}, t_2^{\mathcal{A}})) \neq \mathrm{ite}(bsel_i(t_1^{\mathcal{A}}) = bsel_i(t_2^{\mathcal{A}}),\ 1,\ 0)$. For both modes sum and bitwise, this means $\mathcal{A} \not\models \mathrm{IAND}_A(t_1, t_2)$. For the former, the sums will evaluate differently, while for the latter, a direct disequality will be obtained. This is a contradiction to the assumption that $\mathcal{A} \models \mathcal{T}_A\, \varphi$. $\square$

We use $\mathcal{T}_A$ in the eager procedure $\mathrm{EAGER}_{BV}^A(\varphi)$ of Figure 3, in which the input $\Sigma_{\mathrm{BV}}$-formula $\varphi$ is processed through $\mathcal{T}$ to obtain an equisatisfiable formula $\mathcal{T}\,\varphi$ in $T_{\mathrm{IA}(\&^{\mathbb{N}})}$, and then through $\mathcal{T}_A$ to get an equisatisfiable formula in $T_{\mathrm{IAUF}}$. The result is then handed to a $T_{\mathrm{IAUF}}$-solver $P_{T_{\mathrm{IAUF}}}$ for bounded formulas, which is expected to be a decision procedure for the $T_{\mathrm{IAUF}}$-satisfiability of quantifier-free formulas that also returns (a finite representation of) a $T_{\mathrm{IAUF}}$-model satisfying the input formula whenever that formula is $T_{\mathrm{IAUF}}$-satisfiable. Notice that $\mathcal{T}$ always generates bounded formulas due to $\mathrm{LEM}^{\leq}$, and $\mathcal{T}_A$ preserves boundedness as it does not introduce any new variables or terms of the form $\&_k^{\mathbb{N}}(t_1, t_2)$. This leads to the following correctness result for $\mathrm{EAGER}_{BV}^A$.

**Proposition 2.** $\mathrm{EAGER}_{BV}^A$ *is a decision procedure for the* $T_{\mathrm{BV}}$*-satisfiability of quantifier-free formulas.*

## 4.4 $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-satisfiability: Lazy Approach

We now examine a CEGAR-based approach, which applies the function $\mathrm{LEM}_A^{\&}$ in the $\mathcal{T}_A$ translation in a lazy and incremental way. Our CEGAR-procedure $\mathrm{LAZY}_{BV}^A$ is described in Figure 3. It maintains a set $\Gamma$ of assertions, initially set to the translation of the input $\Sigma_{\mathrm{BV}}$-formula $\varphi$ using $\mathcal{T}$, and a set $\Delta$ of terms of the form $\&_k^{\mathbb{N}}(t_1, t_2)$ in $\mathcal{T}\varphi$. Similarly to the eager approach, we utilize the decision procedure $P_{T_{\mathrm{IAUF}}}$ for $T_{\mathrm{IAUF}}$-satisfiability. If, at any point, $P_{T_{\mathrm{IAUF}}}$ determines that $\Gamma$ is $T_{\mathrm{IAUF}}$-unsatisfiable, $\mathrm{LAZY}_{BV}^A$ returns “unsat”. Otherwise, the model $\mathcal{I}$ of $\Gamma$ returned by $P_{T_{\mathrm{IAUF}}}$ is validated against a set $\Lambda$ of lemmas, instantiated with the terms in $\Delta$. The set $\Lambda$ is a union of two sets of lemmas: $(i)$ a set of basic lemmas $Prop(\Delta)$ that capture basic properties of bitwise *and*: upper bounds, idempotence, symmetry, and values for special inputs; and $(ii)$ lemmas based on $\mathrm{LEM}_A^{\&}$, as defined in Figure 2. Any lemmas falsified by $\mathcal{I}$ make the model unsuitable for $\varphi$. Such lemmas are then added to $\Gamma$, and the process repeats. If all of the lemmas in $\Lambda$ are satisfied, the algorithm returns “sat”.

The correctness argument for $\mathrm{LAZY}_{BV}^A$ is similar to that of Proposition 2. At any point in the procedure, $\Gamma$ consists of $\mathcal{T}\varphi$, as well as a subset of $\Lambda$. It is routine to check that every formula in $\Lambda$ is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-valid. If the procedure returns "unsat", this means that the abstraction $\Gamma$ is not $T_{\mathrm{IAUF}}$-satisfiable, which means that $\mathcal{T}\,\varphi$ itself is $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-unsatisfiable. By Theorem 1, $\varphi$ is $T_{\mathrm{BV}}$-unsatisfiable. In contrast, when the procedure returns "sat", a satisfying $T_{\mathrm{IA}(\&^{\mathbb{N}})}$-interpretation for $\mathcal{T}\,\varphi$ can be constructed according to Definition 1 from the $T_{\mathrm{IAUF}}$-interpretation $\mathcal{I}$, in a similar fashion to the proof of Theorem 2. In turn, this interpretation can be translated to a $T_{\mathrm{BV}}$-interpretation following Theorem 1. Since $\mathcal{T}\,\varphi$ is bounded, we then have the following.

**Proposition 3.** $\textsc{Lazy}^{A}_{BV}$ *is a decision procedure for the* $T_{\mathrm{BV}}$*-satisfiability of quantifier-free formulas.*

*Remark 1.* At this point, it is instructive to compare the translation presented here to that by Niemetz et al. [35]. Although the solutions offered in the two works are similar, they differ on the problem they address. Niemetz et al. study the satisfiability of formulas over bit-vectors with parametric bit-widths, while this paper focuses on the regular SMT-LIB 2 theory of *fixed*-width bit-vectors. Since the translation to integers involves the bit-width of the terms in the input formula, parametric bit-widths require the introduction of quantifiers in the translation in practically all cases. In contrast, by considering only inputs over fixed bit-widths, our approach requires no quantifiers at all. Also, the solving technique we present here has both eager and lazy variants, with two alternative encodings in each. Instead, Niemetz et al. present only eager translations. The most successful translation there mostly resembles our eager sum mode, with some additional quantified axioms that correspond $\mathit{Prop}(t_1, t_2)$ from Figure 3. A counterpart to the bitwise mode was not considered there. Furthermore, their method was only evaluated on benchmarks with a single parametric bit-width due to the limited expressiveness supported by the prototype implementation. In contrast, our technique is fully implemented within the cvc5 solver.

## 5 Experimental Results

### 5.1 Implementation and Experiments

We implemented both $\textsc{Eager}^{A}_{BV}$ and $\textsc{Lazy}^{A}_{BV}$ in the cvc5 SMT solver and evaluated the implementation on three classes of benchmarks.[4] The eager translations are implemented in a preprocessing pass that translates the entire input formula to a formula over the SMT-LIB 2 theory of integers, without any extension. The lazy translations use the same preprocessing pass; however, the translated formulas include the $\&^{\mathbb{N}}_{k}$ operators. The CEGAR loop for $\&^{\mathbb{N}}_{k}$ is implemented as part of the non-linear extension of the arithmetic solver of cvc5.

Note that cvc5 does not have built-in support for pow2. For all $\Sigma_{\mathrm{BV}}$-operators except $\ll^{\mathrm{BV}}$ and $\gg^{\mathrm{BV}}$ this does not matter in practice since the argument to pow2 is a concrete constant. For the shift operators, the argument $t$ to pow2

[4] An artifact that includes the implementation, benchmarks, and results is available at https://doi.org/10.5281/zenodo.5652826.

| | SMT-LIB | | | | ECRW | | | | SC | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *slvd* | *sat* | *uns* | *m* | *slvd* | *sat* | *uns* | *m* | *slvd* | *sat* | *uns* | *m* |
| $eager_b$ | 35031 | 10447 | 24584 | 38 | 41989 | 119 | 41870 | 0 | **24** | 9 | 15 | 0 |
| $eager_s$ | 35035 | 10459 | 24576 | 28 | 41435 | 119 | 41316 | 77 | **24** | 9 | 15 | 0 |
| $lazy_b$ | 35001 | 10383 | 24618 | 23 | **47071** | 119 | 46952 | 0 | **24** | 9 | 15 | 0 |
| $lazy_s$ | 34819 | 10297 | 24522 | 27 | 45350 | 119 | 45231 | 138 | **24** | 9 | 15 | 0 |
| *Bitwuzla* | 41220 | 14233 | 26987 | 19 | 37297 | 265 | 37032 | 11120 | 16 | 8 | 8 | 0 |
| cvc5 | 40543 | 14204 | 26339 | 36 | 33187 | 220 | 32967 | 17535 | - | - | - | - |
| Yices | **41228** | 14280 | 26948 | 11 | 31646 | 255 | 31391 | 15801 | 9 | 3 | 6 | 0 |
| bw-ind | - | - | - | - | 25608 | 0 | 25608 | 0 | - | - | - | - |

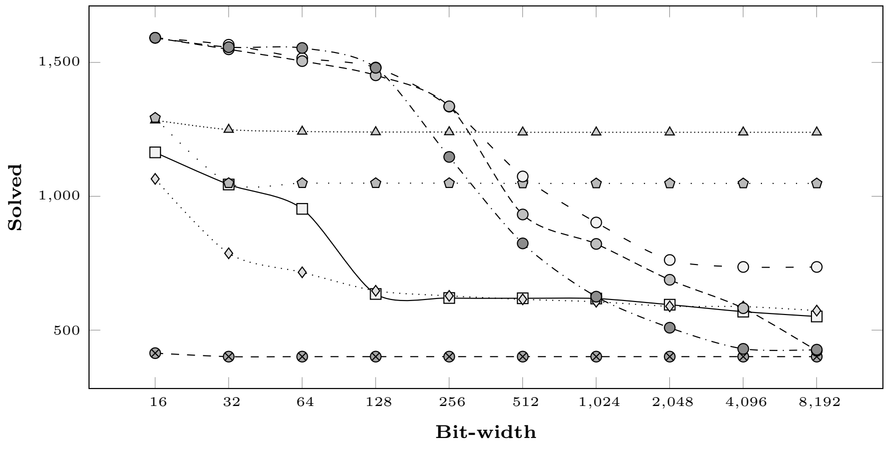
Table 2: Overall results on all three benchmark sets.

may include variables, but the value of pow2($t$) only matters when $0 \leq t < k$, where $k$ is the bit-width of the original $\Sigma_{\mathrm{BV}}$-term. Thus, we are able to eliminate pow2-terms by enumerating a finite set of cases using ite-terms.

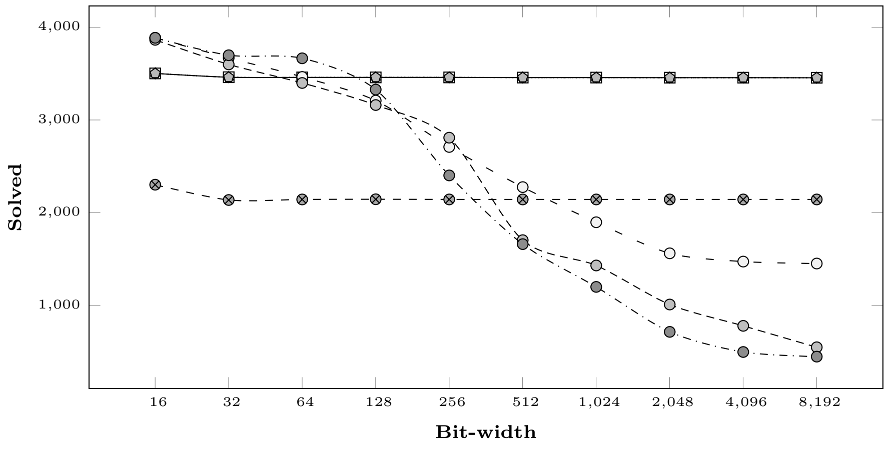
In accordance with Section 4, our implementation focuses on finding and improving strategies for lemma instantiation. Another aspect of integer reasoning is the evaluation of operations over constants, especially when the constants are large, as in our experience, operations on big integers can take up to 30-40% of the overall runtime. In the experiments described below, these are handled by the CLN library [25], which is supported by cvc5. Our focus on lemma instantiation is meant to reduce how often expensive numeric operations must be invoked.

We evaluated our int-blasting approaches $\textsc{Eager}^{A}_{BV}$ and $\textsc{Lazy}^{A}_{BV}$ for $A \in \{\mathsf{sum}, \mathsf{bitwise}\}$ on three sets of benchmarks: (1) the QF_BV benchmarks from SMT-LIB, (2) a set of benchmarks consisting of equivalence checks of bit-vector rewrite rule candidates, and (3) 35 benchmarks originating from a smart contract verification application.[5] We compared our four int-blasting configurations, denoted $eager_s$, $eager_b$, $lazy_s$, $lazy_b$, where $b$ stands for bitwise and $s$ stands for sum, against (1) cvc5 running its eager bit-vector solver using CaDiCaL [10] as the SAT back end, (2) *Bitwuzla* [31] version 0.1-202011 (the QF_BV winner of the 2020 SMT competition), (3) Yices [20] version 2.6.2 with CaDiCaL as the SAT back end (the QF_BV runner-up at the same competition), and (4) bw-ind, the prototype implementation for proving bit-width independent properties used by Niemetz et al. [35], which uses the arithmetic solver of cvc5 as a back-end, the same arithmetic solver used in our int-blasting approaches. We used bw-ind only for the second benchmark set since its support is limited to benchmarks that contain a single bit-width. We performed all experiments on a cluster with Intel Xeon CPU E5-2620 v4 CPUs with 2.1GHz and 128GB memory.

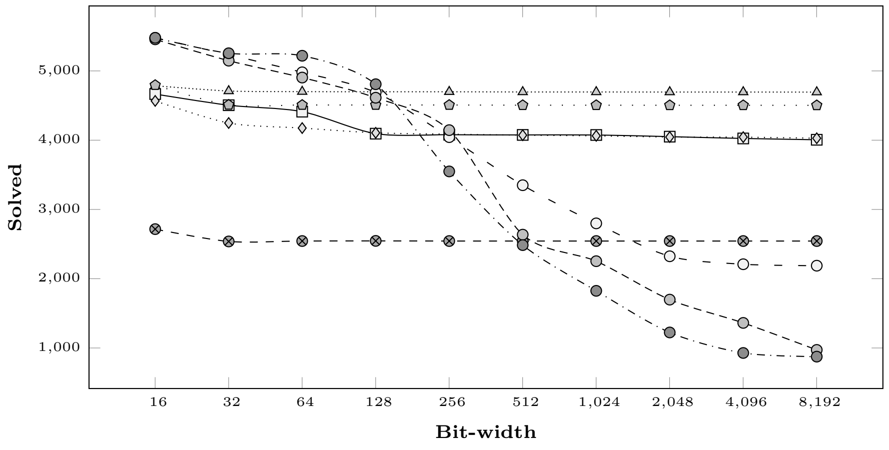
[5] Provided to us by collaborators at Certora.



(a) With bitwise *and* operator.



(b) Without bitwise *and* operator.



(c) All ECRW benchmarks.

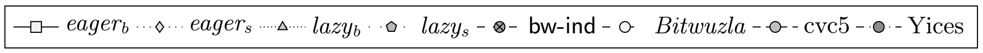


Fig. 4: Number of solved benchmarks grouped by bit-width.

## 5.2 Results

Table 2 summarizes the overall results for all benchmark sets. For each set and running configuration, it shows the total number of solved benchmarks (*slvd*), sat results (*sat*), unsat result (*uns*) and number of memory-outs (*m*).

*QF_BV Benchmarks (SMT-LIB).* The QF_BV benchmark set includes all 41,713 benchmarks from the 2020 SMT-LIB release. We used a limit of 600s of CPU time and a memory limit of 8GB for each solver/benchmark pair. None of the int-blasting configurations is competitive with the other bit-blasting solvers. This is as expected since the QF_BV benchmark set contains few benchmarks with bit-widths larger than 64, the target of our approach. The *pspace* family of QF_BV benchmarks consists of benchmarks with bit-widths ranging from 5,000 to 30,000. The more challenging benchmarks in this set, however, contain the bit-wise *and* operator, and our int-blasting approach cannot solve them within the time limit. All four int-blasting approaches are more competitive on unsatisfiable benchmarks than satisfiable ones. This is because int-blasting relies heavily on the performance of cvc5's procedure for non-linear integer arithmetic. This procedure is based on instantiating a set of lemma schemas [16, 41], which may show unsatisfiability quickly when useful lemmas are discovered, but may take longer to converge when the problem is satisfiable. Overall, each of our int-blasting configurations is able to solve 18 benchmarks that none of the bit-blasting approaches is able to solve; 14 of these are from the arithmetic-heavy *Sage2* family, which includes a wide range of both arithmetic and bitwise operators, including shifts and bitwise *and*, *or*, and *xor*.

*Equivalence Checks of Rewrite Rule Candidates (ECRW).* The ECRW benchmark set consists of equivalence checks of rewrite rule candidates for $T_{\mathrm{BV}}$-terms and formulas. They were automatically generated using a state-of-the-art Syntax-Guided Synthesis (SyGuS) [2] solver implemented in cvc5 [40]. We enumerated pairs of $\Sigma_{\mathrm{BV}}$-terms that are equivalent for bit-vectors of bit-width 4. These pairs of terms were generated over a sub-signature of $\Sigma_{\mathrm{BV}}$ consisting of the constants 0 and 1, the $=$ operator, and the unsigned comparison operators $<_{\mathrm{u}}^{\mathrm{BV}}$ and $\leq_{\mathrm{u}}^{\mathrm{BV}}$, as well as the operators $-^{\mathrm{BV}}$, $\sim^{\mathrm{BV}}$, $+^{\mathrm{BV}}$, $\cdot^{\mathrm{BV}}$, $\mathrm{div}^{\mathrm{BV}}$, $\&^{\mathrm{BV}}$, and $\mathrm{mod}^{\mathrm{BV}}$. In total, we generated 5,491 distinct equivalence checks with bit-width 4. Each equivalence check was then instantiated with bit-widths 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, and 8192, resulting in a total of 54,910 benchmarks. An important feature of the generated checks is that they exclude equivalences that are already derivable solely by the rewriter of cvc5. We used a CPU time limit of 300s and a memory limit of 8GB per solver/benchmark pair. For this benchmark set, our evaluation included bw-ind, whose primary purpose is to prove bit-width independent properties via bit-vectors of parametric widths. Since this benchmark set consists of fixed-width bit-vectors and not parametric ones, we added a constraint that specifies the concrete bit-width of each benchmark, by comparing it to the parametric bit-width. It is evident that bw-ind does not perform well on this benchmark set. This is expected given that this approach is the only one that makes any use of quantifiers.

On this benchmark set, all int-blasting approaches outperformed all other approaches. Figures 4a to 4c provide a more fine-grained analysis for this set by depicting the number of solved benchmarks grouped by bit-width for each solver on (a) benchmarks with applications of the bitwise *and* operator $\&^{\mathrm{BV}}$ (29%), (b) benchmarks without $\&^{\mathrm{BV}}$ (71%), and (c) the full ECRW benchmark set. The bit-blasting approaches are marked with circles, while the int-blasting approaches are marked with other shapes. For each subset of benchmarks there is a bit-width $k$ for which the best int-blasting configuration, the lazy bitwise mode, outperforms all other configurations and solvers: 512 for those benchmarks with $\&^{\mathrm{BV}}$, 128 for those without, and 256 for the full set. This shows that int-blasting can be a useful tool to add to the tool-box of bit-precise reasoning engines, in the presence of large bit-widths. Surprisingly, even for bit-width 16, there were benchmarks for which int-blasting performed better than bit-blasting. For example, there are 78 benchmarks of bit-width 16, without the $\&^{\mathrm{BV}}$ operator that were solved by the int-blasting approaches in less than 1 second, while all the bit-blasting approaches required more than 10 seconds (in many of these cases, bit-blasting required more than 100 seconds).

Comparing the different int-blasting configurations, Figure 4b clearly shows that for benchmarks without $\&^{\mathrm{BV}}$ applications, the lazy and eager int-blasting configurations are almost bit-width independent, and perform equally well (in turn, their markings overlap in the figure). This is expected because the translations differ from one another only in the way they handle $\&^{\mathrm{BV}}$. Moreover, the $\&^{\mathrm{BV}}$-free part of our translations is actually bit-width independent, as the size of the generated terms does not depend on it, except for shift operators, which are not included in this benchmark set. The differences between the translations are visible, also as expected, for benchmarks with $\&^{\mathrm{BV}}$ applications, as shown in Figure 4a. There, the best int-blasting configuration is $lazy_b$. In the presence of bitwise operators, both the eager and lazy translations introduce terms whose size does depend on the bit-width. Accordingly, we see a clear decrease in the performance of the eager translations as the bit-width increases, while little performance degradation is observable for the lazy translations. This can be explained by the fact that the eager approach introduces bit comparison lemmas or sum-based lemmas before the integer solver comes into play. In contrast, the lazy approach introduces those lemmas only if the model generated in the CEGAR loop falsifies them, so there are generally fewer terms whose size depends on the bit-width.

As for the better performance of bitwise compared to sum, we conjecture that the bitwise translation outperforms the sum translation because it is a more direct translation to SAT. The sum translation relies on the linear arithmetic solver generating simple conflicts and lemmas over linear arithmetic literals that correspond to the same reasoning in a more indirect way. While this choice is not obvious, our experiments have confirmed that the former is superior.

*Smart Contract Verification Benchmarks (SC).* This benchmark set consists of 35 benchmarks from a smart contract verification application. They contain (linear and non-linear) arithmetic operators, bitwise operators, as well as uninterpreted functions, and reason about bit-vectors of width 256. These benchmarks originate from verification conditions that are directly produced by Certora's verification tool for Ethereum smart contracts [15]. They encode algebraic properties of low-level methods in smart contracts (e.g., commutativity of balance updates). The application requires the generation of models, which the eager bit-blasting configuration of cvc5 does not support for uninterpreted functions. We imposed a CPU time limit of 3,600s and a memory limit of 32GB per solver/benchmark pair.

The int-blasting configurations are able to solve 24 benchmarks, whereas the bit-blasting solvers solve less (*Bitwuzla* solves 16 and Yices solves 9). In addition to solving more benchmarks in this benchmark set, the int-blasting approaches are also faster: The 24 benchmarks that are solved by int-blasting take a total of 232 seconds, to be solved, where 22 out of these benchmarks are solved in a total time of 20 seconds. This is the case for all int-blasting configurations. In contrast, *Bitwuzla* solves 16 benchmarks in 5,900 seconds, and Yices solves 9 benchmarks in 3,900 seconds. Notice that unsatisfiable benchmarks seem to be better suited for int-blasting, while satisfiable benchmarks are solved better with bit-blasting. This positions int-blasting as a useful complement to bit-blasting.

## 6 Related Work, Conclusion, and Future Work

*Related Work.* Earlier integer-based techniques for bit-precise reasoning focus on translating hardware register transfer level (RTL) constraints into integer linear programming (ILP) and are thus limited to the linear arithmetic subset of the theory of bit-vectors [13, 48]. Similarly, Achterberg's PhD thesis [1] studies translations of bit-vector constraints over linear arithmetic to integers in the context of constraint programming, while bit-blasting non-linear and bitwise operators. Kafle et al. [27] present an approach based on Benders Decomposition [9] for solving modular arithmetic problems after translating them to linear integer arithmetic (LIA). Another approach to solving modular arithmetic problems that originates from software verification was studied by Vizel et al. [45], using a model checking approach. The MathSAT5 solver [19] applies a layered approach for computing Craig's interpolants for the theory of bit-vectors by first converting the problem into an overapproximated LIA problem [24]. When that approach is unsuccessful, MathSAT5 automatically falls back to finding a propositional interpolant via bit-blasting. Earlier versions of MathSAT also utilized this approach for solving bit-vector problems [12]. A similar but more sophisticated approach [3, 4] is implemented in the Princess theorem prover [42]. Another recent LIA-based interpolation method is presented in [37]. Although similar in spirit to that of MathSAT5 [24], it is often able to recover the word-level structure from the propositional interpolant.

In contrast to [3, 13, 27, 48], we focus on general bit-vector problems, and unlike [3, 12, 13, 24, 27, 48], we translate bit-vector problems into an extension of non-linear integer arithmetic. As a result, our approach can handle all operators of the theory of bit-vectors. We present several variants of our technique, including a CEGAR-based one similar in spirit to the lazy approaches discussed above.

Alternative approaches to bit-blasting based on bit-vector reasoning and the so-called *model constructing satisfiability calculus* (mcSAT) [30] have shown promising results [23, 47]. Other orthogonal bit-vector-based alternatives include local search techniques which, while refutationally incomplete, are particularly effective in combination with bit-blasting [22, 32–34]. We reduce the amount of bit-blasting by converting bit-vector formulas to non-linear integer arithmetic formulas and relying on a DPLL(T)-based SMT approach [8] to solve them.

Our translation of bit-vector formulas to integer formulas is similar to the one for solving formulas with bit-vectors of *parametric bit-width* we proposed in previous work [35]. However, in this case, the bit-width is not parametric but fixed, which eliminates the need for the translation to introduce quantifiers. A more detailed comparison with that work is provided in Remark 1.

We implemented an earlier prototype of this approach in lazybv2int [49] that used our SMT solver cvc5 as a black box, via the solver-agnostic API of Smt-switch [29]. Initial evaluation led us to the conclusion that it is preferable to implement int-blasting inside cvc5, thus utilizing its efficient mechanisms such as handling of terms and rewriting.

*Conclusion.* We studied eager and lazy translations from bit-vector formulas to an extension of integer arithmetic, and implemented them in the SMT solver cvc5. The translations reduce arithmetic bit-vector operators as defined in the SMT-LIB 2 standard, and differ in the way they handle bitwise operators. For those, we examined sum-based and bit-based approaches. The experiments we conducted on equivalence checks for rewrite rule candidates show promising results for formulas that involve multiplications and divisions of large bit-vectors. For SMT-LIB benchmarks, our approach is less effective than state-of-the-art approaches largely based on bit-blasting, though not in all cases. Finally, the smart contracts benchmarks show that our approach provides a complement to bit-blasting, especially for unsatisfiable formulas.

*Future Work.* We believe that alternative approaches for bit-precise reasoning, including mcSAT, local search, and integer-based approaches, can be further developed and improved to the point where they can become a true complement to bit-blasting in applications where bit-blasting struggles to scale up. We plan to continue this line of research by studying integer-based abstractions of other bit-vector operators, in particular, the shift operators. Interestingly, our translations also generate challenging benchmarks for non-linear integer arithmetic solvers. We plan to use these benchmarks to improve non-linear integer reasoning, specifically in the presence of division and modulo operations. For that, we target a submission of such benchmarks to the SMT-LIB library.

## References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, Berlin Institute of Technology (2007)
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., anjit A. Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8 (2013)
3. Backeman, P., Rümmer, P., Zeljic, A.: Bit-vector interpolation and quantifier elimination by lazy reduction. In: FMCAD. pp. 1–10. IEEE (2018)
4. Backeman, P., Rümmer, P., Zeljić, A.: Interpolating bit-vector formulas using uninterpreted predicates and presburger arithmetic. Formal Methods in System Design pp. 1–36 (2021)
5. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 171–177. CAV'11, Springer-Verlag (2011), http://dl.acm.org/citation.cfm?id=2032305.2032319
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2020)
7. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
9. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numer. Math. **4**(1), 238252 (Dec 1962). https://doi.org/10.1007/BF01386316, https://doi.org/10.1007/BF01386316
10. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
11. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
12. 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)
13. Brinkmann, R., Drechsler, R.: Rtl-datapath verification using integer linear programming. In: VLSI Design. pp. 741–746. IEEE Computer Society (2002)
14. Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT - A CDCL(LA) solver. In: CADE. Lecture Notes in Computer Science, vol. 11716, pp. 111–122. Springer (2019)
15. Buterin, V.: Ethereum whitepaper, https://ethereum.org/en/whitepaper/
16. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. pp. 58–75 (2017). https://doi.org/10.1007/978-3-662-54577-5_4, https://doi.org/10.1007/978-3-662-54577-5_4

17. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: SAT. Lecture Notes in Computer Science, vol. 10929, pp. 383–398. Springer (2018)
18. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. ACM Trans. Comput. Log. **19**(3), 19:1–19:52 (2018)
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013)
20. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) 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. Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
21. Enderton, H., Enderton, H.B.: A mathematical introduction to logic. Elsevier (2001)
22. Fröhlich, A., Biere, A., Wintersteiger, C.M., Hamadi, Y.: Stochastic local search for satisfiability modulo theories. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA. pp. 1136–1143. AAAI Press (2015)
23. Graham-Lengrand, S., Jovanovic, D., Dutertre, B.: Solving bitvectors with MCSAT: explanations from bits and pieces. In: IJCAR (1). Lecture Notes in Computer Science, vol. 12166, pp. 103–121. Springer (2020)
24. Griggio, A.: Effective word-level interpolation for software verification. In: FMCAD. pp. 28–36. FMCAD Inc. (2011)
25. Haible, B., Kreckel, R.: CLN, a class library for numbers (1996), http://www.ginac.de/CLN
26. Jovanovic, D.: Solving nonlinear integer arithmetic with MCSAT. In: VMCAI. Lecture Notes in Computer Science, vol. 10145, pp. 330–346. Springer (2017)
27. Kafle, B., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: A benders decomposition approach to deciding modular linear integer arithmetic. In: SAT. Lecture Notes in Computer Science, vol. 10491, pp. 380–397. Springer (2017)
28. Kroening, D., Groce, A., Clarke, E.M.: Counterexample guided abstraction refinement via program execution. In: ICFEM. Lecture Notes in Computer Science, vol. 3308, pp. 224–238. Springer (2004)
29. Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovick, C., Guman, A., Tinelli, C., Barrett, C.W.: Smt-Switch: A solver-agnostic C++ API for SMT solving. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 377–386. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_26, https://doi.org/10.1007/978-3-030-80223-3_26
30. de Moura, L.M., Jovanovic, D.: A model-constructing satisfiability calculus. In: VMCAI. Lecture Notes in Computer Science, vol. 7737, pp. 1–12. Springer (2013)
31. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621
32. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: FMCAD. pp. 214–224. IEEE (2020)
33. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. Formal Methods Syst. Des. **51**(3), 608–636 (2017)

34. Niemetz, A., Preiner, M., Biere, A., Fröhlich, A.: Improving local search for bit-vector logics in SMT with path propagation. In: Proceedings of the Fourth International Workshop on Design and Implementation of Formal Tools and Systems, Austin, TX, USA, September 26-27, 2015. pp. 1–10 (2015)
35. Niemetz, A., Preiner, M., Reynolds, A., Zohar, Y., Barrett, C.W., Tinelli, C.: Towards bit-width-independent proofs in SMT solvers. In: CADE. Lecture Notes in Computer Science, vol. 11716, pp. 366–384. Springer (2019)
36. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C.W., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: SAT. Lecture Notes in Computer Science, vol. 11628, pp. 279–297. Springer (2019)
37. Okudono, T., King, A.: Mind the gap: Bit-vector interpolation recast over linear integer arithmetic. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 79–96. Springer (2020)
38. Ranise, S., Tinelli, C., Barrett, C.: Definition of the logic QF_BV in the SMT-LIB standard, http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV
39. Ranise, S., Tinelli, C., Barrett, C.: Definition of the theory FixedSizeBitVectors in the SMT-LIB standard, http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml
40. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: CAV (2). Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019)
41. Reynolds, A., Tinelli, C., Jovanovic, D., Barrett, C.: Designing theory solvers with extensions. In: Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília, Brazil, September 27-29, 2017, Proceedings. pp. 22–40 (2017). https://doi.org/10.1007/978-3-319-66167-4_2, https://doi.org/10.1007/978-3-319-66167-4_2
42. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS, vol. 5330, pp. 274–289. Springer (2008)
43. Tinelli, C.: Definition of the theory Int in the SMT-LIB standard, http://smtlib.cs.uiowa.edu/theories-Ints.shtml
44. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) Logics in Artificial Intelligence. pp. 641–653. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
45. Vizel, Y., Nadel, A., Malik, S.: Solving linear arithmetic with sat-based model checking. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 47–54 (2017). https://doi.org/10.23919/FMCAD.2017.8102240
46. Warren, H.S.: Hacker's delight. Pearson Education (2013)
47. Zeljic, A., Wintersteiger, C.M., Rümmer, P.: Deciding bit-vector formulas with mcsat. In: SAT. Lecture Notes in Computer Science, vol. 9710, pp. 249–266. Springer (2016)
48. Zeng, Z., Kalla, P., Ciesielski, M.J.: LPSAT: a unified approach to RTL satisfiability. In: DATE. pp. 398–402. IEEE Computer Society (2001)
49. Zohar, Y., Irfan, A., Mann, M., Notzli, A., Reynolds, A., Barrett, C.: lazybv2int at the SMT competition 2020 (2020), https://arxiv.org/abs/2105.09743