



# Bitwuzla <sup>★</sup>



Aina Niemetz  and Mathias Preiner 

Stanford University, Stanford, USA  
{niemetz, preiner}@cs.stanford.edu



**Abstract.** Bitwuzla is a new SMT solver for the quantifier-free and quantified theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions. This paper serves as a comprehensive system description of its architecture and components. We evaluate Bitwuzla’s performance on all benchmarks of supported logics in SMT-LIB and provide a comparison against other state-of-the-art SMT solvers.

## 1 Introduction

Satisfiability Modulo Theories (SMT) solvers serve as back-end reasoning engines for a wide range of applications in formal methods (e.g., [13, 14, 21, 23, 35]). In particular, the theory of fixed-size bit-vectors, in combination with arrays, uninterpreted functions and floating-point arithmetic, have received increasing interest in recent years, as witnessed by the high and increasing numbers of benchmarks submitted to the SMT-LIB benchmark library [5] and the number of participants in corresponding divisions in the annual SMT competition (SMT-COMP) [42]. State-of-the-art SMT solvers supporting (a subset of) these theories include Boolector [31], cvc5 [3], MathSAT [15], STP [19], Yices2 [17] and Z3 [25]. Among these, Boolector had been largely dominating the quantifier-free divisions with bit-vectors and arrays in SMT-COMP over the years [2].

Boolector was originally published in 2009 by Brummayer and Biere [11] as an SMT solver for the quantifier-free theories of fixed-size bit-vectors and arrays. Since 2012, Boolector has been mainly developed and maintained by the authors of this paper, who have extended it with support for uninterpreted functions and lazy handling of non-recursive lambda terms [32, 38, 39], local search strategies for quantifier-free bit-vectors [33, 34], and quantified bit-vector formulas [40].

While Boolector is still competitive in terms of performance, it has several limitations. Its code base consists of largely monolithic C code, with a rigid architecture focused on a very specialized, tight integration of bit-vectors and arrays. Consequently, it is cumbersome to maintain, and adding new features is difficult and time intensive. Further, Boolector requires manual management of memory and reference counts from API users; terms and sorts are tied to a specific solver instance and cannot be shared across instances; all preprocessing

---

\* This work was supported in part by the Stanford Center for Automated Reasoning, the Stanford Agile Hardware Center, the Stanford Center for Blockchain Research and a gift from Amazon Web Services.

techniques are destructive, which disallows incremental preprocessing; and due to architectural limitations, incremental solving with quantifiers is not supported.

In 2018, we forked Boolector in preparation for addressing these issues, and entered an improved and extended version of this fork as Bitwuzla in the SMT competition 2020 [26]. At that time, Bitwuzla extended Boolector with: support for floating-point arithmetic by integrating SymFPU [8] (a C++ library of bit-vector encodings of floating-point operations); a novel generalization of its propagation-based local search strategy [33] to ternary values [27]; unsat core extraction; and since 2022, support for reasoning about quantified formulas for all supported theories and their combinations. This version of Bitwuzla was already made available on GitHub at [28], but not officially released. However, architectural and structural limitations inherited from Boolector remained. Thus, to overcome these limitations and address the above issues, we decided to discard the existing code base and rewrite Bitwuzla from scratch.

In this paper, we present the first official release of Bitwuzla, an SMT solver for the (quantified and quantifier-free) theories of fixed-size bit-vectors, arrays, floating-point arithmetic, uninterpreted functions and their combinations. Its name (pronounced as *bitvootslah*) is derived from an Austrian dialect expression that can be translated as *someone who tinkers with bits*. Bitwuzla is written in C++, inspired by techniques implemented in Boolector. That is, rather than only redesigning problematic aspects of Boolector, we carefully dissected and (re)evaluated its parts to serve as guidance when writing a new solver from scratch. In that sense, it is not a reimplementaion of Boolector, but can be considered its superior successor. Bitwuzla is available on GitHub [28] under the MIT license, and its documentation is available at [29].

## 2 Architecture

Bitwuzla supports reasoning about quantifier-free and quantified formulas over fixed-size bit-vectors, floating-point arithmetic, arrays and uninterpreted functions as standardized in SMT-LIB [4]. In this section, we provide an overview of Bitwuzla’s system architecture and its core components as given in Figure 1.

Bitwuzla consists of two main components: the *Solving Context* and the *Node Manager*. The Solving Context can be seen as a solver instance that determines satisfiability of a set of formulas and implements the lazy, abstraction/refinement-based SMT paradigm *lemmas on demand* [6, 24] (in contrast to SMT solvers like *cvc5* and *Z3*, which are based on the CDCL( $\mathcal{T}$ ) [36] framework). The Node Manager is responsible for constructing and maintaining nodes and types and is shared across multiple Solving Context instances.

Bitwuzla provides a comprehensive C++ API as its main interface, with a C and Python API built on top. All features of the C++ API are also accessible to C and Python users. The API documentation is available at [29]. The C++ API exports *Term*, *Sort*, *Bitwuzla*, and *Option* classes for constructing nodes and types, configuring solver options and constructing Bitwuzla solver instances (the external representation of Solving Contexts). *Term* and *Sort* objects may be

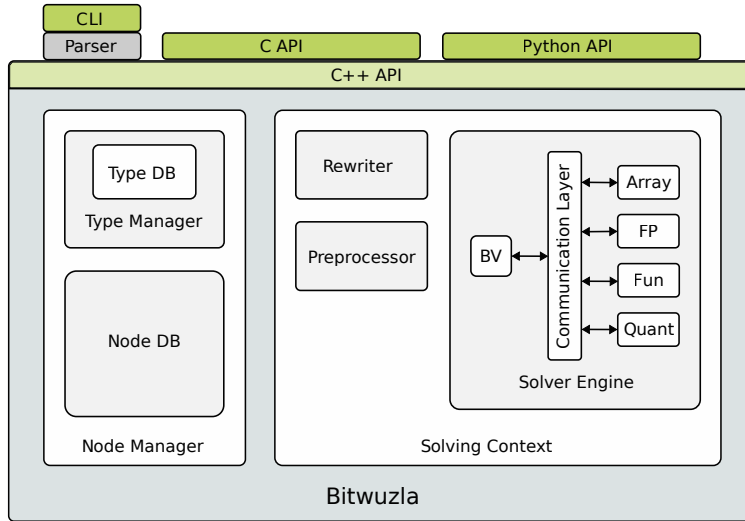


Fig. 1. Bitwuzla system architecture.

used in multiple Bitwuzla instances. The parser interacts with the solver instance via the C++ API. A textual command line interface (CLI) builds on top of the parser, supporting SMT-LIBv2 [4] and BTOR2 [35] as input languages.

## 2.1 Node Manager

Bitwuzla represents formulas and terms as reference-counted, immutable nodes in a directed acyclic graph. The Node Manager is responsible for constructing and managing these nodes and employs hash-consing to maximize sharing of subgraphs. Automatic reference counting allows the Node Manager to determine when to delete nodes. Similarly, types are constructed and managed by the *Type Manager*, which is maintained by the Node Manager. Nodes and types are stored globally (thread-local) in the Node Database and Type Database, which has the key advantage that they can be shared between arbitrarily many solving contexts within one thread. This is one of the key differences to Boolector’s architecture, where terms and types are manually reference counted and tied to a single solver instance, which does not allow sharing between solver instances.

## 2.2 Solving Context

A *Solving Context* is the internal equivalent of a solver instance and determines the satisfiability of a set of asserted formulas (assertions). Solving Contexts are fully configurable via options and provide an incremental interface for adding and removing assertions via push and pop. Incremental solving allows users to perform multiple satisfiability checks with similar sets of assertions

while reusing work from earlier checks. On the API level, Bitwuzla also supports satisfiability queries under a given set of assumptions (SMT-LIB command `check-sat-assuming`), which are internally handled via push and pop.

Nodes and types constructed via the Node Manager may be shared between multiple Solving Contexts. If the set of assertions is satisfiable, the Solving Context provides a model for the input formula. It further allows to query model values for any term, based on this model (SMT-LIB command `get-value`). In case of unsatisfiable queries, the Solving Context can be configured to extract an unsatisfiable core and unsat assumptions.

A Solving Context consists of three main components: a *Rewriter*, a *Preprocessor* and a *Solver Engine*. The Rewriter and Preprocessor perform local (node level) and global (over all assertions) simplifications, whereas the Solver Engine is the central solving engine, managing theory solvers and their interaction.

**Preprocessor.** As a first step of each satisfiability check, prior to solving, the preprocessor applies a pipeline of preprocessing passes in a predefined order to the current set of assertions until fixed-point. Each preprocessing pass implements a set of satisfiability-preserving transformations. All passes can be optionally disabled except for one mandatory transformation, the reduction of the full set of operators supported on the API level to a reduced operator set: Boolean connectives are expressed by means of  $\{\neg, \wedge\}$ , quantifier  $\exists$  is represented in terms of  $\forall$ , inequalities are represented in terms of  $<$  and  $>$ , signed bit-vector operators are expressed in terms of unsigned operators, and more. These reduction transformations are a subset of the term rewrites performed by the Rewriter, and rewriting is implemented as one preprocessing pass. Additionally, Bitwuzla implements 7 preprocessing passes, which are applied sequentially, after rewriting, until no further transformations are possible: *and flattening*, which splits a top-level  $\wedge$  into its subformulas, e.g.,  $a \wedge (b \wedge (c = d))$  into  $\{a, b, c = d\}$ ; *substitution*, which replaces all occurrences of a constant  $x$  with a term  $t$  if  $x = t$  is derived on the top level; *skeleton preprocessing*, which simplifies the Boolean skeleton of the input formula with a SAT solver; *embedded constraints*, which substitutes all occurrences of top-level constraints in subterms of other top-level constraints with *true*; *extract elimination*, which eliminates bit-vector extracts over constants; *lambda elimination*, which applies beta reduction on lambda terms; and *normalization* of arithmetic expressions.

Preprocessing in Bitwuzla is *fully incremental*: all passes are applied to the current set of assertions, from all assertion levels, and simplifications derived from lower levels are applied to all assertions of higher levels (including assumptions). Assertions are processed per assertion level  $i$ , starting from  $i = 0$ , and for each level  $i > 0$ , simplifications are applied based on information from all levels  $j \leq i$ . Note that when solving under assumptions, Bitwuzla internally pushes an assertion level and handles these assumptions as assertions of that level. When a level  $i$  is popped, the assertions of that level are popped, and the state of the preprocessor is backtracked to the state that was associated with level  $i - 1$ . Note that preprocessing assertion levels  $i < j$  with information derived from level  $j$  requires to not only restore the state of the preprocessor, but to also reconstruct

the assertions on levels  $i < j$  when level  $j$  is popped to the state before level  $j$  was pushed, and is left to future work.

Boolector, on the other hand, only performs preprocessing based on top-level assertions (assertion level 0) and does not incorporate any information from assumptions or higher assertion levels.

**Rewriter.** The rewriter transforms terms via a predefined set of rewrite rules into semantically equivalent normal forms. This transformation is local in the sense that it is independent from the current set of assertions. We distinguish between required and optional rewrite rules, and further group rules into so-called rewrite levels from 0–2. The set of required rules consists of operator elimination rewrites, which are considered level 0 rewrites and ensure that nodes only contain operators from a reduced base set. For example, the two’s complement  $-x$  of a bit-vector term  $x$  is rewritten to  $(\sim x + 1)$  by means of one’s complement and bit-vector addition. Optional rewrite rules are grouped into level 1 and level 2. Level 1 rules perform rewrites that only consider the immediate children of a node, whereas level 2 rules may consider multiple levels of children. If not implemented carefully, level 2 rewrites can potentially destroy sharing of subterms and consequently increase the overall size of the formula. For example, rewriting  $(t + 0)$  to  $t$  is considered a level 1 rewrite rule, whereas rewriting  $(a - b = c)$  to  $(b + c = a)$  is considered a level 2 rule since it may introduce an additional bit-vector addition  $(b + c)$  if  $(a - b)$  occurs somewhere else in the formula. The maximum rewrite level of the rewriter can be configured by the user.

Rewriting is applied on the current set of assertions as a preprocessing pass and, as all other passes, applied until fixed-point. That is, on any given term, the rewriter applies rewrite rules until no further rewrite rules can be applied. For this, the rewriter must guarantee that no set of applied rewrite rules may lead to cyclic rewriting of terms. Additionally, all components of the solving context apply rewriting on freshly created nodes to ensure that all nodes are always fully normalized. In order to avoid processing nodes more than once, the rewriter maintains a cache that maps nodes to their fully rewritten form.

**Solver Engine.** After preprocessing, the solving context sends the current set of assertions to the Solver Engine, which implements a lazy SMT paradigm called *lemmas on demand* [6, 24]. However, rather than using a propositional abstraction of the input formula as in [6, 24], it implements a bit-vector abstraction similar to Boolector [12, 38]. At its core, the Solver Engine maintains a bit-vector theory solver and a solver for each supported theory. Quantifier reasoning is handled by a dedicated quantifiers module, implemented as a theory solver. The Solver Engine manages all theory solvers, the distribution of relevant terms, and the processing of lemmas generated by the theory solvers.

The bit-vector solver is responsible for reasoning about the bit-vector abstraction of the input assertions and lemmas generated during solving, which includes all propositional and bit-vector terms. Theory atoms that do not belong to the bit-vector theory are abstracted as Boolean constants, and bit-vector terms whose operator does not belong to the bit-vector theory are abstracted as

bit-vector constants. For example, an array select operation of type bit-vector is abstracted as a bit-vector constant, while an equality between two arrays is abstracted as a Boolean constant.

If the bit-vector abstraction is satisfiable, the bit-vector solver produces a satisfying assignment, and the floating-point, array, function and quantifier solvers check this assignment for theory consistency. If a solver finds a theory inconsistency, i.e., a conflict between the current satisfying assignment and the solver’s theory axioms, it produces a lemma to refine the bit-vector abstraction and rule out the detected inconsistency. Theory solvers are allowed to send any number of lemmas, with the only requirement that if a theory solver does not send a lemma, the current satisfying assignment is consistent with the theory.

Finding a satisfying assignment for the bit-vector abstraction and the subsequent theory consistency checks are implemented as an abstraction/refinement loop as given in Algorithm 1. Whenever a theory solver sends lemmas, the loop is restarted to get a new satisfying assignment for the refined bit-vector abstraction. The loop terminates if the bit-vector abstraction is unsatisfiable, or if the bit-vector abstraction is satisfiable and none of the theory solvers report any theory inconsistencies. Note that the abstraction/refinement algorithm may return *unknown* if the input assertions include quantified formulas.

---

**Algorithm 1** Abstraction/refinement loop in Solver Engine. Function  $SOLVE(\mathcal{A})$  is called on the current set of preprocessed assertions  $\mathcal{A}$ , which is iteratively refined with a set of Lemmas  $\mathcal{L}$ .

---

```

function SOLVE( $\mathcal{A}$ )
   $r \leftarrow UNKNOWN$ ,  $\mathcal{L} \leftarrow \emptyset$ 
  repeat
     $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{L}$ 
     $r, \mathcal{M} \leftarrow T_{BV}::SOLVE(\mathcal{A})$  ▷ Solve bit-vector abstraction of  $\mathcal{A}$ 
    if  $r = UNSAT$  then break end if
     $\mathcal{L} \leftarrow T_{FP}::CHECK(\mathcal{M})$  ▷ Check FP theory consistency of  $\mathcal{M}$ 
    if  $\mathcal{L} \neq \emptyset$  then continue end if
     $\mathcal{L} \leftarrow T_A::CHECK(\mathcal{M})$  ▷ Check array theory consistency of  $\mathcal{M}$ 
    if  $\mathcal{L} \neq \emptyset$  then continue end if
     $\mathcal{L} \leftarrow T_{UF}::CHECK(\mathcal{M})$  ▷ Check UF theory consistency of  $\mathcal{M}$ 
    if  $\mathcal{L} \neq \emptyset$  then continue end if
     $\mathcal{L} \leftarrow T_Q::CHECK(\mathcal{M})$  ▷ Check quantified formulas in  $\mathcal{M}$ 
  until  $\mathcal{L} = \emptyset$ 
  return  $r$ 
end function

```

---

**Backtrackable Data Structures.** Every component of the Solver Context except for the Rewriter depends on the current set of assertions. When solving incrementally, the assertion stack is modified by adding (SMT-LIB com-

mand `push`) and removing (SMT-LIB command `pop`) assertions. In contrast to Boolector, Bitwuzla supports saving and restoring the internal solver state, i.e., the state of the Solving Context, corresponding to these push and pop operations by means of *backtrackable data structures*. These data structures are custom variants of mutable data structures provided in the C++ standard library, extended with an interface to save and restore their state on push and pop calls. This allows the solver to take full advantage of incremental solving by reusing work from previous satisfiability checks and backtracking to previous states. Further, this enables incremental preprocessing. Bitwuzla’s backtrable data structures are conceptually similar to context-dependent data structures in `cvc5` [3].

### 3 Theory Solvers

The Solver Engine maintains a theory solver for each supported theory and implements a module for handling quantified formulas as a dedicated theory solver. The central engine of the Solver Engine is the bit-vector theory solver, which reasons about a bit-vector abstraction of the current set of input assertions, refined with lemmas generated by other theory solvers. The theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions are combined via a model-based theory combination approach similar to [12, 38].

Theory combination is based on candidate models produced by the bit-vector theory solver for the bit-vector abstraction (function  $T_{BV}::\text{solve}()$  in Algorithm 1). For each candidate model, each theory solver checks consistency with the axioms of the corresponding theory (functions  $T_*::\text{check}()$  in Algorithm 1). If a theory solver requests a model value for a term that is not part of the current bit-vector abstraction, the theory solver who “owns” that term is queried for a value. If this value or the candidate model is inconsistent with the axioms of the theory querying the value, it sends a lemma to refine the bit-vector abstraction.

#### 3.1 Arrays

The array theory solver implements and extends the array procedure from [12] with support for reasoning over (equalities of) nested arrays and non-extensional constant arrays. This is in contrast to Boolector, which generalizes the lemmas on demand procedure for extensional arrays as described in [12] to non-recursive first-order lambda terms [37, 38], without support for nested arrays. Generalizing arrays to lambda terms allows to use the same procedure for arrays and uninterpreted functions and enables a natural, compact representation and extraction of extended array operations such as *memset*, *memcpy* and array initialization patterns as described in [39]. As an example,  $\text{memset}(a, i, n, e)$ , which updates  $n$  elements of array  $a$  within range  $[i, i + n[$  to a value  $e$  starting from index  $i$ , can be represented as  $\lambda j. \text{ite}(i \leq j < i + n, e, a[j])$ . Reasoning over equalities involving arbitrary lambda terms (including these operations), however, requires higher-order reasoning, which is not supported by Boolector. Further, extensionality over standard array operators that are represented as lambda terms (e.g.,

store) requires special handling, which makes the procedure unnecessarily complex. Bitwuzla, on the other hand, implements separate theory solvers for arrays and uninterpreted functions. Consequently, since it does not generalize arrays to lambda terms, it cannot utilize the elegant representation of Boolector for the extended array operations of [39]. Thus, currently, extracting and reasoning about these operations is not yet supported. Instead of representing such operators as lambda terms, we plan to introduce specific array operators. This will allow a seamless integration into Bitwuzla’s array procedure, with support for reasoning about extensionality involving these operators. We will also add support for reasoning about extensional constant arrays in the near future.

### 3.2 Bit-Vectors

The bit-vector theory solver implements two orthogonal approaches: the classic *bit-blasting* technique employed by most state-of-the-art bit-vector solvers, which eagerly translates the current bit-vector abstraction to SAT; and the *ternary propagation-based local search* approach presented in [27]. Since local search procedures only allow to determine satisfiability, they are particularly effective as a complementary strategy, in combination with (rather than instead of) bit-blasting [27,33]. Bitwuzla’s bit-vector solver allows to combine local search with bit-blasting in a sequential portfolio setting: the local search procedure is run until a predefined resource limit is reached before falling back on the bit-blasting procedure. Currently, Bitwuzla allows combining these two approaches only in this particular setting. We plan to explore more interleaved configurations, possibly while sharing information between the procedures as future work.

*Bit-Blasting.* Bitwuzla implements the eager reduction of the bit-vector abstraction to propositional logic in two phases. First, it constructs an And-Inverter-Graph (AIG) circuit representation of the abstraction while applying AIG-level rewriting techniques [10]. This AIG circuit is then converted into Conjunctive Normal Form (CNF) via Tseitin transformation and sent to the SAT solver back-end. Note that for assertions from levels  $> 0$ , Bitwuzla leverages solving under assumptions in the SAT solver in order to be able to backtrack to lower assertion levels on pop. Bitwuzla supports CaDiCaL [7], CryptoMiniSat [41], and Kissat [7] as SAT back-ends and uses CaDiCaL as its default SAT solver.

*Local Search.* Bitwuzla implements an improved version of the ternary propagation-based local search procedure described in [27]. This procedure is a generalization of the propagation-based local search approach implemented in Boolector [33] and addresses one of its main weaknesses: its obliviousness to bits that can be simplified to constant values. Propagation-based local search is based on propagating target values from the outputs to the inputs, does not require bit-blasting, brute-force randomization or restarts, and lifts the concept of backtracing of Automatic Test Pattern Generation (ATPG) [22] to the word-level. Boolector additionally implements the stochastic local search (SLS) approach presented in [18], optionally augmented with a propagation-based strategy [34].



Bitwuzla, however, only implements our ternary propagation-based approach since it was shown to significantly outperform these approaches [33].

### 3.3 Floating-Point Arithmetic

The solver for the theory of floating-point arithmetic implements an eager translation of floating-point atoms in the bit-vector abstraction to equisatisfiable formulas in the theory of bit-vectors, a process sometimes referred to as *word-blasting*. To translate floating-point expressions to the word-level, Bitwuzla integrates SymFPU [9], a C++ library of bit-vector encodings of floating-point operations. SymFPU uses templated types for Booleans, (un)signed bit-vectors, rounding modes and floating-point formats, which allows utilizing solver-specific representations. SymFPU has also been integrated into cvc5 [3].

### 3.4 Uninterpreted Functions

For the theory of uninterpreted functions (UF), Bitwuzla implements *dynamic Ackermannization* [16], which is a lazy form of Ackermann’s reduction. The UF solver checks whether the current satisfying assignment of the bit-vector abstraction is consistent with the function congruence axiom  $\bar{a} = \bar{b} \rightarrow f(\bar{a}) = f(\bar{b})$  and produces a lemma whenever the axiom is violated.

### 3.5 Quantifiers

Quantified formulas are handled by the quantifiers module, which is treated as a theory solver and implements model-based quantifier instantiation [20] for all supported theories and their combinations. In the bit-vector abstraction, quantified formulas are abstracted as Boolean constants. Based on the assignment of these constants, the quantifiers solver produces instantiation or Skolemization lemmas. If the constant is assigned to true, the quantifier is treated as universal quantifier and the solver produces instantiation lemmas. If the constant is assigned to false, the solver generates a Skolemization lemma. Bitwuzla allows to combine quantifiers with all supported theories as well as incremental solving and unsat core extraction. This is in contrast to Boolector, which only supports sequential reasoning about quantified bit-vector formulas and, generally, does not provide unsat cores for unsatisfiable instances.

## 4 Evaluation

We evaluate the overall performance of Bitwuzla on all non-incremental and incremental benchmarks of all supported logics in SMT-LIB [5]. We further include logics with floating-point arithmetic that are classified as containing linear integer arithmetic (LRA). Bitwuzla does not support LRA reasoning, but the benchmarks in these logics currently only involve to-floating-point conversion (SMT-LIB command `to_fp`) from real values, which is supported.

Logic	Boolector	Z3	cvc5	SC22	Bitwuzla
ABV (169)	-	<b>89</b>	32	0	1
ABVFP (30)	-	<b>25</b>	19	0	16
ABVFPLRA (75)	-	<b>47</b>	36	0	31
AUFBV (1,522)	-	403	486	597	<b>983</b>
AUFBVFP (57)	-	7	21	24	<b>39</b>
BV (6,045)	5,659	5,593	<b>5,818</b>	5,624	5,705
BVFP (205)	-	176	171	148	<b>188</b>
BVFPLRA (209)	-	189	107	140	<b>199</b>
FP (2,669)	-	2,128	2,353	<b>2,513</b>	2,481
FPLRA (87)	-	72	51	55	<b>83</b>
QF_ABV (15,084)	15,041	14,900	14,923	<b>15,043</b>	15,041
QF_ABVFP (18,129)	-	18,017	18,113	<b>18,125</b>	<b>18,125</b>
QF_ABVFPLRA (74)	-	69	<b>74</b>	34	<b>74</b>
QF_AUFBV (67)	45	50	42	46	<b>55</b>
QF_AUFBVFP (1)	-	1	1	1	1
QF_BV (42,472)	41,958	40,876	41,574	42,039	<b>42,049</b>
QF_BVFP (17,244)	-	17,229	17,238	<b>17,242</b>	17,241
QF_FP (40,409)	-	40,303	40,357	<b>40,368</b>	40,358
QF_FPLRA (57)	-	41	48	<b>56</b>	<b>56</b>
QF_UFBV (1,434)	1,403	1,404	1,387	<b>1,413</b>	1,411
QF_UFFP (2)	-	2	2	2	2
UFBV (192)	-	<b>156</b>	141	146	147
UFBVFP (2)	-	1	1	1	1
<b>Total (146,235)</b>	64,106	141,778	142,995	143,617	<b>144,287</b>
<b>Time (solved) [s]</b>	417,643	1,212,584	1,000,466	563,832	580,435

Table 1. Solved instances and total runtime on solved instances (non-incremental).

Logic	Boolector	Z3	cvc5	SC22	Bitwuzla
ABVFPLRA (2,269)	-	2,220	818	55	<b>2,269</b>
BV (38,856)	-	<b>37,188</b>	36,169	35,567	35,246
BVFP (458)	-	<b>458</b>	<b>458</b>	274	<b>458</b>
BVFPLRA (5,597)	-	<b>5,507</b>	2,964	3,144	4,797
QF_ABV (3,411)	3,238	2,866	2,746	<b>3,242</b>	2,939
QF_ABVFP (550,088)	-	515,714	534,629	550,034	<b>550,041</b>
QF_ABVFPLRA (1,876)	-	48	<b>1,876</b>	<b>1,876</b>	<b>1,876</b>
QF_AUFBV (967)	23	860	320	23	<b>956</b>
QF_BV (53,684)	52,218	51,826	51,683	51,581	<b>52,305</b>
QF_BVFP (3,465)	-	3,403	3,437	<b>3,444</b>	3,438
QF_BVFPLRA (32,736)	-	31,287	32,681	<b>32,736</b>	<b>32,736</b>
QF_FP (663)	-	663	663	663	663
QF_FPLRA (48)	-	48	48	48	48
QF_UFBV (5,492)	4,634	5,422	5,148	2,317	<b>5,489</b>
QF_UFFP (2)	-	2	2	2	2
<b>Total (699,612)</b>	60,113	657,512	673,642	685,006	<b>693,263</b>
<b>Time (solved) [s]</b>	102,812	3,359,645	1,516,672	157,083	172,534

Table 2. Solved queries and total runtime on solved queries (incremental).

We compare against Boolector [31] and the SMT-COMP 2022 version of Bitwuzla [26] (configuration SC22), which, at that time, was an improved and extended version of Boolector and won several divisions in all tracks of SMT-COMP 2022 [2]. Boolector did not participate in SMT-COMP 2022, thus we use the current version of Boolector available on GitHub (commit 13a8a06d) [1]. Further, since Boolector does not support logics involving floating-point arithmetic, quantified logics other than pure quantified bit-vectors and incremental solving when quantifiers are involved, we also compare against the SMT-COMP 2022 versions of cvc5 [3] and Z3 [25]. Both solvers are widely used, high performance SMT solvers with support for a wide range of theories, including the theories supported by Bitwuzla. Note that this version of cvc5 uses a sequential portfolio of multiple configurations for some logics.

We ran all experiments on a cluster with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 8GB of RAM for each solver and benchmark pair, and used a 1200s time limit, the same time limit as used in SMT-COMP 2022 [2].

Table 1 shows the number of solved benchmarks for each solver in the non-incremental quantifier-free (QF-) and quantified divisions. Overall, Bitwuzla solves the largest number of benchmarks in the quantified divisions, considerably improving over SC22 and Boolector with over 600 and 4,200 solved benchmarks, respectively. Bitwuzla also takes the lead in the quantifier-free divisions, with 44 more solved instances compared to SC22, and more than 650 solved benchmarks compared to cvc5. On the 140,438 commonly solved instances between Bitwuzla, SC22, cvc5, and Z3 over all divisions, Bitwuzla is the fastest solver with 203,838s, SC22 is slightly slower with 208,310s, cvc5 is  $2.85\times$  slower (586,105s), and Z3 is  $5.1\times$  slower (1,049,534s).

Table 2 shows the number of solved incremental check-sat queries for each solver in the incremental divisions. Again, Bitwuzla solves the largest number of queries overall and in the quantifier-free divisions. For the quantified divisions, Bitwuzla solves 42,770 queries, the second largest number of solved queries after Z3 (45,373), and more than 3700 more queries than SC22 (39,040). On benchmarks of the ABVFPLRA division, Bitwuzla significantly outperforms SC22 due to the occurrence of nested arrays, which were unsupported in SC22.

The artifact of this evaluation is archived and available in the Zenodo open-access repository at <https://zenodo.org/record/7864687>.

## 5 Conclusion

Our experimental evaluation shows that Bitwuzla is a state-of-the-art SMT solver for the quantified and quantifier-free theories of fixed-size bit-vectors, arrays, floating-point arithmetic, and uninterpreted functions. Bitwuzla has been extensively tested for robustness and correctness with Murxla [30], an API fuzzer for SMT solvers, which is an integral part of its development workflow. We have outlined several avenues for future work throughout the paper. We further plan to add support for the upcoming SMT-LIB version 3 standard, when finalized.

## References

1. Boolector. <https://github.com/boolector/boolector> (2023)
2. The International Satisfiability Modulo Theories Competition (SMT-COMP) (2023), <https://smt-comp.github.io>
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) 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. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24), [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at <http://smt-lib.org>
5. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2023), <http://smt-lib.org>
6. Barrett, C.W., Dill, D.L., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2404, pp. 236–249. Springer (2002). [https://doi.org/10.1007/3-540-45657-0\\_18](https://doi.org/10.1007/3-540-45657-0_18), [https://doi.org/10.1007/3-540-45657-0\\_18](https://doi.org/10.1007/3-540-45657-0_18)
7. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froylyks, N., Heule, M., Iser, M., Jarvisalo, 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)
8. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11427, pp. 79–98. Springer (2019). [https://doi.org/10.1007/978-3-030-17462-0\\_5](https://doi.org/10.1007/978-3-030-17462-0_5), [https://doi.org/10.1007/978-3-030-17462-0\\_5](https://doi.org/10.1007/978-3-030-17462-0_5)
9. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: TACAS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I. LNCS, vol. 11427, pp. 79–98. Springer (2019). [https://doi.org/10.1007/978-3-030-17462-0\\_5](https://doi.org/10.1007/978-3-030-17462-0_5), [https://doi.org/10.1007/978-3-030-17462-0\\_5](https://doi.org/10.1007/978-3-030-17462-0_5)
10. Brummayer, R., Biere, A.: Local Two-Level And-Inverter Graph Minimization without Blowup. In: 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006, Proceedings (2006)
11. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009,

- Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5505, pp. 174–177. Springer (2009). [https://doi.org/10.1007/978-3-642-00768-2\\_16](https://doi.org/10.1007/978-3-642-00768-2_16), [https://doi.org/10.1007/978-3-642-00768-2\\_16](https://doi.org/10.1007/978-3-642-00768-2_16)
12. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. *J. Satisf. Boolean Model. Comput.* **6**(1-3), 165–201 (2009). <https://doi.org/10.3233/sat190067>, <https://doi.org/10.3233/sat190067>
  13. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), [http://www.usenix.org/events/osdi08/tech/full\\_papers/cadar/cadar.pdf](http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf)
  14. Champion, A., Mebsout, A., Stickel, C., Tinelli, C.: The kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016). [https://doi.org/10.1007/978-3-319-41540-6\\_29](https://doi.org/10.1007/978-3-319-41540-6_29)
  15. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7795, pp. 93–107. Springer (2013). [https://doi.org/10.1007/978-3-642-36742-7\\_7](https://doi.org/10.1007/978-3-642-36742-7_7), [https://doi.org/10.1007/978-3-642-36742-7\\_7](https://doi.org/10.1007/978-3-642-36742-7_7)
  16. Dutertre, B., de Moura, L.: The Yices SMT Solver. (2006), <https://yices.csl.sri.com/papers/tool-paper.pdf>
  17. 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](https://doi.org/10.1007/978-3-319-08867-9_49), [https://doi.org/10.1007/978-3-319-08867-9\\_49](https://doi.org/10.1007/978-3-319-08867-9_49)
  18. 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), <http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9896>
  19. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 519–531. Springer (2007). [https://doi.org/10.1007/978-3-540-73368-3\\_52](https://doi.org/10.1007/978-3-540-73368-3_52)
  20. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009). [https://doi.org/10.1007/978-3-642-02658-4\\_25](https://doi.org/10.1007/978-3-642-02658-4_25)
  21. Godefroid, P., Levin, M.Y., Molnar, D.A.: SAGE: whitebox fuzzing for security testing. *Commun. ACM* **55**(3), 40–44 (2012). <https://doi.org/10.1145/2093548.2093564>

22. Kunz, W., Stoffel, D.: Reasoning in Boolean Networks - Logic Synthesis and Verification Using Testing Techniques. *Frontiers in electronic testing*, Springer (1997). <https://doi.org/10.1007/978-1-4757-2572-8>, <https://doi.org/10.1007/978-1-4757-2572-8>
23. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.W.: Pono: A flexible and extensible smt-based model checker. In: Silva, A., Leino, K.R.M. (eds.) *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. *Lecture Notes in Computer Science*, vol. 12760, pp. 461–474. Springer (2021). [https://doi.org/10.1007/978-3-030-81688-9\\_22](https://doi.org/10.1007/978-3-030-81688-9_22), [https://doi.org/10.1007/978-3-030-81688-9\\_22](https://doi.org/10.1007/978-3-030-81688-9_22)
24. Moura, L.D., Rueß, H.: Lemmas on demand for satisfiability solvers. In: *The 5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002, Cincinnati, USA, May 15, 2002* (2002)
25. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24), [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
26. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. *CoRR abs/2006.01621* (2020), <https://arxiv.org/abs/2006.01621>
27. Niemetz, A., Preiner, M.: 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*. pp. 214–224. IEEE (2020). [https://doi.org/10.34727/2020/isbn.978-3-85448-042-6\\_29](https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29), [https://doi.org/10.34727/2020/isbn.978-3-85448-042-6\\_29](https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29)
28. Niemetz, A., Preiner, M.: Bitwuzla. <https://github.com/bitwuzla/bitwuzla> (2023)
29. Niemetz, A., Preiner, M.: Bitwuzla Documentation. <https://bitwuzla.github.io> (2023)
30. Niemetz, A., Preiner, M., Barrett, C.W.: Murxla: A modular and highly extensible API fuzzer for SMT solvers. In: Shoham, S., Vizel, Y. (eds.) *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II. Lecture Notes in Computer Science*, vol. 13372, pp. 92–106. Springer (2022). [https://doi.org/10.1007/978-3-031-13188-2\\_5](https://doi.org/10.1007/978-3-031-13188-2_5), [https://doi.org/10.1007/978-3-031-13188-2\\_5](https://doi.org/10.1007/978-3-031-13188-2_5)
31. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. *J. Satisf. Boolean Model. Comput.* **9**(1), 53–58 (2014). <https://doi.org/10.3233/sat190101>, <https://doi.org/10.3233/sat190101>
32. Niemetz, A., Preiner, M., Biere, A.: Turbo-charging lemmas on demand with don't care reasoning. In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. pp. 179–186. IEEE (2014). <https://doi.org/10.1109/FMCAD.2014.6987611>, <https://doi.org/10.1109/FMCAD.2014.6987611>
33. Niemetz, A., Preiner, M., Biere, A.: Propagation based local search for bit-precise reasoning. *Formal Methods Syst. Des.* **51**(3), 608–636 (2017). <https://doi.org/10.1007/s10703-017-0295-6>, <https://doi.org/10.1007/s10703-017-0295-6>
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 (DIFTS)*, affiliated with FMCAD, Austin, TX. pp. 1–10 (2015)

35. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: BTOR2 , BtorMC and Boolec-tor 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verifica-tion - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018). [https://doi.org/10.1007/978-3-319-96145-3\\_32](https://doi.org/10.1007/978-3-319-96145-3_32)
36. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to  $dpll(T)$ . J. ACM **53**(6), 937–977 (2006). <https://doi.org/10.1145/1217856.1217859>
37. Preiner, M.: Lambdas, Arrays and Quantifiers. Ph.D. thesis, Informatik, Johannes Kepler University Linz (2017)
38. Preiner, M., Niemetz, A., Biere, A.: Lemmas on demand for lambdas. In: Ganai, M.K., Sen, A. (eds.) Proceedings of the Second International Workshop on Design and Implementation of Formal Tools and Systems, Portland, OR, USA, October 19, 2013. CEUR Workshop Proceedings, vol. 1130. CEUR-WS.org (2013), [http://ceur-ws.org/Vol-1130/paper\\_7.pdf](http://ceur-ws.org/Vol-1130/paper_7.pdf)
39. Preiner, M., Niemetz, A., Biere, A.: Better lemmas with lambda extraction. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015. pp. 128–135. IEEE (2015). <https://doi.org/10.1109/FMCAD.2015.7542262>
40. Preiner, M., Niemetz, A., Biere, A.: Counterexample-guided model synthesis. In: Legay, A., Margaria, T. (eds.) 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. Lecture Notes in Computer Science, vol. 10205, pp. 264–280 (2017). [https://doi.org/10.1007/978-3-662-54577-5\\_15](https://doi.org/10.1007/978-3-662-54577-5_15), [https://doi.org/10.1007/978-3-662-54577-5\\_15](https://doi.org/10.1007/978-3-662-54577-5_15)
41. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceed-ings. LNCS, vol. 5584, pp. 244–257. Springer (2009). [https://doi.org/10.1007/978-3-642-02777-2\\_24](https://doi.org/10.1007/978-3-642-02777-2_24), [https://doi.org/10.1007/978-3-642-02777-2\\_24](https://doi.org/10.1007/978-3-642-02777-2_24)
42. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1), 221–259 (2019). <https://doi.org/10.3233/SAT190123>, <https://doi.org/10.3233/SAT190123>