CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver

Xiaomu Shi[1], Yu-Fu Fu[2], Jiaxiang Liu[1], Ming-Hsien Tsai[3], Bow-Yaw Wang[3], and Bo-Yin Yang[3]

[1] Shenzhen University, China
[2] Georgia Institute of Technology, USA
[3] Academia Sinica, Taiwan

Abstract. We present a certified SMT QF_BV solver CoqQFBV built from a verified bit blasting algorithm, Kissat, and the verified SAT certificate checker GratChk in this paper. Our verified bit blasting algorithm supports the full QF_BV logic of SMT-LIB; it is specified and formally verified in the proof assistant Coq. We compare CoqQFBV with CVC4, Bitwuzla, and Boolector on benchmarks from the QF_BV division of the single query track in the 2020 SMT Competition, and real-world cryptographic program verification problems. CoqQFBV surprisingly solves more program verification problems with certification than the 2020 SMT QF_BV division winner Bitwuzla without certification.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers for the Quantifier-Free Bit-Vector (QF_BV) logic have been used to verify programs with bit-level accuracy [9,10]. In such applications, a program verification problem is reformulated as an SMT QF_BV query. An SMT QF_BV solver is then invoked to compute a query result. The query result in turn decides the answer to the program verification problem. For cryptographic assembly programs, a missing carry or borrow flag will result in incorrect computation. Bit-accurate verification is thus necessary for cryptographic programs. SMT QF_BV solvers in fact have been employed to verify such programs [8,25]. These solvers nonetheless are very complex programs with possibly unknown bugs [7,18]. Since bugs in SMT QF_BV solvers may induce incorrect query results, program verification cannot be taken without a grain of salt when SMT QF_BV solvers are employed.

In order to check SMT QF_BV query results independently, SMT QF_BV solvers can generate certificates to validate their answers. In the LFSC certificates [23,14], for instance, an SMT QF_BV query result is certified by correct bit blasting and Boolean Satisfiability (SAT) solving. Such certificates demonstrate that the SMT QF_BV query is reduced to a Boolean SAT query correctly *and* the corresponding SAT query is solved correctly. Although one can certify SAT query results with certificates from SAT solvers [24], it is not always easy to certify correct bit blasting due to complex arithmetic operations in SMT

QF_BV queries. Developing correct and efficient checkers for SMT QF_BV certificates can be very challenging. Indeed, an LFSC certificate checker based on the proof assistant Coq has been developed to improve confidence [12]. Yet the Coq-based certificate checker does not fully support arithmetic operations and thus cannot certify results of SMT QF_BV queries with complicated arithmetic operations. Consequently, the correctness of cryptographic programs still relies on the correctness of SMT QF_BV solvers or their unverified certificate checkers.

In this paper, we take a more direct approach to ensure the correctness of SMT QF_BV query results. Instead of certifying correct bit blasting for every SMT QF_BV query, we specify a bit blasting algorithm and prove its correctness in the proof assistant Coq. In order to formalize the correctness of our bit blasting algorithm, we develop a formal bit-vector theory in Coq. Naturally, the formal theory has to support all arithmetic functions (addition, subtraction, multiplication, division, and remainder) for both signed and unsigned representations as needed in SMT-LIB [3]. Based on our new bit-vector theory, we give a formal semantics for SMT QF_BV queries in Coq. Our semantics follows the SMT-LIB semantics carefully. Particularly, division and remainder are total arithmetic operations even when the divisor is zero. Using our Coq bit-vector theory and semantics, we prove that our bit blasting algorithm always returns a corresponding Boolean formula correctly on any SMT QF_BV query. Since our algorithm has been formally verified, bit blasting is always correct and need not be certified. Through the OCaml program extracted from our verified bit blasting algorithm, a corresponding SAT query is obtained for each SMT QF_BV query and sent to a SAT solver. A SAT certificate checker suffices to validate SAT query results and hence the correctness of answers to SMT QF_BV queries. Since neither complicated SMT QF_BV solvers nor their certificate checkers are trusted, our work can improve the confidence of SMT QF_BV query results.

To our knowledge, our bit-vector theory is the first Coq formalization designed for bit blasting queries from the QF_BV logic of SMT-LIB. Our semantics is the first Coq formalization for full SMT QF_BV queries. We are not aware of any verified bit blasting algorithm or program for full SMT QF_BV queries of SMT-LIB at the time of writing. Even the correctness of its results could be ensured, our certified SMT QF_BV solver CoqQFBV would not be very useful if it were extremely inefficient. In order to evaluate its performance, we run CoqQFBV on benchmarks from the QF_BV division of the single query track in the 2020 SMT Competition. With the same memory and time limits in the competition, our solver successfully finishes 88.72% of the 6861 queries with certification. In comparison, CVC4 with its certificate checker solves 55.97% with certification, and the division winner Bitwuzla solves 98.22% of the benchmarks without certification. Our certified solver outperforms CVC4 with certification significantly. Generating and checking certificates make our certified solver finish about 10% of the queries less than the division winner. The price of accuracy perhaps is not unacceptable for the benchmarks in the competition. To further evaluate CoqQFBV, the certified solver is used to verify linear arithmetic assembly programs from various cryptography libraries such as OpenSSL [30]. CoqQFBV gives certified answers to 96.88% out of the 96 SMT QF_BV queries from real-world cryptographic program verification. CVC4 with its certificate checker certifies 19.79%. Compared with efficient SMT QF_BV solvers without certification, Boolector is able to solve 100% and Bitwuzla solves 91.67% of the queries. Intriguingly, our certified SMT QF_BV solver outperforms the 2020 division winner Bitwuzla in queries from real-world verification problems. Our certified solver is probably useful for real-world verification problems.

Related Work. As mentioned, SMT certificate generating and checking are challenging. There are few efforts developing SMT QF_BV certificate checkers, let alone verified ones. CVC4 is able to produce unsatisfiability certificates for QF_BV queries, and also equipped with an (unverified) certificate checker [14]. SMTCoq [12] is proposed to check certificates from SMT solvers veriT and CVC4. It supports fragments of several logics including the QF_BV logic. Moreover, its correctness is formally proved in Coq. However, the QF_BV logic is not fully supported by SMTCoq. Z3 also supports certificate generation for the QF_BV logic [19]. The proofs can be reconstructed, thus checked, within proof assistants HOL4 and Isabelle [6]. But the lack of details in Z3's generated certificates makes proof reconstruction particularly challenging.

With a similar approach in this paper, GL is a framework for bit blasting finitely bounded ACL2 theorems into SAT queries [28]. Its bit blasting algorithm is formally verified in ACL2. Though it is not designed for SMT-LIB, most of the operations defined in the QF_BV logic are supported, except division and concatenation for instance. A bit blasting algorithm is defined and verified in HOL4 as well [13]. Neither [28] nor [13] aims to develop a scalable SMT QF_BV solver. CoqQFBV accepts SMT-LIB inputs with fully supported QF_BV logic while adopting performance optimizations such as caches.

In Isabelle and HOL4, one can use the bit-vector libraries to conform SMT-LIB operations, see [17] for example. Under the frame of Coq, coq-bits is a formalization of logical and arithmetic operations on bit vectors [15]. The library provides the mapping between bit-vector operations and abstract number operations. Different from our theory, it does not support division/remainder or signed operations. Why3 [11] provides a bit-vector theory which is formalized in Coq too. It defines the division by zero in a different way from SMT-LIB. Moreover, the operations are defined based on integer operations. Our new bit-vector theory instead defines bit-vector operations through bit manipulation. It is more suitable for the correctness proof of bit blasting algorithms.

We have the following organization. After the introduction, an overview is given in Section 2. Section 3 reviews preliminaries. Our formal bit vector theory is presented in Section 4. It is followed by the formal semantics of SMT QF_BV queries (Section 5). The correctness of our bit blasting algorithm is established in Section 6. Section 7 outlines the construction of our certified SMT QF_BV solver. Experiments are presented in Section 8. Section 9 concludes our presentation.

2 Methodology Overview

Given an SMT QF_BV query, a bit blasting algorithm computes a Boolean formula such that the SMT QF_BV query is satisfiable if and only if the Boolean formula is satisfiable. The QF_BV logic contains arithmetic operations for bit vectors. Computing an equi-satisfiable Boolean formula for an arbitrary SMT QF_BV query can be very complicated and susceptible to errors. Our goal is to construct a correct bit blasting program for every SMT QF_BV query. The correctness of the program moreover is verified by the proof assistant Coq to minimize gaps or even errors in hand-written proofs.

Our construction is based on a new formal bit-vector theory `coq-nbits` (Section 4). In `coq-nbits`, we define bit vectors and their functions on top of the Coq data type for Boolean sequences. In order to support the QF_BV logic of SMT-LIB fully, five arithmetic bit-vector functions (addition, subtraction, multiplication, division, and remainder) are defined in our formal theory. To establish the correctness of our definitions, formal proofs are provided to relate bit-vector functions with their arithmetic counterparts. For instance, we show the number represented by the output of the bit-vector negation function is indeed the arithmetic negation of the number represented by the input bit vector.

Using our `coq-nbits` theory, we then give a formal semantics for SMT QF_BV queries as defined in SMT-LIB (Section 5). In our formalization, a QF_BV predicate denotes a Boolean value; and a QF_BV expression denotes a bit vector. An SMT QF_BV query is formalized as a Boolean combination of QF_BV predicates on QF_BV expressions over QF_BV variables and bit-vector constants. In order to demonstrate the correctness of our formal semantics for SMT QF_BV queries, formal proofs are provided to show that our formal semantics coincides with those defined in SMT-LIB.

Our bit blasting algorithm is given in Coq (Section 6). It extends Tseitin transformation for Boolean formulae to SMT QF_BV queries. More precisely, a QF_BV predicate is transformed to a literal with a Boolean formula; a QF_BV expression is transformed to a literal sequence with a Boolean formula. Using our formalization of SMT QF_BV queries, the correctness of bit blasting algorithm is established in Coq by mutual induction. To improve efficiency, our bit blasting algorithm is further optimized with more economic transformations and a cache. The optimized bit blasting algorithm is also verified with formal Coq proofs.

Our formally verified bit blasting algorithm is written in the Coq specification language. It is not yet a program compilable into executable binary codes. Using the code extraction mechanism in Coq, an OCaml program is extracted from our verified bit blasting algorithm. The OCaml program takes expressions in our formal SMT QF_BV query syntax as inputs and returns expressions in our formal syntax for Boolean formulae as outputs. SAT solvers can be employed to decide satisfiability of output Boolean formulae. Their certificates can be validated by SAT certificate checkers independently (Section 7).

3 Preliminaries

Let v be a Boolean *variable* with values *ff* and *tt*. A *literal* is of the form v or $\neg v$. A *clause* is a disjunction $l_0 \vee l_1 \vee \cdots \vee l_k$ of literals $l_0, l_1, \ldots, l_k$. A Boolean formula in the *conjunctive normal form (CNF)* is a conjunction $c_0 \wedge c_1 \wedge \cdots \wedge c_m$ of clauses $c_0, c_1, \ldots, c_m$. A SAT *query* is a Boolean CNF formula. An *environment* maps Boolean variables to their values. Given a SAT query, the *Boolean satisfiability problem* is to decide if the query evaluates to *tt* on some environments.

A bit vector of *width* w is written as $\texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ with $b_i \in \{0,1\}$ for $0 \leq i < w$. In the *unsigned* representation, the bit vector $\texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ denotes the natural number (non-negative integer) $\sum_{0 \leq i < w} b_i 2^i$; in *two's complement (signed)* representation, it denotes the integer $\sum_{0 \leq i < w-1} b_i 2^i - 2^{w-1} b_{w-1}$. For instance, #b1010 denotes 10 and -6 in the unsigned and two's complement representations respectively. We use $\mathit{bv2nat}(bv)$ for the natural number denoted by the bit vector bv in the unsigned representation; and $\mathit{nat2bv}(w, i)$ stands for the bit vector of width w representing the natural number i modulo 2^w.

Let $bv = \texttt{\#b}b_{w-1}b_{w-2}\cdots b_0$ and $cv = \texttt{\#b}c_{u-1}c_{u-2}\cdots c_0$ be bit vectors of widths w and u respectively. The following QF_BV *operations* are defined in the QF_BV logic of SMT-LIB: $\mathit{concat}\ bv\ cv \triangleq \texttt{\#b}b_{w-1}b_{w-2}\cdots b_0 c_{u-1}c_{u-2}\cdots c_0$ is the concatenation of bv and cv; $\mathit{extract}\ i\ j\ bv \triangleq \texttt{\#b}b_i b_{i-1}\cdots b_j$ extracts bits from bv where $0 \leq j \leq i < w$; $\mathit{bvnot}\ bv$, $\mathit{bvand}\ bv\ cv$, and $\mathit{bvor}\ bv\ cv$ are the bitwise complement, and, or operations respectively. Additionally, $\mathit{bvneg}\ bv \triangleq \mathit{nat2bv}(w, 2^w - \mathit{bv2nat}(bv))$ is the arithmetic negation operation; $\mathit{bvadd}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) + \mathit{bv2nat}(cv))$ is the arithmetic addition operation; and $\mathit{bvmul}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \times \mathit{bv2nat}(cv))$ is the arithmetic multiplication operation. The arithmetic division and remainder operations are

$$\mathit{bvudiv}\ bv\ cv \triangleq \begin{cases} \mathit{nat2bv}(w, 2^w - 1) & \text{if } \mathit{bv2nat}(cv) = 0 \\ \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \div \mathit{bv2nat}(cv)) & \text{otherwise} \end{cases}$$
$$\mathit{bvurem}\ bv\ cv \triangleq \begin{cases} bv & \text{if } \mathit{bv2nat}(cv) = 0 \\ \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \bmod \mathit{bv2nat}(cv)) & \text{otherwise.} \end{cases}$$

Note that the arithmetic division and remainder operations are defined even when the divisor represents the number zero. Finally, the operations $\mathit{bvshl}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \times 2^{\mathit{bv2nat}(cv)})$ shifts the bit vector bv to the left by $\mathit{bv2nat}(cv)$ bits; $\mathit{bvlshr}\ bv\ cv \triangleq \mathit{nat2bv}(w, \mathit{bv2nat}(bv) \div 2^{\mathit{bv2nat}(cv)})$ shifts the bit vector bv to the right by $\mathit{bv2nat}(cv)$ bits. In addition to bit-vector operations, the QF_BV logic of SMT-LIB defines QF_BV *predicates* on bit vectors. The predicate $\mathit{bveq}\ bv\ cv$ is true when the bit vectors bv and cv are equal; $\mathit{bvult}\ bv\ cv$ is true if $\mathit{bv2nat}(bv) < \mathit{bv2nat}(cv)$. In the QF_BV logic of SMT-LIB, both operands of binary operations and predicates must have the same width. Overall, seventeen bit-vector operations and predicates are defined in the QFBV logic of SMT-LIB. Particularly, arithmetic division and remainder operations with operands in both unsigned and two's complement signed representations are defined in SMT-LIB.

A QF_BV *variable* denotes a bit vector. A QF_BV *expression* is constructed from QF_BV operations over QF_BV variables and bit vectors. An SMT QF_BV *query* is a Boolean combination of QF_BV predicates on QF_BV expressions. Let *stores* be mappings from QF_BV variables to bit vectors. Given an SMT QF_BV query, the *satisfiability modulo* QF_BV *theory problem* is to decide if the query evaluates to *tt* on some stores.

4 Bit-Vector Theory

We present our formal Coq bit-vector theory `coq-nbits` in this section. The `coq-nbits` theory supports bit vectors in both unsigned and two's complement signed representations. In `coq-nbits`, a bit vector is represented by a Boolean sequence of the data type bits in the least significant bit-first order.

```
Definition bits : Set := seq bool.
```

In the definition, `bool` and `seq` are the data types for Boolean values (`false` and `true`) and sequences in Coq respectively. For instance, the bit vector `#b100` is represented by [:: `false`; `false`; `true`] in `coq-nbits`.

Coq functions defined for sequences are applicable to bit vectors. Particularly, size *bv* computes the width of the bit vector *bv* and *bv* ++ *cv* is the concatenation of the bit vectors *bv* and *cv*. It is also straightforward to define auxiliary bit-vector functions. For example, zeros n returns the bit vector of n `false`'s; ones n returns the bit vector of n `true`'s; extract i j *bv* returns the sub-sequence of the bit vector *bv* with indices from j to i where $0 \leq j \leq i <$ size *bv*. Let a $\triangleq$ [:: `false`; `false`; `true`]. Then size a $= 3$ and extract 2 1 a $=$ [:: `false`; `true`].

Bitwise functions are defined as easily. For instance, the bitwise inverse function maps each Boolean value to its complement:

```
Definition invB bv : bits := map (fun b => ~~b) bv.
```

Other bitwise functions are defined similarly. Specifically, bitwise and andB, bitwise or orB, logical left shift shlB, logical right shift shrB are all defined in `coq-nbits`. Let b $\triangleq$ [:: `false`; `true`; `true`]. We have invB b $=$ [:: `true`; `false`; `false`], andB a b $=$ [:: `false`; `false`; `true`], and shlB 1 b $=$ [:: `false`; `false`; `true`].

Arithmetic bit-vector functions are slightly more complicated. To prove properties about arithmetic functions, `coq-nbits` provides conversion functions between bit vectors and natural numbers.

```
Definition to_N (bv : bits) : N :=
  foldr (fun b res => N_of_bool b + res * 2) 0 bv.
```

In the definition, to_N *bv* converts the bit vector *bv* to a natural number where `N_of_bool` `false` $= 0$ and `N_of_bool` `true` $= 1$. The to_N function multiplies the previous result by two and adds the least significant bit b. For instance, to_N a $=$ to_N [:: `false`; `false`; `true`] $= 4$. The function from_N w n, on the other hand, converts any natural number n to a bit vector of width w.

```
Fixpoint from_N (w : nat) (n : N) : bits :=
  match w with
  | O => [::]
```

```
    | S w' => (N.odd n)::(from_N w' (N.div n 2))
    end.
```

The function first checks the width w. If the width is zero, it returns the empty bit vector. Otherwise, the function returns the bit vector with the least significant bit N.odd n and the remaining $w - 1$ bits representing n divided by two. Observe that two Coq formalizations of natural numbers are used. The nat theory uses the unary representation suitable for inductive proofs; N uses the succinct binary representation. The following lemma is proved in Coq:

Lemma 1. *The following properties hold:*

1. $\forall bv, \mathsf{from\_N}\ (\mathsf{size}\ bv)\ (\mathsf{to\_N}\ bv) = bv$.
2. $\forall w\ n, n < 2^w \implies \mathsf{to\_N}\ (\mathsf{from\_N}\ w\ n) = n$.

The first property shows that bit vectors can be converted to natural numbers and back to themselves. The second property shows that natural numbers can be converted to bit vectors with sufficient widths and back to themselves. To see how they are used to prove properties about bit-vector functions in coq-nbits, consider the definition of the successor bit-vector function.

```
Fixpoint succB (bv : bits) : bits :=
  match bv with
  | [::] => [::]
  | hd::tl => if hd then false::(succB tl) else true::tl
  end.
```

If the input is the empty bit vector, the function returns the empty bit vector. Otherwise, succB checks the least significant bit of the input bit vector. If the bit is true, the function computes the successor of the remaining bits and appends false as the least significant bit. If the least significant bit of the input is false, the function simply changes the least significant bit to true and copies the remaining bits. Using the conversion functions, the bit-vector successor is related to the arithmetic successor in the following lemma:

Lemma 2. $\forall bv, \mathsf{succB}\ bv = \mathsf{from\_N}\ (\mathsf{size}\ bv)\ ((\mathsf{to\_N}\ bv) + 1)$.

Lemma 2 says that succB bv does compute the bit vector representing the arithmetic successor of the natural number represented by the bit vector bv. Observe that the successor bit vector function is correct when the input bit vector is empty. It is also correct when there is overflow. Indeed, both sides are zeros of width size bv when overflow occurs.

Other arithmetic bit-vector functions are defined and proved in coq-nbits similarly. Specifically, the arithmetic negation negB, addition addB, subtraction subB, unsigned multiplication mulB, unsigned division divB, and unsigned remainder remB functions are supported by coq-nbits. We give properties to relate the arithmetic functions for bit vectors and natural numbers.

Lemma 3. *The following properties hold:*

1. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_N}\ (\mathsf{addB}\ bv\ cv) = (\mathsf{to\_N}\ bv + \mathsf{to\_N}\ cv)\ \mathtt{mod}\ 2^{\mathsf{size}\ bv}$.
2. $\forall bv\ cv, \mathsf{to\_N}\ (\mathsf{mulB}\ bv\ cv) = (\mathsf{to\_N}\ bv \times \mathsf{to\_N}\ cv)\ \mathtt{mod}\ 2^{\mathsf{size}\ bv}$.
3. $\forall bv\ n, \mathsf{divB}\ bv\ (\mathsf{zeros}\ n) = \mathsf{ones}\ (\mathsf{size}\ bv)$.
4. $\forall bv\ bv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies cv \neq \mathsf{zeros}\ (\mathsf{size}\ cv) \implies \mathsf{to\_N}\ (\mathsf{divB}\ bv\ cv) = (\mathsf{to\_N}\ bv)\ \mathtt{div}\ (\mathsf{to\_N}\ cv)$.
5. $\forall bv\ n, \mathsf{remB}\ bv\ (\mathsf{zeros}\ n) = bv$.
6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies cv \neq \mathsf{zeros}\ (\mathsf{size}\ cv) \implies \mathsf{to\_N}\ (\mathsf{remB}\ bv\ cv) = (\mathsf{to\_N}\ bv)\ \mathtt{mod}\ (\mathsf{to\_N}\ cv)$.
7. $\forall bv\ n, \mathsf{to\_N}\ (\mathsf{shlB}\ n\ bv) = ((\mathsf{to\_N}\ bv) \times 2^n)\ \mathtt{mod}\ 2^{\mathsf{size}\ bv}$
8. $\forall bv\ n, \mathsf{to\_N}\ (\mathsf{shrB}\ n\ bv) = (\mathsf{to\_N}\ bv)\ \mathtt{div}\ 2^n$.

Let bv, cv be bit vectors of width w. Lemma 3 shows that the natural number represented by the bit vector addB bv cv is equal to the modular sum of the natural numbers represented by bv and cv. Similarly, the natural number represented by mulB bv cv is equal to the modular product of the natural numbers represented by bv and cv. The division and remainder functions in `coq-nbits` follow the SMT-LIB semantics. Specifically, the quotient of any bit vector divided by zero is equal to the bit vector of all `true`'s; the remainder of a bit vector divided by zero is the bit vector itself. For non-zero divisors, the division and remainder functions behave as expected. The natural number represented by the bit vector divB bv cv is the quotient of the number represented by bv divided by the number represented by cv; and the bit vector remB bv cv represents the remainder of the number represented by bv divided by the number represented by cv. Last but not least, the logical left (shlB) and right (shrB) shifts correspond to multiplication and division by powers of two respectively.

`coq-nbits` also provides comparison predicates. In addition to the equality predicate == inherited from Boolean sequences, ltB bv cv and leB bv cv compare the natural numbers represented by the bit vectors bv and cv. Properties about comparison predicates have also been proved in Coq.

Lemma 4. *The following properties hold:*

1. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{ltB}\ bv\ cv = (\mathsf{to\_N}\ bv < \mathsf{to\_N}\ cv)$.
2. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{leB}\ bv\ cv = (\mathsf{to\_N}\ bv \leq \mathsf{to\_N}\ cv)$.

In addition to arithmetic functions and predicates in the unsigned representation, our formal bit-vector theory moreover defines arithmetic functions and predicates for bit vectors in two's complement representation. For the signed representation, bit vectors are converted to integers by the to_Z function. Arithmetic bit-vector functions and predicates in the signed representation are related to arithmetic integer functions and predicates as follows.

Lemma 5. *The following properties hold:*

1. $\forall bv, \neg(\mathsf{msb}\ bv \wedge \mathsf{dropmsb}\ bv = \mathsf{zeros}\ (\mathsf{size}\ bv - 1)) \implies \mathsf{to\_Z}\ (\mathsf{negB}\ bv) = -\mathsf{to\_Z}\ bv$.

2. $\forall bv\ n, 1 < \mathsf{size}\ bv \implies \mathsf{to\_Z}\ (\mathsf{sarB}\ n\ bv) = (\mathsf{to\_Z}\ bv)\ \mathtt{quot}\ 2^n$.
3. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_Z}\ (\mathsf{mulB}\ (\mathsf{sext}\ (\mathsf{size}\ cv)\ bv)\ (\mathsf{sext}\ (\mathsf{size}\ bv)\ cv)) = \mathsf{to\_Z}\ bv \times \mathsf{to\_Z}\ cv$.
4. $\forall bv\ cv, 1 < \mathsf{size}\ bv \implies \mathsf{size}\ bv = \mathsf{size}\ cv \implies [\neg(\mathsf{msb}\ bv \wedge \mathsf{dropmsb}\ bv = \mathsf{zeros}\ (\mathsf{size}\ bv - 1)) \vee cv \neq \mathsf{ones}\ (\mathsf{size}\ cv)] \implies \mathsf{to\_Z}\ (\mathsf{sdivB}\ bv\ cv) = (\mathsf{to\_Z}\ bv)\ \mathtt{quot}\ (\mathsf{to\_Z}\ cv)$.
5. $\forall bv\ cv, 1 < \mathsf{size}\ bv \implies \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{to\_Z}\ (\mathsf{sremB}\ bv\ cv) = (\mathsf{to\_Z}\ bv)\ \mathtt{rem}\ (\mathsf{to\_Z}\ cv)$.
6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{sltB}\ bv\ cv = (\mathsf{to\_Z}\ bv < \mathsf{to\_Z}\ cv)$.
7. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{sleB}\ bv\ cv = (\mathsf{to\_Z}\ bv \leq \mathsf{to\_Z}\ cv)$.

In the lemma, sext n bv extends the bit vector bv by n bits with the sign bit of bv, msb bv returns the sign bit of bv, and dropmsb bv drops the sign bit of bv. `quot` and `rem` are the quotient and remainder functions for Coq integers. Consider, for instance, the signed division function sdivB bv cv in `coq-nbits` (Lemma 5(4)). If the dividend bv is of width > 1, the widths of bv and the divisor cv are equal, and bv is not of the form `#b100` $\cdots$ `0` or cv is not of the form `#b11` $\cdots$ `1`, then the bit vector sdivB bv cv represents the quotient of the integers represented by bv and cv. The condition may appear counter-intuitive. To see why it is necessary, consider bv = `#b100` $\cdots$ `0` and cv = `#b11` $\cdots$ `1` both of width w. bv and cv thus represent the integers -2^{w-1} and -1 respectively. Their quotient 2^{w-1} however cannot be represented by bit vectors of width w in two's complement representation. The corner input case is hence excluded. The corner case is also excluded from the arithmetic negation function (Lemma 5(1)).

The `coq-nbits` theory has several important differences from the prior Coq formalization in [15]. Our formal bit-vector theory supports both unsigned and two's complement signed representations. It also provides the arithmetic division and remainder functions. Since these features are needed in the QF_BV logic of SMT-LIB, they are essential to the formalization of SMT QF_BV queries. Such important features unfortunately are lacking in the prior formalization. Another noted difference is the numeric representations used in theory developments. Since integers are needed for the QF_BV logic, `coq-nbits` naturally uses binary representations for integers and natural numbers in Coq. The prior formalization on the other hand is mainly based on the unary natural number representation but provides conversion to positive integers in the binary representation.

5 Theory for SMT QF_BV Queries

Using `coq-nbits`, we formalize SMT QF_BV queries. Our formalization consists of two parts: a syntactic representation for SMT QF_BV queries in Coq inductive types and a formal semantics in our bit-vector theory `coq-nbits`.

5.1 Syntax of SMT QF_BV Queries

An SMT QF_BV query is a Coq term of the data type `bexp`. It can be constants `Bfalse` or `Btrue`, a unary predicate `Bnot`, or binary predicates `Band` or `Bor` for

Boolean connectives. Additionally, `Bbveq` and `Bbvult` with two arguments of the data type `exp` are binary QF_BV predicates.

```
Inductive bexp : Type := Bfalse : bexp | Btrue : bexp
| Bnot : bexp -> bexp
| Band : bexp -> bexp -> bexp | Bor : bexp -> bexp -> bexp
| Bbveq : exp -> exp -> bexp  | Bbvult : exp -> exp -> bexp
(* other QFBV predicates *)
end with exp : Type :=
| Evar : var -> exp           | Econst : bits -> exp
| Ebvnot : exp -> exp
| Ebvand : exp -> exp -> exp  | Ebvor : exp -> exp -> exp
| Ebvshl : exp -> exp -> exp  | Ebvlshr : exp -> exp -> exp
| Ebvneg : exp -> exp
| Ebvadd : exp -> exp -> exp  | Ebvmul : exp -> exp -> exp
| Ebvudiv : exp -> exp -> exp | Ebvurem : exp -> exp -> exp
| Eextract : nat -> nat -> exp -> exp
| Econcat : exp -> exp -> exp
(* other QFBV operations *)
| Ebvsub : exp -> exp -> exp
end.
```

A Coq term of the data type `exp` represents a QF_BV expression. It can be a QF_BV variable `Evar` *vid* with a variable identifier *vid* : `var`, a bit vector constant `Econst` *bv* with *bv* : bits, a bitwise-not operation `Ebvnot` e_0, a bitwise-and operation `Ebvand` e_0 e_1, a bitwise-or operation `Ebvor` e_0 e_1, a logical left-shift operation `Ebvshl` e_0 e_1, or a logical right-shift operation `Ebvlshr` e_0 e_1. For arithmetic operations, there are `Ebvneg` e_0 for negation, `Ebvadd` e_0 e_1 for addition, `Ebvmul` e_0 e_1 for multiplication, `Ebvudiv` e_0 e_1 for unsigned division, and `Ebvurem` e_0 e_1 for unsigned remainder with e_0, e_1 : `exp`. Finally, the extraction `Eextract` i j e_0 and the concatenation `Econcat` e_0 e_1 operations have the data type `exp` with i, j : `nat` and e_0, e_1 : `exp`.

5.2 Semantics of SMT QF_BV Queries

In our Coq formalization, an SMT QF_BV query is interpreted on stores. A *store* is a mapping from QF_BV variables to bits. Let σ be a store. The interpretation of *be* : `bexp` on σ is a Boolean value; the interpretation of e : `exp` on σ is a bit vector. Semantic functions `eval_bexp` and `eval_exp` are as follows.

```
Fixpoint eval_bexp (be : bexp) (σ : store) : bool :=
  match be with
  | Bfalse => false
  | Btrue => true
  | Bnot be0 => ~~ (eval_bexp be0 σ)
  | Band be0 be1 => (eval_bexp be0 σ) && (eval_bexp be1 σ)
  | Bor be0 be1 => (eval_bexp be0 σ) || (eval_bexp be1 σ)
  | Bbveq e0 e1 => (eval_exp e0 σ) == (eval_exp e1 σ)
  | Bbvult e0 e1 => ltB (eval_exp e0 σ) (eval_exp e1 σ)
```

```
(* other QFBV predicates *)
end with eval_exp (e : exp) (σ : store) : bits :=
match e with
| Evar v => Store.acc v σ
| Econst bv => bv
| Ebvnot e0 => invB (eval_exp e0 σ)
| Ebvand e0 e1 => andB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvor e0 e1 => orB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvshl e0 e1 => shlB (to_nat (eval_exp e1 σ)) (eval_exp e0 σ)
| Ebvlshr e0 e1 => shrB (to_nat (eval_exp e1 σ)) (eval_exp e0 σ)
| Ebvneg e0 => negB (eval_exp e0 σ)
| Ebvadd e0 e1 => addB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvmul e0 e1 => mulB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvudiv e0 e1 => divB (eval_exp e0 σ) (eval_exp e1 σ)
| Ebvurem e0 e1 => remB (eval_exp e0 σ) (eval_exp e1 σ)
| Eextract i j e0 => extract i j (eval_exp e0 σ)
| Econcat e0 e1 => (eval_exp e1 σ) ++ (eval_exp e0 σ)
(* other QFBV operations *)
| Ebvsub e0 e1 => subB (eval_exp e0 σ) (eval_exp e1 σ)
end.
```

An SMT QF_BV query denotes a value in the Coq data type `bool`. `Bfalse` and `Btrue` denote `false` and `true` respectively. Boolean negation, conjunction, and disjunction correspond to `~~`, `&&`, and `||` in `bool` respectively. For QF_BV predicates, the bit-vector equality `Bbveq` is interpreted by the equality `==` for Boolean sequences. The `coq-nbits` function ltB is used to interpret `Bbvult`.

A QF_BV expression denotes a bit vector. For basic cases, QF_BV variables are interpreted by corresponding bit vectors in the store σ through the store access function `Store.acc`; bit vector constants are interpreted by themselves. Bitwise logical operations `Ebvnot`, `Ebvand`, and `Ebvor` are interpreted by corresponding `coq-nbits` functions invB, andB, and orB respectively. For logical shift operations, the offset e_1 is first converted to a natural number through to_nat (`eval_exp` e_1 σ) and then passed to the corresponding logical shift functions shlB or shrB in `coq-nbits`. QF_BV arithmetic operations are interpreted by corresponding `coq-nbits` arithmetic functions as expected. Finally, the extraction `Eextract` and concatenation `Econcat` operations are interpreted by extract and ++ in `coq-nbits` respectively.

In an SMT QF_BV query, a QF_BV variable designates a bit vector of a certain width. An SMT QF_BV query is hence associated with a *signature* Σ mapping QF_BV variables to their respective widths. A store σ *conforms* to a signature Σ if the interpretation of each QF_BV variable on σ has the same width as specified in Σ. Given an SMT QF_BV query *be* : `bexp` with its signature Σ, *be* is *satisfiable* if there is a store σ conforming to Σ and `eval_bexp` *be* σ = `true`.

5.3 Derived QF_BV Operations and Predicates

In the QF_BV logic of SMT-LIB, a number of QF_BV operations and predicates are derived from a small set of core operations and predicates. Consider the signed comparison predicate *bvslt bv cv* in SMT-LIB:

$$\begin{array}{lll} \mathit{bvslt}\ bv\ cv \triangleq (\mathit{or} & (\mathit{and} & (=\ (\mathit{extract}\ (w-1)\ (w-1)\ bv)\ \texttt{\#b1}) \\ & & (=\ (\mathit{extract}\ (w-1)\ (w-1)\ cv)\ \texttt{\#b0})) \\ & (\mathit{and} & (=\ (\mathit{extract}\ (w-1)\ (w-1)\ bv) \\ & & \quad (\mathit{extract}\ (w-1)\ (w-1)\ cv)) \\ & & (\mathit{bvult}\ bv\ cv))). \end{array}$$

To compare two bit vectors of width w in two's complement representation, the sign bits are checked. If bv is negative but cv is positive, *bvslt bv cv* is true. Otherwise, the signed predicate checks that both operands have the same sign and compares the operands using the unsigned comparison predicate. Interestingly, the arithmetic subtraction operation is actually a derived operation in SMT-LIB: $\mathit{bvsub}\ bv\ cv \triangleq \mathit{bvadd}\ bv\ (\mathit{bvneg}\ cv)$. The arithmetic operation is defined to be the bit-vector sum of minuend and the negation of subtrahend. It is *not*, for instance, defined as $\mathit{nat2bv}(w, \mathit{bv2nat}(bv) - \mathit{bv2nat}(cv))$ because $\mathit{bv2nat}(bv) - \mathit{bv2nat}(cv)$ may not be a natural number.

For derived operations and predicates, there is a subtle yet important difference between our formal semantics and those defined in SMT-LIB. In our formal bit-vector theory `coq-nbits`, most functions and predicates are defined directly. Particularly, the arithmetic subtraction function subB is defined by one-bit subtractors in `coq-nbits`. Our formal semantics for the QF_BV arithmetic operation *bvsub* therefore is defined by the corresponding bit-vector function subB. Since our formal semantics did not define *bvsub* by *bvadd* and *bvneg*, it could be different from those in SMT-LIB. In order to build a certified solver for the QF_BV logic of SMT-LIB, it is necessary to establish semantic equivalences between both semantic definitions for all derived QF_BV operations and predicates.

To justify our formal semantics, we show the semantics of our definitions and those of SMT-LIB indeed denote the same bit-vector functions or predicates. Consider again the subtraction operation. Recall the semantics of the arithmetic operations *bvadd* and *bvneg* are defined by the bit-vector functions addB and negB respectively. The next lemma is useful to show the semantic equivalence:

Lemma 6. $\forall bv\ cv, \mathsf{size}\ bv = \mathsf{size}\ cv \implies \mathsf{subB}\ bv\ cv = \mathsf{addB}\ bv\ (\mathsf{negB}\ cv)$.

For all derived QF_BV operations and predicates, we give Coq proofs for the equivalence between our formal semantics and those of SMT-LIB. Particularly, semantics of all QF_BV arithmetic operations and predicates over two's complement representation are equivalent to those in SMT-LIB. Our formal semantics for QF_BV queries is thus certified to be equivalent to SMT-LIB.

6 Certified Bit Blasting

Recall that a SAT query is a Boolean CNF formula. Given an SMT QF_BV query, a bit blasting algorithm computes a SAT query that is satisfiable if and only if the given SMT QF_BV query is satisfiable. Although it is the standard technique for solving SMT QF_BV queries, bit blasting can be very complex due to arithmetic operations and various optimizations. Bit blasting algorithms therefore can be tedious to construct and thus prone to errors. We verify a bit blasting algorithm for SMT QF_BV queries using our Coq formalization.

Let us start with a simple formalization of Boolean CNF formulae. In our formalization, a clause is represented by a sequence of literals; a CNF formula in turn is represented by a sequence of clauses. Let `bvar` be the data type for Boolean variables. We have the following data types in Coq:

```
Inductive lit : Set := Pos of bvar | Neg of bvar.
Definition clause : Set := seq lit.
Definition CNF : Set := seq clause.
```

Define an *environment* ϵ to be a mapping from `bvar` to `bool`. Given a literal ℓ, a CNF formula f, and an environment ϵ, it is straightforward to define the semantic functions `eval_lit` ℓ ϵ : `bool` and `eval_cnf` f ϵ : `bool`. A SAT query f is *satisfiable* if there is an environment ϵ such that `eval_cnf` f ϵ = `true`.

To illustrate how our Coq proof works, consider Tseitin transformation for the logical negation operation:

```
Definition bit_blast_Bnot ℓ : lit * CNF :=
  let r := a fresh literal in
  (r, [:: [:: r; ℓ]; [:: !r; !ℓ] ]).
```

Given a literal ℓ, `bit_blast_Bnot` ℓ returns a new literal r and the CNF formula $(r \vee \ell) \wedge (\neg r \vee \neg \ell)$. Tseitin transformation ensures the interpretations of ℓ and r are complementary on any environment ϵ evaluating the CNF formula to true. We give a formal proof using our formalization in Coq:

Lemma 7. $\forall r\ cnf\ \ell\ \epsilon, (r, cnf) =$ `bit_blast_Bnot` $\ell \implies$ `eval_cnf` $cnf\ \epsilon =$ `true` $\implies$ `eval_lit` $r\ \epsilon =$ ~~ (`eval_lit` $\ell\ \epsilon$).

The idea is generalized to QF_BV operations naturally. For each QF_BV operation, we construct a literal sequence $\vec{r}$ and a Boolean CNF formula *cnf*. If *cnf* evaluates to true on an environment ϵ, the interpretation of $\vec{r}$ on ϵ needs to reflect the semantics of the QF_BV operation. For instance, a Coq proof is given for the QF_BV addition operation:

Lemma 8. $\forall \vec{r}\ cnf\ \vec{\ell_0}\ \vec{\ell_1}\ \epsilon, (\vec{r}, cnf) =$ `bit_blast_Ebvadd` $\vec{\ell_0}\ \vec{\ell_1} \implies$ `eval_cnf` $cnf\ \epsilon$ = `true` $\implies$ `eval_lits` $\vec{r}\ \epsilon =$ `addB` (`eval_lits` $\vec{\ell_0}\ \epsilon$) (`eval_lits` $\vec{\ell_1}\ \epsilon$).

Given two literal sequences $\vec{\ell_0}$ and $\vec{\ell_1}$, `bit_blast_Ebvadd` $\vec{\ell_0}\ \vec{\ell_1}$ returns a literal sequence $\vec{r}$ and a CNF formula *cnf*. If *cnf* evaluates to true on an environment ϵ, then the interpretation of the literal sequence $\vec{r}$ on ϵ is indeed the bit-vector sum of the interpretations of $\vec{\ell_0}$ and $\vec{\ell_1}$ on ϵ. Bit blasting algorithms for other QF_BV operations are given and shown to reflect the semantics of corresponding functions defined in the bit-vector theory `coq-nbits`. Particularly, our bit blasting algorithms for arithmetic division and remainder correctly reflect corresponding arithmetic bit-vector functions in `coq-nbits`.

Recall that the semantics for SMT QF_BV queries is defined over stores for QF_BV variables. In order to prove the correctness of bit blasting algorithms, one has to relate stores for QF_BV variables with environments for Boolean variables. The relation is explicated through literal correspondences. A *literal correspondence* π is a mapping from QF_BV variables to sequences of literals. For each QF_BV variable v, the literal sequence $\pi(v)$ is meant to interpret v on environments for Boolean variables. More formally, let `eval_lits` $\vec{\ell}\ \epsilon$: `bits` be the bit vector for the literal sequence $\vec{\ell}$ interpreted on the environment ϵ. The bit vector `eval_lits` $\pi(v)\ \epsilon$ is hence the interpretation of the QF_BV variable v on the environment ϵ. Let σ be a store and π a literal correspondence. An environment ϵ is *consistent with* σ *through* π if the bit vectors `eval_lits` $\pi(v)\ \epsilon$ and `Store.acc` $v\ \sigma$ are equal for every QF_BV variable v in σ. Thus, an environment is consistent with a store if their interpretations of variables coincide.

It is now straightforward to give our bit blasting algorithm for SMT QF_BV queries. For each QF_BV expression, our algorithm first computes literals and CNF formulae for operands recursively. It then invokes an auxiliary bit blasting algorithm to construct result literals and a CNF formula for the QF_BV operation. The literal correspondence is also updated when literals are allocated for QF_BV variables. Finally, the result literals and the updated literal correspondence are returned along with the concatenation of all CNF formulae.

```
Definition bit_blast_bexp Σ π b : lit * correspondence * CNF :=
match be with
| Bnot be₀ =>
    let (r₀, π', cnf₀) := bit_blast_bexp Σ π be₀ in
    let (r, cnf) := bit_blast_Bnot r₀ in
    (r, π', cnf ++ cnf₀)
(* other QFBV predicates *)
end with bit_blast_exp Σ π e : seq lit * correspondence * CNF :=
match e with
| Evar v =>
    if π(v) is defined then (π(v), π, [::])
    else let r⃗ := fresh literals for v according to Σ in
         let π' := update π with v ↦ r⃗ in
         (r⃗, π', [::])
| Ebvadd e₀ e₁ =>
    let (r⃗₀, π', cnf₀) := bit_blast_exp Σ π e₀ in
    let (r⃗₁, π'', cnf₁) := bit_blast_exp Σ π' e₁ in
    let (r⃗, cnf) := bit_blast_Ebvadd r⃗₀ r⃗₁ in
    (r⃗, π'', cnf ++ cnf₀ ++ cnf₁)
(* other QFBV operations *)
end.
```

The following Coq theorem establishes the connection between the output literals and the input SMT QF_BV query or expression of the algorithm.

Theorem 1. *Let be* : `bexp` *be an* SMT QF_BV *query with the signature* Σ_{be}, e : `exp` *a* QF_BV *expression with the signature* Σ_e, *and* π_0 *the empty literal correspondence.*

1. *$\forall r\ \pi\ cnf\ \sigma\ \epsilon, (r, \pi, cnf) =$* `bit_blast_bexp` *$\Sigma_{be}\ \pi_0\ be \implies \sigma$ conforms to Σ_{be} $\implies \epsilon$ is consistent with σ through $\pi \implies$* `eval_cnf` *$cnf\ \epsilon =$* `true` *$\implies$* `eval_lit` *$r\ \epsilon =$* `eval_bexp` *$be\ \sigma$.*
2. *$\forall \vec{r}\ \pi\ cnf\ \sigma\ \epsilon, (\vec{r}, \pi, cnf) =$* `bit_blast_exp` *$\Sigma_e\ \pi_0\ e \implies \sigma$ conforms to Σ_e $\implies \epsilon$ is consistent with σ through $\pi \implies$* `eval_cnf` *$cnf\ \epsilon =$* `true` *$\implies$* `eval_lits` *$\vec{r}\ \epsilon =$* `eval_exp` *$e\ \sigma$.*

Let *be* be an SMT QF_BV query with the signature Σ_{be}, r and *cnf* the literal and CNF formula returned by `bit_blast_bexp` respectively. Consider any store conforming to Σ_{be} and any environment consistent with the store. If the environment evaluates the formula *cnf* to true, Theorem 1 says that the literal r and the SMT QF_BV query *be* evaluate to the same Boolean value on the environment and store respectively. In other words, the algorithm `bit_blast_bexp` is a generalized Tseitin transformation for SMT QF_BV queries. Particularly, all QF_BV arithmetic operations (addition, subtraction, multiplication, division, and remainder in the unsigned and two's complement representations) are transformed to CNF formulae with formal proofs of correctness in Coq.

A useful corollary to Theorem 1 is the reduction of the satisfiability of SMT QF_BV queries to the satisfiability of SAT queries.

Corollary 1. *Let be* : `bexp` *be an* SMT QF_BV *query with the signature Σ_{be} and π_0 the empty literal correspondence. Then*

$$\forall r\ \pi\ cnf, (r, \pi, cnf) = \texttt{bit\_blast\_bexp}\ \Sigma_{be}\ \pi_0\ be \implies$$
$$[(\exists \sigma, \sigma \textit{ conforms to } \Sigma_{be} \wedge \texttt{eval\_bexp}\ be\ \sigma = \texttt{true}) \iff$$
$$(\exists \epsilon, \texttt{eval\_cnf}\ ([::\ [::\ r]]\ \texttt{++}\ cnf)\ \epsilon = \texttt{true})].$$

Corollary 1 gives the formal proof of correctness for our bit blasting algorithm `bit_blast_bexp`. Let *be* be an arbitrary SMT QF_BV query, r and *cnf* the literal and the CNF formula returned by the algorithm. The corollary shows that the query *be* is satisfiable if and only of the SAT query $r \wedge cnf$ is satisfiable. An equi-satisfiable SAT query is indeed obtained from the bit blasting algorithm on every input SMT QF_BV query with a formal proof of correctness.

Recall that several QF_BV operations and predicates are derived from a small number of operations and predicates in SMT-LIB. A naïve bit blasting algorithm could expand derived operations or predicates, and then perform bit blasting on a small set of operations and predicates. Such an algorithm would have a simpler proof of correctness but generate more intermediate literals and clauses. For instance, the naïve algorithm for *bvsub* would perform bit blasting on *bvneg* followed by *bvadd* with intermediate literals and clauses. Our bit blasting algorithm for *bvsub* on the other hand reflects our semantics defined by the bit-vector function subB. Intermediate literals or clauses are not needed. Our bit blasting algorithm hence transforms *bvsub* more economically than the naïve algorithm.

To improve our bit blasting algorithm further, a cache for QF_BV expressions and predicates is added. In large queries, QF_BV expressions and predicates can occur a number of times. If a QF_BV expression has several occurrences, our basic bit blasting algorithm will generate result literals and CNF formulae for each occurrence. Consider the SMT QF_BV query

$$(\mathit{and}\ (\mathit{bvslt}\ \texttt{\#b1000}\ (\mathit{bvadd}\ x\ y))\ (\mathit{bvslt}\ (\mathit{bvadd}\ x\ y)\ \texttt{\#b0111})).$$

The query checks whether the sum of the QF_BV variables x and y can be in a proper range. Since the Boolean predicate *and* has two operands, our basic algorithm invokes the auxiliary bit blasting algorithm for the two comparison predicates. It in turn blasts the same expression *bvadd* x y twice. Repeated bit blasting on the same expression or predicate is redundant. A hash function can detect repeated QF_BV expressions and predicates easily. When an expression or a predicate recur, the previously computed literals with the empty CNF formula are returned from a cache as the result. More importantly, we give a formal Coq proof of Corollary 1 for the bit blasting algorithm with a cache.

7 A Certified SMT QF_BV Solver

We have so far built a formally verified bit blasting algorithm for SMT QF_BV queries. Using the code extraction mechanism in Coq, an OCaml program corresponding to the verified bit blasting algorithm is obtained. Using a SAT solver and a SAT certificate checker, a certified SMT QF_BV solver can be constructed. Figure 1 gives the flow of our certified solver.

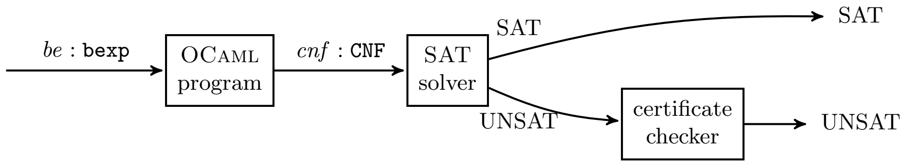

Fig. 1: Certified SMT QF_BV Solver

In the figure, the extracted OCaml program takes an OCaml expression *be* of the type `bexp` as an input (Section 5). The verified program performs bit blasting on the SMT QF_BV query and returns an OCaml expression *cnf* of the type `lit list list` representing a SAT query (Section 6). Precisely, an OCaml term of the type `lit` represents a literal. The OCaml type `lit list` corresponds to the data type for clauses; and the type `lit list list` corresponds to the data type for CNF formulae. The expression *cnf* is sent to a SAT solver to check satisfiability. If the SAT solver reports SAT, the SMT QF_BV query represented by *be* is satisfiable. Otherwise, the SAT solver reports UNSAT with a certificate. The certificate is sent to a SAT certificate checker for validation. If it is validated, the SMT QF_BV query *be* is unsatisfiable with certification.

8 Experiments

In order to evaluate the performance of our verified OCaml bit blasting program, we instantiate our SMT QF_BV solver CoqQFBV based on Figure 1 as follows. We write an OCaml parser to translate a text file in the SMT-LIB format to an SMT QF_BV query in our formal syntax. The query is sent to the verified OCaml program for bit blasting. We then add an OCaml program to transform the output SAT query to a text file in the DIMACS format. The 2020 SAT Competition winner Kissat [5] is used to check the satisfiability of the SAT query. If the SAT solver reports UNSAT with a certificate in the DRAT format [31], the certificate is sent to the verified certificate checker GratChk [16] for validation. Certificate checkers for SAT solvers use much simpler algorithms than certificate checkers for SMT solvers. They are hence easier to build and prove correct. The correctness of GratChk is in fact verified by the proof assistant Isabelle [22]. We need not trust the certificate checker either.

We ran two experiments to evaluate our certified SMT QF_BV solver. The first experiment is the QF_BV division of the single query track in the 2020 SMT Competition [2]. The second experiment consists of verification problems from various assembly implementations for linear field arithmetic in cryptography libraries such as OpenSSL [30], RELIC [1], and BLST [29]. We compare CoqQFBV against three SMT QF_BV solvers: CVC4 [4] with an LFSC certificate checker [27], the 2020 SMT QF_BV division winner Bitwuzla [20], and the 2019 SMT QF_BV division winner Boolector [21]. Bitwuzla and Boolector are designed for efficiency without certification. CVC4 provides an LFSC certificate checker implemented in C [26]. The certificate checker can validate certificates from different theories but is itself not verified. All experiments were run on a Linux machine with a 3.20GHz CPU and 1TB memory.

8.1 SMT QFBV Competition

The first experiment is running our certified solver CoqQFBV on tasks from the QF_BV division of the 2020 SMT Competition. We set 60GB memory limit and 20 minutes timeout for each task as in the competition. A task solves a single SMT-LIB file sequentially. The SMT QF_BV division contains 6861 files in the SMT-LIB format. All files are marked with *unsat*, *sat*, or *unknown* indicating expected query results. To save running time, we ran 10 tasks concurrently. The experimental results are summarized in Table 1.

In the table, the column N_{SC} indicates the number of solved tasks with certification. O_{SC} is the number of timeouts. E_{SC} shows the number of unsolved tasks due to tool errors. T_{SC} is the average time for solved tasks. CoqQFBV solves 6087 (88.72%) and CVC4 with its certificate checker solves 3840 (55.97%) with certification. We observe three stack overflow errors during bit blasting in CoqQFBV. These errors are induced by deep recursion. Among 328 errors from CVC4, 249 are segmentation faults raised by the LFSC certificate checker.

The same table also compares against efficient but uncertified solvers. To evaluate the overhead from certificate checking, the two certified solvers Co-

Table 1: Experimental Results on the 2020 SMT QF_BV Division

Tool	N_{SC}	O_{SC}	E_{SC}	T_{SC}	N_S	O_S	E_S	T_S
CoqQFBV	6087 (88.72%)	771	3	119.69	6169 (89.91%)	689	3	81.74
CVC4	3840 (55.97%)	2693	328	74.63	4255 (62.02%)	2544	62	56.87
Bitwuzla	-	-	-	-	6739 (98.22%)	122	0	16.09
Boolector	-	-	-	-	6719 (97.93%)	142	0	15.44

Table 2: Experimental Results on the 2020 SMT QF_BV Division by Categories

Tool	N_{SC}	T_{SC}	P_{SU}	N_S	T_S
4238 *unsat* tasks					
CoqQFBV	3838 (90.56%)	146.72	291.35 MB	3920 (92.50%)	86.51
CVC4	1762 (41.58%)	86.68	266.61 MB	2177 (51.37%)	49.68
Bitwuzla	- -	-	-	4188 (98.82%)	12.75
Boolector	- -	-	-	4180 (98.63%)	11.72
2553 *sat* tasks					
CoqQFBV	- -	-	-	2242 (87.82%)	73.26
CVC4	- -	-	-	2078 (81.39%)	64.41
Bitwuzla	- -	-	-	2524 (98.86%)	21.08
Boolector	- -	-	-	2516 (98.55%)	21.31
70 *unknown* tasks					
CoqQFBV	5 (7.14%)	173.17	203.52 MB	7 (10.00%)	128.26
CVC4	- -	-	-	0 (0.00%)	-
Bitwuzla	- -	-	-	27 (38.57%)	66.36
Boolector	- -	-	-	23 (32.86%)	48.58

qQFBV and CVC4 still generate certificates but do not validate them. The column N_S gives the number of solved tasks without certification. O_S is the number of timeouts. E_S indicates the number of errors, and T_S is the average time for solved tasks. Our certified solver CoqQFBV finishes 6169 (89.91%) tasks. The CVC4 solver finishes 4255 (62.02%) tasks. CoqQFBV and CVC4 solve $82(= 6169 - 6087)$ and $415(= 4255 - 3840)$ more tasks without certification respectively. Since our bit blasting algorithm is verified for all inputs, CoqQFBV does not certify bit blasting on each query and hence induces less overhead. The 2020 and 2019 SMT QF_BV division winners Bitwuzla and Boolector finish 6739 (98.22%) and 6719 (97.93%) tasks without certification respectively. CoqQFBV solves about 10% less tasks with certification than the 2020 track winner Bitwuzla without certification. It also performs significantly better than CVC4 with a general SMT certificate checker.

Table 2 compares the four solvers by tasks from the three expected query results. Among the 4238 *unsat* tasks, CoqQFBV and CVC4 give certified answers to 3838 (90.56%) and 1762 (41.58%) of them respectively. The column P_{SU} gives the average size of certificates. Efficient solvers Bitwuzla and Boolector give 4188 (98.82%) and 4180 (98.63%) uncertified answers respectively.

Among the 2553 *sat* tasks, Bitwuzla and Boolector finish 2524 (98.86%) and 2516 (98.55%) of them respectively. CoqQFBV and CVC4 solve only 2242 (87.82%) and 2078 (81.39%) *sat* tasks respectively. For the 70 tasks marked *unknown*, Bitwuzla and Boolector respectively answer 27 (38.57%) and 23 (32.86%) of them without certification. Our certified SMT QF_BV solver finds two *sat* and five *unsat* tasks. Answers to the five *unsat* tasks are all certified. CVC4 with its certificate checker fails to solve any *unknown* task. For the benchmarks from the 2020 SMT QF_BV division, our certified solver CoqQFBV appears to be more scalable than CVC4 with its general SMT certificate checker.

Table 3: Average Time for CoqQFBV Components

Task	T_{BB}	T_{SAT}	T_{Cert}
unsat	41.84	49.92	73.51
sat	37.08	62.09	-
unknown	32.34	121.99	62.86

Table 3 further decomposes the time spent on different components in CoqQFBV. The column T_{BB} gives the average time for our verified OCaml bit blasting program; T_{SAT} gives the average time used by the SAT solver Kissat; and T_{Cert} contains the average time for the certificate checker GratChk. For the tasks in the QF_BV division, the time for SAT solving and certificate checking are comparable. In comparison, the OCaml bit blasting program seems to take an unexpectedly large amount of time and hence can still be improved.

8.2 Linear Field Arithmetic in Cryptography

In this section, we evaluate our certified SMT QF_BV solver on benchmarks from real-world assembly implementations in various cryptography libraries such as OpenSSL [30], RELIC [1], and BLST [29]. In elliptic curve cryptography, arithmetic operations over large finite fields are needed. A field element is typically represented by hundreds of bits. A field arithmetic operation takes two field elements and returns a field element as the result. In the signature scheme Ed25519 used in OpenSSH, for instance, a field element belongs to the residue system modulo the prime number $2^{255} - 19$. Field sum of two field elements is obtained by the arithmetic sum modulo $2^{255} - 19$. Commodity processors however do not support arithmetic instructions with operands in hundreds of bits natively. Field arithmetic has to be implemented by 32- or 64-bit instructions. The functional specification of the field addition used in Ed25519 may look as

Table 4: Experimental Results on Cryptographic Assembly Program Verification

Tool	N_{SC}	T_{SC}	P_{SU}	N_S	T_S
CoqQFBV	93 (96.88%)	121.42	168.45 MB	93 (96.88%)	68.96
CVC4	19 (19.79%)	6.66	267.92 MB	46 (47.92%)	40.16
Bitwuzla	- -	-	-	88 (91.67%)	16.07
Boolector	- -	-	-	96 (100.00%)	18.25

follows.

$$\{\textstyle\sum_{i=0}^{3} a_i \times 2^{64\times i} < 2^{255} - 19 \land \sum_{i=0}^{3} b_i \times 2^{64\times i} < 2^{255} - 19\}$$
$$\texttt{x25519\_fe64\_add}(r_0, r_1, r_2, r_3, a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$$
$$\left\{ \begin{array}{c} \sum_{i=0}^{3} r_i \times 2^{64\times i} \equiv \sum_{i=0}^{3} a_i \times 2^{64\times i} + \sum_{i=0}^{3} b_i \times 2^{64\times i} (\mathrm{mod}\ 2^{255} - 19) \\ \land \\ \sum_{i=0}^{3} r_i \times 2^{64\times i} < 2^{255} - 19 \end{array} \right\}$$

Let a_i, b_i, c_i be 64-bit variables (registers) for $0 \leq i \leq 3$. The specification says that the output field element represented by r_i's computed by the program `x25519_fe64_add` is the field arithmetic sum of the input elements represented by a_i's and b_i's. In finite field arithmetic programs, over- or under-flow in assembly instructions lead to incorrect results, and bit-accurate program verification is required. We obtain 46 implementations and generate 96 SMT QF_BV queries from verification conditions in order to evaluate our certified solver in this experiment.

Table 4 shows the verification results with the same memory and time limits in the 2020 SMT Competition. All SMT QF_BV queries are expected to be unsatisfiable. Boolector successfully solves all queries (100%) without certification. The 2020 QF_BV track winner Bitwuzla finishes 88 queries (91.67%) without certification. Surprisingly, CoqQFBV gives certified answers to 93 queries (96.88%). The verified SAT certificate checker GratChk used in CoqQFBV successfully validates all certificates for the real-world cryptographic program verification problems. In comparison, CVC4 solves 46 queries (47.92%) but certifies only 19 (19.79%). The CVC4 certificate checker raises segmentation faults on the 27 (= 46 − 19) solved but uncertified queries. These certificates are perhaps too complicated to be validated by the unverified LFSC certificate checker. For the SMT QF_BV queries from real-world program verification problems, our certified solver CoqQFBV seems to perform slightly better than the efficient but uncertified SMT QF_BV solver Bitwuzla. Our certified solver is probably scalable enough for certain bit-accurate program verification problems.

9 Conclusion

We combine algorithm design with interactive theorem proving to build a scalable certified SMT QF_BV solver CoqQFBV in this work. Our certified solver employs a verified OCaml bit blasting program and the verified certificate checker GratChk to improve the confidence in SMT QF_BV query results. Experiments on the QF_BV division of the 2020 SMT Competition and real-world cryptographic program verification suggest that CoqQFBV is useful.

For future work, we plan to specify and verify more heuristics to further optimize CoqQFBV. Particularly, cryptographic program verification requires more sophisticated range checks. More verified bit blasting algorithms for such checks will undoubtedly improve the confidence of bit-accurate program verification.

Acknowledgements We thank all the anonymous reviewers for their insightful comments and suggestions. We thank the authors of SSReflect for its powerful language and libraries. We would like to give special thanks to Prof. Moshe Vardi for his encouragement. The work is supported by National Natural Science Foundation of China under the Grant Numbers 62002228, 61802259 and 61836005; the Guangdong Science and Technology Department under the Grant Number 2018B010107004; Ministry of Science and Technology of Taiwan under the Grant Numbers MOST108-2221-E-001-010-MY3 and MOST108-2221-E-001-009-MY2; Academia Sinica for Sinica Investigator Award AS-IA-109-M01.

References

1. Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`
2. Barbosa, H., Hoenicke, J., Hyvarinen, A.: International Satisfiability Modulo Theories Competition (SMT-COMP). `https://smt-comp.github.io/2020/` (2020)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf`
4. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 171–177. Springer (2011)
5. 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.) SAT Competition 2020 - Solver and Benchmark Descriptions. B, vol. B-2020-1, pp. 50–53. University of Helsinki (2020)
6. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Jouannaud, J., Shao, Z. (eds.) Certified Programs and Proofs (CPP). LNCS, vol. 7086, pp. 183–198. Springer (2011)
7. Brummayer, R., Biere, A.: Fuzzing and delta-debugging smt solvers. In: Dutertre, B., Strichman, O. (eds.) Satisfiability Modulo Theories (SMT). pp. 1–5. ACM (2009)
8. Chen, Y.F., Hsu, C.H., Lin, H.H., Schwabe, P., Tsai, M.H., Wang, B.Y., Yang, B.Y., Yang, S.Y.: Verifying Curve25519 software. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM Computer and Communications Security (CCS). pp. 299–309. ACM (2014)

9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 2988, pp. 168–176. Springer (2004)
10. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Chechik, S.B.M. (ed.) Verified Software. Theories, Tools, and Experiments. pp. 56–72. LNCS, Springer (2016)
11. Dross, C., Fumex, C., Gerlach, J., Marché, C.: High-Level Functional Properties of Bit-Level Programs: Formal Specifications and Automated Proofs. Research Report RR-8821, INRIA Saclay (Dec 2015), `https://hal.inria.fr/hal-01238376`
12. Ekici, B., Katz, G., Keller, C., Mebsout, A., Reynolds, A.J., Tinelli, C.: Extending SMTCoq, a certified checker for SMT (extended abstract). Electronic Proceedings in Theoretical Computer Science **210**, 21–29 (Jun 2016)
13. Fox, A.C.J.: LCF-style bit-blasting in HOL4. In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving (ITP). LNCS, vol. 6898, pp. 357–362. Springer (2011)
14. Hadarean, L., Barrett, C., Andrew Reynolds, C.T., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (LPAR). pp. 340–355. Springer (2015)
15. Kennedy, A., Benton, N., Jensen, J.B.., Dagand, P.E.: Coq: The world's best macro assembler? In: Schrijvers, T. (ed.) Principles and Practice of Declarative Programming (PPDP). pp. 13–24. ACM (2013)
16. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura, L. (ed.) International Conference on Automated Deduction (CADE). LNCS, vol. 10395, pp. 237–254. Springer (2017)
17. Lochbihler, A.: Fast machine words in Isabelle/HOL. In: Avigad, J., Mahboubi, A. (eds.) Interactive Theorem Proving (ITP). LNCS, vol. 10895, pp. 388–410. Springer (2018)
18. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In: Devanbu, P., Cohen, M., Zimmermann, T. (eds.) ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 701–712. ACM (2020)
19. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008), `http://ceur-ws.org/Vol-418/paper10.pdf`
20. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), `https://arxiv.org/abs/2006.01621`
21. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 53–58 (2014)
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
23. Oe, D., Reynolds, A., Stump, A.: Fast and flexible proof checking for SMT. In: Dutertre, B., Strichman, O. (eds.) Satisfiability Modulo Theories (SMT). pp. 6–13. ACM (2009)

24. Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., Barrett, C.: DRAT-based bit-vector proofs in CVC4. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 11628, pp. 298–305. Springer (July 2019)
25. Polyakov, A., Tsai, M.H., Wang, B.Y., Yang, B.Y.: Verifying arithmetic assembly programs in cryptographic primitives. In: Schewe, S., Zhang, L. (eds.) Concurrency Theory (CONCUR). pp. 4:1–4:16. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
26. Reynolds, A., Stump, A.: LFSC checker. `https://github.com/CVC4/LFSC`
27. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**, 91–118 (2013)
28. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. In: Hardin, D., Schmaltz, J. (eds.) The ACL2 Theorem Prover and its Applications (ACL2). EPTCS, vol. 70, pp. 84–102 (2011)
29. The blst Developers: The blst BLS12-381 signature library. `https://github.com/supranational/blst`
30. The OpenSSL Project: The OpenSSL repository. `https://github.com/openssl/openssl`
31. Wetzler, N., Heule, M.J., Hunt Jr., W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429. Springer (2014)