An Efficient Floating-Point Bit-Blasting API for Verifying C Programs

Mikhail R. Gadelha[1] , Lucas C. Cordeiro[2] , and Denis A. Nicole[3]

[1] SIDIA Instituto de Ciência e Tecnologia, Manaus, Brazil
mikhail.gadelha@sidia.com
[2] University of Manchester, Manchester, UK
lucas.cordeiro@manchester.ac.uk
[3] University of Southampton, UK,
dan@ecs.soton.ac.uk

Abstract. We describe a new SMT bit-blasting API for floating-points and evaluate it using different out-of-the-shelf SMT solvers during the verification of several C programs. The new floating-point API is part of the SMT backend in ESBMC, a state-of-the-art bounded model checker for C and C++. For the evaluation, we compared our floating-point API against the native floating-point APIs in Z3 and MathSAT. We show that Boolector, when using floating-point API, outperforms the solvers with native support for floating-points, correctly verifying more programs in less time. Experimental results also show that our floating-point API implemented in ESBMC is on par with other state-of-the-art software verifiers. Furthermore, when verifying programs with floating-point arithmetic, our new floating-point API produced no wrong answers.

Keywords: Floating-Point Arithmetic · Satisfiability Modulo Theories · Software Verification.

1 Introduction

Software verification tools operate by converting their input (e.g., a program source code) into a format understandable by an automated theorem prover, encoding high-level program properties (e.g., arithmetic overflow) and algorithms (e.g., bounded model checking) into low-level equations (e.g., SMT). The encoding process of a program usually involves several intermediate steps, designed to generate a formula that can be efficiently solved by the theorem provers. In this domain, the analysis of programs with floating-point arithmetic has received much attention, primarily when safety depends on the correctness of these programs. In essence, the Ariane 5 rocket exploded mid-air in 1996 due to an exception thrown by an invalid floating-point conversion [42]. It is a complex problem because the semantics may change beyond code level, including the optimization performed by compilers [46].

There exist various static analysis tools that are able to check for floating-point computations [9,10,21,22,30,52,56]. For example, Astrée is a static analysis

tool that considers all possible rounding errors when verifying C programs with floating-point numbers [9]. It has been applied to verify embedded software in the flight control software of the Airbus. CBMC [21] is also another notable example of a software model checking tool, which implements a bit-precise decision procedure for the theory of floating-point arithmetic [11]. It has been applied to verify industrial applications from the automotive industry, which rely on floating-point reasoning [55]. CBMC is also the main verification engine employed by other software verifiers that efficiently verify C programs with floating-point numbers such as PeSCo [53] and VeriAbs [19]. It is a challenging verification task to prove the correctness of C programs with floating-points mainly because of 32/64 bits floating-point computations. Given the current knowledge in software verification, there exists no other study that shows a thorough comparative evaluation of software verifiers and SMT solvers concerning the verification of C programs that contain floating-points.

Here we present the new floating-point technologies developed in one bounded model checker, ESBMC [31], and evaluate it using a large set of floating-point benchmarks [7]. In particular, we evaluate a new floating-point API on top of our SMT backend that extends the floating-point feature to all solvers supported by ESBMC (including Boolector [14] and Yices [26] that currently do not support the SMT FP logic [13]). For evaluation, we used the benchmarks of the 2020 International Competition on Software Verification (SV-COMP) [7], from the floating-point sub-category. The five different solvers supported by ESBMC were evaluated (Z3 [25], Yices [26], Boolector [14], MathSAT [20], and CVC4 [4]) and ESBMC is able to evaluate more benchmarks within the usual time and memory limits (15 minutes and 15GB, respectively) when using Boolector. In particular, results show that Boolector can solve more floating-point problems using the new floating-point API than MathSAT or Z3, which have native floating-point APIs. Our experimental results also show that our floating-point API implemented in ESBMC is competitive to other state-of-the-art software verifiers, including CBMC [21], PeSCo [53], and VeriAbs [19].

2   Floating-point Arithmetic

The manipulation of real values in programs is a necessity in many fields, e.g., scientific programming [46]. The set of real numbers, however, is infinite, and some numbers cannot be represented with finite precision, e.g., irrational numbers. Over the years, computer manufacturers have experimented with different machine representations for real numbers [34]. The two fundamental ways to encode a real number are the fixed-point representation, usually found in embedded microprocessors and microcontrollers [28], and the floating-point representation, in particular, the IEEE floating-point standard (IEEE 754-2008 [36]), which has been formally accepted by many processors [35].

Each encoding can represent a range of real numbers depending on the word-length and how the bits are distributed. A fixed-point representation of a number consists of an integer component, a fractional component, and a bit for the sign.

In contrast, the floating-point representation consists of an exponent component, a significand component, and a bit for the sign. Floating-point has a higher dynamic range than fixed-point (e.g., a `float` in C has 24 bits of precision, but can have values up to 2^{128}), while fixed-point can have higher precision than floating-point [49]. Furthermore, the IEEE floating-point standard defines values that have no equivalent in a fixed-point or real encoding, e.g., positive and negative infinities. In general, IEEE floating-points are of the following kinds: *zeroes*, *NaNs*, *infinities*, *normal*, *denormal (or subnormal)* [36].

Definition 1 *(Infinities) Both* `+inf` *and* `-inf` *are defined in the standard. These floating-points represent overflows or the result of non-zero floating-point divisions by zero (Annex F of the C language specification [38]).*

Definition 2 *(Zeroes) Both* `+0` *and* `-0` *are defined in the standard. Most of the operations will behave identically when presented with* `+0` *or* `-0` *except when extracting the sign bit or dividing by zero (usual rules about signedness apply and will result in either* `+inf` *or* `-inf`*). Equalities will even be evaluated to true when comparing positive against negative zeroes.*

Definition 3 *(NaNs) The* **N**ot **a** **N**umber *special values represent undefined or unrepresentable values, e.g.,* $\sqrt{-1}$ *or* `0.f/0.f`*. As a safety measure, most of the operations will return NaN if at least one operator is NaN, as a way to indicate that the computation is invalid. NaNs are not comparable: except for the not equal operator (*`!=`*), all other comparisons will evaluate to false (even comparing a NaN against itself). Furthermore, casting NaNs to integers is undefined behavior.*

Definition 4 *(Normal) A non-zero floating-point that can be represented within the range supported by the encoding.*

Definition 5 *(Denormal (or subnormal)) A non-zero floating-point representing values very close to zero, filling the gap between what can be usually represented by the encoding and zero.*

The IEEE standard also defines five kinds of exceptions, to be raised under specific conditions, which are: *invalid operation*, *overflow*, *division by zero*, *underflow*, and *inexact* [36].

Exception 1 *(Invalid Operation) This exception is raised when the operation produces a NaN as a result.*

Exception 2 *(Overflow) This exception is raised when the result of an operation is too large to be represented by the encoding. By default, these operations return* $\pm$`inf`*.*

Exception 3 *(Division By Zero) It is raised by* `x/`$\pm$`0`*, for* `x`$\neq$`0`*. By default, these operations return* $\pm$`inf`*.*

Exception 4 *(Underflow) Raised when the result is too small to be represented by the encoding. The result is a denormal floating-point.*

Exception 5 *(Inexact) This exception is raised when the encoding cannot represent the result of an operation unless it is rounded. By default, these operations will round the result.*

The standard defines five rounding modes. Given a real number x, a rounded floating-point $r(x)$ will be rounded using: *Round Toward Positive (RTP)*, *Round Toward Negative (RTN)*, *Round Toward Zero (RTZ)*, *Round to Nearest ties to Even (RNE)*, and *Round to Nearest ties Away from zero (RNA)*:

Mode 1 *(RTP) $r(x)$ is the least floating-point value $\geq x$.*

Mode 2 *(RTN) $r(x)$ is the greatest floating-point value $\leq x$.*

Mode 3 *(RTZ) $r(x)$ is the floating-point with the same sign of x, such that $|r(x)|$ is the greatest floating-point value $\leq |x|$.*

Mode 4 *(RNE) $r(x)$ is the floating-point value closest to x; if two floating-point values are equidistant to x, $r(x)$ is the one which the least significant bit is zero.*

Mode 5 *(RNA) $r(x)$ is the floating-point value closest to x; if two floating-point values are equidistant to x, $r(x)$ is the one further away from zero.*

The standard also defines some arithmetic operations (add, subtract, multiply, divide, square root, fused multiply-add, remainder), conversions (between formats, to and from strings), and comparisons and total ordering. In particular, the standard defines how floating-point operations are to be encoded using bit-vectors. Table 1 shows four primitive types usually available in the x86 family of processors that follow the standard; each type is divided into three parts: one bit for the sign, an exponent, and a significant part which depends on the bit length of the type. The significands also include a hidden bit: a 1 bit that is assumed to be the leading part of the significand, unless the floating-point is denormal.

Name	Common Name	Size (exponent + significand)
fp16	Half precision	16 (5 + 10)
fp32	Single precision	32 (8 + 23)
fp64	Double precision	64 (11 + 53)
fp128	Quadruple precision	128 (15 + 113)

Table 1: IEEE floating-point types.

In Annex F of the C language specification [38], fp32 and fp64 are defined as `float` and `double`. The standard does not define any types for fp16, and compilers usually implement two formats: `__fp16` as defined in the ARM C language extension (ACLE) [2] and `_Float16` as defined by the ISO/IEC 18661-3:2015

standard [39]. While `__fp16` is only a storage and interchange format (meaning that it is promoted when used in arithmetic operations), `_Float16` is an actual type, and arithmetic operations are performed using half-precision. The standard only weakly specifies how an fp128 (`long double` in C) should be implemented, and compilers usually implement it using an 80-bit long double extended precision format [35].

Floating-points are represented as $(-1)^{sign} \times significand \times 2^{exponent}$. Here, $1 \leq significand \leq 2$ and $2^{exponent}$ is the scaling factor [51]. Regular floating-points are encoded assuming that the leading hidden bit is 1 and the exponent is in the range $[-exponent_{max} + 1, exponent_{max}]$, e.g., the number 0.125 is represented as $\langle 0011000000000000 \rangle$ in the floating-point format. Denormals are encoded assuming that the leading hidden bit is zero and the exponent is $-exponent_{max}$. Zeros are represented as an all-zero bit-vector (except for the sign bit if the zero is negative). Finally, a bit-vector with the exponent equal to $exponent_{max}$ and significand all zero is an infinity. In contrast, a bit-vector with an exponent equal to $exponent_{max}$ and significand not zero is a NaN.

3   A Floating-Point Bit-Blasting API for Verifying C Programs

When ESBMC was created, all floating-point types and operations were encoded using fixed-points [1,5,6,18,37]. A fixed-point number is represented in ESBMC as a pair (m, n) where m is the total number of bits and $n \leq m$ is the number of fractional bits, e.g., the number 0.125 is represented as $\langle 0000.0010 \rangle$ (assuming it is 8 bits long) in the fixed-point format. The fixed-point arithmetic is performed similarly to the bit-vector arithmetic, except that the operations are applied separately to the integral and fractional parts of the fixed-points and concatenated at the end (overflow in the fractional parts are treated accordingly). Different from floating-points, all bit-vectors represent one number in the real domain.

The lack of proper floating-point encoding, however, meant that ESBMC was unable to accurately verify an entire class of programs, such as the famous floating-point "issue" [57] illustrated in Figure 1.

The assertion in line 7 holds if the program is encoded using fixed-point arithmetic, but fails if floating-point arithmetic is used. The assertion violation arises from the fact that floating-points in the IEEE standard are represented as whole numbers × a power of two, so the only numbers that use a prime factor of the base two that can be correctly expressed as fractions. Since in binary (or base 2) the only prime factor is 2, only $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ... would be correctly expressed as decimals, so the constants 0.1, 0.2, 0.3 (or $\frac{1}{10}$, $\frac{1}{5}$, $\frac{1}{3}$) in the program are only approximations. In the program in Figure 1, the constants are:

- `x` is 0.1000000000000000055511151231257827021181583404541015625
- `y` is 0.200000000000000011102230246251565404236316680908203125
- `w` is 0.3000000000000000444089209850062616169452667236328125
- `z` is 0.299999999999999988897769753748434595763683319091796875

```
1  int  main ( )
2  {
3    double  x  =  0 . 1 ;
4    double  y  =  0 . 2 ;
5    double  w  =  0 . 3 ;
6    double  z  =  x  +  y ;
7    assert ( w  ==  z ) ;
8    return  0 ;
9  }
```

Fig. 1: The assertions in line 7 does not hold when using floating-point arithmetic.

The discrepancy happens in the C program because the closest floatint-point to `0.3` is smaller than the real `0.3` but the closest floating-point to `0.2` is greater than the real `0.2`, so adding the floating-points `0.1` and `0.2` results in a floating-point slightly greater than the constant floating-point `0.3`.

To address this limitation, ESBMC was extended to support floating-point arithmetic [32] but was only able to encode it using SMT solvers that offered native support for the floating-point theory, i.e., Z3 and MathSAT. A floating-point is represented in ESBMC following the IEEE-754 standard for the size of the exponent and significand precision. For example, a half-precision floating-point (16 bits) has 1 bit for the sign, 5 bits for the exponent, and 11 bits for the significand (1 hidden bit) [36].

The work described in this paper, namely a new floating-point API in our SMT backend, is the natural evolution of our research: the support of floating-point arithmetic for the remaining SMT solvers in ESBMC (Boolector [48], Yices [26], and CVC4 [4]). The new floating-point API works by converting all floating-point types and operations to bit-vectors (a process called bit-blasting), thus extending the support for floating-point arithmetic to any solver that supports bit-vector arithmetic [33].

3.1   Bit-blasting Floating-Point Arithmetic

The SMT `FP` logic is an addition to the SMT standard, first proposed in 2010 by Rümmer and Wahl [54]. The current version of the theory largely follows the IEEE standard 754-2008 [36]. It formalizes floating-point arithmetic, positive and negative infinities and zeroes, NaNs, relational and arithmetic operators, and five rounding modes: round nearest with ties choosing the even value, round nearest with ties choosing away from zero, round towards positive infinity, round towards negative infinity and round towards zero.

There exist some functionalities from the IEEE standard that are not yet supported by the `FP` logic as described by Brain et al. [13]; however, when encoding C programs using the `FP` logic, most of the process is a one-to-one conversion, as we described in our previous work [32].

Encoding programs using the SMT floating-point theory has several advantages over a fixed-point encoding. However, the main one is the correct modeling of ANSI-C/C++ programs that use the IEEE floating-point arithmetic. ESBMC ships with models for most of the current C11 standard functions [38]; floating-point exception handling, however, is not yet supported.

The encoding algorithms, however, can be very complex, and it is not uncommon to see the SMT solvers struggling to support every corner case [27,50]. Currently, various SMT solvers support the SMT floating-point theory, e.g., Z3 [25], MathSAT [20], CVC4 [4], Colibri [44], Solonar [41], and UppSAT [58]; ESBMC implements the floating-point encoding for all of them using their native API. Regarding the support from the solvers, Z3 implements all operators, Math-SAT implements all but two: `fp.rem` (remainder operator) and `fp.fma` (fused multiply-add) and CVC4 implements all but the conversions to other sorts.

The three solvers offer two (non-standard) functions to reinterpret floating-points to and from bit-vectors: `fp_as_ieeebv` and `fp_from_ieeebv`, respectively. These functions can be used to circumvent any lack of operators, and only require the user to write the missing operators. Note that this is different from converting floating-points to bit-vectors and vice-versa: converting to bit-vectors follows the rounding modes defined by the IEEE-754 standard while reinterpreting floating-point as bit-vectors returns the bit-vector format of the floating-point. We use these functions in our backend to implement the fused multiply-add operator for MathSAT.

The implementation of the floating-point API is based on the encoding of Muller et al. [47], however, before we can discuss the algorithms in the floating-point API, we first need to describe the basic operations performed by most of them, the four-stage pipeline [12]: unpack, operate, round, and pack.

1. *Unpack stage*: the floating-point is split into three bit-vectors, one for the sign, one for the exponent, and one for the significand. In our floating-point API, the unpack operation also adds the hidden bit to the significand, unbias the exponent. It offers an option to normalize subnormals exponents and significands if requested.
2. *Operate stage*: in this stage, conversion and arithmetic operations are performed in the three bit-vectors. Depending on the operation, the bit-vectors need to be extended, e.g., during a fused multiply-add operation, the significand has length `2 * sb + 3`, and the exponent has length `eb + 2`.
3. *Round stage*: since the previous stage was performed using extended bit-vectors, this stage needs to round the bit-vectors back to the nearest representable floating-point of the target format. Here, *guard* and *sticky* bits in the significand are used to determine how far the bit-vector is from the nearest representable, and the rounding mode is used to determine in which direction the floating-point will be rounded. The exponent bit-vector is also checked for under- or overflow when rounding, to create the correct floating-point, e.g., infinity might be created if the exponent is too large for the target format.

4. *Pack stage*: in the final stage, the three bit-vectors are concatenated to form the final floating-point.

The four-stage pipeline will be used when performing operations with the floating-points. We grouped the operations into seven groups: sorts constructors, rounding modes constructors, value constructors, classification operators, comparison operators, conversion operators, and arithmetic operators.

In the three constructors groups (sorts, rounding modes, and value), the floating-points are encoded using bit-vectors:

Sorts constructors. The sorts follow the definitions in Table 1 for the bit-vector sizes. We do not support the 80-bit long double extended precision format used in some processors [35]; instead, we use 128 bits for quadruple precision.

Rounding mode constructors. The floating-point API supports all rounding modes described in Section 2, even though the C standard does not support RNA [38]. These are encoded as 3-bits long bit-vectors.

Value constructors. Floating-point literals, plus and minus infinity, plus and minus zeroes and NaNs can be created. For the later, the same NaN is always created (positive, the significand is $000\ldots01$). All values are bit-vectors with total length `1 + eb + sb`, where `eb` is the number of exponent bits and `sb` is the number of significand bits. All algorithms in the floating-point API assume one hidden-bit in the significand.

The remaining four operators groups use at least one of the stages in the pipeline to reason about floating-points:

Classification operators. Algorithms to classify normals, subnormals, zeros (regardless of sign), infinities (regardless of sign), NaNs, and negatives and positives. The operators work by unpacking the floating-point and comparing the bit-vectors against the definitions.

Comparison operators. The operators "greater than or equal to", "greater than", "less than or equal to", "less than", and "equality" are supported. The first three are written in terms of the last two. All of them evaluate to false if one of their arguments is NaN; this check is done using the NaN classification operator.

Conversion operators. The floating-point API can convert:

– Floating-points to signed bit-vectors and floating-points to unsigned bit-vectors: converts the floating-point to bit-vectors always rounding towards zero. These operations generate a free variable if it can not represent the floating-point using the target bit-vector, i.e., if the floating-point is out-of-range, $\pm$NaN or $\pm$ infinity. Minus zero is converted to zero.
– Floating-points to another floating-point: converts the floating-point to a different format using a rounding mode. $\pm$NaN, $\pm$infinity, and $\pm$zeroes are always convertible between floating-points, but converting between formats might create $\pm$infinity if the target format can not represent the original floating-point.
– Signed bit-vectors to floating-points and unsigned bit-vectors to floating-points: converts bit-vectors to the nearest representable floating-point, using

a rounding mode. It might create ±infinity if the target format can not represent the original bit-vector.

Arithmetic operators. The floating-point API implements:

- *Absolute value operator*: sets the sign bit of the floating-point to `0`.
- *Negation operator*: flips the sign bit of the floating-point.
- *Addition operator*: the significands are extended by 3 bits to perform the addition and the exponent are extended by 2 bits to check for overflows. The algorithm first aligns the significands then it adds them.
- *Subtraction operator*: negates the right-hand side of the expression and uses the addition operator, i.e., $x - y = x + (-y)$.
- *Multiplication operator*: the length of the significand bit-vectors are doubled before multiplying them, and the exponents are added together. The final sign bit is the result of xor'ing the sign of both operands of the multiplication.
- *Division operator*: the length of both significand and exponent are extended by 2 bits, then bit-vector subtractions are used to calculate the target significand and exponent.
- *Fused multiply-add*: the significand is extended to length `2 * sb + 3` to accommodate both the multiplication and the addition, and the exponent is extended by 2 bits. The first two operands are multiplied, and the result is aligned with the third operand before adding them.
- *Square root operator*: neither the significand nor the exponent is extended since the result always fits the original format and can never underflow or overflows as per the operator definition. Here, $\sqrt{x} = l * 2^d$, where the final exponent d is half the unbiased exponent minus the leading zeros, and l is calculated using a restorative algorithm [47, Chapter 10].

All operators but the absolute value and negation handle special values (±NaN, ±infinity, and ±zeroes) before performing the operations, e.g., in the multiplication operator, if the left-hand side argument is positive infinity, the result is NaN if the right-hand side argument is `0`; otherwise, the result is an infinity with the right-hand side argument sign. Furthermore, all arithmetic operations in the floating-point API that take more than one floating-point as an argument assume that the floating-points have the same format. This assumption is not a problem when converting C programs, as type promotion rules already ensure this pre-condition [38].

A detailed table with all the supported features of the floating-point API, and the comparison with the features from other solvers can be found in Appendix A.

4   Experimental Evaluation

Our experimental evaluation consists of three parts. In Section 4.1, we present the benchmarks used to evaluate the implementation of our floating-point API. In Section 4.2, we compare the verification results of the new floating-point API in ESBMC using several solvers. In Section 4.3, we compare the best solver found

in Section 4.2 against all the tools that competed in the *ReachSafety-Floats* sub-category in SV-Comp 2020. Our experimental evaluation aims to answer two research questions:

> **RQ1 (Soundness and completeness)** Is our floating-point API sound and complete?
>
> **RQ2 (Performance)** How does the implementation of our floating-point API compare to other software verifiers?

4.1   Experimental Setup

We evaluate our approach using all the verification tasks in SV-COMP 2020 [7]. In particular, we considered 466 benchmarks for the sub-category *ReachSafety-Floats*, described as *"containing tasks for checking programs with floating-point arithmetics"*.

The *ReachSafety-Floats* sub-category is part of the *ReachSafety* category. In this category, a function call is checked for reachability; the property is formally defined in the competition as `G ! call(__VERIFIER_error())` or *"The function __VERIFIER_error() is not called in any finite execution of the program"*.

We have implemented our floating-point API in ESBMC. We run ESBMC on each benchmark in that sub-category once per solver, with the following set of options: `--no-div-by-zero-check`, which disables the division by zero check (an SV-COMP requirement); `--incremental-bmc`, which enables the incremental BMC; `--unlimited-k-steps`, which removes the upper limit of iteration steps in the incremental BMC algorithm; `--floatbv`, which enables SMT floating-point encoding; `--32`, which assumes a 32 bits architecture; and `--force-malloc-success`, which forces all dynamic allocations succeed to (also an SV-COMP requirement). We also disable pointer safety checks and array bounds check (`--no-pointer-check`, `--no-bounds-check`) as, per the competition definition, these benchmarks only have reachability bugs. Finaly, in order to select an SMT solver for verification, the options `--boolector`, `--z3`, `--cvc`, `--mathsat`, and `--yices` are used.

All experiments were conducted on our mini cluster at the University of Manchester, UK. The compute node used are equipped with Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 180GB of RAM, where nine instances of ESBMC were executed in parallel. For each benchmark, we set time and memory limits of 900 seconds and 15GB, respectively, as per the competition definitions. We, however, do not present the results as scores (as it is done in SV-COMP) but show the number of correct and incorrect results, and the verification time.

4.2   Floating-Point API evaluation

Figure 2 shows the number of correctly verified programs out of the 466 benchmarks from the *ReachSafety-Floats* sub-category, using several solvers and how

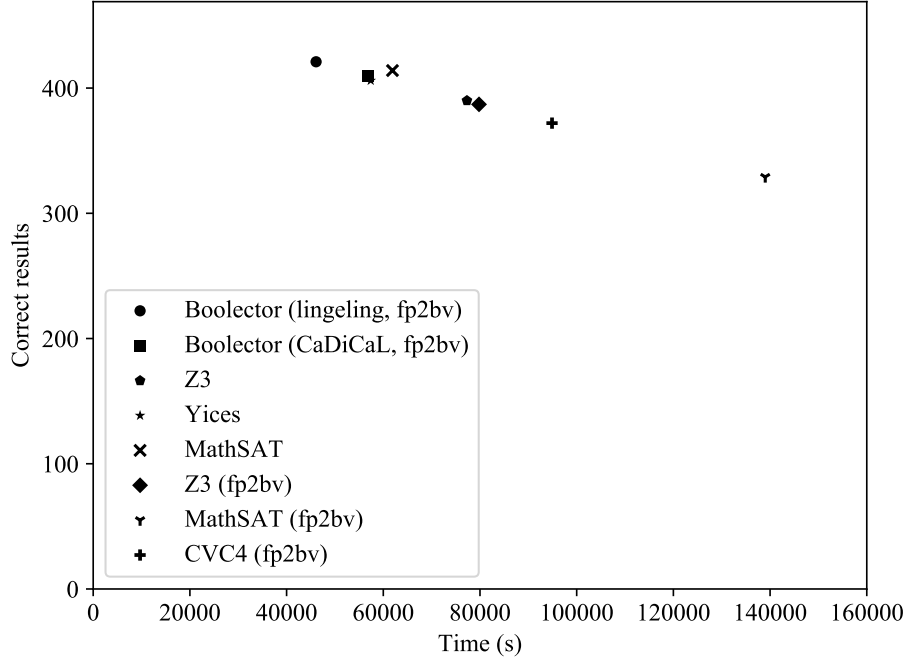long it took to complete the verification. There exists no case where ESBMC reports an incorrect result.

Fig. 2: *ReachSafety-Floats* results for each solver, using the incremental BMC. The "fp2bv" next to the solver name means that our floating-point API was used to bit-blast floating-point arithmetic.

Boolector (lingeling, fp2bv) reports the highest number of correct results (421), followed by MathSAT using their native floating-point API (414). This evaluation produced a slightly better result than our previous one of these solvers, where MathSAT was able to solve floating-point problems quickly but suffered slowdowns in programs with arrays [32]. MathSAT (fp2bv) presented the fewest number of correct results (329).

The results show that Z3 with its native floating-point API and Z3 with our fp2bv API produce very similar results: 390 and 387, respectively; this result is expected since our fp2bv API is heavily based on the bit-blasting performed by Z3 when solving floating-points. The number of variables and clauses generated in the CNF format, when using Z3 with its native floating-point API, is 1%-2% smaller than the number generated when using our fp2bv API. The smaller number explains the slightly better results: we assume this is the result of optimizations when Z3 performs the bit-blasting internally.

MathSAT results show that their API can solve 85 more benchmarks than MathSAT (fp2bv) within time and memory limits. These benchmarks contain chains of multiplications. They thus require a high computational effort during the propositional satisfiability search. Given that we replace all higher-level operators by bit-level circuit equivalents (bit-blasting), we end up destroying structural word-level information in the problem formulation. Therefore, these results lead us to believe that the MathSAT ACDL algorithm is somehow optimized for FP operations; unfortunately, MathSAT is a free but closed source tool, so we cannot confirm this.

The total verification time for each solver is also illustrated in Figure 2, and again Boolector (lingeling, fp2bv) was the faster solver, thereby solving all programs in 46100 seconds. It is followed by Boolector (CaDiCal, fp2bv) with 56900, and Yices (fp2bv) with 57400 seconds. Overall, Boolector (lingeling, fp2bv) presented the best results. It correctly verified more programs while also being the faster solver, almost 20% faster than the second faster solver, which is also Boolector but with a different SAT solver (CaDiCaL).

> ESBMC produced no incorrect result in this evaluation, which partially answers **RQ1**: although we can not formally prove that our algorithm is sound and complete, empirical evidence suggests it.

4.3   Comparison to other Software Verifiers

We compare the implementation of our floating-point API with other software verifiers: 2LS [43], CBMC [40], CPA-Seq [8], DIVINE [3], PeSCo [53], Pinaka [17], Symbiotic [16], VeriAbs [19]. Figure 3 illustrates the *ReachSafety-Floats* results from our best approach against tools that participated in SV-COMP 2020. In particular, we have used the binary and scripts of these tools that are available at the SV-COMP 2020 website under "Participating Teams".[4] Overall, VeriAbs achieved the highest number of correct results (435) in 53600 s followed by Pinaka (422) with 27800 s, ESBMC (421) with 46100 s, and CBMC (420) with 49200 s.

VeriAbs can verify C programs with floating-points via abstraction using SAT solvers. In particular, VeriAbs replaces loops in the original code by abstract loops of small known bounds; it performs value analysis to compute loop invariants and then applies an iterative refinement using k-induction. The VeriAbs tool uses CBMC as its backend to prove properties and find errors, which thus allows VeriAbs to verify C programs with floating-points. By contrast, ESBMC uses an iterative technique and verifies the program for each unwind bound until it exhausts the time or memory limits. Intuitively, ESBMC can either find a counterexample with up to k loop unwinding or fully unwinds all loops using the same unwinding bound so that it can provide a correct result. ESBMC also relies on SMT solvers to check the satisfiability of the verifications conditions that contain floating-point arithmetic.

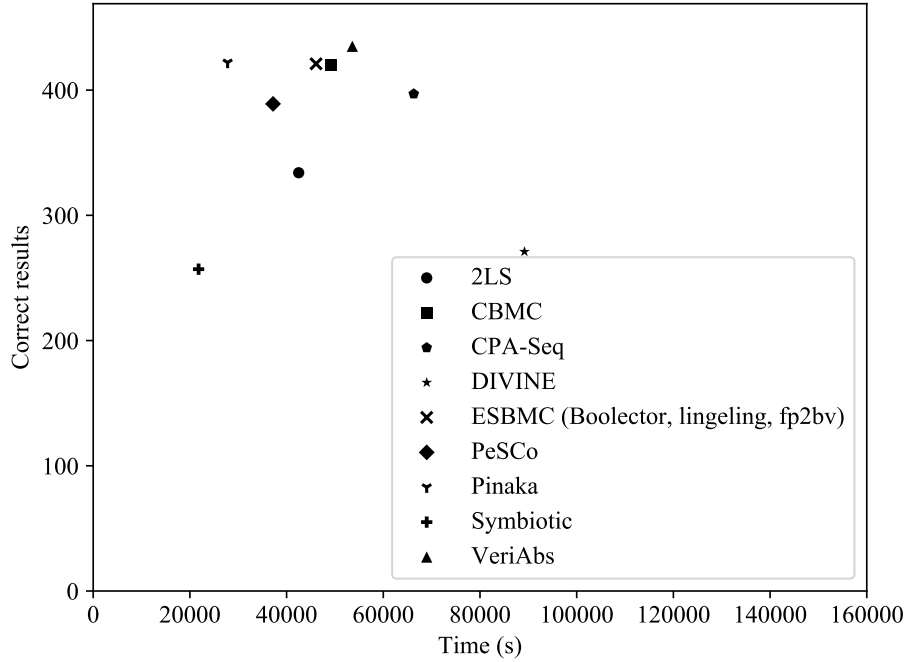[4] https://sv-comp.sosy-lab.org/2020/systems.php

Fig. 3: *ReachSafety-Floats* results from our best approach against tools from SV-COMP 2020.

Pinaka verifies C programs using CBMC, but it relies on an incremental SAT solving coupled with eager state infeasibility checks. Additionally, Pinaka extends CBMC to support both Breadth-First Search and Depth-First Search as state exploration strategies along with partial and full incremental modes. Here we have not evaluated the SMT incremental mode implemented in ESBMC since this feature is currently supported for the SMT solver Z3 only. Other SMT solvers do support incremental solving, but ESBMC does not provide support for incremental solving for other SMT solvers yet.

CBMC [21] implements a bit-precise decision procedure for the theory of floating-point arithmetic [11]. Both VeriAbs and Pinaka rely on CBMC to verify the underlying C program using that decision procedure. ESBMC originated as a fork of CBMC in 2008 with an improved SMT backend [24] and support for the verification of concurrent programs using an explicit interleaving approach [23]. CBMC uses SAT solvers as their primary engine but offers support for the generation of an SMT formula for an external SMT solver. ESBMC supports SMT solvers directly, through their APIs, along with the option to output SMT formulae. As a result, the main difference between CBMC and ESBMC here relies on the encoding and checking of the verification conditions that contain floating-point arithmetic.

These results answer our **RQ2**: our floating-point API is on par with other state-of-the-art tools. VeriAbs and Pinaka implement several heuristics to simplify the check for satisfiability using CBMC, while ESBMC used an incremental approach produced close results. ESBMC was also slightly faster and provided a few more results than CBMC, which lead us to believe that our tool would also greatly benefit VeriAbs and Pinaka if used as backend.

5   Related Work

Several symbolic execution tools try to verify programs with floating-point arithmetic by employing different strategies. CoverMe [30] reformulates floating-point constraints as mathematical optimization problems and uses a specially built solver called XSat [29] to check for satisfiability. Pex [56] uses a similar approach and reasons for floating-point constraints as a search problem, and they are solved by using meta-heuristics search methods. FPSE [10] models floating-point arithmetic by using an interval solver over real arithmetic combined with projection functions.

HSE [52] extends KLEE [15] to execute the program and convert floating-points into bit-vectors symbolically. It then uses SMT solvers to reason about satisfiability. Astrée is a static analysis tool that considers all possible rounding errors when verifying C programs with floating-point numbers [9]. It has been applied to verify embedded software in the flight control software of the Airbus.

Bounded model checkers have also been applied to verify programs with floating-point arithmetic: CBMC [21] and 2LS [55] convert floating-point operations to bit-vectors and use SAT solvers to reason about satisfiability. CPBPV [22] uses bounded model checking combined with their FPCS [45] interval solver to generate tests that violate output constraints in the program.

Brain et al. [12] describe an approach called SymFPU for handling the theory of floating-point by reducing it to the theory of bit-vectors. In particular, the authors describe a library of encodings, which can be included in SMT solvers to add support for the theory of floating-point by taking into account floating-point reasoning and the fundamentals of circuit design. Brain et al. have integrated SymFPU into the SMT solver CVC4 and evaluate it using a broad set of benchmarks; they conclude that SymFPU+CVC4 can substantially outperforms all previous systems despite using a straightforward bit-blasting approach for floating-point problems. We could not compare our approach against SymFPU because of bugs in the CVC4 C API; we contacted the author, and we will create bug reports about the issues we identified.

6   Conclusions

We have described our new SMT floating-point API, which bit-blasts floating-point arithmetic and extends the floating-point support for SMT solvers that

only support bit-vector arithmetic. The floating-point API was implemented in the SMT backend of ESBMC. Our experimental results show that Boolector (with lingeling as SAT solver) presented the best results: the highest number of correct results within the shortest verification time. We also show that our floating-point API implemented in ESBMC is on par with other state-of-the-art software verifiers. VeriAbs and Pinaka implement several heuristics to simplify the check for satisfiability using CBMC, while ESBMC with a straightforward incremental approach produced close results.

ESBMC was already extensively used to verify digital systems [1,5,6]. However, these projects were limited to fixed-point arithmetic; supporting floating-point encoding will allow researchers to expand their activities in the scientific community. The extensive evaluation performed during the development of these technologies also identified areas to be improved in the solvers and other verification tools. In particular, we submitted patches to Z3 to optimize the generation of unsigned less-than operations during the bit-blast of floating-points[5] (accepted, part of Z3 4.6.1). We also reported bugs to both CBMC[6] and MathSAT, concerning floating-point arithmetic issues, which were later confirmed by the developers.

References

1. Abreu, R.B., Gadelha, M.R., Cordeiro, L.C., Filho, E.B.d.L., da Silva Jr., W.S.: Bounded model checking for fixed-point digital filters. Journal of the Brazilian Computer Society **22**(1), 1:1–1:20 (2016)
2. ARM: ARM C Language Extensions 2.1 (2016), IHI 0053D
3. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology For Verification And Analysis. LNCS, vol. 10482, pp. 201–207 (2017)
4. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer-Aided Verification. LNCS, vol. 6806, pp. 171–177 (2011)
5. Bessa, I., Ismail, H., Cordeiro, L.C., Filho, J.E.C.: Verification of fixed-point digital controllers using direct and delta forms realizations. Design Automation for Embedded Systems **20**(2), 95–126 (2016)
6. Bessa, I., Ismail, H., Palhares, R., Cordeiro, L.C., Filho, J.E.C.: Formal non-fragile stability verification of digital control systems with uncertainty. IEEE Transactions on Computers **66**(3), 545–552 (2017)
7. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 12079, pp. 347–367 (2020)
8. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Computer-Aided Verification. LNCS, vol. 6806, pp. 184–190 (2011)
9. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation. pp. 196–207 (2004)

---

[5] https://github.com/Z3Prover/z3/pull/1501

[6] https://github.com/diffblue/cmbc/issues/1944

10. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations: Research articles. Software Testing, Verification & Reliability **16**(2), 97–121 (2006)
11. Brain, M., D'Silva, V., Griggio, A., Haller, L., Kroening, D.: Deciding floating-point logic with abstract conflict driven clause learning. Formal Methods in System Design **45**(2), 213–245 (2014)
12. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 11427, pp. 79–98 (2019)
13. Brain, M., Tinelli, C., Ruemmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: Symposium On Computer Arithmetic. pp. 160–167 (2015)
14. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 5505, pp. 174–177 (2009)
15. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Symposium On Operating Systems Design And Implementation. pp. 209–224 (2008)
16. Chalupa, M., Vitovská, M., Jonás, M., Slaby, J., Strejcek, J.: Symbiotic 4: Beyond reachability - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10206, pp. 385–389 (2017)
17. Chaudhary, E., Joshi, S.: Pinaka: Symbolic execution meets incremental solving - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 11429, pp. 234–238 (2019)
18. Chaves, L., Bessa, I., Cordeiro, L.C., Kroening, D., Filho, E.B.d.L.: Verifying digital systems with MATLAB. In: Symposium On Software Testing And Analysis. pp. 388–391 (2017)
19. Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 10206, pp. 404–408 (2017)
20. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The mathSAT5 SMT solver. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 7795, pp. 93–107 (2013)
21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 2988, pp. 168–176 (2004)
22. Collavizza, H., Michel, C., Ponsini, O., Rueher, M.: Generating test cases inside suspicious intervals for floating-point number programs. In: Constraints In Software Testing Verification And Analysis. pp. 7–11 (2014)
23. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: International Conference on Software Engineering. pp. 331–340 (2011)
24. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Automated Software Engineering. pp. 137–148 (2009)
25. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 4963, pp. 337–340 (2008)
26. Dutertre, B.: Yices 2.2. In: Computer-Aided Verification. LNCS, vol. 8559, pp. 737–744 (2014)

27. Erkk, L.: Bug in floating-point conversions. https://github.com/Z3Prover/z3/issues/1564 (2018), [Online; accessed January-2020]
28. Frantz, G., Simar, R.: Comparing fixed- and floating-point DSPs. Tech. rep., SPRY061, Texas Instruments (2004)
29. Fu, Z., Su, Z.: XSat: A fast floating-point satisfiability solver. In: Computer-Aided Verification. LNCS, vol. 9780, pp. 187–209 (2016)
30. Fu, Z., Su, Z.: Achieving high coverage for floating-point code via unconstrained programming. In: Programming Language Design And Implementation. pp. 306–319 (2017)
31. Gadelha, M.R., Monteiro, F., Cordeiro, L., Nicole, D.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 11429, pp. 209–213 (2019)
32. Gadelha, M.Y.R., Cordeiro, L.C., Nicole, D.A.: Encoding floating-point numbers using the SMT theory in ESBMC: An empirical evaluation over the SV-COMP benchmarks. In: Simpósio Brasileiro De Métodos Formais. LNCS, vol. 10623, pp. 91–106 (2017)
33. Gadelha, M.Y.R., Menezes, R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC: scalable and precise test generation based on the floating-point theory - (competition contribution). In: Wehrheim, H., Cabot, J. (eds.) International Conference Fundamental Approaches to Software Engineering. LNCS, vol. 12076, pp. 525–529. Springer (2020)
34. Gerrity, G.W.: Computer representation of real numbers. IEEE Transactions on Computers **C-31**(8), 709–714 (1982)
35. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1), 5–48 (1991)
36. IEEE: IEEE Standard For Floating-Point Arithmetic (2008), IEEE 754-2008
37. Ismail, H., Bessa, I., Cordeiro, L.C., Filho, E.B.d.L., Filho, J.E.C.: DSVerifier: A bounded model checking tool for digital systems. In: Symposium On Model Checking Software. LNCS, vol. 9232, pp. 126–131 (2015)
38. ISO: C11 Standard (2011), ISO/IEC 9899:2011
39. ISO: C++ Standard (2015), ISO/IEC 18661-3:2015
40. Kroening, D., Tautschnig, M.: CBMC - C bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 8413, pp. 389–391 (2014)
41. Lapschies, F.: Sonolar the solver for non-linear arithmetic. http://www.informatik.uni-bremen.de/agbs/florian/sonolar (2017), [Online; accessed April-2020]
42. Lions, J.L.: ARIANE 5 flight 501 failure. Tech. rep., Inquiry Board (1996)
43. Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.: 2LS: Memory safety and non-termination. In: Tools And Algorithms For The Construction And Analysis Of Systems. LNCS, vol. 10806, pp. 417–421 (2018)
44. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. http://smt-workshop.cs.uiowa.edu/2017/papers/SMT2017_paper_21.pdf (2017), [Online; accessed April-2020]
45. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: Principles And Practice Of Constraint Programming. pp. 524–538 (2001)
46. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems **30**(3), 12:1–12:41 (2008)
47. Muller, J.M., Brisebarre, N., Dinechin, F., Jeannerod, C.P., Lefvre, V., Melquiond, G., Revol, N., Stehl, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhuser Boston, 1st edn. (2010)

48. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9**, 53–58 (2014)
49. Nikolić, Z., Nguyen, H.T., Frantz, G.: Design and implementation of numerical linear algebra algorithms on fixed point DSPs. European Association for Signal Processing **2007**(1), 1–22 (2007)
50. Noetzli, A.: Failing precondition when multiplying 4-bit significand/4-bit exponent floats. https://github.com/CVC4/CVC4/issues/2182 (2018), [Online; accessed January-2020]
51. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition). Academic Press (2012)
52. Quan, M.: Hotspot symbolic execution of floating-point programs. In: Symposium On Foundations Of Software Engineering. pp. 1112–1114 (2016)
53. Richter, C., Wehrheim, H.: Pesco: Predicting sequential combinations of verifiers - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 11429, pp. 229–233 (2019)
54. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. In: SMT Workshop (2010)
55. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Incremental bounded model checking for embedded software (extended version). Formal Aspects of Computing **29**(5), 911–931 (2017)
56. Tillmann, N., De Halleux, J.: Pex: White box test generation for .NET. In: Tests And Proofs. pp. 134–153 (2008)
57. Wiffin, E.: 0.30000000000000004.com. https://0.30000000000000004.com/ (2012), [Online; accessed April-2020]
58. Zeljic, A., Backeman, P., Wintersteiger, C.M., Rümmer, P.: Exploring approximations for floating-point arithmetic using uppsat. In: International Joint Conference on Automated Reasoning. LNCS, vol. 10900, pp. 246–262. Springer (2018)

A Support for the FP logic

SMT FP operations	Z3 v4.7.1	MathSAT v5.5.1	CVC4 v1.6-prerelease	ESBMC FP API
Create floating point sort	√	√	√	√
Create rounding mode sort	√	√	√	√
Create floating point literal	√	√	√	√
Create plus and minus infinity	√	√	√	√
Create plus and minus zeroes	√	√	√	√
Crete NaN	√	√	√	√
Absolute value operator	√	√	√	√
Negation operator	√	√	√	√
Addition operator	√	√	√	√
Subtraction operator	√	√	√	√
Multiplication operator	√	√	√	√
Division operator	√	√	√	√
Fused multiply-add operator	√	×[7]	√	√
Square root operator	√	√	√	√
Remainder operator	√	×	√	×
Rounding to Integral operator	√	√	√	√
Minimum operator	√	√	√	×
Maximum operator	√	√	√	×
Less than or equal to operator	√	√	√	√
Less than operator	√	√	√	√
Greater than or equal to operator	√	√	√	√
Greater than operator	√	√	√	√
Equality operator	√	√	√	√
IsNormal check	√	√	√	√
IsSubnormal check	√	√	√	√
IsZero check	√	√	√	√
IsInfinite check	√	√	√	√
IsNaN check	√	√	√	√
IsNegative check	√	√	√	√
IsPositive check	√	√	√	√
Convert to FP from real	√	√	×	×
Convert to FP from signed BV	√	√	×	√
Convert to FP from unsigned BV	√	√	×	√
Convert to FP from another FP	√	√	×	√
Convert to unsigned BV from FP	√	√	×	√
Convert to signed BV from FP	√	√	×	√
Convert to real from FP	√	√	×	×

---

[7] In ESBMC, the fused multiply-add operation uses the bit-blasting API when using MathSAT.

SMT FP operations	Z3 v4.7.1	MathSAT v5.5.1	CVC4 v1.6-prerelease	ESBMC FP API
Convert to IEEE BV from FP[8]	√	√	√	√
Convert to floating-point from IEEE BV[8]	√	√	√	√

Table 2: Support in each SMT solver and in the ESBMC floating-point API for the operations described in the SMT FP logic. A √ indicates a supported feature while × indicates an unsupported feature.

---

[8] Not part of the SMT FP logic.