





Overapproximation of Non-Linear Integer Arithmetic for Smart Contract Verification

Petra Hozzová¹, Jaroslav Bendík², Alexander Nutz², and Yoav Rodeh²

¹ TU Wien

² Certora

Abstract

The need to solve non-linear arithmetic constraints presents a major obstacle to the automatic verification of smart contracts. In this case study we focus on the two overapproximation techniques used by the industry verification tool Certora Prover: overapproximation of non-linear integer arithmetic using linear integer arithmetic and using non-linear real arithmetic. We compare the performance of contemporary SMT solvers on verification conditions produced by the Certora Prover using these two approximations against the natural non-linear integer arithmetic encoding. Our evaluation shows that the use of the overapproximation methods leads to solving a significant number of new problems.

1 Introduction

Smart contracts [19] are an ideal target for automated program verification in many ways: Their source code is often relatively small. Due to high execution costs (also known as gas consumption), they contain few loops or recursive methods. Finally, since the underlying applications are often financial in nature, there is a large incentive to detect bugs early, and hacks that lead to a loss of money in the tens of millions of US dollars are not rare. However, a major pain point is that the verification conditions of smart contracts often contain non-linear arithmetic, which makes them undecidable in general and sometimes intractable in practice.

In this case study, we evaluate two methods to prove verification conditions stemming from smart contract verification that were previously intractable due to their use of non-linear integer arithmetic (NIA). The methods are based on the observation that the integer inequalities that occur in the smart contract verification conditions can often be effectively reasoned about at lower precision than the one demanded by the standard NIA semantics, thus allowing for a precision-complexity trade-off. We evaluate two techniques to overapproximate a NIA formula, one using linear integer arithmetic (LIA) and one using nonlinear real arithmetic (NRA).

The motivation leading to this case study was to augment the portfolio of methods that the Certora Prover, a leading industrial tool for automatic verification of smart contracts, uses for verifying properties of Ethereum [20] smart contracts containing NIA. To verify a smart contract property, the Certora Prover builds a logical formula, called a *verification condition (VC)*, such that the VC is satisfiable iff the specification can be violated by the contract. Moreover, every model of the VC corresponds to a particular counter-example, i.e., an execution of the smart

contract violating the specification. To check the VC satisfiability, the Certora Prover converts the VC to an SMT formula and passes it to an SMT solver, such as `CVC5` [1] or `Z3` [11].

The smart contract VC is encoded in a fragment of first-order logic with theories, uninterpreted sorts, and uninterpreted function symbols. Depending on the smart contract, specification, and tool configuration, the VC may contain quantifiers, datatypes, LIA or NIA, and bitwise operations. This study focuses on the cases with quantifier-free VC using NIA. Since satisfiability of formulas with NIA is undecidable [18] it is not too surprising that SMT solvers often struggle with proving such VCs. Our evaluation shows this difficulty, as well as a significant number of benchmarks being newly solved using the two overapproximation techniques.

The idea to overapproximate the underlying non-linearity was developed in the Certora Prover already from its conception with a simple LIA overapproximation, which was then gradually refined by adding axioms specialized for the domain of smart contracts. Motivated by empirical success of the LIA-based method, we recently implemented another overapproximation based on NRA. Both methods are designed for proving the VC unsatisfiability, not finding counter-examples: overapproximation unsatisfiability implies unsatisfiability of the original VC, but not vice versa. Interestingly, contemporary SMT solvers also internally apply various overapproximation techniques while solving NIA instances [8, 16, 17]. As our results show, these techniques seem to be largely complementary to the overapproximations we investigate in this paper.

Contributions. The main focus of this paper is on experimental evaluation of two SMT-based overapproximation techniques for smart contract verification, employed by the Certora Prover. Namely, we compare the performance of contemporary SMT solvers on the natural NIA encoding against the LIA and NRA overapproximations. Using the two overapproximations, we are able to verify 39% new examples in our benchmark suite, which could not be verified using the NIA encoding. These benchmarks are real-world benchmarks: they stem from actual customer contracts that were under verification by Certora, and are made available online. Secondary contributions of this paper are (brief) descriptions of the LIA and NRA encodings.

Outline. Section 2 lays out the preliminaries and introduces an illustrative example. Section 3 presents the overapproximation methods and demonstrates one of them on our example. Section 4 reports on the experiment setup and results. Section 5 briefly reviews related work.

2 Preliminaries and Example

We assume familiarity with standard first-order logic with equality and the theories of integer and real arithmetic (for details see [4, 14]). We use the theories of linear integer arithmetic, and non-linear arithmetic for integers and reals in accordance with SMT-LIB definitions [3], denoted by LIA, NIA and NRA, respectively. These theories define the predicates less, less equal, greater, greater equal, and the functions of addition, subtraction, (only NIA and NRA) multiplication, division, (only NIA) absolute value¹, modulo. We respectively denote these predicate and function symbols for integers by $<_I, \leq_I, >_I, \geq_I, +_I, -_I, *_I, \text{div}_I, \text{abs}_I, \text{mod}_I$ and for reals by $<_R, \leq_R, >_R, \geq_R, +_R, -_R, *_R, /_R$. We use $-_I$ and $-_R$ to denote both the unary and binary minus. Further, we use the same symbols $0, 1, -1, \dots$ for both integer and real constants.

¹In SMT-LIB, only NIA defines the absolute value function. However, since abs_I does not occur in the VCs produced by the Certora Prover, and since its counterpart for LIA and NRA could be straightforwardly defined as $\text{abs}(x) = \text{if } x > 0 \text{ then } x \text{ else } -x$, we do not consider it in the overapproximations described in this paper.

In this paper we work with the quantifier-free fragment of first-order logic, denoted by the prefix “QF_”. Free variables are implicitly assumed to be existentially quantified, hence the (SMT) variables correspond to first-order constants. We denote the SMT variables by a, b ; first-order variables by x, y, z, w ; terms by t, s, u, v ; formulas by f ; all possibly with indices. We use $t <_I u \leq_I v$ as an abbreviation for $t <_I u \wedge u \leq_I v$, and similarly with other comparison predicate symbols. We use the standard notions of position in a formula and of (sub)formula polarity. We denote the position in a formula by a sequence of positive integers. We write ϵ for the empty sequence denoting the top-most position in a formula. When f' is a subformula of a formula f at position π , where the top-level logical connective of f' is n -ary, then the subformula of f at position $\pi.i$ for any $i \in \{1, \dots, n\}$ is the i th argument of f' . We denote the polarity of a (sub)formula of f at position π by $p_f(\pi)$. Let f be a formula and let $p_f(\epsilon) \stackrel{\text{def}}{=} 1$. Then, for each (sub)formula f' at position π of f : If f' has the form $f_1 \wedge \dots \wedge f_n$ or $f_1 \vee \dots \vee f_n$, then $\forall i \in \{1, \dots, n\}: p_f(\pi.i) = p_f(\pi)$. If f' has the form $\neg f_1$, then $p_f(\pi.1) = -p_f(\pi)$. If f' has the form $f_1 \rightarrow f_2$, then $p_f(\pi.1) = -p_f(\pi)$ and $p_f(\pi.2) = p_f(\pi)$. If f' has the form $f_1 \leftrightarrow f_2$, then $p_f(\pi.1) = p_f(\pi.2) = 0$. We say that the (sub)formula of f at position π has positive polarity if $p_f(\pi) = 1$ and negative polarity if $p_f(\pi) = -1$.

We use the model domains of integers \mathbb{Z} and real numbers \mathbb{R} , and we consider $\mathbb{Z} \subset \mathbb{R}$, i.e., we consider e.g. 1 and 1.0 to denote the same number. We use m, n to denote numbers from either \mathbb{Z} or \mathbb{R} . The theory predicate and function symbols are interpreted in the standard way, and we denote their interpretations by $<, \leq, >, \geq, +, -, *, \text{div}, \text{abs}, \text{mod}, /$, respectively.²

Example 1. We illustrate the verification problem by a simplified Ethereum smart contract verification example. Assume assets in the form of shares and money. We model all the smart contract and specification variables, typically of type `uint256`, as non-negative integer variables.

Let there be a total supply of t_S shares, altogether having the total monetary value of t_M , where $t_S \neq 0$ and $t_M \neq 0$. If we withdrew w_S shares, these shares would be sold and we would obtain the monetary value $w_M = (w_S * t_M) \text{div } t_S$. Note that div is the integer division: since the smart contract works with `uint256`s, we might get a little less money for the shares than the real value that they have. Now, assume that n_S new shares are added to the total supply t_S , having the same per-share value as the old shares (with precision up to the precision of div). We obtain new total supply of shares $t'_S = t_S + n_S$ and the new total monetary value $t'_M = t_M + ((n_S * t_M) \text{div } t_S)$. If we now withdrew the same number w_S of shares from the new total supply, we would obtain the monetary value $w'_M = (w_S * t'_M) \text{div } t'_S$. Given this scenario, we would like to prove the following property: the monetary value from the original withdrawal is greater or equal to the monetary value from the second withdrawal, i.e., $w_M \geq w'_M$.

We encode this example in QF_NIA. We construct the verification condition for the example as a conjunction of all the assumptions and a negation of the conjecture:

$$\begin{aligned} & t_S >_I 0 \wedge t_M >_I 0 \wedge w_S \geq_I 0 \wedge w_M \geq_I 0 \wedge n_S \geq_I 0 \wedge t'_S \geq_I 0 \wedge t'_M \geq_I 0 \wedge w'_M \geq_I 0 \\ & \wedge w_M = (w_S *_I t_M) \text{div}_I t_S \wedge t'_S = t_S +_I n_S \wedge t'_M = t_M +_I ((n_S *_I t_M) \text{div}_I t_S) \quad (1) \\ & \wedge w'_M = (w_S *_I t'_M) \text{div}_I t'_S \wedge \neg(w_M \geq_I w'_M) \end{aligned}$$

If the formula (1) is unsatisfiable, the property holds. If (1) is satisfiable, its model constitutes a counterexample to the property. In this case, the formula (1) is indeed unsatisfiable (it has no integer model), as will be later confirmed using the NRA overapproximation. However, none of the SMT solvers we considered³ can prove the unsatisfiability of its NIA encoding within

²For positive m , n $\text{div } m$ is defined as $\lfloor n/m \rfloor$, for negative m as $\lceil n/m \rceil$.

³CVC5 v1.0.2 [1], MATHSAT v5.6.8 [9], YICES2 v2.6.4 [12], and Z3 v4.11.0 [11]

a 5-minute time limit. The SMT-LIB2 encoding of formula (1) is displayed in Appendix A.1. Further, the smart contract verification problem this example is based on is also displayed in Appendix A.3.

The Certora Prover uses formulas similar in character to (1), and encodes them in the SMT-LIB2 [3] format. Wraparound semantics for `uint256s` are modelled by computing all operations that potentially under- or overflow modulo 2^{256} . In addition to NIA, the Certora Prover also utilizes uninterpreted function symbols (UF) and the theory of datatypes (DT), resulting in the logic QF_UFDTNIA.

3 Overapproximation Methods

In this section, we describe the techniques for constructing the LIA and NRA overapproximations f_{LI}, f_{NR} of the formula f_{NI} using QF_NIA (and possibly other theories). Formula f is an *overapproximation* of f_{NI} , if from unsatisfiability of f follows that f_{NI} is also unsatisfiable. We say that f_2 is a *tighter* overapproximation of f_{NI} than f_1 if both f_1, f_2 are overapproximations of f_{NI} , and the set of models of f_1 is a superset of models of f_2 . There is a trade-off between the overapproximation tightness and the practical complexity. The overapproximation f could precisely capture f_{NI} – but that would essentially force the solvers proving f to reason with NIA instead of LIA or NRA. Rather, we construct formulas f_{LI}, f_{NR} which can be solved using LIA, NRA, respectively, yet are typically unsatisfiable if f_{NI} was unsatisfiable. To construct an overapproximation of f_{NI} , we rewrite the original formula. We denote rewriting of the term t to the term s , i.e., replacing all occurrences of t in a given formula by s , by $t \rightsquigarrow s$.

3.1 Linear Integer Arithmetic Overapproximation

The LIA-based overapproximation proceeds in two steps. First, it replaces nonlinear operations in f_{NI} by replacing the NIA function symbols by uninterpreted function symbols as follows:

$$t *_I s \rightsquigarrow \text{umul}(t, s) \quad t \text{ div}_I s \rightsquigarrow \text{udiv}(t, s) \quad t \text{ mod}_I s \rightsquigarrow \text{umod}(t, s)$$

The resulting formula f_1 is a very coarse overapproximation of f_{NI} .

Second, to tighten the overapproximation, the transformation adds axioms constraining `umul`, `udiv`, `umod` to the formula. To retain the overapproximation property, the axioms must correspond to valid properties of integer arithmetic – e.g., an axiom regarding `umul` must only state valid properties of `*I`. We do not add quantified axioms, but rather we instantiate axioms with selected terms occurring in f_1 . Due to space limitations we do not explicitly list all axioms, nor the instantiation strategy. We rather list the axiomatized properties: (i) arithmetic axioms: commutativity, neutral and absorbing element with respect to `umul`, `udiv` and `umod`, ranges of `udiv` and `umod` results; (ii) relating `umul` to the linear multiplication `*I` if one operand is an interpreted constant; (iii) overflow-related: multiplicative inverse property only holding if the multiplication did not overflow modulo 2^{256} ; ⁴ (iv) twos-complement related: certain standard identities regarding twos-complement multiplications in a modular ring; (v) relating pairs of multiplications: monotonicity and distributivity.

These axioms are instantiated over single applications of the operators, as well as pairs of multiplications that lie on a common program path. They are not instantiated recursively, i.e., if an axiom produces a new multiplication, we don't generate an axiom for that.

⁴The formula f_{NI} usually contains many occurrences of `t modI 2256` (because $2^{256} - 1$ is the maximal value of the integer type `uint256`), and might also contain `0 ≤I t` and `t <I 2256` as the encoding of overflow checks.

Adding the axioms to f_1 results in a formula f_{LI} as a ground formula using the theories UF and LIA in addition to any theories that f_{NI} used except for NIA. E.g., if f_{NI} is in QF_DTNIA, f_{LI} will be in QF_UFDTLIA.

3.2 Non-Linear Real Arithmetic Overapproximation

We construct the NRA overapproximation f_{NR} of f_{NI} in three steps. The first two steps are similar to the LIA overapproximation: we replace the function symbols and add axioms. Then we tighten the overapproximation by rewriting selected inequalities. Similarly to f_{LI} , the formula f_{NR} is ground and uses UF and NRA in addition to any theories that f_{NI} used except for NIA.

In contrast to the LIA overapproximation, NRA overapproximation uses the real sort. Hence, we change the sorts of all variables and functions from integers to reals. We replace all interpreted integer constants by their real counterparts, and the symbols $+_I, -_I, *_I$ by $+_R, -_R, *_R$. We replace NIA function symbols $\text{div}_I, \text{mod}_I$, which do not correspond to any interpreted function in NRA, using freshly added function symbols frac, umod , obtaining formula f_1 :

$$t \text{ div}_I s \rightsquigarrow t/_R s -_R \text{frac}(t, s) \quad t \text{ mod}_I s \rightsquigarrow \text{umod}(t, s)$$

Intuitively, we subtract $\text{frac}(t, s)$ from $t/_R s$ to make sure that any integer model of $t \text{ div}_I s$ will also be a model of its real translation.

Next we axiomatize the new function symbols. We provide a definition for the function umod , two axioms partially constraining frac , and an axiom constraining all the real variables:

$$\text{umod}(x, y) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } 0 \leq_R x <_R y \\ x -_R y & \text{if } y \leq_R x <_R 2 *_R y \\ x +_R y & \text{if } -_R y \leq_R x <_R 0 \\ y *_R \text{frac}(x, y) & \text{else} \end{cases} \quad (2)$$

$$\text{ax_frac_bound}(x, y) \stackrel{\text{def}}{=} (y >_R 0 \rightarrow 0 \leq_R \text{frac}(x, y) <_R 1) \wedge (y <_R 0 \rightarrow 0 \geq_R \text{frac}(x, y) >_R -1) \quad (3)$$

$$\text{ax_frac_zero}(x, y, z, w) \stackrel{\text{def}}{=} (x = z *_R w \wedge (y = z \vee y = w)) \rightarrow \text{frac}(x, y) = 0 \quad (4)$$

$$\text{ax_int_approx}(x) \stackrel{\text{def}}{=} x \leq_R -1 \vee x = 0 \vee x \geq_R 1 \quad (5)$$

The definition (2) is tailored to the occurrences of mod_I in f_{NI} produced by the Certora Prover, as described in Footnote 4. Axioms (3), (4) constrain $\text{frac}(x, y)$ to be between 0 and 1 (or 0 and -1 for negative y), and to be 0 when y is known to be a divisor of x . We instantiate them by selected terms occurring as arguments of $/_R, \text{umod}$ and $*_R$ in f_1 . Finally, we add an instance of (5) for each real variable in f_1 , to ensure that for all non-zero values n, m both $\text{abs}(n * m) \geq \text{abs}(n)$ and $\text{abs}(n * m) \geq \text{abs}(m)$ hold, and for all n and non-zero m also $\text{abs}(n/m) \leq \text{abs}(n)$ holds, as it would for integer n, m . We obtain the formula f_2 from f_1 by adding the definitions (2)-(5), and instances of axioms (3)-(5) as described above. Intuitively, f_2 is an overapproximation of f_{NI} , because any model of f_{NI} can be extended to a model of f_2 by interpreting $\text{frac}(n, m)$ as $n/m - (n \text{ div } m)$ for integers n, m . The instances of axioms (3)-(5) hold in this model, because they are only instantiated for terms from f_1 , which are in the considered model interpreted by integer values.

The third step of the NRA overapproximation method is based on the following observation: If we want to prove the conjecture $a \leq_I b$ in NIA, we can do it by proving $a -_R 1 <_R b$ in NRA.

From the proof-by-refutation point of view, $\neg(a -_R 1 <_R b)$ is an overapproximation of $\neg(a \leq_I b)$. Thus, to tighten the overapproximation f_2 of f_{NI} , we relax some of the inequalities in f_1 (not the inequalities in the axiom and function definitions (2)-(5)). We first replace each inequality $t <_R s$ or $t >_R s$ occurring in the f_1 part of f_2 with positive polarity by an equivalent inequality adding one more negation to obtain an inequality using \geq_R or \leq_R with negative polarity:

$$t <_R s \rightsquigarrow \neg(t \geq_R s) \quad t >_R s \rightsquigarrow \neg(t \leq_R s)$$

Then we relax each inequality $t \leq_R s$ and $t \geq_R s$ occurring in the f_1 part of f_2 with negative polarity, obtaining the formula f_{NR} :

$$t \leq_R s \rightsquigarrow t -_R 1 <_R s \quad t \geq_R s \rightsquigarrow t +_R 1 >_R s$$

Example 2. Using the method of this subsection on the formula (1) from Example 1, we obtain the formula consisting of the axiom definitions (3)-(5) (since (1) does not use `umod`, we omit the definition (2)) and the following:

$$\begin{aligned} & \neg(t_S -_R 1 <_R 0) \wedge \neg(t_M -_R 1 <_R 0) \wedge w_S \geq_R 0 \wedge w_M \geq_I 0 \wedge n_S \geq_R 0 \wedge t'_S \geq_R 0 \wedge t'_M \geq_R 0 \wedge w'_M \geq_R 0 \\ & \wedge w_M = (w_S *_R t_M) / r_{t_S} -_R \mathbf{ufrac}(w_S *_R t_M, t_S) \wedge t'_M = t_M +_R ((n_S *_R t_M) / r_{t_S} -_R \mathbf{ufrac}(n_S *_R t_M, t_S)) \\ & \wedge t'_S = t_S +_R n_S \wedge w'_M = (w_S *_R t'_M) / r_{t'_S} -_R \mathbf{ufrac}(w_S *_R t'_M, t'_S) \wedge \neg(w_M +_R 1 >_R w'_M) \\ & \wedge \mathbf{ax_frac_bound}(w_S *_R t_M, t_S) \wedge \mathbf{ax_frac_zero}(w_S *_R t_M, t_S, w_S, t_M) \wedge \mathbf{ax_frac_zero}(w_S *_R t_M, t_S, n_S, t_M) \\ & \wedge \mathbf{ax_frac_zero}(w_S *_R t_M, t_S, w_S, t'_M) \wedge \mathbf{ax_frac_bound}(n_S *_R t_M, t_S) \wedge \mathbf{ax_frac_zero}(n_S *_R t_M, t_S, w_S, t_M) \\ & \wedge \mathbf{ax_frac_zero}(n_S *_R t_M, t_S, n_S, t_M) \wedge \mathbf{ax_frac_zero}(n_S *_R t_M, t_S, w_S, t'_M) \wedge \mathbf{ax_frac_bound}(w_S *_R t'_M, t'_S) \\ & \wedge \mathbf{ax_frac_zero}(w_S *_R t'_M, t'_S, w_S, t_M) \wedge \mathbf{ax_frac_zero}(w_S *_R t'_M, t'_S, n_S, t_M) \wedge \mathbf{ax_int_approx}(t_S) \\ & \wedge \mathbf{ax_frac_zero}(w_S *_R t'_M, t'_S, w_S, t'_M) \wedge \mathbf{ax_int_approx}(t_M) \wedge \mathbf{ax_int_approx}(w_S) \wedge \mathbf{ax_int_approx}(w_M) \\ & \wedge \mathbf{ax_int_approx}(n_S) \wedge \mathbf{ax_int_approx}(t'_S) \wedge \mathbf{ax_int_approx}(t'_M) \wedge \mathbf{ax_int_approx}(w'_M) \end{aligned}$$

When encoded in the SMT-LIB2 syntax, this formula is proved unsatisfiable by the SMT solvers YICES2 and CVC5. Its SMT-LIB2 encoding is displayed in Appendix A.2.

4 Experimental Evaluation

Experimental setup. Our experimental setup mirrors the setup used by the Certora Prover. We considered the NIA encoding of 792 verification problems stemming from customer contracts that were under verification by Certora, and compared the performance of four SMT solvers on these benchmarks against overapproximated versions of the benchmark set.

The LIA overapproximation method was implemented directly in the Certora Prover. The Certora Prover hence produces both the natural NIA encoding of the problems and their LIA overapproximations. In order to prototype the NRA overapproximation, we implemented the method as a formula-to-formula transformation in Python using the tokenizer of PYSMT [15]. The formulas produced by the Certora Prover and our artifact are in SMT-LIB2 format [4]. All benchmarks are available at <https://github.com/Certora/PublicBenchmarks/tree/master/Sep2022>. The artifact for NRA overapproximation is available at https://github.com/hzzv/scripts/tree/main/real_relaxation.

The 792 benchmarks originate from two different encodings of 396 problems: one encoding in QF_UFNIA, one in QF_UFDTNIA. The encodings differ in how they model hash functions used by smart contracts for memory addressing. The QF_UFNIA encoding represents the hash functions using uninterpreted functions while the QF_UFDTNIA encoding uses inductive datatypes. Since the hash functions are only used for memory layout, the hash function representation

mostly does not interact with non-linear arithmetic. Thus, there is in general no interplay of datatypes and NIA in the QF_UFDTNIA benchmark set. Out of both benchmark sets, we excluded 300 benchmarks found to be satisfiable in QF_UFNIA or QF_UFDTNIA during our experiments, resulting in 246 benchmarks in each set. For these benchmarks we generated the overapproximation sets QF_UFLIA, QF_UFDTLIA, and QF_UFNRA, QF_UFDTNRA. In the sequel we refer to the sets QF_UFNIA, QF_UFLIA, QF_UFNRA together as QF_UF+, and to the sets QF_UFDTNIA, QF_UFDTLIA, QF_UFDTNRA together as QF_UFDT+.

We compared the performance of CVC5 v1.0.2 [1], MATHSAT v5.6.8 [9], YICES2 v2.6.4 [12], and Z3 v4.11.0 [11] on the original and the overapproximated benchmarks. These are all the non-portfolio and non-wrapper SMT solvers that competed in SMT-COMP 2022 [2] in the single query track of division QF_NonLinearIntArith. We ran all the experiments on computers with 32 cores (AMD Epyc 7502, 2.5 GHz) and 1 TB RAM using the benchmarking tool BENCHEXEC [5]. We used the time limit of 300 seconds and the memory limit of 16 GB per problem, corresponding to the time limits used in practice by the Certora Prover.

Experimental results. From the 492 benchmarks, 482 use integer division and 490 use modulo. Multiplication is used on average 83 times per problem, division 37 times per problem, and modulo 47 times per problem. Both overapproximations therefore added many instances of axioms related to multiplication, division and modulo. Further, since the Certora Prover internally converts the smart contracts to a static-single-assignment form and hence introduces many auxiliary variables (the average number of variables in a problem is 1361), the NRA overapproximation also added many instances of the axiom (5). Consequently, the LIA and NRA benchmarks were on average 70% and 243% larger in file size, respectively, than the original benchmarks. We note that even though instantiating the axioms increases the benchmark size significantly, in our experience it leads to better solver performance than using quantified axioms instead.

We say that a benchmark was solved to convey that it was proved unsatisfiable. The main goal of using the overapproximations is to verify smart contracts that were not verified in the NIA encoding. The key metric is therefore the number of *newly solved problems*, i.e., problems solved in an overapproximation but not solved in the NIA encoding, no matter which solver solves them. Hence in the result analysis, we use the concept of a *virtual best solver* (VBS) – a hypothetical solver which solves each problem as fast as the fastest solver we ran. E.g., if we have two benchmarks, one benchmark solved by CVC5 in 1 second and by Z3 in 2 seconds, and another benchmark solved only by Z3 in 1 second, then the VBS is considered to have solved both benchmarks with the runtime of 1 second for each of them.

We ran all four SMT solvers on the benchmarks in QF_UF+, but since MATHSAT and YICES2 do not support datatypes, only the solvers CVC5 and Z3 on the benchmarks in QF_UFDT+. The results are displayed in Figure 1. Both the LIA and NIA overapproximations led to 30 newly solved benchmarks compared to the original benchmark set (28% increase). If we combined the results for both overapproximations, the VBS solved 50 and 59 benchmarks in at least one of the overapproximations originating from QF_UFNIA and QF_UFDTNIA, respectively. 19 and 23 benchmarks were newly solved in at least one of the overapproximations of QF_UFNIA and QF_UFDTNIA, respectively, corresponding to a 39% increase in the number of smart contracts we were able to verify compared to the NIA encoding.

Interestingly, the NRA overapproximation significantly increased the number of solved benchmarks for all solvers except for Z3. A large majority of these benchmarks were newly solved, suggesting that the techniques used by the solvers for NIA are orthogonal to our NRA overapproximation, and that the solvers might potentially benefit from using a similar method.

QF_UF+	NIA	LIA (new)	NRA (new)
cvc5	8	7 (7)	13 (13)
MATHSAT	10	1 (1)	40 (31)
YICES2	0	7 (7)	34 (34)
Z3	47	25 (9)	20 (7)
virtual best solver	52	34 (11)	42 (14)
QF_UFDT+	NIA	LIA (new)	NRA (new)
cvc5	9	8 (6)	43 (39)
Z3	55	49 (19)	25 (13)
virtual best solver	57	52 (19)	47 (16)

Figure 1: Numbers of solved problems in the respective benchmark sets. Columns “LIA” and “NRA” refer to the benchmark overapproximations. In the parentheses is the number of problems solved in the respective overapproximated benchmark set, but not in the original NIA benchmark set. The “virtual best solver” row corresponds to the best result for each benchmark – i.e., the numbers in this row are the counts of benchmarks solved by any solver.

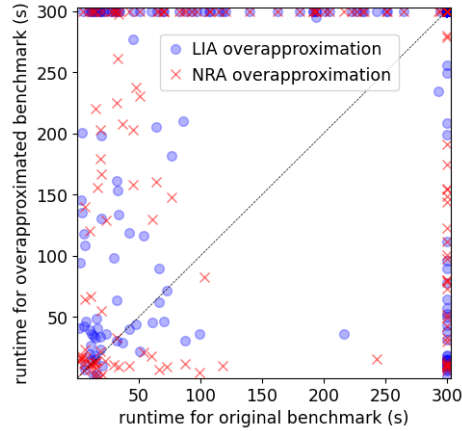


Figure 2: Comparison of the runtimes on original and overapproximated benchmark sets. We compare runtimes for the virtual best solver – for each problem we consider the best runtime achieved by any of the four solvers.

Figure 2 displays a comparison of the runtimes for the VBS for all the problems in the original and the two overapproximated benchmark sets. We used the actual best runtime for problems solved as unsatisfiable, but for problems with only non-unsatisfiable results we set the runtime to 300 seconds to convey that none of the solvers was able to solve the problem in the given time limit. There were 56 problems solved as unsatisfiable in both the original and the LIA overapproximated set, and 59 such problems for the original and the NRA overapproximated set. 40 problems were solved faster in the original set compared to LIA overapproximations, and 35 faster in the original set compared to NRA overapproximations.

Measuring the Overapproximation Tightness. In order to measure how tight are both overapproximations, we look at the numbers of problems solved as unsatisfiable in the original set, but either solved as satisfiable or not solved at all in the overapproximated set. Out of the 109 originally unsatisfiable problems, in the LIA overapproximation no problems were solved as satisfiable, while 53 were not solved at all. In the NRA overapproximation, 24 originally unsatisfiable problems were solved as satisfiable and 26 were not solved at all. We thus conclude that the LIA overapproximation is quite tight and is unlikely to benefit from adding more axioms. On the other hand, the NRA overapproximation could be tightened.⁵

Applications in Other Domains. We also tried running our NRA overapproximation tool on the QF_NIA and QF_UFNIA sets of the SMT-LIB benchmark library [3]. However, the QF_NIA set only contains a small number of problems using either division or modulo, and the QF_UFNIA set only contains non-linear operations in function definitions, for which the method

⁵We tried using a modification of the axiom (3) bounding $\text{ufrac}(x, y)$ to be at most $1 - \frac{1}{R} \frac{1}{y}$ for $y >_R 0$ and at least $-1 - \frac{1}{R} \frac{1}{y}$ for $y <_R 0$. However, compared to (3), this tightening did not increase the number of newly solved problems, nor decrease the number of originally unsatisfiable problems being solved as satisfiable.

does not add any axiom instances. Hence, it is not surprising that the SMT solvers were in general less successful on the overapproximated benchmarks compared to the original benchmarks. The only notable exception was the performance of Yices on the QF_NIA benchmarks, where out of 10343 total problems it solved 5243 in the original set, but 5992 benchmarks in the overapproximated set. Therefore, we conclude that for successful application of overapproximation methods, it is crucial to fine-tune the methods for the specific domain.

5 Related Work

Contemporary SMT solvers implement a range of methods for reasoning with QF_NIA, such as bit-blasting [13] (APROVE [13], CVC5 [1], Z3 [11], and SMT-RAT [10]), linearization [7] (BARCELOGIC [6]), incremental linearization [8] (MATHSAT [9], CVC5, and Z3), and NRA overapproximation combined with branch-and-bound [16, 17] (SMT-RAT, Z3, and YICES2 [12]).

The overapproximations we evaluate in this case study are similar in spirit to that of [8, 16, 17] (not [7], since that is geared towards finding models, while the overapproximations we focused on aim for preservation of unsatisfiability). However, all these approaches use iterative refinements of the overapproximation, while the overapproximation methods we compared work on top of SMT solvers: they only create one overapproximation and pass it to a solver supporting LIA or NRA. Further, the overapproximation methods we focused on are specialized for the domain of smart contract verification (e.g., using a specialized definition of `umod`).

6 Conclusions

In this case study we focused on overapproximation methods for verification of smart contracts using NIA. We described and evaluated two overapproximation methods using LIA and NRA. Our results show that both methods lead to solving a large number of industry benchmarks that were not solved in their natural NIA encoding, emphasizing the benefit of domain-specific verification methods. Further, our evaluation indicates that the NRA overapproximation can be further refined. Finally, our results also suggest that the NRA overapproximation method could be combined with existing methods for real overapproximation used by SMT solvers.

Acknowledgements. We thank Ján Hozza, Laura Kovács, and Gereon Kremer for fruitful discussions. We also thank Shelly Grossman, Jochen Hoenicke, and Mooly Sagiv. This work was partially funded by the ERC CoG ARTIST 101002685, and the FWF grants LogiCS W1255-N23 and LOCOTES P 35787.

References

- [1] Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: `cvc5`: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science*, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
- [2] Barbosa, H., Bobot, F., Hoenicke, J.: 17th International Satisfiability Modulo Theories Competition (SMT-COMP 2022). <https://smt-comp.github.io/2022/> (2022)

- [3] Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
- [4] Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard Version 2.6. Tech. rep. (2021)
- [5] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer* **21**(1), 1–29 (2019)
- [6] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: *International Conference on Computer Aided Verification*. pp. 294–298. Springer (2008)
- [7] Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning* **48**(1), 107–131 (2012)
- [8] Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Experimenting on solving nonlinear integer arithmetic with incremental linearization. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2018*. pp. 383–398. Springer International Publishing, Cham (2018)
- [9] Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) *Proceedings of TACAS. LNCS*, vol. 7795. Springer (2013)
- [10] Corzilius, F., Kremer, G., Junges, S., Schupp, S., Abraham, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: *International Conference on Theory and Applications of Satisfiability Testing*. pp. 360–368. Springer (2015)
- [11] De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Proc. of TACAS. LNCS*, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
- [12] Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) *Computer-Aided Verification (CAV’2014)*. *Lecture Notes in Computer Science*, vol. 8559, pp. 737–744. Springer (July 2014)
- [13] Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) *Theory and Applications of Satisfiability Testing – SAT 2007*. pp. 340–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
- [14] Gallier, J.H.: *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications (2015)
- [15] Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: *SMT Workshop 2015* (2015)
- [16] Jovanović, D.: Solving nonlinear integer arithmetic with MCSAT. In: *International Conference on Verification, Model Checking, and Abstract Interpretation*. pp. 330–346. Springer (2017)
- [17] Kremer, G., Corzilius, F., Abraham, E.: A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In: *International Workshop on Computer Algebra in Scientific Computing*. pp. 315–335. Springer (2016)
- [18] Matiyasevich, Y.V.: *Hilbert’s Tenth Problem*. MIT Press, Cambridge, MA, USA (1993)
- [19] Szabo, N.: *Smart contracts* (1994)
- [20] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* **151**(2014), 1–32 (2014)

A Appendix

A.1 SMT-LIB Encoding of Example 1

To encode our running example into SMT-LIB2, we rename the variables $t_S, t'_S, t_M, t'_M, w_M, w'_M, n_S, w_S$ to `ts1, ts2, tm1, tm2, wm1, wm2, ns, ws`, respectively.

The encoding of the formula (1) from Example 1:

```
(set-logic QF_NIA)

(declare-const ts1 Int)
(declare-const ts2 Int)
(declare-const tm1 Int)
(declare-const tm2 Int)
(declare-const wm1 Int)
(declare-const wm2 Int)
(declare-const ns Int)
(declare-const ws Int)

(assert (and
  (> ts1 0)
  (> tm1 0)
  (>= ws 0)
  (>= wm1 0)
  (>= ns 0)
  (>= ts2 0)
  (>= tm2 0)
  (>= wm2 0)
  (assert (= wm1 (div (* ws tm1) ts1)))
  (assert (= ts2 (+ ts1 ns)))
  (assert
    (= tm2 (+ tm1 (div (* ns tm1) ts1))))
  (assert (= wm2 (div (* ws tm2) ts2)))
  (assert (not (>= wm1 wm2)))
))

(check-sat)
```

A.2 SMT-LIB Encoding of Example 2

We overapproximated the SMT-LIB2 formula above using our script. The script considers relaxing each inequality individually, resulting in the inequalities $t_S -_R 1 \geq 0$ and $t_M -_R 1 \geq 0$ instead of the equivalent $\neg(t_S -_R 1 <_R 0)$ and $\neg(t_M -_R 1 <_R 0)$ from formula (2). We also added comments annotating parts of the encoding. Otherwise the resulting formula matches the complete relaxed formula from Example 2:

```
(set-logic QF_UFNRA)

;; Newly added functions
(declare-fun ufrac (Real Real) Real)
(define-fun ax_frac_bound
  ((x Real) (y Real)) Bool
  (and
    (> (> y 0) (and
      (<= 0.0 (ufrac x y))
      (< (ufrac x y) 1.0)))
    (> (< y 0) (and
      (>= 0.0 (ufrac x y))
      (> (ufrac x y) (- 1.0))))
  ))
(declare-fun ax_frac_zero
  ((x Real) (y Real) (z Real) (w Real))
  Bool
  (> (and (= x (* z w))
    (or (= y z) (= y w)))
    (= (ufrac x y) 0))
  )
(declare-fun ax_int_approx
  ((x Real)) Bool
  (or (= x 0) (>= x 1) (<= x (- 1)))
  )

;; Translated declarations
(declare-const ts1 Real)
(declare-const ts2 Real)
(declare-const tm1 Real)
(declare-const tm2 Real)
(declare-const wm1 Real)
(declare-const wm2 Real)
(declare-const ns Real)
(declare-const ws Real)

;; Newly added axiom instances:
(assert (ax_frac_bound (* ws tm2) ts2))
(assert
  (ax_frac_zero (* ws tm2) ts2 ws tm2))
(assert
  (ax_frac_zero (* ws tm2) ts2 ns tm1))
(assert
  (ax_frac_zero (* ws tm2) ts2 ws tm1))
(assert (ax_frac_bound (* ns tm1) ts1))
(assert
  (ax_frac_zero (* ns tm1) ts1 ws tm2))
(assert
```

<pre> (ax_frac_zero (* ns tm1) ts1 ns tm1)) (assert (ax_frac_zero (* ns tm1) ts1 ws tm1)) (assert (ax_frac_bound (* ws tm1) ts1)) (assert (ax_frac_zero (* ws tm1) ts1 ws tm2)) (assert (ax_frac_zero (* ws tm1) ts1 ns tm1)) (assert (ax_frac_zero (* ws tm1) ts1 ws tm1)) (assert (ax_int_approx wm2)) (assert (ax_int_approx ts1)) (assert (ax_int_approx ns)) (assert (ax_int_approx ws)) (assert (ax_int_approx wm1)) (assert (ax_int_approx tm1)) (assert (ax_int_approx tm2)) (assert (ax_int_approx ts2)) ;; Relaxed assertions: (assert (and (>= (- ts1 1) 0) </pre>	<pre> (>= (- tm1 1) 0) (>= ws 0) (>= wm1 0) (>= ns 0) (>= ts2 0) (>= tm2 0) (>= wm2 0))) (assert (= wm1 (- (/ (* ws tm1) ts1) (ufrac (* ws tm1) ts1)))) (assert (= ts2 (+ ts1 ns))) (assert (= tm2 (+ tm1 (- (/ (* ns tm1) ts1) (ufrac (* ns tm1) ts1)))))) (assert (= wm2 (- (/ (* ws tm2) ts2) (ufrac (* ws tm2) ts2)))) (assert (not (> (+ wm1 1) wm2))) (check-sat) </pre>
---	--

A.3 A Sample Smart Contract Verification Problem

Example 1 corresponds to a part of a simplified version of the following smart contract verification problem. The variables $t_S, t_M, w_M, w'_M, n_S, w_S$ correspond to variables `elastic1, base1, toBase1, toBase2, addAmount, someValue`, respectively.

The original specification in the Certora Verification Language syntax:⁶

<pre> methods { function getElastic() external returns (uint128) envfree; function getBase() external returns (uint128) envfree; function toBaseFloor(uint256 elastic) external returns (uint256) envfree; function addFloor(uint256 elastic) external returns (uint256) envfree; } /** x and y are almost equal; y may be * smaller than x up to epsilon */ definition only_slightly_larger_than(uint x, uint y, uint epsilon) returns bool = y <= x && x <= require_uint256(y + epsilon); /** Check that adding some amount does * not impact the elastic/base * quotient beyond the given error * margin. */ rule integrityOnAdd { uint128 addAmount; uint128 someValue; </pre>	<pre> uint128 elastic1 = getElastic(); uint128 base1 = getBase(); /** these cases are handled * separately */ require base1 != 0; require elastic1 != 0; uint256 toBase1 = toBaseFloor(someValue); /** using ~Floor version so the * rounding error is in favour * of the "bank" */ uint256 addAmountBase = addFloor(addAmount); uint256 toBase2 = toBaseFloor(someValue); uint256 sum1 = require_uint256(elastic1 + addAmount); uint256 error_margin = require_uint256((someValue / sum1) + 1); </pre>
---	--

⁶<https://docs.certora.com/en/latest/docs/cvl/index.html>

```

assert only_slightly_larger_than(
    toBase1, toBase2, error_margin),
    "addFloor(..) may not change the
}
    result of toBaseFloor(someValue)
    significantly; but does so.";
}

```

The corresponding contract in Solidity:

```

pragma solidity 0.6.12;
pragma experimental ABIEncoderV2;

/** verification harness contract */
contract RebaseWrapper {
    using BoringMath for uint256;
    using BoringMath128 for uint128;

    Rebase public rebase;

    function getElastic() public view
        returns (uint128) {
        return rebase.elastic;
    }

    function getBase() public view
        returns (uint128) {
        return rebase.base;
    }

    function toBaseFloor(uint256 elastic)
        public view returns
        (uint256 base) {
        if (rebase.elastic == 0) {
            base = elastic;
        } else {
            base = elastic.mul(rebase.base) /
                rebase.elastic;
        }
    }

    function addFloor(uint256 elastic)
        public returns (uint256 base) {
        base = toBaseFloor(elastic);
        rebase.elastic =
            rebase.elastic.add(
                elastic.to128());
        rebase.base =
            rebase.base.add(base.to128());
        return base;
    }
}

/** from BoringMath.sol */
/** @notice A library for performing
 * over-/underflow-safe math, updated
 * with awesomeness from DappHub
 * https://github.com/dapphub/ds-math
 */
library BoringMath {
    function mul(uint256 a, uint256 b)
        internal pure returns
        (uint256 c) {
        require(b == 0 ||
            (c = a * b) / b == a,
            "BoringMath: Mul Overflow");
    }

    function to128(uint256 a) internal
        pure returns (uint128 c) {
        require(a <= uint128(-1),
            "BoringMath: uint128 Overflow");
        c = uint128(a);
    }
}

/** @notice A library for performing
 * over-/underflow-safe addition
 * and subtraction on uint128. */
library BoringMath128 {
    function add(uint128 a, uint128 b)
        internal pure returns
        (uint128 c) {
        require((c = a + b) >= b,
            "BoringMath: Add Overflow");
    }
}

/** from BoringRebase.sol */
struct Rebase {
    uint128 elastic;
    uint128 base;
}

```

Note that the code and comments above were edited for conciseness and clarity. The code includes the smart contract written by Certora during the verification process, and the relevant parts from libraries `BoringMath.sol` and `BoringRebase.sol`, version corresponding to commit [9f6d870⁷](https://github.com/boringcrypto/BoringSolidity/tree/9f6d8708aa5df8b5d6cdad4a917e37b14c348684/contracts/libraries).

⁷<https://github.com/boringcrypto/BoringSolidity/tree/9f6d8708aa5df8b5d6cdad4a917e37b14c348684/contracts/libraries>