SolType: Refinement Types for Arithmetic Overflow in Solidity

BRYAN TAN, University of California, Santa Barbara, USA
BENJAMIN MARIANO, University of Texas at Austin, USA
SHUVENDU K. LAHIRI, Microsoft Research, USA
ISIL DILLIG, University of Texas at Austin, USA
YU FENG, University of California, Santa Barbara, USA

As smart contracts gain adoption in financial transactions, it becomes increasingly important to ensure that they are free of bugs and security vulnerabilities. Of particular relevance in this context are arithmetic overflow bugs, as integers are often used to represent financial assets like account balances. Motivated by this observation, this paper presents SolType, a refinement type system for Solidity that can be used to prevent arithmetic over- and under-flows in smart contracts. SolType allows developers to add refinement type annotations and uses them to prove that arithmetic operations do not lead to over- and under-flows. SolType incorporates a rich vocabulary of refinement terms that allow expressing relationships between integer values and aggregate properties of complex data structures. Furthermore, our implementation, called Solid, incorporates a type inference engine and can automatically infer useful type annotations, including non-trivial contract invariants.

To evaluate the usefulness of our type system, we use Solid to prove arithmetic safety of a total of 120 smart contracts. When used in its fully automated mode (i.e., using Solid's type inference capabilities), Solid is able to eliminate 86.3% of redundant runtime checks used to guard against overflows. We also compare Solid against a state-of-the-art arithmetic safety verifier called VeriSmart and show that Solid has a significantly lower false positive rate, while being significantly faster in terms of verification time.

CCS Concepts: • **Theory of computation → Program verification**; *Invariants*; *Type theory*.

Additional Key Words and Phrases: refinement type inference, smart contracts, integer overflow

ACM Reference Format:
Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: Refinement Types for Arithmetic Overflow in Solidity. *Proc. ACM Program. Lang.* 6, POPL, Article 4 (January 2022), 29 pages. https://doi.org/10.1145/3498665

1 INTRODUCTION

Smart contracts are programs that run on top of the blockchain and perform financial transactions in a distributed environment without intervention from trusted third parties. In recent years, smart contracts have seen widespread adoption, with over 45 million [Etherscan 2021] instances covering financial products, online gaming, real estate [M. 2018], shipping, and logistics [Straight 2020].

Authors' addresses: Bryan Tan, University of California, Santa Barbara, USA, bryantan@cs.ucsb.edu; Benjamin Mariano, University of Texas at Austin, USA, bmariano@cs.utexas.edu; Shuvendu K. Lahiri, Microsoft Research, USA, Shuvendu.Lahiri@microsoft.com; Isil Dillig, University of Texas at Austin, USA, isil@cs.utexas.edu; Yu Feng, University of California, Santa Barbara, USA, yufeng@cs.ucsb.edu.

2475-1421/2022/1-ART4
https://doi.org/10.1145/3498665

Because smart contracts deployed on a blockchain are freely accessible through their public methods, any functional bugs or vulnerabilities inside the contracts can lead to disastrous losses, as demonstrated by recent attacks [Atzei et al. 2017; Mix 2018; Palladino 2017; Siegel 2016]. Therefore, unsafe smart contracts are increasingly becoming a serious threat to the success of the blockchain technology. For example, recent infamous attacks on the Ethereum blockchain such as the DAO [Siegel 2016] and the Parity Wallet [Palladino 2017] attacks were caused by unsafe smart contracts. To make things worse, smart contracts are immutable once deployed, so bugs cannot be easily fixed.

One common type of security vulnerability involving smart contracts is arithmetic overflows. In fact, according to a recent study, such bugs account for over 96% of CVEs assigned to Ethereum smart contracts [So et al. 2020]. Furthermore, because smart contracts often use integers to represent financial assets, arithmetic bugs, if exploited, can cause significant financial damage [Mix 2018]. For this reason, smart contracts rarely *directly* perform arithmetic operations (e.g., addition) but instead perform arithmetic indirectly by calling a library called SafeMath. Since this library inserts run-time checks for over- and under-flows, this approach is effective at preventing exploitable vulnerabilities but comes with run-time overhead. Furthermore, because computation on the blockchain costs money (measured in a unit called *gas*), using the SafeMath library can significantly increase the cost of running these smart contracts.

Motivated by this observation, we are interested in *static* techniques that can be used to prevent integer overflows. Specifically, we propose a refinement type system for Solidity, called SolType, that can be used to prove that arithmetic operations do not over- or under-flow. Arithmetic operations that are proven type-safe by SolType are guaranteed to not overflow; thus, SolType can be used to eliminate unnecessary runtime checks.

While SolType is inspired by prior work on logically qualified types [Rondon et al. 2010, 2008; Vekris et al. 2016], it needs to address a key challenge that arises in Solidity programs. Because smart contracts often use non-trivial data structures (e.g., nested mappings or mappings of structs) to store information about accounts, it is important to reason about the relationship between integer values and aggregate properties of such data structures. Based on this observation, SolType allows relational-algebra-like refinements that can be used to perform projections and aggregations over mappings. While this design choice makes Solid expressive enough to discharge many potential overflows, it nonetheless enjoys decidable type checking. Furthermore, our implementation, called Solid, incorporates a type inference engine based on Constrained Horn Clause (CHC) solvers [Bjørner et al. 2015] and can therefore automatically infer useful type annotations, including non-trivial contract invariants.

We evaluate Solid on 120 smart contracts and compare it against Verismart [So et al. 2020], a state-of-the-art verifier for checking arithmetic safety in smart contracts. Our evaluation shows that (1) Solid can discharge 86.3% of unnecessary runtime overflow checks, and (2) Solid is both faster and has a lower false positive rate compared to Verismart. We also perform ablation studies to justify our key design choices and empirically validate that they are important for achieving good results.

In summary, this paper makes the following contributions:

- We present a new refinement type system called SolType for proving arithmetic safety properties of smart contracts written in Solidity. Notably, our type system allows expressing relationships between integers and aggregate properties of complex data structures.
- We implement the proposed type system in a tool called Solid, which also incorporates a type inference procedure based on Constrained Horn Clause solvers.

```
 1 contract ExampleToken is IERC20 {
 2     address owner;
 3     uint tot;
 4     mapping(address => uint) /* { sum(v) <= tot } */ bals;
 5     constructor(address _owner) public {
 6         owner = _owner;
 7     }
 8
 9     function mint(uint amt) public {
10         require(msg.sender == owner);
11         require(tot + amt >= tot);
12         tot = tot + amt;
13         bals[msg.sender] = bals[msg.sender] + amt;
14     }
15
16     function transfer(address recipient, uint amt) public {
17         require(bals[msg.sender] >= amt);
18         bals[msg.sender] = bals[msg.sender] - amt;
19         bals[recipient] = bals[recipient] + amt;
20     }
21
22     /* ... other functions ... */
23 }
```

Fig. 1. Excerpt from a typical ERC20 token.

- We evaluate Solid on 120 real-world smart contracts and show that it is useful for eliminating many run-time checks for arithmetic safety. We also compare Solid against Verismart and demonstrate its advantages in terms of false positive rate and running time.

2 OVERVIEW

In the section, we go through the high-level idea of our refinement type system using a representative example.

2.1 Verifying Overflow Safety: A Simple Example

Fig. 1 shows a snippet from a typical implementation of an ERC20 token. Here, the `mint` function allows an authorized "owner" address to generate tokens, and the `transfer` function allows users of the contract to transfer tokens between each other. User token balances are tracked in the mapping `bals`, with the total number of issued tokens stored in `tot`. It is crucial that the arithmetic does not overflow in either function; otherwise it would be possible for a malicious agent to arbitrarily create or destroy tokens.

This contract does not contain any arithmetic overflows because it maintains the invariant that the summation over the entries in `bals` is at most `tot`. This invariant, together with the `require` clauses in `mint` and `transfer`, ensures that there are no overflows. Such *contract invariants* can be expressed as refinement type annotations in Solid. In particular, as shown in line 4, `bals` has the refinement type $\{\nu \mid \text{sum}(\nu) \leq \text{tot}\}$, which is sufficient to prove the absence of overflows in this program. In the remainder of this section, we illustrate some of the features of Solid using this example and its variants.

2.2 Intermediate Representation

Due to imperative features like loops and mutation, Solidity is not directly amenable to refinement type checking. Thus, similar to prior work [Rondon et al. 2010; Vekris et al. 2016], we first translate the contract to a intermediate representation (IR) in SSA form that we call MiniSol. Our IR has the following salient features:

- Variables are assigned exactly once (i.e., SSA form).
- Integers are represented as unbounded integers (i.e., integers in $\mathbb{Z}$) rather than as machine integers (i.e., integers modulo $2^{256} - 1$).
- Mappings, structs, and arrays are converted into immutable data structures.
- A *fetch* statement is used to load the values of the storage variables at function entry, and a *commit* statement is used to store the values of the storage variables at function exit.

The resulting IR of the example contract is shown in Fig. 2, which we will use for the examples in this section. Our tool automatically converts Solidity programs to this IR in a pre-processing step and automatically inserts *fetch*/*commit* statements at the necessary places. Similar to the *fold*/*unfold* operations from prior work [Rondon et al. 2010], the use of these *fetch*/*commit* statements in the MiniSol IR allows temporary invariant violations.

2.3 Discharging Overflow Safety on Local Variables

We now demonstrate how our refinement type system can be used to verify arithmetic safety in this example. Consider the `mint` function in Fig. 2. The require statement on line 16 is a runtime overflow check for `tot1 + amt`. If the runtime check fails, then the contract will abort the current transactions, meaning that it discards any changes to state variables. However, `tot1 + amt >= tot1` only works as a runtime overflow check when the integers are machine integers. Since Solid operates on mathematical integers, Solid will heuristically rewrite common overflow check patterns over machine integers[1], such as the one above, into a predicate over mathematical integers:

$$tot1 + amt \le \mathsf{MaxInt}$$

where MaxInt is a constant equal to the maximum machine integer ($= 2^{256} - 1$ for Solidity's `uint` type). Since Solid keeps track of this predicate as a *guard predicate*, it is able to prove the safety of the addition on line 17.

Next, we explain how Solid proves the safety of the addition on line 19. For now, let us assume that the refinement type annotation for `bals` is indeed valid, i.e., `bals1` has the following refinement type (we omit base type annotations for easier readability):

$$bals1 : \{v \mid \mathsf{Sum}(v) \le tot1\} \tag{1}$$

Since `bals1` contains only unsigned integers, we refine `bals1[msg.sender]` with the type:

$$bals1[msg.sender] : \{v \mid v \le \mathsf{Sum}(bals1)\}$$

From transitivity of $\le$, this implies that $bals1[msg.sender] \le tot1$. Finally, using the guard predicate $tot1 + amt \le \mathsf{MaxInt}$, Solid is able to prove:

$$bals1[msg.sender] + amt \le \mathsf{MaxInt}$$

which implies that the addition does not overflow. In a similar manner, Solid can also prove the safety of the operations in `transfer` using the refinement type of state variable `bals` and the `require` statement at line 25.

[1]Specifically, the patterns include `X + Y >= X` and `X + Y >= Y`, with similar patterns for multiplication.

```
 1  contract ExampleToken {
 2      owner : addr;
 3      bals : map(addr => uint) /* { sum(v) <= tot } */;
 4      tot : uint;
 5
 6      constructor(_owner : addr) public {
 7          let owner0 : addr = _owner;
 8          let bals0 : map(addr => uint) = zero_map[addr, uint];
 9          let tot0 : uint = 0;
10          commit owner0, bals0, tot0;
11      }
12
13      fun mint(amt : uint) {
14          fetch owner1, bals1, tot1;
15          require(msg.sender == owner1);
16          require(tot1 + amt >= tot1);
17          let tot2 : uint = tot1 + amt;
18          let bals2 : map(addr => uint) =
19            bals1[msg.sender ◁ bals1[msg.sender] + amt];
20          commit owner1, bals2, tot2;
21      }
22
23      fun transfer(recipient : address, amt : uint) {
24          fetch owner3, bals3, tot3;
25          require(bals3[msg.sender] >= amt);
26          let bals4 : map(addr => uint) =
27            bals3[msg.sender ◁ bals3[msg.sender] - amt];
28          let bals5 : map(addr => uint) =
29            bals4[recipient ◁ bals4[recipient] + amt];
30          commit owner3, bals5, tot3;
31      }
32  }
```

Fig. 2. MiniSol version of the example token (with some irrelevant details omitted)

2.4 Type Checking the Contract Invariant

In the previous subsection, we showed that contract invariants (expressed as refinement type annotations on state variables) are useful for discharging overflows. However, Solid needs to establish that these type annotations are valid, meaning that contract invariants are preserved at the end of functions. For instance, at the end of the `mint` function, the type checker needs to establish that `bals2` has the following refinement type:

$$bals2 : \{v \mid \mathsf{Sum}(v) \leq tot2\} \tag{2}$$

To establish this, we proceed as follows: First, from line 19, Solid infers the type of `bals2` to be:

$$bals2 : \{v \mid \mathsf{Sum}(v) = \mathsf{Sum}(bals1) + amt)\} \tag{3}$$

Then, since `tot2` has the refinement type $\{v \mid v = tot1 + amt\}$ and `bals1` has the refinement type $\{v \mid \mathsf{Sum}(v) \leq tot1\}$, we can show that $\mathsf{Sum}(bals2) \leq tot2$. This establishes the desired contract invariant (i.e., Eq. 2). Using similar reasoning, Solid can also type check that the contract invariant is preserved in `transfer`.

2.5 Contract Invariant Inference

Although the type annotation is given explicitly in this example, Solid can actually prove the absence of overflows without *any* annotations by performing type inference. To do so, Solid first introduces a ternary uninterpreted predicate I (i.e., unknown contract invariant) for each state variable (`owner, bals, tot`) that relates its value v to the other two state variables. This corresponds to the following type "annotations" for the state variables:

$$\begin{aligned} owner &: \{v \mid I_1(v, tot, bals)\} \\ tot &: \{v \mid I_2(owner, v, bals)\} \\ bals &: \{v \mid I_3(owner, tot, v)\} \end{aligned}$$

Given these type annotations over the unknown predicates I_1-I_3, Solid generates constraints on I_1-I_3 during type inference and leverages a CHC solver to find an instantiation of each I_i. However, one complication is that we cannot feed all of these constraints to a CHC solver because the generated constraints may be unsatisfiable. In particular, consider the scenario where the program contains a hundred arithmetic operations, only one of which is unsafe. If we feed all constraints generated during type checking to the CHC solver, it will correctly determine that there is no instantiation of the invariants that will allow us to prove the safety of *all* arithmetic operations. However, we would still like to infer a type annotation that will allow us to discharge the remaining 99 arithmetic operations.

To deal with this difficulty, Solid tries to solve the CHC variant of the well-known MaxSAT problem. That is, in the case where the generated Horn clauses are unsatisfiable, we would like to find an instantiation of the unknown predicates so that the maximum number of constraints are satisfied. Solid solves this "MaxCHC"-like problem by generating one overflow checking constraint at a time and iteratively strengthening the inferred contract invariant. For instance, for our running example, Solid can automatically infer the desired contract invariant I_3, namely `sum(bals)<= tot` (I_1 and I_2 are simply true).

2.6 Refinements for Aggregations over Nested Data Structures

A unique aspect of our type system is its ability to express more complex relationships between integer values and data structures like nested mappings over structs. To illustrate this aspect of the type system, consider Fig. 3, which is a modified version of Fig. 1 that stores balance values inside a `User` struct in a nested mapping. The addition on line 14 cannot overflow because the contract maintains the invariant that the summation over the `bal` fields of all the nested `User` entries in `usrs` is at most `tot`. Solid is able to infer that the type of *usrs* is

$$\{v \mid \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(v))) \leq tot\}$$

In particular, this refinement type captures the contract invariant that the sum of all user balances nested inside the struct of `usrs` is bounded by `tot`. Using such a refinement type, we can prove the safety of the addition on line 14. Note that the use of operators like Flatten and Fld_{bal} is vital for successfully discharging the potential overflow in this example.

3 LANGUAGE

In this section, we present the syntax of MiniSol (Figures 4 and 5), an intermediate representation for modeling Solidity smart contracts. In what follows, we give a brief overview of the important features of MiniSol relevant to the rest of the paper.

```
contract ExampleTokenWithStruct {
    struct User {
        uint bal;
        /* ... other fields ... */
    }

    /* ... other state variables ... */
    mapping(address => mapping(uint => User)) usrs;

    function mint(uint amt, uint accno) public {
        require(msg.sender == owner);
        require(tot + amt >= tot);
        tot += amt;
        usrs[msg.sender][accno].bal += amt;
    }

    /* ... other functions ... */
}
```

Fig. 3. The Solidity source code of the running example (Fig. 1) modified to use nested data structures.

3.1 MiniSol Syntax

In MiniSol, a program is a contract C that contains fields (referred to as *state variables*), struct definitions S, and function declarations f. State variables are declared and initialized in a constructor $ctor$. Functions include a type signature, which provides a type τ_i for each argument x_i as well as return type τ_r, and a method body, which is a sequence of statements, followed by an expression.

Statements in MiniSol include let bindings, conditionals, function calls (with pass-by-value semantics), assertions, and assumptions. As MiniSol programs are assumed to be in SSA form, we include a *join point* j for if statements and while loops. In particular, a join point consists of a list of Φ-node variable declarations, where each x_i is declared to be of type τ_i and may take on the value of either x_{i1} or x_{i2}, depending on which branch is taken. In line with this SSA assumption, we also assume variables are not redeclared.

As mentioned previously, another feature of MiniSol is that it contains fetch and commit statements for reading the values of all state variables from the blockchain and writing them to the blockchain respectively. Specifically, the construct

$$\text{fetch } x'_1 \text{ as } x_1, \ldots, x'_n \text{ as } x_n$$

simultaneously retrieves the values of state variables $x'_1, \ldots, x'_n$ and writes them into local variables $x_1, \ldots, x_n$. Similarly, the commit statement

$$\text{commit } e_1 \text{ to } x_1, \ldots, e_n \text{ to } x_n$$

simultaneously stores the result of evaluating expressions $e_1, \ldots, e_n$ into state variables $x_1, \ldots, x_n$. Note that Solidity does not contain such fetch and commit constructs; however, we include them in MiniSol to allow temporary violations of contract invariants inside procedures. Such mechanisms have also been used in prior work [Rondon et al. 2010; Smith et al. 2000] for similar reasons.

Expressions in MiniSol consist of variables, unsigned integer and boolean constants, binary operations, and data structure operations. Data structures include maps and structs, and we use the same notation for accessing/updating structs and maps. In particular, $e_1[e_2]$ yields the value stored at key (resp. field) e_1 of map (resp. struct) e_2. Similarly, $e_1[e_2 \triangleleft e_3]$ denotes the new map (resp.

C	::=	$\text{contract}\, C\ \{ctor\ \overrightarrow{decl}\}$	**Contract**
$ctor$	::=	$\text{ctor}(\overrightarrow{x : \tau}) = s$	**Constructor**
$decl$	::=		**Body declaration:**
		$\textbf{fun}\, f(\overrightarrow{x_i : \tau_i}) : \tau_r = s; e$	function
		$\textbf{struct}\, S\{\overrightarrow{x_i : T_i}\}$	struct
s	::=		**Statement:**
		$\text{let}\, x : \tau = e$	(immutable) assignment
	|	$s_1; s_2$	sequence
	|	skip	skip
	|	$\text{assert}\, e$	assertion
	|	$\text{assume}\, e$	assumption
	|	$\text{if}\, e_1\, \text{then}\, s_1\, \text{else}\, s_2\, \text{join}\, j$	if-then-else
	|	$\text{while}\, j; e\, \text{do}\, s$	while loop
	|	$\text{fetch}\, x'_1\, \text{as}\, x_1, \ldots, x'_n\, \text{as}\, x_n$	fetch state variables
	|	$\text{commit}\, e_1\, \text{to}\, x_1, \ldots, e_n\, \text{to}\, x_n$	commit state variables
	|	$\text{call}\, x : \tau = f(e_1, \ldots, e_n)$	function call
j	::=	$\overrightarrow{x_i : \tau_i = \Phi(x_{i1}, x_{i2})}$	**Join point**
e	::=		**Expression:**
		$n \mid \text{true} \mid \text{false} \mid ()$	constants
	|	x	variable
	|	$e_1 \oplus e_2 \mid \neg e$	binary/unary operation
	|	$\text{mapping}(\overrightarrow{n_i \mapsto e_i})$	mapping with uint keys
	|	$\text{struct}\, S(\overrightarrow{x_i \mapsto e_i})$	struct
	|	$e_1[e_2]$	data structure index
	|	$e_1\ [e_2 \triangleleft e_3]$	data structure update
	|	$.x$	field selector
$\oplus$	$\in$	$\{+, -, *, /, =, \neq, \geq, \leq, >, <, \wedge, \vee\}$	**Binary operators**

Fig. 4. Syntax of MiniSol programs

struct) obtained by writing value e_3 at key (resp. field) e_2 of e_1. For structs, we use a special type of expression called a *field selector* to indicate struct access (e.g. $e_1[.x]$).

3.2 Types

Similar to previous refinement type works, MiniSol provides both *base type* annotations as well as *refinement type* annotations (Figure 5). Base types correspond to standard Solidity types, such as unsigned integers (256-bit), booleans, mappings, and structs. We shorten "mappings" to Map and assume that all keys are UInts, as most key types in Solidity mappings are coercible to UInt. Thus, Map(T) is a mapping from unsigned integers to Ts.

A refinement type $\{\nu : T \mid \phi\}$ refines the base type T with a logical qualifier ϕ. In this paper, logical qualifiers belong to the quantifier-free combined theory of rationals, equality with uninterpreted functions, and arrays. Note that determining validity in this theory is decidable, and we intentionally

T	$::=$		**Base type:**
		UInt	unsigned integer
	$\mid$	Bool	boolean
	$\mid$	$\mathsf{Map}(T)$	mapping with unsigned int keys
	$\mid$	$\mathsf{Struct}\,S$	struct
τ	$::=$	$\{v : T \mid \phi\}$	**Refinement type**
ϕ	$::=$	$t \oplus t \mid \phi \land \phi \mid \phi \lor \phi \mid \neg\phi$	**Logical qualifiers**
t	$::=$		**Refinement terms:**
		e	expressions
	$\mid$	$\mathsf{Sum}(t)$	aggregate properties
	$\mid$	$\mathsf{Fld}_x(t) \mid \mathsf{Flatten}(t)$	aggregate operators
	$\mid$	MaxInt	largest actual integer

Fig. 5. Syntax of MiniSol types

use the theory of rationals rather than integers to keep type checking decidable in the presence of non-linear multiplication. In more detail, logical qualifiers are boolean combinations of binary relations $\oplus$ over refinement terms t, which include both MiniSol expressions as well as terms containing the special constructs Sum, Fld, and Flatten (represented as uninterpreted functions) that operate over terms of type Map. In particular,

- $\mathsf{Sum}(t)$ represents the sum of all values in t.
- Given a mapping t containing structs, $\mathsf{Fld}_x(t)$ returns a mapping that only contains field x of the struct. Thus, Fld is similar to the projection operator from relational algebra.
- Given a nested mapping t, $\mathsf{Flatten}(t)$ flattens the map. In particular, given a nested mapping $\mathsf{Map}(\mathsf{Map}(t))$, Flatten produces a mapping $\mathsf{Map}(t)$ where each value of the mapping corresponds to a value of one of the nested mappings from $\mathsf{Map}(\mathsf{Map}(t))$.

In addition to these constructs, refinement terms in MiniSol also include a constant called MaxInt, which indicates the largest integer value representable in UInt.

Example 1. Consider the `usrs` mapping in Fig. 3. The predicate

$$\mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(usrs))) \leq \mathsf{MaxInt}$$

expresses that if we take the `usrs` mapping (which is of type $\mathsf{Map}(\mathsf{Map}(User))$) and (a) flatten the nested mapping so that it becomes a non-nested mapping of *User*; (b) project out the `bal` field of each struct to obtain a $\mathsf{Map}(\mathsf{UInt})$; and (c) take the sum of values in this final integer mapping, then the result should be at most MaxInt.

4 TYPE SYSTEM

In this section, we introduce the SolType refinement type system for the MiniSol language.

4.1 Preliminaries

We start by discussing the environments and judgments used in our type system; these are summarized in Fig. 6.

Environments. Our type system employs a local environment Γ and a set of *guard* predicates γ. A local environment Γ is a sequence of type bindings (i.e., $x : \tau$), and we use the notation $\mathsf{dom}(\Gamma)$ to denote the set of variables bound in Γ. The guard predicates γ consist of (1) a set of path conditions

$$\begin{array}{lcll}
\Gamma & ::= & \epsilon \mid x : \tau, \Gamma & \text{(Local variable bindings)} \\
L & ::= & \text{Unlocked} \mid \text{Locked} & \text{(Lock status)} \\
\gamma & ::= & \epsilon \mid t, \gamma & \text{(Guard predicates)} \\
\Delta & ::= & \epsilon \mid \Delta, f \mapsto (\overrightarrow{x_i : \tau_i}, \tau_r) \mid & \text{(Global declarations)} \\
 & & \Delta, S \mapsto \overrightarrow{x_i : T_i} \mid \Delta, x : \tau &
\end{array}$$

$$\begin{array}{ll}
\Gamma; \gamma \vdash e : \tau & \text{(Expression typing judgment)} \\
\Gamma; L, \gamma \vdash s \dashv \Gamma'; L', \gamma' & \text{(Statement typing judgment)} \\
\Gamma; \gamma; (\Gamma_1; \gamma_1); (\Gamma_2; \gamma_2) \vdash j \dashv \Gamma' & \text{(Join point typing judgment)}
\end{array}$$

Fig. 6. Environments and judgments used in our typing rules

under which an expression is evaluated and (2) a *lock status* L (Locked or Unlocked) that tracks whether the state variables are currently being modified.

In addition to Γ and γ, our typing rules need access to struct and function definitions. For this purpose, we make use of another environment Δ that stores function signatures, struct definitions, and types of the state variables. To avoid cluttering notation, we assume that Δ is implicitly available in the expression and statement typing judgments, as Δ does not change in a function body. We use the notation $\Delta(S)$ to indicate looking up the fields of struct S, and we write $\Delta(S, x)$ to retrieve the type of field x of struct S. Similarly, the notation $\Delta(f)$ yields the function signature of f, which consists of a sequence of argument type bindings as well as the return type. Finally, given a state variable x, $\Delta(x)$ returns the type of x.

Typing Judgments. As shown in Fig. 6, there are three different typing judgments in MiniSol:

- *Expression typing judgments*: We use the judgment $\Gamma; \gamma \vdash e : \tau$ to type check a MiniSol expression e. In particular, this judgment indicates that expression e has refinement type τ under local environment Γ, guard γ, and (implicit) global environment Δ.
- *Statement typing judgments*: Next, we use a judgment of the form $\Gamma; L, \gamma \vdash s \dashv \Gamma'; L', \gamma'$ to type check statements. The meaning of this judgment is that statement s type checks under local environment Γ, lock status L, and guard γ, and it produces a new local type environment Γ', lock status L', and guard γ'. Note that the statement typing judgments are flow-sensitive in that they modify the environment, lock status, and guard; this is because (1) let bindings can add new variables to the local type environment, and (2) fetch/commit statements will modify the lock status and guard. When the guards are empty, we omit them from the judgment to simplify the notation.
- *Join typing judgments*: Finally, we have a third typing judgment for join points of conditionals. This judgment is of the form $\Gamma; \gamma; (\Gamma_1; \gamma_1); (\Gamma_2; \gamma_2) \vdash j \dashv \Gamma'$ where Γ, γ pertain to the state before the conditional, Γ_i, γ_i pertain to the state after executing each of the branches, and Γ' is the resulting environment after the join point.

Subtyping. Finally, our type system makes use of a subtyping judgment of the form:

$$\Gamma; \gamma \vDash \tau_1 <: \tau_2$$

which states that, under Γ and γ, the set of values represented by τ_1 is a subset of those represented by τ_2. As expected and as shown in Fig. 7, the subtyping judgment reduces to checking logical implication. In the Sub-Base rule, we use a function Encode to translate Γ, γ and the two qualifiers ϕ_1, ϕ_2 to logical formulas which belong to the quantifier-free fragment of the combined theory of rationals, arrays, equality with uninterpreted functions. Thus, to check whether τ_1 is a subtype

$$\textsc{Sub-Base}\ \frac{(\mathsf{Encode}(\Gamma) \land \mathsf{Encode}(\gamma) \land \mathsf{Encode}(\phi_1) \implies \mathsf{Encode}(\phi_2))\ \text{valid}}{\Gamma; \gamma \vDash \{v : T \mid \phi_1\} <: \{v : T \mid \phi_2\}}$$

Fig. 7. Subtyping relation

of τ_2, we can use an off-the-shelf SMT solver. Since our encoding of refinement types into SMT is the standard scheme used in [Rondon et al. 2010, 2008; Vekris et al. 2016], we do not explain the Encode procedure in detail.

Example 2. Consider the following subtyping judgment:

$$\Gamma; c \geq d \vDash \{v : \mathsf{UInt} \mid v = a\} <: \{v : \mathsf{UInt} \mid v \geq d\}$$

where Γ contains four variables a, b, c, d all with base type UInt and a has the refinement $v = b + c$. This subtyping check reduces to querying the validity of the following formula:

$$\phi_T \land a = b + c \land c \geq d \land v = a \implies v \geq d$$

where ϕ_T are additional clauses that restrict the terms of type UInt to be in the interval $[0, \mathsf{MaxInt}]$.

4.2 Refinement Typing Rules

In this section, we discuss the basic typing rules of SolType, starting with expressions. Note that the typing rules we describe in this subsection are *intentionally* imprecise for complex data structures involving nested mappings or mapping of structs. Since precise handling of complex data structures requires introducing additional machinery, we delay this discussion until the next subsection.

4.2.1 Expression Typing. Fig. 8 shows the key typing rules for expressions. Since the first two rules and the rules involving structs are standard, we focus on the remaining rules for arithmetic expressions and mappings.

Arithmetic expressions. The TE-Plus and TE-Mul rules capture the overflow safety properties of 256-bit integer addition and multiplication respectively. In particular, they check that the sum/product of the two expressions is less than MaxInt (when treating them as mathematical integers). The next rule, TE-Minus, checks that the result of the subtraction is not negative (again, when treated as mathematical integers). Finally, the TE-Div rule disallows division by zero, and constrains the result of the division to be in the appropriate range.

Mappings. The next two rules refer to reading from and writing to mappings. As stated earlier, we only focus on precise typing rules for variables of type Map(UInt) in this section and defer precise typing rules for more complex maps to the next subsection. Here, according to the Te-MapInd rule, the result of evaluating $e_1[e_2]$ is less than or equal to $\mathsf{Sum}(e_1)$, where Sum is an uninterpreted function. According to the next rule, called Te-MapUpd, if we write value e_3 at index e_2 of mapping e_1, the sum of the elements in the resulting mapping is given by:

$$\mathsf{Sum}(e_1) - e_1[e_2] + e_3$$

Thus, this rule, which is based on an axiom from a previous work [Permenev et al. 2020], allows us to precisely capture the sum of elements in mappings whose value type is UInt.

4.2.2 Statement Typing. Next, we consider the statement typing rules shown in Fig. 9. Again, since some of the rules are standard, we focus our discussion on aspects that are unique to our type system.

$$\frac{\Gamma;\gamma \vdash e : \tau \qquad \Gamma;\gamma \vDash \tau <: \tau'}{\Gamma;\gamma \vdash e : \tau'}\ \text{TE-Sub} \qquad \frac{\Gamma(x) = \{v : T \mid \phi\}}{\Gamma;\gamma \vdash x : \{v : T \mid v = x\}}\ \text{TE-Var}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{UInt} \qquad \Gamma;\gamma \vdash e_2 : \{v : \mathsf{UInt} \mid v + e_1 \leq \mathsf{MaxInt}\}}{\Gamma;\gamma \vdash e_1 + e_2 : \{v : \mathsf{UInt} \mid v = e_1 + e_2\}}\ \text{TE-Plus}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{UInt} \qquad \Gamma;\gamma \vdash e_2 : \{v : \mathsf{UInt} \mid v * e_1 \leq \mathsf{MaxInt}\}}{\Gamma;\gamma \vdash e_1 * e_2 : \{v : \mathsf{UInt} \mid v = e_1 * e_2\}}\ \text{TE-Mul}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{UInt} \qquad \Gamma;\gamma \vdash e_2 : \{v : \mathsf{UInt} \mid v \leq e_1\}}{\Gamma;\gamma \vdash e_1 - e_2 : \{v : \mathsf{UInt} \mid v = e_1 - e_2\}}\ \text{TE-Minus}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{UInt} \qquad \Gamma;\gamma \vdash e_2 : \{v : \mathsf{UInt} \mid v > 0\}}{\Gamma;\gamma \vdash e_1/e_2 : \{v : \mathsf{UInt} \mid e_2 * v \leq e_1 \land e_1 < (v+1) * e_2\}}\ \text{TE-Div}$$

$$\frac{\begin{array}{c}\Gamma;\gamma \vdash e_1 : \mathsf{Map}(T) \qquad \Gamma;\gamma \vdash e_2 : \mathsf{UInt} \\ \phi_a = \begin{cases} v \leq \mathsf{Sum}(e_1) & \text{if } T = \mathsf{UInt} \\ \mathsf{true} & \text{otherwise}\end{cases}\end{array}}{\Gamma;\gamma \vdash e_1[e_2] : \{v : T \mid v = e_1[e_2] \land \phi_a\}}\ \text{TE-MapInd}$$

$$\frac{\begin{array}{c}\Gamma;\gamma \vdash e_1 : \mathsf{Map}(T) \qquad \Gamma;\gamma \vdash e_2 : \mathsf{UInt} \qquad \Gamma;\gamma \vdash e_3 : T \\ \phi_a = \begin{cases} \mathsf{Sum}(v) = \mathsf{Sum}(e_1) - e_1[e_2] + e_3 & \text{if } T = \mathsf{UInt} \\ \mathsf{true} & \text{otherwise}\end{cases}\end{array}}{\Gamma;\gamma \vdash e_1\,[e_2 \triangleleft e_3] : \{v : \mathsf{Map}(T) \mid v = e_1\,[e_2 \triangleleft e_3] \land \phi_a\}}\ \text{TE-MapUpd}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{Struct}\,S \qquad \Delta(S, x) = T}{\Gamma;\gamma \vdash e_1[.x] : \{v : T \mid v = e_1[.x]\}}\ \text{TE-SctInd}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{Struct}\,S \qquad \Gamma;\gamma \vdash e_2 : T \qquad \Delta(S, x) = T}{\Gamma;\gamma \vdash e_1\,[.x \triangleleft e_2] : \{v : T \mid v = e_1\,[.x \triangleleft e_2]\}}\ \text{TE-SctUpd}$$

Fig. 8. Main refinement typing rules for expressions

Conditionals. In the conditional rule TS-If, we separately type check the true and false branches, adding the appropriate predicate (i.e., e or $\neg e$) to the statement guard for each branch. The results of the two branches are combined using the join typing judgment T-Join (also shown in Fig. 9). In particular, the join rule checks the type annotations for variables that are introduced via Φ nodes in SSA form and requires that the type of the x_i variants in each branch is a subtype of the declared type τ_i of x_i. Note also that the join rule requires agreement between the lock states in the two branches. This means that either (1) both branches commit their changes to state variables, or (2) both have uncommitted changes.

$$\frac{\begin{array}{c}\Gamma_1; L_1, \gamma_1 \vdash s_1 \dashv \Gamma_2; L_2, \gamma_2 \\ \Gamma_2; L_2, \gamma_2 \vdash s_2 \dashv \Gamma_3; L_3, \gamma_3\end{array}}{\Gamma_1; L_1, \gamma_1 \vdash s_1; s_2 \dashv \Gamma_3; L_3, \gamma_3}\ \text{TS-SEQ} \qquad \frac{\Gamma; \gamma \vdash e : \tau \qquad x \notin \mathsf{dom}(\Gamma)}{\Gamma; L, \gamma \vdash \mathsf{let}\, x : \tau = e \dashv x : \tau, \Gamma; L, \gamma}\ \text{TS-LET}$$

$$\frac{\begin{array}{c}\Gamma_1; \gamma \vdash e : \mathsf{Bool} \qquad \Gamma_1; L, e, \gamma \vdash s_1 \dashv \Gamma_{11}; L', \gamma_1 \\ \Gamma_1; L, \neg e, \gamma \vdash s_2 \dashv \Gamma_{12}; L', \gamma_2 \qquad \Gamma_1; \gamma; (\Gamma_{11}, \gamma_1); (\Gamma_{12}, \gamma_2) \vdash j \dashv \Gamma_2\end{array}}{\Gamma_1; L, \gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2\, \mathsf{join}\, j \dashv \Gamma_2; L', \gamma}\ \text{TS-IF}$$

$$\frac{\Gamma; \gamma \vdash e : \{\nu : \mathsf{Bool} \mid \nu = \mathsf{true}\}}{\Gamma; L, \gamma \vdash \mathsf{assert}\, e \dashv \Gamma; L, \gamma}\ \text{TS-ASSERT} \qquad \frac{\Gamma; \gamma \vdash e : \mathsf{Bool}}{\Gamma; L, \gamma \vdash \mathsf{assume}\, e \dashv \Gamma; L, e, \gamma}\ \text{TS-ASSUME}$$

$$\frac{\begin{array}{c}\text{for all } i = 1..n, \qquad x_i \notin \mathsf{dom}(\Gamma_1) \qquad \Delta(x'_i) = \tau_i \\ \tau'_i = \tau_i[x'_1 \mapsto x_1, \ldots, x'_i \mapsto \nu, \ldots, x'_n \mapsto x_n] \\ \Gamma_2 = x_1 : \tau'_1, \ldots, x_n : \tau'_n, \Gamma_1\end{array}}{\Gamma_1; \mathsf{Unlocked}, \gamma \vdash \mathsf{fetch}\, \overrightarrow{x'_i\, \mathsf{as}\, x_i} \dashv \Gamma_2; \mathsf{Locked}, \gamma}\ \text{TS-FETCH}$$

$$\frac{\begin{array}{c}\text{for all } i = 1..n, \qquad \Gamma; \gamma \vdash e_i : \tau_i \qquad \Delta(x_i) = \tau_i \\ \Gamma; \gamma \vDash \tau_i <: \tau'_i[x_1 \mapsto e_1, \ldots, x_i \mapsto \nu, \ldots, x_n \mapsto e_n]\end{array}}{\Gamma; \mathsf{Locked}, \gamma \vdash \mathsf{commit}\, e_1\, \mathsf{to}\, x_1, \ldots, e_n\, \mathsf{to}\, x_n \dashv \Gamma; \mathsf{Unlocked}, \gamma}\ \text{TS-COMMIT}$$

$$\frac{\begin{array}{c}\Delta(f) = ((x'_1 : \tau'_1, \ldots, x'_n : \tau'_n), \tau_r) \qquad \Gamma_2 = x : \tau_r[x'_1 \mapsto e_1, \ldots, x'_n \mapsto e_n], \Gamma_1 \\ \text{for all } i = 1..n, \qquad \Gamma_1; \gamma \vdash e_i : \tau_i \\ \Gamma_1; \gamma \vDash \tau_i <: \tau'_i[x'_1 \mapsto e_1, \ldots, x'_i \mapsto \nu, \ldots, x'_n \mapsto e_n]\end{array}}{\Gamma_1; \mathsf{Unlocked}, \gamma \vdash \mathsf{call}\, x : \tau_r = f(e_1, \ldots, e_n) \dashv \Gamma_2; \mathsf{Unlocked}, \gamma}\ \text{TS-CALL}$$

$$\frac{\begin{array}{c}\text{for all } x_i, \\ x_i \notin \mathsf{dom}(\Gamma) \qquad \Gamma_1; \gamma_1 \vdash x_{i1} : \{\nu : T_{i1} \mid \phi_{i1}\} \qquad \Gamma_2; \gamma_2 \vdash x_{i2} : \{\nu : T_{i2} \mid \phi_{i2}\} \\ \Gamma_1; \gamma_1 \vDash \{\nu : T_{i1} \mid \nu = x_{i1}\} <: \tau_i[(\overrightarrow{x_i \mapsto x_{i1}})] \\ \Gamma_2; \gamma_2 \vDash \{\nu : T_{i2} \mid \nu = x_{i2}\} <: \tau_i[(\overrightarrow{x_i \mapsto x_{i2}})] \qquad \Gamma' = \overrightarrow{x_i : \tau_i}, \Gamma\end{array}}{\Gamma; \gamma; (\Gamma_1; L, \gamma_1); (\Gamma_2; L, \gamma_2) \vdash \overrightarrow{x_i : \tau_i = \Phi(x_{i1}, x_{i2})} \dashv \Gamma'}\ \text{T-JOIN}$$

$$\frac{\begin{array}{c}\overrightarrow{x_i : \tau_i}; \mathsf{Unlocked} \vdash s \dashv \Gamma; \mathsf{Unlocked}, \gamma \\ \gamma = \mathsf{Unlocked}, \gamma' \qquad \Gamma; \gamma \vdash e : \tau'_r \qquad \Gamma; \gamma \vDash \tau'_r <: \tau_r\end{array}}{\vdash \mathbf{fun}\, f(\overrightarrow{x_i : \tau_i}) : \tau_r = s; e}\ \text{T-FUNDECL}$$

Fig. 9. Main typing rules for statements and function definitions

Fetch and Commit. As mentioned earlier, MINISOL contains fetch and commit statements, with the intention of allowing temporary violations of contract invariants. The TS-FETCH and TS-COMMIT rules provide type checking rules for these two statements. Specifically, the TS-FETCH rule says that, when no "lock" is held on the state variables (indicated by a lock status Unlocked), a fetch statement can be used to "acquire" a lock and copy the current values of the state variables into a set of freshly declared variables. The contract invariant will be assumed to hold on these variables.

Next, according to the TS-COMMIT rule, a commit statement can be used to "release" the lock and write the provided expressions back to the state variables. The contract invariant will be checked to hold on the provided expressions. Thus, our fetch and commit constructs allow temporary violations of the contract invariant *in between* these fetch and commit statements.

Function Calls. The TS-Call rule in Fig. 9 is used to type check function calls. In particular, since the contract invariant should not be violated between different transactions, our type system enforces that all state variables have been committed by requiring that the current state is Unlocked. Since the type checking of function arguments and return value is standard, the rest of the rule is fairly self-explanatory.

Function Definitions. Finally, we consider the type checking rule for function definitions (rule T-FunDecl rule in Fig. 9). According to this rule, a function declaration is well-typed if its body s is well-typed (with the environment initially set to the arguments and lock status being Unlocked), and the returned expression has type τ_r after executing s.

4.3 Generalizing Sum Axioms to Deeply Nested Mappings

The TE-MapInd and TE-MapUpd rules that we presented in Fig. 8 are sufficient for handling maps over integers; however, they are completely imprecise for more complex data structures, such as nested mappings or mappings of structs. Since Solidity programs often employ deeply nested mappings involving structs, it is crucial to have precise typing rules for more complex data structures. Therefore, in this section, we show how to generalize our precise typing rules for mappings over integers to arbitrarily nested mappings.

To gain some intuition about how to generalize these typing rules, consider the `usrs` mapping in Fig. 3. The update to `usrs` on line 14 is expressed in our IR as a sequence of map lookup operations followed by a sequence of updates:

```
//  usrs0 has type mapping(address => mapping(uint => User))
let m0 = usrs0[msg.sender];          // lookup
let u0 = m0[accno];                  // lookup
let b0 = u0[.bal];                   // access bal field
let u1 = u0[.bal ◂ b0 + amt];        // update bal field of struct
let m1 = m0[accno ◂ u1];             // update usrs[msg.sender] with updated struct
let usrs1 = usrs0[msg.sender ◂ m1]; // update usrs with updated nested mapping
```

Before we consider the revised typing rules, let us first consider the constraints we would *want* to generate for this example:

- For the first line, `usrs0` is a nested mapping containing a struct called `User` which has an integer field called `bal`, and `m0` is a mapping over `Users`. Based on the semantics of our refinement terms, the assignment on the first line imposes the following constraint on `m0`:

$$\mathsf{Sum}(\mathsf{Fld}_{bal}(m0)) \leq \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(usrs0)))$$

- Using similar reasoning, we can infer the following constraint on `u0`, which is of type `User`:

$$u0[.bal] \leq \mathsf{Sum}(\mathsf{Fld}_{bal}(m0))$$

- When we update `m1` on the fourth line, this again imposes a constraint involving Sum:

$$\mathsf{Sum}(\mathsf{Fld}_{bal}(m1)) = \mathsf{Sum}(\mathsf{Fld}_{bal}(m0)) - m0[accno][.bal] + u1[.bal]$$

- Finally, for the update on the last line, we can infer:

$$\begin{aligned}&\mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(usrs1))) = \\ &\qquad \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(m0))) - \mathsf{Sum}(\mathsf{Fld}_{bal}(m0)) + \mathsf{Sum}(\mathsf{Fld}_{bal}(m1))\end{aligned}$$

$$\frac{T_h \Vdash H : \mathsf{Map}(\mathsf{UInt}), w}{T_h \Vdash \mathsf{Sum}(H) : \mathsf{UInt}, w}\ \text{TH-Sum} \qquad \frac{T_h \Vdash H : \mathsf{Map}(\mathsf{Map}(T)), w}{T_h \Vdash \mathsf{Flatten}(H) : \mathsf{Map}(T), w}\ \text{TH-Flatten}$$

$$\frac{T_h \Vdash H : \mathsf{Map}(\mathsf{Struct}\, S), w \qquad \Delta(S, x) = T}{T_h \Vdash \mathsf{Fld}_x(H) : \mathsf{Map}(T), wx}\ \text{TH-Fld}$$

$$\frac{T_h \Vdash H : \mathsf{Struct}\, S, w \qquad \Delta(S, x) = T}{T_h \Vdash H[.x] : T, wx}\ \text{TH-FldAcc} \qquad \frac{}{T_h \Vdash \Box : T_h, \epsilon}\ \text{TH-Hole}$$

Fig. 10. Inference rules for type-directed synthesis of refinement term templates

As we can see from this example, the *shape* of the constraints we infer for complex map reads and writes is very similar to what we had in the TE-MapInd and the TE-MapUpd rules from Fig. 8. In particular, reading from a map imposes constraints of the form:

$$H_2(m[k]) \leq H_1(m) \tag{4}$$

whereas updating a map imposes constraints of the form:

$$H_1(m\,[k \triangleleft v]) = H_1(m) - H_2(m[k]) + H_2(v) \tag{5}$$

Therefore, we can easily generalize the TE-MapInd and the TE-MapUpd rules to arbitrarily complex mappings as long as we have a way of figuring out how to instantiate H_1, H_2 in Equations 4 and 5.

4.3.1 Refinement Term Templates. Based on the above observation, given a source expression e of some base type T, we want a way to automatically generate relevant refinement terms of the form $H(e)$. To facilitate this, we first introduce the notion of *templatized refinement terms*:

Definition 1. **(Templatized refinement term)** A *refinement term template* H is a refinement term containing a unique hole, denoted $\Box$. Given such a template H, we write $H(e)$ to denote the refinement term obtained by filling the hole $\Box$ in H with e.

For instance, $H = \mathsf{Sum}(\mathsf{Fld}_{bal}(\Box))$ is a valid template, and $H(m0)$ yields $\mathsf{Sum}(\mathsf{Fld}_{bal}(m0))$.

Next, given a type T_h, we need a way to generate all templates that can be applied to expressions of type T_h. Towards this purpose, we introduce a *template synthesis judgment* of the following form:

$$T_h \Vdash H : T, w \quad \text{(Template synthesis judgment)}$$

The meaning of this judgment is that, given an expression e of type T_h, $H(e)$ is a well-typed term of type T. In this judgment, w is a so-called *access path* that keeps track of the field accesses we need to perform to get from e to $H(e)$. In particular, an access path is a string over the language of field names, defined according to the following grammar:

$$w \quad ::= \quad \epsilon \mid xw \quad \text{(Access path)}$$

where x is the name of a field. As we will see shortly, we need this concept of access path to ensure that we do not generate non-sensical constraints in our generalized type checking rules for map reads and writes.

Fig. 10 shows our rules for synthesizing refinement term templates. The first rule, TH-Sum, states that we can apply the Sum operator to an expression whose base type is Map(UInt). The second rule, TH-Flatten, allows applying a Flatten operation to any term of type Map(Map(T)).

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{Map}(T) \qquad \Gamma;\gamma \vdash e_2 : \mathsf{UInt} \qquad \phi_a = \bigwedge_i H_{i2}(v) \leq H_{i1}(e_1) \\ \text{for each } i, \qquad \mathsf{Map}(T) \Vdash H_{i1} : \mathsf{UInt}, w_i \qquad T \Vdash H_{i2} : \mathsf{UInt}, w_i}{\Gamma;\gamma \vdash e_1[e_2] : \{v : T \mid v = e_1[e_2] \land \phi_a\}} \text{ TE-MapInd}$$

$$\frac{\Gamma;\gamma \vdash e_1 : \mathsf{Map}(T) \qquad \Gamma;\gamma \vdash e_2 : \mathsf{UInt} \\ \Gamma;\gamma \vdash e_3 : T \qquad \phi_a = \bigwedge_i (H_{i1}(v) = H_{i1}(e_1) - H_{i2}(e_1[e_2]) + H_{i2}(e_3)) \\ \text{for each } i, \qquad \mathsf{Map}(T) \Vdash H_{i1} : \mathsf{UInt}, w_i \qquad T \Vdash H_{i2} : \mathsf{UInt}, w_i}{\Gamma;\gamma \vdash e_1\ [e_2 \triangleleft e_3] : \{v : \mathsf{Map}(T) \mid v = e_1\ [e_2 \triangleleft e_3] \land \phi_a\}} \text{ TE-MapUpd}$$

Fig. 11. Updated typing rules for mappings, where refinement term templates are used to generate sum properties over nested data structures.

The third rule, TH-Fld, states that we can apply a projection operator to an expression that is a mapping of structs. In other words, if H is a mapping over structs S and S has a field called x, then we can generate the template $\mathsf{Fld}_x(H)$. Since this involves a field access, note that the TH-Fld rule adds x to the access path. The next rule, Th-FldAcc, is very similar but applies to structs instead of mappings of structs. In particular, if H has struct type S with field x, it is valid to access the x field of H. As in the previous case, this rule appends x to the resulting access path. The final rule, TH-Hole is a base case and states that the hole is constrained to have the specified type T_h.

4.3.2 Generalized Typing Rules for Map Reads and Writes. Equipped with the template synthesis rules, we are now ready to present the generalized and precise versions of TH-MapInd and TH-MapUpd rules for arbitrarily complex mappings.

The new TE-MapInd rule shown in Fig. 11 generates precise constraints on $e_1[e_2]$ where e_1 is has type $\mathsf{Map}(T)$. As illustrated earlier through the example, this rule generates constraints of the form $H_2(v) \leq H_1(e_1)$, where H_1, H_2 are templates that can be applied to terms of type $\mathsf{Map}(T)$ and T respectively. Since a given struct can have multiple integer fields, this rule generates a conjunction of such constraints, one for each integer field. Note that, for each constraint of the form $H_{i2}(v) \leq H_{i1}(e_1)$ in this rule, we enforce that the corresponding access paths w_i match, as, otherwise, the generated constraints would not make sense.

Next, we consider the new TE-MapUpd rule in Fig. 11 for updating maps. Specifically, given an expression $e_1[e_1 \triangleleft e_3]$ where e_1 is of type $\mathsf{Map}(T)$, this rule generates a conjunction of constraints of the form:

$$H_{i1}(v) = H_{i2}(e_1) - H_{i2}(e_1[e_2]) + H_{i2}(e_3),$$

one for each integer field accessible from T. As in the previous case, we use the notion of access paths to ensure that the hole templates H_{i1} and H_{i2} correspond to the same field sequence.

Example 3. Consider again the update to usrs on line 14 of Fig 3. We demonstrate how the TE-MapInd rule generates typing constraints. First, since usrs has type $\mathsf{Map}(\mathsf{Map}(\mathit{User}))$, we have $T = \mathsf{Map}(\mathit{User})$. From the synthesis rules, one such instantiation for H_{i1} is $\mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(\square)))$, as shown by the following derivation tree:

$$\textsc{TH-Sum}\ \dfrac{\textsc{TH-Fld}\ \dfrac{\textsc{TH-Flatten}\ \dfrac{\textsc{TH-Hole}\ \dfrac{}{\mathsf{Map}(\mathsf{Map}(User)) \Vdash \square : \mathsf{Map}(\mathsf{Map}(User)), \epsilon}}{\mathsf{Map}(\mathsf{Map}(User)) \Vdash \mathsf{Flatten}(\square) : \mathsf{Map}(User), bal} \qquad \Delta(User, bal) = \mathsf{UInt}}{\mathsf{Map}(\mathsf{Map}(User)) \Vdash \mathsf{Fld}_{bal}(\mathsf{Flatten}(\square)) : \mathsf{Map}(\mathsf{UInt}), bal}}{\mathsf{Map}(\mathsf{Map}(User)) \Vdash \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(\square))) : \mathsf{UInt}, bal}$$

Next, we find an instantiation for H_{i2} that accesses the same fields $w = bal$. It can be proven that

$$\mathsf{Map}(\mathsf{Struct}) \Vdash \mathsf{Sum}(\mathsf{Fld}_{bal}(\square)) : \mathsf{UInt}, bal$$

which yields $H_{i2} = \mathsf{Sum}(\mathsf{Fld}_{bal}(\square))$. Then, using the TE-MapInd rule, we obtain the constraint

$$\mathsf{Sum}(\mathsf{Fld}_{bal}(v)) \leq \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(usrs))) \tag{6}$$

Note that this is only *one* of the predicates that we can derive. For example, if the `User` had another integer field called `frozen`, we would also generate the following predicate:

$$\mathsf{Sum}(\mathsf{Fld}_{frozen}(v)) \leq \mathsf{Sum}(\mathsf{Fld}_{frozen}(\mathsf{Flatten}(usrs))) \tag{7}$$

Taking the conjunction of Eqs. (6) and (7), we obtain the following type for `usrs[msg.sender]`:

$$\begin{aligned} usrs[msg.sender] : \{v \mid\, & v = usrs[msg.sender] \\ & \wedge \mathsf{Sum}(\mathsf{Fld}_{bal}(v)) \leq \mathsf{Sum}(\mathsf{Fld}_{bal}(\mathsf{Flatten}(usrs))) \\ & \wedge \mathsf{Sum}(\mathsf{Fld}_{frozen}(v)) \leq \mathsf{Sum}(\mathsf{Fld}_{frozen}(\mathsf{Flatten}(usrs)))\} \end{aligned}$$

4.4 Soundness

We characterize the soundness of our type system in the typical way through progress and preservation theorems for both expressions and statements. We briefly state the key propositions and refer the reader to the extended version of the paper [Tan et al. 2021] for details.

The execution of a MiniSol program can be modeled as evaluation of a closed statement. Motivated by this observation, expressions are evaluated under the empty environment and empty guard predicates.

Proposition 2 (Progress for Expressions). *If $\epsilon; \epsilon \vdash e : \tau$, then e is a value or there exists an e' such that $e \longrightarrow e'$.*

Here, $e \longrightarrow e'$ is the standard expression evaluation relation, and progress can be proven by induction in the standard way. On the other hand, proving preservation is more interesting, mainly due to the rules from Fig. 11 for complex data structures. To prove preservation, we first need to prove a lemma that that relates each sum function generated in the refinement of a mapping to that of the corresponding sum function generated for each entry (value) in the mapping. With such a lemma, we can prove the following preservation theorem in a fairly standard way:

Proposition 3 (Preservation for Expressions). *If $\epsilon; \epsilon \vdash e : \tau$ and $e \longrightarrow e'$, then $\epsilon; \epsilon \vdash e' : \tau$.*

Next, to formulate the progress and preservation theorems for statements, we first introduce the following notation:

$$\begin{array}{lr} \sigma & \text{(State variable store)} \\ (s, \sigma) \longrightarrow (s', \sigma', \sigma_{sub}) & \text{(Statement evaluation relation)} \\ \vDash \sigma & \text{(Store typing)} \end{array}$$

The state variable store σ contains the value bindings for the state variables. The statement evaluation relation $(s, \sigma) \longrightarrow (s', \sigma', \sigma_{sub})$ indicates that, starting with state variable bindings σ, the statement s will step to a statement s' with updated bindings σ' and a set of local variable bindings

σ_{sub} to apply to the statements following s'. The store typing judgment $\vDash \sigma$ asserts that the bindings in σ are well-typed.

With this notation in place, we formulate the progress and preservation theorems for statements as Propositions 4 and 5. Note that assume false in Proposition 4 corresponds to a failed runtime check.

Proposition 4 (Progress for Statements). *If* $\epsilon; L \vdash s \dashv \Gamma; L', \gamma$ *and* $\vDash \sigma$, *then* $s = \text{skip}$, *or* $s = \text{assume false}$, *or there exist* $s', \sigma', \sigma_{sub}$ *such that* $(s, \sigma) \longrightarrow (s', \sigma', \sigma_{sub})$.

Proposition 5 (Preservation for Statements). *If all of the following hold:*

(a) $\epsilon; L_0 \vdash s \dashv \Gamma; L', \gamma$
(b) $\vDash \sigma$
(c) $(s, \sigma) \longrightarrow (s', \sigma', \sigma_{sub})$

then there exist L_0', Γ', γ' *such that*

(i) $\epsilon; L_0' \vdash s' \dashv \Gamma'; L', \gamma'$
(ii) $\vDash \sigma'$

While progress for statements can be proven in the standard way, preservation is trickier to prove since the resulting statement s' may be typed under different environment and guard predicates than s. To this end, we instead prove a strengthened version of preservation that implies Proposition 5; we refer the reader to the extended version of the paper [Tan et al. 2021] for the details.

5 TYPE INFERENCE

In this section, we describe our algorithm for automatically inferring refinement type annotations. Since writing refinement type annotations for every variable can be cumbersome, Solid infers types for both local and global (state) variables.

As mentioned earlier, the key idea underlying our type inference algorithm is to reduce the type inference problem to Constrained Horn Clause (CHC) solving [Bjørner et al. 2015]. However, since not all arithmetic operations in the program are guaranteed to be overflow-safe, the CHC constraints generated by the type checking algorithm will, in general, not be satisfiable. Thus, our high-level idea is to treat the overflow safety constraints as *soft constraints* and find type annotations that try to satisfy as many soft clauses as possible.

Before we explain our algorithm, we first clarify some assumptions that we make about the type checking phase. First, we assume that, during constraint generation, the refinement type of every variable v without a type annotation is represented with a fresh uninterpreted predicate symbol p_v. Second, we assume that the generated constraints are marked as either being hard or soft. In particular, the overflow safety constraints generated using the TE-Plus, TE-Minus etc. rules are considered to be soft constraints, while all other constraints are marked as hard. Third, we assume that the constraint generation phase imposes a partial order on the soft constraints. In particular, if c, c' are soft constraints generated when analyzing expressions e, e' and e must be evaluated before e', then we have $c \prec c'$. As we will see shortly, this partial order allows us to assume, when trying to satisfy some soft constraint c, that overflow safety checks that happened before c did not fail.

Our type inference procedure InferTypes is shown in Figure 12 and takes three inputs, namely a set of soft and hard constraints, C_s and C_h respectively, and a partial order $\preceq$ on soft constraints as described above. The output of the algorithm is a mapping Σ from each uninterpreted predicate symbol to a refinement type annotation.

Initially, the algorithm starts by initializing every uninterpreted predicate symbol p_v to true (line 5) and then enters a loop where the interpretation for each p_v is gradually strengthened. In

```
1:  procedure InferTypes($C_s$, $C_h$, $\preceq$)
2:      Input: Soft clauses $C_s$, hard clauses $C_h$, partial order $\preceq$ on $C_s$
3:      Output: Mapping $\Sigma$ from uninterpreted predicates $C_s \cup C_h$ to refinement type annotations
4:      $\mathcal{P} \leftarrow \mathsf{Preds}(C_s \cup C_h)$
5:      $\Sigma \leftarrow \{p \mapsto \top \mid p \in \mathcal{P}\}$
6:      for $c \in C_s$ do
7:          $\Phi_A \leftarrow \bigwedge_{c_i \prec c} c_i$; $\Phi_H \leftarrow \bigwedge_{c_j \in C_h} c_j$
8:          $(r, \sigma) \leftarrow \mathsf{SolveCHC}((\Phi_A \rightarrow c) \wedge \Phi_H)$
9:          if $r = \mathsf{UNSAT}$ then
10:             continue
11:         for $p \in \mathsf{Dom}(\sigma)$ do
12:             $\phi \leftarrow \Sigma(p) \wedge \sigma(p)$
13:             $\Sigma \leftarrow \Sigma[p \mapsto \phi]$
14:     return $\Sigma$
```

Fig. 12. Type inference procedure

particular, in every loop iteration, we pick one of the soft constraints c and try to satisfy c along with all the hard constraints C_H. In doing so, we can assume all the overflow safety checks that happened to prior to c; thus, at line 8, we feed the following constraint to an off-the-shelf CHC solver:

$$\Big(\Big(\bigwedge_{c_i \prec c} c_i\Big) \rightarrow c\Big) \wedge \bigwedge_{c_j \in C_h} c_j$$

In other words, we try to satisfy this particular soft constraint together with all the hard constraints, under the assumption that previous overflow checks did not fail. This assumption is safe since Solid inserts run-time overflow checks for any operation that cannot be statically verified. If this constraint is unsatisfiable, we move on and try to satisfy the next soft constraint. On the other hand, if it is satisfiable, we strengthen the type annotation for the unknown predicates by conjoining it with the assignment produced by the CHC solver (line 12).

6 IMPLEMENTATION

We implemented our type checking and type inference algorithms in a prototype called Solid. Given a Solidity source file, Solid will run the aforementioned algorithms and, for each arithmetic operation, output whether its corresponding soft constraint is satisfied or violated. Our tool is written in Haskell, with about 3000 lines of code for the frontend (e.g., Solidity preprocessing, lowering, SSA transformation) and 4000 lines of code for the backend (e.g., constraint generation, SMT embedding, solving). Solid uses the Z3 SMT solver [de Moura and Bjørner 2008] and the Spacer CHC solver [Komuravelli et al. 2014] in its backend, and it leverages the Slither [Feist et al. 2019] analyzer as part of its front-end. In what follows, we discuss various aspects of Solidity and how we handle them in the MiniSol IR.

Preprocessing Solidity code. To facilitate verification, we preprocess Solidity smart contracts before we translate them to the MiniSol IR. In particular, we partially evaluate constant arithmetic expressions such as `6 * 10**26`, which frequently appear in constructors and initialization expressions. We also inline internal function calls.

Function call handling. Our implementation automatically inserts fetch and commit statements between function call boundaries, such as at the beginning of a function, before and after function calls or calls to external contracts, and before returning from a function. This includes most function calls, including monetary transfers like `msg.sender.send(..)` and calls to "non-pure" functions in the same file. In Solidity, functions may be marked "pure", meaning they do not read or write to state variables. We modified the TS-Call rule to not require the state variables to be unlocked when making calls to a pure function.

Loops. Our implementation supports while-loops and for-loops using a variation of the TS-If rule. Performing effective type inference in the presence of loops (particularly, doubly-nested loops) is challenging, as existing CHC solvers have a hard time solving the resulting constraints. Inferring invariants for complex loops could be an interesting direction for future work, however, in practice, such loops are rare [Mariano et al. 2020] and we found they are mostly irrelevant to proving overflow safety. Our prototype implementation does not support complex control flow inside loops such as break or continue statements.

References and aliasing. Since everything is passed by value in MiniSol, the language shown in Figure 4 does not have aliasing. Thus, as standard [Barnett et al. 2005; Flanagan et al. 2002], our translation from Solidity to MiniSol introduces additional mappings to account for possible aliasing. At a high-level, the idea is that, for any variables $X = \{x_1, \ldots, x_n\}$ that may alias each other, we introduce a mapping M_X and model stores (resp. loads) to any x_i as writing to (resp. reading from) $M_X[x_i]$.

Unsupported features. Our implementation does not support some Solidity features such as inline assembly, states of external contracts, bitwise operations, and exponentiation. When translating to the MiniSol IR, we model these statements using "havoc" expressions as is standard [Barnett et al. 2005].

7 EVALUATION

In this section, we describe a series of experiments that are designed to answer the following research questions:

- **RQ1:** Can Solid successfully remove redundant overflow checks without manual annotations?
- **RQ2:** What is Solid's false positive rate?
- **RQ3:** How long does Solid take to type check real-world smart contracts?
- **RQ4:** How does Solid compare against existing state-of-the art tools?
- **RQ5:** How important are type inference and the proposed refinement typing rules for more complex data structures?

Baseline. To answer our fourth research question, we compare Solid against Verismart, which is a state-of-the-art Solidity verifier that focuses on proving arithmetic safety properties [So et al. 2020]. We choose Verismart as our baseline because it has been shown to outperform other smart contract verifiers for checking arithmetic safety [So et al. 2020]. Verismart infers so-called *transaction invariants* through a domain-specific instantiation of the *counterexample-guided inductive synthesis (CEGIS)* framework [Solar-Lezama 2009] and uses the inferred invariants to discharge potential overflow errors. To perform this evaluation, we built the latest version of Verismart from source (version from May 31, 2020) and ran both tools on all the benchmarks, using versions 0.4.26 and 0.5.17 of the Solidity compiler.

Table 1. Statistics about our benchmarks from ETHERSCAN

Description	Stats
% contracts containing mappings of structs	26.7%
% contracts containing nested mappings	81.3%
Average lines of code	389
Average number of methods	29.6

Setup. We performed all experiments on a computer with an AMD Ryzen 5900X CPU and 32GB of RAM. We used version 4.8.9 of the Z3 SMT solver, which also has the built-in Spacer CHC solver [Komuravelli et al. 2014]. In our experiments, we set a time limit of 10 seconds per Z3 query.

Settings. Our implementation can be used in two modes that we refer to as AUTO-SOLID and SEMI-SOLID. In particular, AUTO-SOLID is fully automated and uses our type inference procedure to infer type annotations for state variables (i.e., contract invariants). On the other hand, SEMI-SOLID requires the programmer to provide contract invariants but automatically infers types for local variables. To use SOLID in this semi-automated mode, we manually wrote refinement type annotations for state variables by inspecting the source code of the contract.

7.1 Benchmarks

We evaluate SOLID on two different sets of benchmark suites that we refer to as "VERISMART benchmarks" and "ETHERSCAN benchmarks". As its name indicates, the former one is taken from the VERISMART evaluation [So et al. 2020] and consists of 60 smart contracts that have at least one vulnerability reported in the CVE database. Since these benchmarks are from 2018 and do not reflect rapid changes in smart contract development, we also evaluate our approach on another benchmark suite collected from ETHERSCAN. The latter benchmark suite consists of another 60 contracts randomly sampled from ETHERSCAN in March 2021. Table 1 presents some relevant statistics about the contracts in this dataset. Since the goal is to discharge redundant SAFEMATH[2] calls, we preprocessed the benchmarks by replacing all SAFEMATH method calls with their equivalent unchecked operations. Thus, the reader should be advised that the "VERISMART benchmarks" are not exactly the same as the ones in [So et al. 2020].

Although our preprocessed benchmarks technically do not contain any SAFEMATH runtime checks, in the following discussion, we will use the terms "run-time checks", "redundant checks", and "necessary checks" to refer to arithmetic operations, provably overflow-safe operations, and operations whose overflow-safety cannot be proven, respectively.

7.2 Discharging Redundant SafeMath Calls

We performed a manual inspection of all 120 benchmarks to determine how many of the SAFEMATH run-time checks are redundant. In total, among the 1309 run-time checks, we determined 853 of them to be redundant (390 in VERISMART and 463 in ETHERSCAN). In this experiment, we evaluate what percentage of these redundant checks VERISMART, AUTO-SOLID, and SEMI-SOLID are able to discharge.

The results from this evaluation are shown in Table 2. The key-take away is that AUTO-SOLID is able to discharge more run-time checks in both benchmark suites. In particular, for the VERISMART benchmarks, AUTO-SOLID can discharge approximately 12% more overflow checks, and for the

[2]Recall that SAFEMATH is a library that inserts runtime checks before each arithmetic operation.

Table 2. Number and percentage of redundant overflow checks that can be eliminated by each tool. The Ops column displays the total number of SafeMath checks. Under each tool, the Safe column lists the number of arithmetic operations that can be proven safe by that tool, and the % column shows the percentage of redundant overflow checks that can be discharged.

Dataset	Ops	#redundant	VERISMART		AUTO-SOLID		SEMI-SOLID	
			Safe	%	Safe	%	Safe	%
VERISMART	642	390	294	75.4%	340	87.2%	351	90.0%
ETHERSCAN	667	463	272	58.7%	396	85.5%	417	90.1%
Total	1309	853	566	66.4%	736	86.3%	768	90.0%

Table 3. Comparison of false positive rates

	VERISMART	AUTO-SOLID	SEMI-SOLID
Verismart Benchmarks			
# false positives	97	50	41
# true positives	251	252	252
False positive rate	27.9%	16.2%	14.0%
Etherscan Benchmarks			
# false positives	195	67	46
# true positives	200	204	204
False positive rate	49.4%	24.7%	18.4%
Overall			
# false positives	292	117	87
# true positives	451	456	456
False positive rate	39.3%	20.4%	16.0%

ETHERSCAN benchmarks, this difference increases to 27%. If we additionally leverage manually-written refinement type annotations, then SEMI-SOLID can discharge almost 90% of the redundant checks across both datasets.

Result for RQ1: SOLID can automatically prove that 86.3% of the redundant SAFEMATH calls are unnecessary. In contrast, VERISMART can only discharge 66.4%.

7.3 False Positive Evaluation

Next, we evaluate SOLID's false positive rate in both the fully-automated and semi-automated modes and compare it against VERISMART. In particular, Table 3 shows the number of true and false positives as well as the false positive rate for each tool for both benchmark suites. Across all benchmarks, VERISMART reports 292 false alarms, which corresponds to a false positive rate of 39.3%. On the other hand, AUTO-SOLID only reports 117 false alarms with a false positive rate of 20.4%. Finally, SEMI-SOLID has an ever lower false positive rate of 16.0%.

It is worth noting that the false positive rate for SEMI-SOLID can be further reduced by spending additional effort in strengthening our refinement type annotations. When performing this evaluation, we did not refine our initial type annotations based on feedback from the type checker.

Table 4. Comparison of average running times in seconds

Dataset	VERISMART	AUTO-SOLID	SEMI-SOLID
VERISMART	476	62	9
ETHERSCAN	425	24	10
Overall	451	41	10

Table 5. Experiments to evaluate type inference. This experiment is conducted on the ETHERSCAN benchmarks.

	SOLID-NoInfer	SOLID-NoSoft	SOLID
False positive rate	48.7%	69.4%	24.7%

Cause of false positives for AUTO-SOLID. As expected, the cause of most false positives for AUTO-SOLID is due to the limitations of the CHC solver. There are several cases where the CHC solver times-out or returns unknown, causing AUTO-SOLID to report false positives.

Qualitative comparison against VERISMART. We believe that SOLID outperforms VERISMART largely due to its ability to express relationships between integers and aggregate properties of data structures. In particular, while VERISMART can reason about summations over basic mappings, it cannot easily express aggregate properties of more complex data structures.

Result for RQ2 and RQ4. In its fully automated mode, SOLID has a false positive rate of 20.4%. In contrast, VERISMART has a significantly higher false positive rate of 39.3%.

7.4 Running Time

Next, we investigate the running time of SOLID and compare it against VERISMART. Table 4 gives statistics about the running time of each tool. As expected, SEMI-SOLID is faster than AUTO-SOLID, as it does not need to rely on the CHC solver to infer contract invariants. However, even AUTO-SOLID is significantly faster than VERISMART despite producing fewer false positives.

Result for RQ3 and RQ4: In its fully automated mode, SOLID takes an average of 41 seconds to analyze each benchmark, and is significantly faster compared to VERISMART despite generating fewer false positives.

7.5 Impact of Type Inference

In this section, we perform an experiment to assess the impact of the type inference procedure discussed in Section 5. Towards this goal, we consider the following two ablations of SOLID:

- SOLID-**NoInfer:** This is a variant of SOLID that does not perform global type inference to infer contract invariants. In particular, SOLID-NoInfer uses `true` as the type refinement of all state variables; however, it still performs local type inference.
- SOLID-**NoSoft:** This variant of SOLID differs from the type inference procedure in Fig. 12 in that it treats all overflow checks as hard constraints.

In this experiment, we compare the false positive rate of each ablated version against SOLID on the ETHERSCAN benchmarks. The results of this experiment are summarized in Table 5.

Table 6. Ablation study to evaluate rules from Section 4.3. This experiment is conducted on Etherscan benchmarks that contain non-trivial mappings.

	Solid-NoNested	Solid
False positive rate	39.0%	23.5%

Importance of contract invariant inference. As we can see by comparing Solid against Solid-NoInfer, global type inference is quite important. In particular, if we do not infer contract invariants, the percentage of false positives increases from 24.7% to 48.7%.

Importance of soft constraints. Next, we compare the false positive rate of Solid against that of Solid-NoSoft. Since Solid-NoSoft treats all overflow checks as hard constraints, type inference fails if *any* potential overflow in the contract cannot be discharged. Since most contracts contain at least one unsafe overflow, Solid-NoSoft fails for most benchmarks, meaning that all overflows are considered as potentially unsafe. Thus, the false positive rate of Solid-NoSoft jumps from 24.7% to 69.4%.

7.6 Impact of the Type Checking Rules from Section 4.3

In this section, we describe an ablation study to assess the impact of the more complex typing rules from Fig. 11 for non-trivial mappings. Towards this goal, we consider the following ablation:

- **Solid-NoNested:** This variant of Solid uses the simpler refinement type checking rules from Section 4.2 but it does not utilize the typing rules from Section 4.3 that pertain to complex data structures.

In this experiment, we compare Solid-NoNested against Solid on those contracts from Etherscan that contain complex data structures. As shown in Table 6, the more involved typing rules from Section 4.3 are quite important for successfully discharging overflows in contracts with non-trivial mappings. In particular, without our refinement templates from Section 4.3, the false positive rate increases from 23.5% to 39.0% on these benchmarks.

Result for RQ5: Our proposed typing rules from Section 4.3 for more complex data structures and the proposed type inference algorithm from Section 5 are both important for achieving good results.

8 LIMITATIONS

In this section, we discuss some of the limitations of both our type system as well as prototype implementation. First, while logical qualifiers in SolType express relationships between integers and aggregate properties of mappings, they do not allow quantified formulas. In principle, there may be situations that necessitate quantified refinements to discharge arithmetic safety; however, this is not very common. Second, Solid uses an off-the-shelf CHC solver to infer refinement type annotations. Since this problem is, in general, undecidable, the CHC solver may return unknown or fail to terminate in a reasonable time. In practice, we set a small time limit of 10 seconds per call to the CHC solver. Third, Solid is designed with checking arithmetic overflows in mind, so the proposed refinement typing rules may not be as effective for checking other types of properties.

9 RELATED WORK

Smart contract correctness has received significant attention from the programming languages, formal methods, and security communities in recent years. In this section, we discuss prior work on refinement type systems and smart contract security.

Refinement types. Our type system is largely inspired by and shares similarities with prior work on liquid types [Rondon et al. 2010, 2008; Vazou et al. 2014; Vekris et al. 2016]. For instance, our handling of temporary contract violations via fetch/commit statements is similar to a simplified version of the `fold` and `unfold` operators used in [Rondon et al. 2010]. However, a key novelty of our type system is its ability to express and reason about relationships between integer variables and aggregate properties of complex data structures (e.g., nested maps over struct) that are quite common in smart contracts. Also, similar to the liquid type work, our system also performs type inference; however, a key novelty in this regard is the use of soft constraints to handle programs where not all arithmetic operations can be proven safe.

Smart contract verification. Many popular security analysis tools for smart contracts are based on symbolic execution [King 1976]. Well-known tools include Oyente [Luu et al. 2016], Mythril [Mythril 2020] and Manticore [Mossberg et al. 2019], and they look for an execution path that violates a given property or assertion. In contrast to these tools, our proposed approach is based on a refinement type system and offers stronger guarantees compared to bug finding tools for smart contracts.

To bypass the scalability issues associated with symbolic execution, researchers have also investigated sound and scalable static analyzers [Grech et al. 2018; Grossman et al. 2018; Kalra et al. 2018; Tsankov et al. 2018]. Both Securify [Tsankov et al. 2018] and Madmax [Grech et al. 2018] are based on abstract interpretation [Cousot and Cousot 1977], which does not suffer from the path explosion problem. ZEUS [Kalra et al. 2018] translates Solidity code into the LLVM IR and uses an off-the-shelf verifier to check the specified policy [Rakamaric and Emmi 2014]. The ECF [Grossman et al. 2018] system is designed to specifically detect the DAO vulnerability. In contrast to these techniques, the focus of our work is on proving arithmetic safety, which is a prominent source of security vulnerabilities.

Similar to Solid, Verismart [So et al. 2020] is also (largely) tailored towards arithmetic properties of smart contracts. In particular, Verismart leverages a CEGIS-style algorithm for inferring contract invariants. As we show in our experimental evaluation, Solid outperforms Verismart in terms of false positive rate as well as running time.

Some systems [Grishchenko et al. 2018; Hirai 2017; Park et al. 2018; Permenev et al. 2020] for reasoning about smart contracts rely on formal verification. These systems typically prove security properties of smart contracts using existing interactive theorem provers [Team 2021], or leverage temporal verification for checking functional correctness [Permenev et al. 2020]. They typically offer strong guarantees that are crucial to smart contracts. However, unlike our system, all of them require significant manual effort to encode the security properties and the semantics of smart contracts. Furthermore, Solythesis [Li et al. 2020] inserts runtime checks into smart contract source code to revert transactions that violate contract invariants. We note that this approach is complementary to static verifiers such as Solid.

Verifying overflow/underflow safety. The problem of checking integer overflow/underflow in software systems is a well-studied problem in the verification community. In particular, Astree [Blanchet et al. 2003] and Sparrow [Oh et al. 2014] are both based on abstract interpretation [Cousot and Cousot 1977] and are tailored towards safety-critical code such as avionics software. As we argue throughout the paper, verifying arithmetic safety of smart contracts requires reasoning about aggregation properties over mappings; hence, standard numeric abstract domains such as the ones used in Astree would not be sufficient for discharging over- and under-flows in Solidity programs.

CHC Solving. We are not the first to use CHC solvers in the context of refinement type checking. For example, Liquid Haskell [Vazou et al. 2014] uses a predicate-abstraction based horn clause solver for refinement type inference. However, in contrast to our work, Liquid Haskell does not tackle the problem of MaxCHC type inference, which is critical for discharging the maximum number of runtime overflow checks in smart contracts.

To the best of our knowledge, the only prior work that can potentially address our MaxCHC problem is in the context of network repair. In particular, [Hojjat et al. 2016] formulate the problem of repairing buggy SDN configurations as an optimization problem over constrained Horn clauses. They generalize their proposed method to Horn clause optimization over different types of lattices, and our type inference algorithm may be viewed as an instantiation of their more general framework. However, in contrast to their approach, our algorithm leverages domain-specific observations for efficient (but potentially suboptimal) type inference in the context of smart contracts. In particular, we use the fact that all soft constraints correspond to overflow checks that are guaranteed to be satisfied (either because they are safe or we insert a runtime check) to check each soft constraint independently.

10 CONCLUSION

We have presented SolType, a refinement type system for Solidity that can be used to prove the safety of arithmetic operations. Since integers in smart contracts often correspond to financial assets (e.g., tokens), ensuring the safety of arithmetic operation is particularly important in this context. One of the distinguishing features of our type system is its ability to express and reason about arithmetic relationships between integer variables and aggregations over complex data structures such as multi-layer mappings. We have implemented our proposed type system in a prototype called Solid, which also has type inference capabilities. We have evaluated Solid on 120 smart contracts from two datasets and demonstrated that it can fully automatically discharge 86.3% of redundant SafeMath calls in our benchmarks. Furthermore, Solid, even when used in a fully automated mode, significantly outperforms Verismart, a state-of-the-art Solidity verifier, in terms of both false positive rate and running time.

While the design of SolType was largely guided from the perspective of proving arithmetic safety, our proposed type system could also be used for proving other types of properties. In future work, we plan to explore the applicability of our refinement type system in other settings and extend it where necessary.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, as well as our shepherd James R. Wilcox, for their helpful comments and valuable suggestions.

This work was partially supported by NSF Grants 2027977, 1908494, 1811865, 1762299, and the Google Faculty Research Award.

REFERENCES

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10204)*, Matteo Maffei and Mark Ryan (Eds.). Springer, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO*

2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (Lecture Notes in Computer Science, Vol. 4111), Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.). Springer, 364–387.

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science, Vol. 9300)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.). Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 196–207. https://doi.org/10.1145/781131.781153

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, Los Angeles, CA, USA, 238–252. https://doi.org/10.1145/512950.512973

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Etherscan 2021. *The Ethereum Blockchain Explorer.* Etherscan. Retrieved June 25, 2021 from https://etherscan.io/

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019.* IEEE / ACM, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 234–245.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 116:1–116:27. https://doi.org/10.1145/3276486

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10804)*, Lujo Bauer and Ralf Küsters (Eds.). Springer, 243–269. https://doi.org/10.1007/978-3-319-89722-6_10

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10323)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer, Sliema, Malta, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33

Hossein Hojjat, Philipp Rümmer, Jedidiah McClurg, Pavol Cerný, and Nate Foster. 2016. Optimizing horn solvers for network repair. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, Ruzica Piskac and Muralidhar Talupur (Eds.). IEEE, 73–80. https://doi.org/10.1109/FMCAD.2016.7886663

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing Smart Contract with Runtime Validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 438–453. https://doi.org/10.1145/3385412.3385982

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, 254–269. https://doi.org/10.1145/2976749.2978309

Adriana M. 2018. *Real Estate Business Integrates Smart Contracts*. Coindoo. https://coindoo.com/real-estate-business-integrates-smart-contracts/

Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. 2020. Demystifying Loops in Smart Contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 262–274. https://doi.org/10.1145/3324884.3416626

Mix. 2018. *Ethereum bug causes integer overflow in numerous ERC20 smart contracts (Update)*. https://thenextweb.com/news/ethereum-smart-contract-integer-overflow

Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

ConsenSys 2020. *Mythril*. ConsenSys. Retrieved November 2020 from https://github.com/ConsenSys/mythril

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 475–484. https://doi.org/10.1145/2594291.2594318

Santiago Palladino. 2017. *On the parity wallet multisig hack*. OpenZeppelin. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Rosu. 2018. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 912–915. https://doi.org/10.1145/3236024.3264591

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy* (San Francisco, CA, USA) *(SP '20)*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 106–113. https://doi.org/10.1007/978-3-319-08867-9_7

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-Level Liquid Types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1706299.1706316

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

David Siegel. 2016. *Understanding The DAO Attack*. CoinDesk. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/

Frederick Smith, David Walker, and Greg Morrisett. 2000. Alias Types. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 366–381. https://doi.org/10.1007/3-540-46425-5_24

S. So, M. Lee, J. Park, H. Lee, and H. Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1678–1694. https://doi.org/10.1109/SP40000.2020.00032

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Brian Straight. 2020. *Smart contracts may offer smart solutions for carriers, truck drivers*. FreightWaves. https://www.freightwaves.com/news/smart-contracts-may-offer-smart-solutions-for-carriers-truck-drivers

Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2021. SolType: extended version. arXiv:2110.00677 [cs.PL]

The Coq Development Team. 2021. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.4501022

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82.

https://doi.org/10.1145/3243734.3243780

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 310–325. https://doi.org/10.1145/2908080.2908110