Verifying smart contracts

SMT real life challenges and solutions

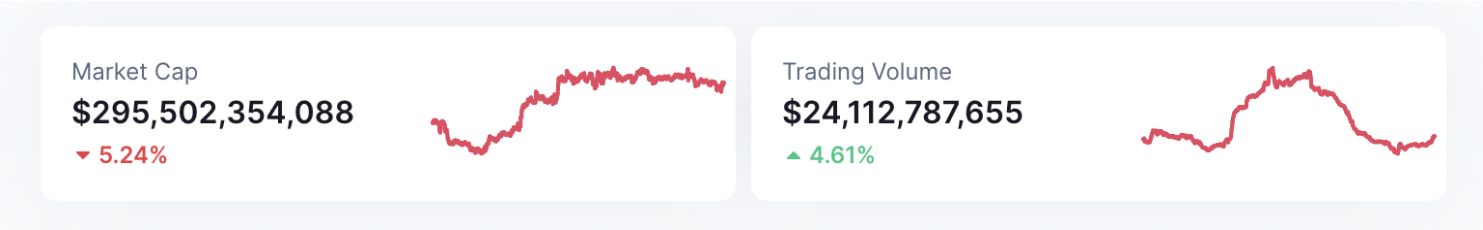
Yoav Rodeh

CERTORA

Smart Contracts

a.k.a EVM bytecode programs

- Short and simple.
  - Running code costs in *gas*.
- Handle heaps of money.



- Bugs cost **Billions** annually.
  - Already more than 2B in 2022.

They are perfect for formal verification!

At Certora

- Specification is in CVL.
- Combine with the compiled byte-code.
  - why?
  - ahh!
- Throw it at SMT-solvers:
  - z3, cvc5, yices.
  - open to suggestions.
- It works!

Certora has investors, paying customers, and is growing fast.

However...

Contracts are not **that** short and simple

- Bytecode is optimized to save gas, not to help out SMT-solvers.
- At Certora we work a lot to make solvers run fast.

In this talk we'll briefly see some interesting examples of our work.

EVM Storage Model

A 2^{256} array of 256-bit words.

```
Contract C {
    uint256 x;
    int16 y;
    int104 z;
    int[] a;
}
```

- `x` is in slot 0.
- `y` and `z` share slot 1.
- slot 2 contains the length of `a`, and the elements are placed continuously starting at ***keccak***(2).

Storage Overview

Not your usual out of the box storage model

- Different than memory.
- Is a 2^{256} array of 256-bit words.
  - Kept as a dictionary

The Location of elements is complex, but:

- The *Keccak256* hash function is used a lot.
- It is assumed there are no collisions.
- Arrays are continuous.

Modeling Keccak256

Way too complex for solvers

An uninterpreted one-to-one function:

$$\boldsymbol{h} : \boldsymbol{Universe} \rightarrow 2^{256}$$

But ***Universe*** is huge $\Longrightarrow$ there is no such ***h***!

Solution: Define an ”inverse” function ***g***, and for every ***keccack***(α) in the code require:

$$\boldsymbol{g}(\boldsymbol{h}(\alpha)) = \alpha$$

That was easy!

Arrays

Recall they are continuous

For arrays $\boldsymbol{a} \neq \boldsymbol{b}$, then forall $\boldsymbol{i}$ and $\boldsymbol{j}$:

$$\boldsymbol{keccak}(\boldsymbol{a}) + \boldsymbol{i} \neq \boldsymbol{keccak}(\boldsymbol{b}) + \boldsymbol{j}$$

Impossible! yet true...

Injectivity of $\boldsymbol{h}$ is not enough. We actually want:

Large Gaps Injectivity

Arithmetic Large Gaps

Why stop at 2^{256}

Array size is restricted by 2^{256}, so model $\boldsymbol{keccak}(\boldsymbol{a})$ by:

$$(2^{256} + 1) \cdot \boldsymbol{h}(\boldsymbol{a})$$

Where $\boldsymbol{h}$ is injective as above.

Why $+1$?

- Result can never be small.
  - EVM uses the low indices of storage for static fields.
- Nor equal to any large number in the code.
  - some hash values are hard coded.

Arithmetic Large Gaps

Works surprisingly well, but:

- Arithmetical overhead.
  - Hashes are frequently nested.
- Cannot use Bit-Vector theory.
  - Because all values must be less than 2^{256}.

Plain Injectivity

Pattern Based

- Examine all the expressions that are used to access storage.
- Rewrite each one as $\textbf{\textit{keccak}}(\textbf{\textit{a}}) + \textbf{\textit{expr}}$.
  - ***expr*** does not contains any ***keccak***s.
  - Almost always possible.
  - ***ite***'s make this more complex.
  - ***a*** itself could be such an expression.
- Require that every pair of such sums is different if either ***a*** is different or ***expr*** is different.

Plain Injectivity

Problems

We get lots of assertions:

- A quadratic number of assertions for the pairs.
  - Even more because of ite expressions
- Small constants and hard-coded hash results should be treated explicitly.

It is pretty brittle...

Datatype Theory

The perfect match!

Expressions that are a result of a hashing operation are of a special recursive datatype ***Skey***:

$$\begin{aligned} \textbf{\textit{Skey}} &\leftarrow \textbf{\textit{Root}} \mid (\textbf{\textit{Skey}}, \textbf{\textit{Offset}}) \\ \textbf{\textit{Root}} &\leftarrow \langle \text{256-bit non-negative num} \rangle \\ \textbf{\textit{Offset}} &\leftarrow \langle \text{256-bit non-negative num} \rangle \end{aligned}$$

So for example, for an array of arrays which is at root slot 100:

$$\textbf{\textit{a}}[5][6] \Longrightarrow ((100, 5), 6)$$

Corresponding to

$$\textbf{\textit{keccak}}(\textbf{\textit{keccak}}(100) + 5)) + 6$$

Datatype Theory

- Datatype theory promises that two such expression trees are equal iff they are exactly the same.
- No aritmetic, and no need for added axioms.
- perfect!

Surprisingly, this is usually harder on the solvers...

Why? perhaps the addition of another theory?

Packing & Unpacking

Storage Splitting

SMT doesn't like packing-unpacking

- Each storage slot has 256 bits.
  - Reads and Writes work with whole slots.
- Smaller elements are packed within a slot.

```
Struct S {
    uint8 x;
    uint16 y;
    uint104 z;
}
```

Write Logic

Writing ***value*** to ***y*** (Reading is similar):

```
old <- slot
remainder := old & 0xff...ff0000ff
slot <- remainder | (value << 8)
```

Treated naively, this is hard for solvers, and LIA and NIA need additional axiomatization.

Rewrite Packing/Unpacking Logic

Just throw it away

Ideal solution is to detect such cases and rewrite it all as if it's not packed.

- This does away with all bitwise operations.
- But Solidity compiler versions and optimizations result in many different code patterns for storage access.
  - e.g., A few fields can be written at once.
- So identifying bytecode patterns doesn't work perfectly.
- A general algorithm may even simplify hand coded packing and unpacking logic.

Split Detection Algorithm

Gather constraints on how variables (slots) can be split into bit ranges. e.g., ***a*** into $\boldsymbol{a}[0-10]$, $\boldsymbol{a}[11-30]$, $\boldsymbol{a}[31-255]$. For Example:

```
a := b | c
```

Means that all three variables should be split in the same way. Then we can rewrite:

```
a[0-10] := b[0-10] | c1[0-10],
a[15-20] := b[15-20] | c[15-20],
...
```

The bitwise-or disappears when one side is 0 (like the ***value*** and ***remainder*** above).

More Constraints

```
a := b + 1
```

Makes both ***b*** and ***a*** unsplittable. But if we know ***b***'s top bits are zeros, this constraint is relaxed.

```
a := b << 10
```

Means ***a***'s and ***b***'s split should be the same, yet shifted. If eventually split, then:

```
a[0-9] = 0,
a[10-30] = b[0-20],
...
```

And the shift operation disappears.

Fixed Point Algorithm

Constraints are encoded in a graph, and via a fixed point algorithm, the best possible split is found.

- Many missing details here.
  - The crucial treatment of upper bits zeros.
  - Handling constants.
  - Both of these need a preliminary over-approximation step for the bit values of variables.
- Best in the sense of not introducing any split that never needs to be ”glued” back.
  - This may actually be sub-optimal.
  - But (so far) always works if there are no ”rogue” storage accesses.

LIA Overapproximation

LIA vs NIA

- There is a dramatic difference in solving time between LIA formulas and NIA ones.
- Our generated SMT-formulas are mostly linear, with usually only a few non-linear operations.
- Their correctness often depends only on simple non-linear properties.

So we created a partial axiomatization of non-linear operations which is itself linear.

Axioms

Just a few examples

Denote by $\odot$ an uninterpreted version of multiplication.

$$a \odot 0 = 0$$
$$a \odot 1 = 1$$
$$a \odot b = b \odot a$$

This one is to model multiplication overflow checks:

$$\frac{(a \odot b) \ \% \ 2^{256}}{a} = b \iff |a \odot b| < 2^{256}$$

Division and modulo are also uninterpreted.

An Overapproximation

This axiomatization is made up only of statements which are true in NIA.

- If a formula is UnSAT, i.e., correct, it is truly correct.
  - The NIA formulation is exact (well, almost).
- If a formula is SAT, we try to use the counter example to direct the NIA search.
- Otherwise, we run a NIA solver.

Quantification

A big no no

We support it, but,

- Really hard on SMT solvers.
- Can't promise we don't have bugs..

So we don't quantify over these axioms, but rather instantiate them on every multiplication we have.

Including:

$$\boldsymbol{a} \odot \langle \text{constant} \rangle = \boldsymbol{a} \cdot \langle \text{constant} \rangle$$

We do a lot more...

On the edge of the fork

- CEGAR.
- Axiomatization of bitwise operations.
- Our own array theory implementation, because we need longstores and array initialization.
- Tons of static analysis to simplify stuff.
- Signed arithmetic in 2s complement.
  - I'm working on it these very days!

And we plan so much more!

Thanks for listening,

and if you're doing your Ph.d., and this sounds exciting to you, please apply for an internship with us!