Verifying smart contracts

SMT real life challenges and solutions

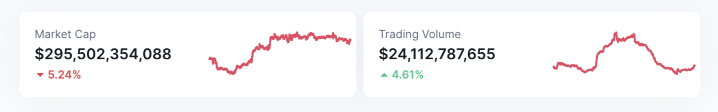
Certora

Yoav Rodeh

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

EVM Storage Model

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

An uninterpreted one-to-one function:

$$\mathit{h} : \mathit{Universe} \to 2^{256}$$

But U is huge $\Longrightarrow$ there is no such h!

Solution: Define an ”inverse” function g, and for every $\mathit{keccack}(\alpha)$ in the code require:

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

That was easy!

Arrays

Recall they are continuous

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

$$keccak(a) + i \neq keccak(b) + j$$

Impossible! yet true

> *Do you know the joke about the difference between theory and practice?*

I won't tell it.

Large Gaps Injectivity

Why stop at 2^{256}

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

$$2^{256} * h(a)$$

Where h is injective as above.

Works surprisingly well, but:

- Arithmetical overhead.
  - Hashes are frequently nested.
- Cannot use Bit-Vector theory.

Plain Injectivity

Pattern Based

- Examine all the expressions that are used to access storage.
- Rewrite each one as $keccak(a) + expr$.
  - $expr$ 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
- $h(a)$ 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.

This is all pretty brittle...

Datatype Theory

The perfect match!

Expressions flowing into storage access get their own datatype. its a tree,

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

- It's all much simpler if fields were not packed.
- Solution 1: detect such patterns, simplify and remove them.
- Problem: Solidity compiler versions and optimizations result in many such patterns.
  - e.g., A few fields can be written at once.
- Solution 2: Follow the bits around and see which ones are never used together.

Split Detection Algorithm

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

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 high 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.

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.

No Quantification

We really don't like it

And many more

On the edge of the fork

- CEGAR.
- Signed arithmetic in 2s complement.
- Axiomatization of bitwise operations.
- Our own array theory implementation, because we need longstores and array initialization.

Lots of other stuff, many of them I have almost no idea about.