Are your Zero-Knowledge Proofs Correct?

Jon Stephens
CTO, Veridise
About Us

Isil Dillig, President
Stanford (BS’06, PhD’11)
CS Prof at UT Austin

Yu Feng, CEO
UT Austin (PhD’18)
CS Prof at UCSB

Jon Stephens, CTO
UT Austin,
Current PhD candidate

Ben Mariano, VP of R&D
UT Austin
Current PhD candidate

Shankara Pailoor, Sr. Scientist
UT Austin
Current PhD candidate

Jacob Van Geffen, Sr. Scientist
University of Washington
Current PhD candidate
Veridise provides state-of-the-art formal methods solutions for all layers in the blockchain ecosystem.

Used to aid our DeFi, ZK Circuit and Blockchain audits. Now available in our SaaS!
Bugs in Zero-Knowledge Circuits
Zero-Knowledge Circuits are pervasive in L2

These circuits present new challenges!
DeFi Bugs

Ethereum DeFi Protocol Beanstalk Hacked for $182 Million—What You Need to Know

Crypto Bridge Nomad Drained of Nearly $200M in Exploit

Harmony's $100M Hack Was Due to a Compromised Multi-Sig Scheme, Says Analyst

Tornado Cash got hacked. By us.

Polygon Dodges $850M Hack, Pays Record $2M Bounty

Solana hit with another network incident causing degraded

Solana's network goes down for the third time in less than six months
What is a Zero-Knowledge Circuit?

Creating a ZK Circuit

- Witness Generator
- Public Input
- Private Input
- Witness
- Prover
- Proof
- Verifier
- Finite Field Constraints
- Prover
- Verifier
What is a Zero-Knowledge Circuit?

ZK Circuit

Witness Generator → Witness → Prover → Proof → Verifier

Public Input

Private Input

Functional Correctness Violation

Witness Generator

Prover

Finite Field Constraints

Verifier
What is a Zero-Knowledge Circuit?

**ZK Circuit**

- **Witness Generator**
- **Prover**
- **Verifier**
- **Public Input**
- **Private Input**
- **Witness**
- **Proof**

**Overconstrained Circuit**

- **Witness Generator**
- **Prover**
- **Verifier**
- **Finite Field Constraints**
What is a Zero-Knowledge Circuit?

**ZK Circuit**

- Witness Generator
- Witness
- Prover
- Proof
- Verifier

**Overconstrained Circuit**

- Witness Generator
- Finite Field Constraints
- Prover
- Verifier

Public Input

Private Input
Formal Methods

Set of techniques for finding bugs and constructing proofs about software

Automated testing
✓ Method: Run program on constructed inputs
✓ Purpose: Bug finding

Static analysis
✓ Method: Analyze source code for specific classes of bugs
✓ Purpose: Bug finding + proof

Formal verification
✓ Method: Analyze source code wrt formal specification
✓ Purpose: Bug finding + proof

Stronger guarantees
More human effort
Common Vulnerability Detection
Uniqueness Bugs

Constraints allow a single input to map to multiple outputs!

```cpp
template Decoder(w) { 
  signal input inp;
  signal output out[w];
  signal output success;
  var lc=0;

  for (var i=0; i<w; i++) {
    out[i] <= (inp == i) ? 1 : 0;
    out[i] * (inp-i) == 0;
    lc = lc + out[i];
  }

  lc ==> success;
  success * (success -1) == 0;
}
```

Entries are zero except `out[in]` if `in < w`

\[ \text{out}[i] \times (\text{inp} - i) = 0 \]

\[ (\text{inp} - i) = 0 \quad \text{out}[i] = 0 \]

When `inp = i`, `out[i]` can be any value and satisfy the above constraint.
Detecting Uniqueness Bugs

Static Analysis of Constraints (ECNE)

Apply predefined rules to quickly detect if circuit is properly constrained

\[
\begin{align*}
\text{input } x \\
\text{output } y \\
z &= 3x + 4 \\
y &= z + 2x
\end{align*}
\]

Since \( y \) is linear in \( x, z \), we immediately infer it is not under constrained

SMT Solver

Underconstrained can be expressed as SMT query

\[ \exists y_1, y_2 \cdot Q[y_1/y] \land Q[y_2/y] \land y_1 \neq y_2 \]

SAT means the circuit is underconstrained

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Analysis</td>
<td>Scalable</td>
</tr>
<tr>
<td>SMT Solver</td>
<td>Precise</td>
</tr>
</tbody>
</table>
If $\mathcal{P}$ is constrained

If $\mathcal{P}$ is unconstrained

Otherwise

Combine the strengths of Static Analysis and SMT!
Static Analysis Phase

Takes Polynomial Constraints (\(\mathcal{P}\)) and set of signals proven unique (K) as input

\[\mathcal{P} \rightarrow \text{Static Analyzer} \rightarrow K'\]

If Output Signals \(\subseteq K'\), return \(\checkmark\)

Otherwise send \(K'\) to SMT Phase
If Output Signals $\subseteq K'$, return $\checkmark$
If Output Signals $\cap K_{\text{uncons}} \neq \emptyset$, return $\times$
If $K = K''$, return $??$
Otherwise invoke Static Analysis phase with $K''$
Results

Picus outperforms Static Analysis and SMT
Functional Correctness Checking
Medjai

Translator -> Symbolic Cairo Program -> Symbolic Execution Engine -> Logical Query -> Query Solver
Symbolic Execution

Starting State:
\{\text{var}[u]=x, \text{var}[v]=y, \text{var}[w]=z\}

If \text{usrs}.contains(\text{u}): 
\text{var}[u] = 5

If \text{usrs}.contains(\text{v}): 
\text{var}[v] = 6

If \text{usrs}.contains(\text{w}): 
\text{var}[w] = 7

...
Optimization

Starting State:
{\text{var}[u]=x, \text{var}[v]=y, \text{var}[w]=z}

if /
\{u\leftarrow5,...\}

else
\{u\leftarrow x,...\}

\{u\leftarrow(\text{ite} \text{usr}.\text{contains}(u) 5 x),...\}

if /
\ ...

else
\ ...

# u, v, w : symbolic felts
func my_func(u, v, w, ...):
    if usr.\text{contains}(u):
        var[u] = 5
    end

    if usr.\text{contains}(v):
        var[v] = 6
    end

    if usr.\text{contains}(w):
        var[w] = 7
    end

...
Example

Goal: Verify and find bugs in ZK Smart Contracts

```
func move(src : felt, dst : felt, rad : Uint256):

    # Sub from src
    let (dai_src) = _dai.read(src)
    let (dai_src) = sub(dai_src, rad)
    _dai.write(src, dai_src)

    # Add to dst
    let (dai_dst) = _dai.read(dst)
    let (dai_dst) = add(dai_dst, rad)
    _dai.write(dst, dai_dst)

    return ()
end
```

Specification:

```plaintext
_dai[dst] >= old(_dai[dst])
_dai[src] <= old(_dai[src])
```

Medjai Output:

```
X
when
rad=Uint256(2**129, 2**129)
```
Example

Goal: Verify and find bugs in ZK Smart Contracts

```plaintext
func move(src : felt, dst : felt, rad : Uint256):
    uint256_check(rad)
    # Sub from src
    let (dai_src) = _dai.read(src)
    let (dai_src) = sub(dai_src, rad)
    _dai.write(src, dai_src)
    # Add to dst
    let (dai_dst) = _dai.read(dst)
    let (dai_dst) = add(dai_dst, rad)
    _dai.write(dst, dai_dst)
    return ()
end
```

Specification:

- `_dai[dst] >= old(_dai[dst])`
- `_dai[src] <= old(_dai[src])`

Medjai Output:

✅
Example

Goal: Verify and find bugs in ZK Smart Contracts

```
func mov(_dai: DAI, dst: DAI, rad: Uint256):
    let (dai_src) = sub(_dai.src, rad)
    _dai.write(src, dai_src)

    # Add to dst
    let (dai_dst) = _dai.read(dst)
    let (dai_dst) = add(dai_dst, rad)
    _dai.write(dst, dai_dst)

    return ()
```

Specification:

- `dai[dst] >= old(_dai[dst])`
- `|dai[src]| <= old(_dai[src])`

maciejka fix fold, suggested by Veridise
Thank you!

Jon Stephens  
CTO, Veridise  
jon@veridise.com

Picus Repository

Medjai Repository

@VeridiseInc