Secure ZK Circuits via Formal Methods

Guest Lecturer: Yu Feng (UCSB & Veridise)

Zero Knowledge Proofs

Instructors: Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang
Bugs in blockchain software are extremely costly.

Bugs in any of these layers can be catastrophic when exploited!
Smart Contract Bugs

Flash loan vulnerability in smart contract
Solana suffered its second outage in a month, sending price plunging

Key Points
- Solana fell more than 50% on Wednesday as the blockchain suffered its second outage in the last month.
- Investors who had been located largely in one area began diversifying into Solana and other alternative blockchains during last year's crypto turmoil.
- But the loss of trust and a lack of confidence in the trade-off as the blockchain network has suffered multiple outages.

DoS vulnerability in consensus protocol
Zcash team fixes serious vulnerability that allowed counterfeiting

- The vulnerability was discovered by a cryptographer from Zcash Company in March 2018.
- Attackers could create fake Zcash coins in large numbers by exploiting this vulnerability.

Bug in arithmetic circuit implementing zkSNARK!
Formal Methods to Rescue

Formal methods can eradicate these bugs
Section 1
Formal Methods in a Nutshell
What is Formal Methods

Set of mathematically rigorous techniques for finding bugs and constructing proofs about software
Formal Methods Techniques on Spectrum

- Fuzzing
- Concolic Execution
- Abstract Interpretation

Stronger guarantees
More human effort
Classification of FM Techniques

DYNAMIC
Execute the program on interesting inputs & monitor what happens

STATIC
Analyze source code and reason about all executions

FUZZING
CONCOLIC EXECUTION

ABSTRACT INTERPRETATION
FORMAL VERIFICATION

EXECUTE THE PROGRAM ON INTERESTING INPUTS & MONITOR WHAT HAPPENS

ANALYZE SOURCE CODE AND REASON ABOUT ALL EXECUTIONS
● Blue irregular shape is the actual states
● Red region corresponds to “bad states”
● Due to undecidability, we can never determine exactly what the blue region is
● Over-approximate blue region with the regular green region above
Fundamentals of Static Analysis

False Positives

Actual states
States computed by static analysis

False positives!

False Negatives

Actual states
Unsound static analysis
Bad states
Concrete Interpretation is Easy

If \( f(x) = x+2 \), then \( f(1) = 3 \)
Static Analysis via Abstract Interpretation

Abstract values

Code snippet

Abstract Interpreter

New abstract values
Idea: Emulate all possible program paths

```python
if(flag):
    x = 1;
else:
    x = -1;
```

When in doubt, conservatively assume either path could be taken and merge information for different paths

$$x \in [-1,1]$$
Abstract Interpretation Tools in Web3

• Slither (TrailOfBits)
• Sailfish (Bose et al, Oakland’22)
• Vanguard (Veridise)
Static Analysis via Formal Verification

- Program implementation: Source code of the program, or intermediate representation
- The specification: A formal description of the property to be verified
- Human annotations (optional): Loop invariants, Contract invariants
Formal Specifications

Formal specification: Precise mathematical description of intended program behavior, typically in some formal logic

\[ ((\text{finish}(\text{bid}, \text{msg} . \text{value} = X \land \text{msg} . \text{sender} = L)) \land \Diamond \text{finish}(\text{close}, L \neq \text{winner})) \rightarrow \Diamond \text{send}(\text{to} = L \land \text{amt} = X)) \]
Formal Verification Tools in Web3

- Certora prover (Certora)
- K framework (Runtime Verification)
Different Flavors of Static Analysis

Formal verification checks program against provided specification

**Abstract Interpretation**
- Looks for known types of bugs
- Doesn’t require specifications

**Formal Verification**
- Can find (prove absence of) any bug
- Requires specifications
Section 2
Formal Methods in ZK: part I
Circuits Workflow

Source Code

Source Code: Witness Generation and Constraints should (generally) be equivalent!

Witness Generation

Halo2

SNARK

Prover

Verifier

Equations

Polynomial Field

C

P

22
What is Equivalence

Every input-output of \( P \) must satisfy \( C \)

Every \((x,y)\) which satisfy \( C \) must be an input-out pair of \( P \)

**For every** \( x, y \). \( P(x) = y \) **if and only if** \( C(x, y) \) **is true**

**How can this be violated?**

**Program:** \( P \)  
**Set of Constraints:** \( C \)  
**Input:** \( x \)  
**Inputs:** \( x, y \)  
**Output:** \( y \)  
**Output:** \( true \) or \( false \)
Equivalence Violations

Two Requirements:

1. Every input-out pair of \( P \) satisfies \( C \)
2. For any \( x, y \) which satisfy \( C \), \( P(x) = y \)

Overconstrained Bugs

\[
\text{Exists } x, y \text{ where } P(x) = y \text{ but } C(x, y) \text{ is false}
\]

Underconstrained Bugs

\[
\text{Exists } x, y \text{ where } C(x, y) \text{ is true but } P(x) \neq y
\]

Most ZK languages (e.g., Circom, Halo2) add field equations as assertions to circuit!
Why Do We Care

ZK Circuit Workflow

Source Code

Compiled

Polynomial Field Equations

SNARK

Prover

Verifier

Could be used to drain all tokens

Underconstrained bugs: Verifier can accept bad inputs/outputs

Tornado Cash
Oct 12, 2019 - 3 min read - Listen

Tornado_cash got hacked. By us.

Double spend

BigMod incorrectly omits range checks on the remainder #10

Disclosure of recent vulnerabilities

We have recently patched two severe bugs in Aztec 2.0. The first was found by an Aztec engineer and the second by community members.

1. Lack of range constraints for the `tree_index` variable
A Taxonomy of ZK Bugs

- Unconstrained Signals
  - Unconstrained Output
  - Unconstrained Public Signal
- Constraint/Computation
  - No Zero Inverse
- Unsafe Component Usage
  - Under-Constrained Sub-Circuit Output
  - Under-Constrained Sub-Circuit Input
Unconstrained Signals

Corresponds to signals whose constraints always evaluate to true, accepting everything
Underconstrained Output

**Buggy Implementation**

```cpp
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1 = 0;
    var e2 = 1;
    for (var i = 0; i < n-1; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] - 1) == 0;
        lc1 += out[i] * e2;
        e2 = e2 + e2;
    }
    lc1 == in;
}
```

**Constraints for \( n = 3 \)**

\[
\begin{align*}
\text{input } in & \\
\text{output } out_0, out_1, out_2 &
\end{align*}
\]

\[
\begin{align*}
out_0 \cdot (out_0 - 1) &= 0 \\
out_1 \cdot (out_1 - 1) &= 0 \\
out_0 + 2 \cdot out_1 &= in
\end{align*}
\]

*Attacker can pass in any value for \( out_2 \)*

Unsafe Component Usage

Sub-circuits often assume constraints are placed on inputs and outputs

Corresponds to cases where the use of a sub-circuit do not follow

Unsafe Component Usage

Under-Constrained Sub-Circuit Output

Under-Constrained Sub-Circuit Input
Example: Under-Constrained Sub-Circuit Output

```c
template withdraw(n) {
    assert(n \leq 252);
    signal input bal;
    signal input amt;
    signal output out;

    component n2b1 = Num2Bits(n); // assert (bal < 2^n)
        n2b1.in \leq bal;
    component n2b2 = Num2Bits(n); // assert (amt < 2^n)
        n2b2.in \leq amt;
    component lt = LessThan(n); // check amt < bal
        lt.in[0] \leq bal;
        lt.in[1] \leq amt;

    out \leq bal - amt;
}
```

Without the missing constraint, attacker can withdraw more funds than they have
Constraint/Computation Discrepancy

Not all computation can be directly expressed as a constraint.

Corresponds to constraints that do not capture a computation’s semantics.
Example: No Zero Inverse

```
template MulInverse() {
  signal input a;
  signal input b;
  signal output out;

  out <-- a / b;
  out * b === a;
}
```

Multiplicative inverse undefined when \( b = 0 \)

Constraints allow \( b = 0 \)

Accepts arbitrary \( out \) when \( a \) and \( b \) are 0!
Circuit Dependence Graphs (CDG)

**Goal:** Identify discrepancies between computation and constraints

Diagram:

- **Input Signal** (i) connected to **Output Signal** (o)
- **Equivalence** (i <-> i) and **Implication** (i -> o) indicated
- **In** and **Out** labels for input and output signals
Vanguard Static Analysis

Used to evaluate 258 circuits from 17 public Circom projects on Github
Evaluation Results

Developers have the most difficulty reasoning about a computation’s semantics!

- Identified 32 previously unknown vulnerabilities!
  - Some Circuits had multiple bugs!
  - Unconstrained Signals
  - Unsafe Component Usage
  - Constraint/Computation Discrepancy

35
Section 3
Formal Methods in ZK: part II
Existing Strategies

**Static Analysis of Constraints (SA)**

- 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

\[
\exists y_1, y_2, P[y_1/y] \land P[y_2/y] \land y_1 \neq y_2
\]

SAT means the circuit is underconstrained
If it can prove $P$ is constrained

If it can prove $P$ is unconstrained

Otherwise

Combine the strengths of Static Analysis and SMT!

Fast but imprecise!

Precise but slow!

Static Analysis and SMT phases interact in a loop
Static Analysis Phase

Takes as input field equations $P$, and set of signals $K$ proven

At the start of the algorithm $K = \{ \}$.  

If $\text{OutputSignals} \subseteq K'$ we return ✓

Otherwise we send $K'$ as input to SMT Phase

New set $K'$ of signals proven unique. $K \subseteq K'$
SMT Phase

If OutputSignals \subseteq K'' we return

If OutputSignals \cap K_{uncons} \neq \emptyset we return

If K = K'' we return

Otherwise we send K'' to Static Analysis phase and repeat.
$ ./picus-solve.sh ./benchmarks/motivating/adder.r1cs
# number of constraints: 9
# parsing alternative r1cs...
# configuring precondition...
safe.

Guaranteed to have no underconstrained signals!
Evaluation

<table>
<thead>
<tr>
<th>Benchmark Set</th>
<th># circuits</th>
<th>Avg. # constraints</th>
<th>Avg. # output signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>circomlib-utils</td>
<td>59</td>
<td>352</td>
<td>10</td>
</tr>
<tr>
<td>circomlib-core</td>
<td>104</td>
<td>6,690</td>
<td>32</td>
</tr>
<tr>
<td>All</td>
<td>163</td>
<td>4,396</td>
<td>24</td>
</tr>
</tbody>
</table>

![Bar chart showing solved (%) for different benchmark sets.](chart.png)
Conclusion

• Automated Detection of Underconstrained Circuits for Zero-Knowledge Proofs, PLDI’23
• Practical Security Analysis of Zero-Knowledge Proof Circuits
• Certifying Zero-Knowledge Circuits with Refinement Types