Formal Verification of Arithmetic Circuits

Marcin Kostrzewa

reilabs

Real bugs in ZK protocols

Under-constrained Circuits

Nondeterministic Circuits

Arithmetic Over/Under Flows

Mismatching Bit Lengths

Unused Public Inputs Optimized Out

Frozen Heart: Forging of Zero Knowledge Proofs

Trusted Setup Leak

Assigned but not Constrained

source: https://github.com/0xPARC/zk-bug-tracker

reilabs

Real bugs in ZK protocols

Under-constrained Circuits

Nondeterministic Circuits

Arithmetic Over/Under Flows

Mismatching Bit Lengths

Unused Public Inputs Optimized Out

Frozen Heart: Forging of Zero Knowledge Proofs

Trusted Setup Leak

Assigned but not Constrained

source: https://github.com/0xPARC/zk-bug-tracker

reilabs

17 out of 26 listed bugs are circuit issues

reilabs

The most powerful design principle

reilabs

The most powerful design principle

Producing values out of thin air, then constraining them.

reilabs

The most powerful design principle

Producing values out of thin air, then constraining them.

For example, to show $N < 8$:

1. Take x_1, x_2, x_3 out of thin air

reilabs

The most powerful design principle

Producing values out of thin air, then constraining them.

For example, to show N < 8:

1. Take x_1, x_2, x_3 out of thin air
2. Constrain:

$$x_1 * (1 - x_1) = 0$$
$$x_2 * (1 - x_2) = 0$$
$$x_3 * (1 - x_3) = 0$$
$$x_1 + 2 * x_2 + 4 * x_3 = N$$

reilabs

The most powerful design principle

Producing values out of thin air, then constraining them.

For example, to show N < 8:

1. Take x_1, x_2, x_3 out of thin air
2. Constrain:

$$x_1 * (1 - x_1) = 0$$
$$x_2 * (1 - x_2) = 0$$
$$x_3 * (1 - x_3) = 0$$
$$x_1 + 2 * x_2 + 4 * x_3 = N$$

Even this simple example has some deep mathematical content!

reilabs

The most powerful design principle

Producing values out of thin air, then constraining them.

For example, to show N < 8:

1. Take x_1, x_2, x_3 out of thin air
2. Constrain:

$$x_1 * (1 - x_1) = 0$$
$$x_2 * (1 - x_2) = 0$$
$$x_3 * (1 - x_3) = 0$$
$$x_1 + 2 * x_2 + 4 * x_3 = N$$

Even this simple example has some deep mathematical content!

And doesn't work at full field bit-width.

reilabs

If you can’t avoid it, prove it!

reilabs

Lean 4

reilabs

Lean 4

→ Dependently-typed functional programming language

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures
- → Proofs are code

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures
- → Proofs are code
  - ↳ Maintainable

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures
- → Proofs are code
  - ↳ Maintainable
  - ↳ Collaborative

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures
- → Proofs are code
  - ↳ Maintainable
  - ↳ Collaborative
- → Rich corpus of mathematics already implemented

reilabs

Lean 4

- → Dependently-typed functional programming language
- → Proof assistant
  - ↳ Can reason about programs & mathematical structures
- → Proofs are code
  - ↳ Maintainable
  - ↳ Collaborative
- → Rich corpus of mathematics already implemented
  - ↳ Recently completed the proof of Fermat's Last Theorem!

reilabs

Modelling circuits in Lean

reilabs

Modelling circuits in Lean

→ Use the `Prop` type to represent a constraint system.

reilabs

Modelling circuits in Lean

- → Use the `Prop` type to represent a constraint system.
- → Use Continuation Passing Style (CPS) for composing constraints:

```
def Constraint (α: Type): Type := (α → Prop) → Prop
```

`Constraint Foo` performs some assertions and passes a value of type Foo to its caller.

reilabs

Modelling circuits in Lean

→ Use existential quantification for “values out of thin air”:

```
def unconstrained (α : Type): Constraint α := fun k => ∃ a, k a
```

i.e. pass a value of the required type to the caller, without saying anything about it

reilabs

Modelling circuits in Lean

→ Use existential quantification for “values out of thin air”:

```
def unconstrained (α : Type): Constraint α := fun k => ∃ a, k a
```

i.e. pass a value of the required type to the caller, without saying anything about it

→ One more useful helper:

```
def assert (p : Prop) : Constraint Unit := fun k => p ∧ k ()
```

i.e. add a condition to the resulting proposition

reilabs

Example: binary representation

reilabs

Example: binary representation

```
def unconstrained_bit : Constraint F := fun k =>
 unconstrained F fun x =>
 assert (x * (1 - x) = 0) fun _ =>
 k x
```

reilabs

Example: binary representation

```
def unconstrained_bit : Constraint F := fun k =>
 unconstrained F fun x =>
 assert (x * (1 - x) = 0) fun _ =>
 k x

def binary_rep_3 (f : F) : Constraint (Vector F 3) := fun k =>
 unconstrained_bit fun x₁ =>
 unconstrained_bit fun x₂ =>
 unconstrained_bit fun x₃ =>
 assert (x₁ + x₂ * 2 + x₃ * 4 = f) fun _ =>
 k (x₁ ::ᵥ x₂ ::ᵥ x₃ ::ᵥ Vector.nil)
```

reilabs

🚨 Monad Alert 🚨

reilabs

Prettier example

```
def unconstrained_bit : Constraint F := do
 let x ← unconstrained F
 assert (x * (1 - x) = 0)
 return x

def binary_rep_3 (f : F) : Constraint (Vector F 3) := do
 let x₁ ← unconstrained_bit
 let x₂ ← unconstrained_bit
 let x₃ ← unconstrained_bit
 assert (x₁ + x₂ * 2 + x₃ * 4 = f)
 return x₁ ::ᵥ x₂ ::ᵥ x₃ ::ᵥ Vector.nil
```

reilabs

Deterministic Constraints

reilabs

Deterministic Constraints

→ `def deterministic : α → Constraint α := fun a k => k a`

Embeds a known value into a constraint system.

reilabs

Deterministic Constraints

- → `def deterministic : α → Constraint α := fun a k => k a`

  Embeds a known value into a constraint system.

- → Our first theorem:

  ```
  theorem binary_rep_3_deterministic:
    ∀ f, f.val < 8 → ∃ rep, binary_rep_3 f = deterministic rep
  ```

  For any element less than 8, we can replace the call to binary_rep_3 with a deterministic computation.

reilabs

Deterministic Constraints

- → `def deterministic : α → Constraint α := fun a k => k a`

  Embeds a known value into a constraint system.

- → Our first theorem:

  ```
  theorem binary_rep_3_deterministic:
    ∀ f, f.val < 8 → ∃ rep, binary_rep_3 f = deterministic rep
  ```

  For any element less than 8, we can replace the call to binary_rep_3 with a deterministic computation.

- → This gadget alone eliminates 3 classes of bugs: **under-constrained circuits**, **nondeterministic circuits** and **assigned but not constrained**.

reilabs

Semantic Properties

reilabs

Semantic Properties

→ We can show that our gadget actually works as a range check:

```
theorem binary_rep_3_rangecheck:
  ∀ f k, binary_rep_3 f k → f.val < 8
```

reilabs

Semantic Properties of Real Protocols

reilabs

Semantic Properties of Real Protocols

```
theorem signaller_is_in_tree:
 ∀ (IdentityNullifier IdentityTrapdoor SignalHash ExtNullifier NullifierHash : F)
   (Tree : MerkleTree F poseidon₂ 20) (Path Proof: Vector F 20),
 (collision_resistant poseidon₂ ∧
  Semaphore.circuit IdentityNullifier IdentityTrapdoor Path Proof
                    SignalHash ExtNullifier NullifierHash Tree.root) →
 Tree.item_at (Dir.create_dir_vec Path.reverse) =
   identity_commitment IdentityNullifier IdentityTrapdoor
```

Correctness of the Semaphore protocol – there is no unauthorized signalling.

reilabs

Semantic Properties of Real Protocols

```
theorem no_double_signal_with_same_commitment:
 ∀ (IdentityNullifier₁ IdentityNullifier₂ IdentityTrapdoor₁ IdentityTrapdoor₂
     SignalHash₁ SignalHash₂ ExtNullifier₁ ExtNullifier₂ NullifierHash₁
     NullifierHash₂ Root₁ Root₂ : F)
   (Path₁ Proof₁ Path₂ Proof₂: Vector F 20),
 (collision_resistant poseidon₂ ∧ collision_resistant poseidon₁ ∧
  Semaphore.circuit IdentityNullifier₁ IdentityTrapdoor₁ Path₁ Proof₁ SignalHash₁
                    ExtNullifier₁ NullifierHash₁ Root₁ ∧
  Semaphore.circuit IdentityNullifier₂ IdentityTrapdoor₂ Path₂ Proof₂ SignalHash₂
                    ExtNullifier₂ NullifierHash₂ Root₂ ∧
  ExtNullifier₁ = ExtNullifier₂ ∧
  identity_commitment IdentityNullifier₁ IdentityTrapdoor₁ =
    identity_commitment IdentityNullifier₂ IdentityTrapdoor₂) →
  NullifierHash₁ = NullifierHash₂
```

reilabs

Compiling to Lean

→ This is all very cool, but we don’t actually use Lean for circuit development.

reilabs

Compiling to Lean

→ This is all very cool, but we don't actually use Lean for circuit development.

→ So we've also built a compiler from Gnark to Lean.

https://github.com/reilabs/gnark-lean-demo

reilabs

Full workflow

reilabs

Full workflow

→ Write your circuits in Gnark

reilabs

Full workflow

- → Write your circuits in Gnark
- → Use our compiler to emit equivalent Lean definitions

reilabs

Full workflow

- → Write your circuits in Gnark
- → Use our compiler to emit equivalent Lean definitions
- → Define the theorems and prove them

reilabs

Full workflow

- → Write your circuits in Gnark
- → Use our compiler to emit equivalent Lean definitions
- → Define the theorems and prove them
- → Run the verification with continuous integration to never break the circuits again!

reilabs

Results in the wild

reilabs

Results in the wild

→ Uncovered an unsoundness in Gnark’s standard library

reilabs

Results in the wild

→ Uncovered an unsoundness in Gnark's standard library

↳ Number comparison gadget was broken – due to full-bit-width binary decompositions

reilabs

Results in the wild

- → Uncovered an unsoundness in Gnark's standard library
  - ↳ Number comparison gadget was broken – due to full-bit-width binary decompositions
- → Verified Worldcoin's state-management circuits - one of the most frequently used production deployments of a ZK circuit

reilabs

Further work

reilabs

Further work

→ Better support for probabilistic reasoning

reilabs

Further work

- → Better support for probabilistic reasoning
  - ↳ Useful for in-circuit commitments, recursion etc.

reilabs

Further work

- → Better support for probabilistic reasoning
  - ↳ Useful for in-circuit commitments, recursion etc.
- → A Noir compiler!

reilabs

Thank you!

@reilabs_io

reilabs