Lecture 3: Programming ZKPs

Guest Lecturers: Pratyush Mishra and Alex Ozdemir

Zero Knowledge Proofs

Instructors: Dan Boneh, Shafi Goldwasser, Dawn Song, Justin Thaler, Yupeng Zhang
Using a ZKP

Idea → Program → R1CS → Params → ZKP

Programmer → Compiler* → Setup → Prove → ZKP

This Lecture

\[ \text{Verify} \rightarrow \{0, 1\} \]
This Lecture

1. Big Picture: ZKP programmability
2. Using an HDL (+ tutorial)
3. Using a library (+ tutorial)
4. Using a compiler (+ tutorial)
5. An overview of prominent ZKP toolchains
ZKP Programmability
Recap: ZKPs for a predicate $\phi$

- Prover knows $\phi, x, w$
- Verifier knows $\phi, x$
- Proof $\pi$ shows that $\phi(x, w)$ holds
  - but does not reveal $w$

- Key Question: what can $\phi$ be?
What is $\phi$?

$\phi$ in theory

- $\psi$ is a factorization of integer $x$
- $\psi$ is the secret key for public key $x$
- $\psi$ is the credential for account $x$
- $\psi$ is a valid transaction

$\phi$ in practice

- $\phi$ is an “arithmetic circuit” over inputs $x, \psi$
Arithmetic Circuits (ACs), Part I

- Domain: “prime field”
  - \( p \): a large (~255 bit) prime
  - \( \mathbb{Z}_p \): the integers, mod \( p \)
    - operations: +,\( \times \), = (mod \( p \))
  - Example in \( \mathbb{Z}_5 \):
    - \( 4 + 5 = 9 = 4 \)
    - \( 4 \times 4 = 16 = 1 \)

- ACs as systems of field equations:
  - Example:
    - \( w_0 \times w_0 \times w_0 = x \)
    - \( w_1 \times w_1 = x \)
    - Addition is also OK
Arithmetic Circuits (ACs), Part II

- **ACs as circuits**
  - Directed, acyclic graph
  - Nodes: inputs, gates, constants
  - Edges: wires/connections

- **Example:**
  - $w_0 \times w_0 \times w_0 = x$
  - $w_1 \times w_1 = x$

- As a circuit:
R1CS: a common Arithmetic Circuit format

- **R1CS: format for ZKP ACs**
  - **Definition:**
    - $x$: field elements $x_1, \ldots, x_\ell$
    - $w$: $w_1, \ldots, w_{m-\ell-1}$
    - $\phi$: $n$ equations of form
      - $\alpha \times \beta = \gamma$
      - where $\alpha, \beta, \gamma$ are affine combinations of variables
  - **Examples:**
    - $w_2 \times (w_3 - w_2 - 1) = x_1$
    - $w_2 \times w_2 = w_2$ (Corrected: $w_2 \times w_2 = w_2$ should be $w_2 \times w_2 = w_4$)
    - $w_2 \times w_2 \times w_2 = x_1$ (Corrected: $w_2 \times w_2 \times w_2 = x_1$ should be $w_4 \times w_2 = x_1$)
R1CS: Matrix Definition

- $x$: vector of $\ell$ field elements
- $w$: vector of $m - \ell - 1$ field elements
- $\phi$: matrices $A, B, C \in \mathbb{Z}_p^{n \times m}$
  - $z = (1 \parallel x \parallel w) \in \mathbb{Z}_p^m$
  - Holds when $Az \odot Bz = Cz$

One rank-1 constraint

Element-wise product
Writing an AC as R1CS (Example)

- **Step 1**: intermediate $w$s
- **Step 2**: write equations
  - $w_0 \times w_1 = w_2$
  - $w_3 = w_2 + x_0$
  - $w_1 \times x_0 = w_4$
  - $w_3 = w_4$
Zooming out: a Programming Languages problem

High-level specification for $\phi$

Program in high-level language

Using:
- Libraries
- Compilers
- Programming Langs.
- Domain Specific Langs.
- ...

R1CS

Assembly
The Idea

- Booleans
- Structures
- Modules
- Functions
- ...

High-level code $\Rightarrow$ Compiler/Library $\Rightarrow$ R1CS $\Rightarrow$ ZK Proof System
An Example

- Merkle tree
- Pedersen Hash
- Signatures
- Spend logic
- ...

Zcash Circuit → Bellman Library → R1CS → Groth16
An HDL for R1CS
**Programming Languages (PLs) vs. Hardware Description Languages (HDLs)**

<table>
<thead>
<tr>
<th>PL objects</th>
<th>HDL objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>Wires</td>
</tr>
<tr>
<td>Operations</td>
<td>Gates</td>
</tr>
<tr>
<td>Program/Functions</td>
<td>Circuit/Sub-circuits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PL actions</th>
<th>HDL actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutate variables</td>
<td>Connect wires</td>
</tr>
<tr>
<td>Call functions</td>
<td>Create sub-circuits</td>
</tr>
</tbody>
</table>
HDLs: From Digital to Arithmetic

HDLs for Digital Circuits
- Verilog
- SystemVerilog
- VHDL
- Chisel
- ...

An HDL for R1CS
- circom
  - wires: R1CS variables
  - gates: R1CS constraints
- a circom circuit does 2 things:
  - sets variable values
  - creates R1CS constraints
A “template” is a (sub)circuit
- A “signal” is a wire
  - “input” or “output”
- “<--” sets signal values
- “===” creates constraints
  - Must be rank-1:
    - one side: linear
    - other side: quadratic
- “<==” does both

```cpp
template Multiply() {
    signal input x;
    signal input y;
    signal output z;
    z <-- x * y;
    z == x * y;
    // ERROR: z == x * x * y
    // OR   : z <= x * y;
}

component main {public [x]} = Multiply();
```
Circom: Metaprogramming Language

- Template arguments
- Signal arrays
- Variables
  - Mutable
  - Not signals
  - Evaluated at compile-time
- Loops
- If statements
- Array accesses

```cpp
template RepeatedSquaring(n) {
    signal input x;
    signal output y;

    signal xs[n];
    xs[0] <= x;
    for (var i = 0; i <= n; i++) {
        xs[i+1] <= xs[i] * xs[i];
    }
    y <= xs[n];
}
component main {public [x]} =
RepeatedSquaring(1000);
```
Circom: Witness Computation & Sub-circuits

- **Witness computation:** more general than R1CS
  - “<--” is more general than “==”
- “component’s hold sub-circuits
- Access inputs/outputs with dot-notation

```
template NonZero() {
    signal input in;
    signal inverse;
    inverse <-- 1 / in; // not R1CS
    1 == in * signal; // is R1CS
}

template Main() {
    signal input a; signal input b;
    component nz = NonZero();
    nz.in <= a;
    0 == a * b;
}
```
Circom Tutorial
Tutorial Example: Sudoku

- 9 by 9 grid
- Some cells have #s
- Goal: fill all cells with 1...9
- Rule: no duplicates in any:
  - Column
  - Row
  - 3x3 sub-grid
A Library for R1CS
Circom: Recap

- An HDL for R1CS
- Key features:
  - Direct control over constraints
  - Custom language
    - Can be good
    - Can be bad
R1CS Libraries

- A library in a *host* language (Eg: Rust, OCaml, C++, Go, ...)
- Key type: *constraint system*
  - Maintains state about R1CS constraints and variables
- Key operations:
  - create variable
  - create *linear combinations* of variables
  - add constraint

![Constraint System Diagram]

```plaintext
A
B
C
```

variables
Variable creation
\[ \text{cs.add_var}(p, v) \rightarrow \text{id} \]
- \text{cs}: constraint system
- \text{p}: visibility of variable
- \text{v}: assigned value
- \text{id}: variable handle

Linear Combination creation
\[ \text{cs.zero}() \rightarrow \text{lc} \]
\[ \text{lc.add}(c, \text{id}) \rightarrow \text{lc}' \]
- \text{id}: variable
- \text{c}: coefficient
- \text{lc'} := \text{lc} + c \times \text{id}

Adding constraints
\[ \text{cs.constrain}(\text{lc}_A, \text{lc}_B, \text{lc}_C) \]
- Adds a constraint \( \text{lc}_A \times \text{lc}_B = \text{lc}_C \)
Example: Boolean AND

```rust
fn and(cs: ConstraintSystem, a: Var, b: Var) → Var {
    let result = cs.new_witness_var(|| a.value() & b.value());
    self.cs.enforce_constraint(
        lc!() + a,
        lc!() + b,
        lc!() + result,
    );
    result
}
```

- Create result variable
- Create linear combinations
- Enforce constraint
Example: Boolean AND

```rust
fn and(cs: ConstraintSystem, a: Var, b: Var) → Var {
    let result = cs.new_witness_var(|| a.value() & b.value());
    self.cs.enforce_constraint(lc!() + a, lc!() + b, lc!() + result);
    result
}
```

This is unpleasant, tedious, and error-prone!

Can you imagine writing a complex algorithm like signature verification in this style?
Idea: Leverage Language Abstractions!

We can use language abstractions like structs, operator overloading, methods, etc. to allow better developer UX:

```rust
struct Boolean { var: Var }

impl BitAnd for Boolean {
    fn and(self: Boolean, other: Boolean) → Boolean {
        // Same as before
        Boolean { var: result }
    }
}
```

Wrap variable in dedicated type
Implement interface for operator overloading
Does it work? Yes!

Can use abstractions like normal code:

```javascript
let a = Boolean::new_witness(|| true);
let b = Boolean::new_witness(|| false);
(a & b).enforce_equal(Boolean::FALSE);
```

Many different gadget libraries:
- libsnark: gadgetlib (C++)
- arkworks: r1cs-std + crypto-primitives (Rust)
- Snarky (Ocaml)
- Gnark (Go)
What about Witness Computation?

- Can perform arbitrary computations to generate witnesses

```rust
let a = Boolean::new_witness(|| (4 == 5) & (x < y));
let b = Boolean::new_witness(|| false);
(a & b).enforce_equal(Boolean::FALSE);
```

Closure (lambda) executed only during proving!
Arkworks Tutorial
Compiling a Programming Language to R1CS
HDLs & Circuit Libraries

- **Difference:**
  - Host language v. custom language

- **Similarities:**
  - explicit wire creation (explicitly wire values)
  - explicit constraint creation

- **Do we need to explicitly build a circuit?**
  - No!
Compiling PLs to Circuits (Idea)

Program

```python
fn main(...) {
  ...
}
```

- Variables
- Mutation
- Functions
- Arrays

Compiler

R1CS

- Wires
- Constraints
ZoKrates: Syntax

- Struct syntax for custom types
- Variables contain values during execution/proving
- Can annotate privacy
- “assert” creates constraints

```plaintext
type F = field;

def main(public F x, private F[2] ys) {
    field y0 = y[0];
    field y1 = y[1];
    assert(x = y0 * y1);
}
```
Zokrates: Language features

- Integer generics
- Arrays
- Variables
  - Mutable
- Fixed-length loops
- If expressions
- Array accesses

```rust
def repeated_squaring<N>(field x) -> field {
    field[N] mut xs;
    xs[0] = x;
    for u32 i in 0..n {
        xs[i + 1] = xs[i] * xs[i];
    }
    return xs[N];
}

def main (public field x) -> field {
    repeated_squaring::<1000>(x)
}
```
What about Witness Computation?

- No way to compute witnesses
- All witnesses must be provided as input

```python
def main(private field a, public field b) {
    assert(a * b == 1)
}
```
ZoKrates Tutorial
ZKP Toolchains: 
A Quick Tour
Toolchain Type

**HDL**
- a language for describing circuit synthesis

**Library**
- a library for describing circuit synthesis

**PL + Compiler**
- a language, compiled to a circuit

```
fn main(...)
{
  circ.add_wire(...)
  circ.add_gate(...)
  ...
}
```
### Toolchain Types, Organized

<table>
<thead>
<tr>
<th>Language Type</th>
<th>Standalone Language?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Circuit</td>
<td>Library (arkworks)</td>
</tr>
<tr>
<td>Program</td>
<td></td>
</tr>
</tbody>
</table>
ZoKrates
Pros:
- Easiest to learn
- Elegant syntax
Cons:
- Limited witness computation
- Few optimizations

Arkworks
Pros:
- Clear constraints
- As expressive as Rust
Cons:
- Need to know Rust
- Few optimizations

Circom
Pros:
- Clear constraints
- Elegant syntax
Cons:
- Hard to learn
- Limited abstraction

HDL types?

===

HDL

just a PL

manual opts

always implicit
## Other toolchains

<table>
<thead>
<tr>
<th>HDL</th>
<th>Library</th>
<th>PL + Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Circom</td>
<td>• Arkworks (Rust)</td>
<td>• ZoKrates</td>
</tr>
<tr>
<td></td>
<td>• Gadgetlib (C++)</td>
<td>• Noir</td>
</tr>
<tr>
<td></td>
<td>• Bellman (Rust)</td>
<td>• Leo</td>
</tr>
<tr>
<td></td>
<td>• Snarky (OCaml)</td>
<td>• Cairo</td>
</tr>
<tr>
<td></td>
<td>• PLONKish (Rust)</td>
<td></td>
</tr>
</tbody>
</table>
Timeline

Gadgetlib 2015

Bellman

Snarky

Arkworks

Noir

PLONKish

Circom

ZoKratess

Cairo

Leo

?
Shared Compiler Infrastructure?

Source
- Cairo
- ZoKrates
- Circom
- PLONKish
- Noir
- Snarky
- Bellman
- Gadgetlib
- Leo

Common Techniques
- RAM
- Boolean
- Fixed-Width Int.
- RAM checking
- Structures
- Mutation
- Variables
- Control Flow
- Optimization

CirC
A library for ZKP languages

Target
- R1CS
- Plonk
- AIR
Summary

Idea

- HDL (circom)
- Library (ark)
- PL (noir)

R1CS

ZKP

\(\pi\)
End of Lecture