How do we use formal methods to harden ZK-rollups

Denis Firsov
Matter Labs

Motivation

“ZK-Rollups bring scalability without compromises in security.”

Motivation

- Scalability
  - Gets a lots of attention
  - Easy to check the claims (TPS, downtime, etc.)

- Security
  - Remains a promise?
  - Evidences and guarantees?

Security

- Evidences of security
  a. Documentation;
  b. Tested;
  c. Audited.

- Are these evidences rocksolid conclusive?

- Does it make sense to ask for more?

Troubles with testing and auditing

- Testing/Fuzzying
  - Can help catch “programming” and correctness mistakes.
  - Meaningless for security properties like zero-knowledge, soundness, and proof-of-knowledge.
  - Never conclusive

- Auditing
  - Quality of code
  - Time spent
  - Experience of auditors
  - Never conclusive

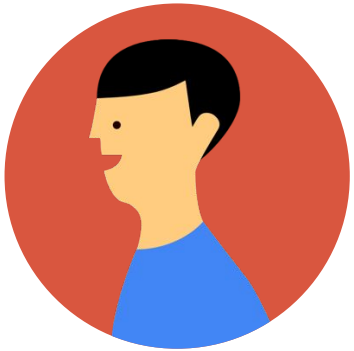
- For critical components we need conclusive proofs of security.

The trouble with security proofs

- **In theory:** Once the protocol is proven secure it is secure forever!

- **In practice:**
  a. Security proof is wrong
  b. Implementation is broken

Crisis of rigor in cryptography

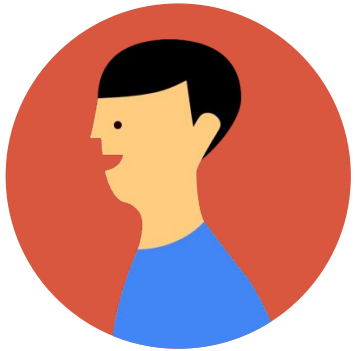


Shai Halevi, 2005

"We generate more proofs than we carefully verify."

Bristol Crypto Group, 2017

"Security proofs for even simple cryptographic systems are dangerous and ugly beasts."



Bellare and Rogeway, 2004

"In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor."

Rigor through formal methods

- The main idea behind formal methods is to implement formal logic as a computer system - theorem prover - in the similar way to programming languages.

- In this way the proofs are implemented by a humans and checked automatically by a computer.

- **Insight:** it (still) takes a human to write (non-trivial) proofs, but computer can efficiently verify formally written proofs (and without compromises).

Formal methods for ZK-Rollups?

- Trusted codebase (TCB) of ZK-Rollups is HUGE and grows.
- Impossible to formally verify entire TCB.
- Must find and focus on the critical targets.

Many possible targets

- Architecture of a rollup
- Rollup contracts on L1
- Consensus
- State diff/compress
- Virtual machine
- ZK-Circuits
- ZK-Verifier

Many possible targets

- Architecture of a rollup
- Rollup contracts on L1
- Consensus
- State diff/compress
- Virtual machine
- ZK-Circuits
- **ZK-Verifier**

ZK-Verifier

- Assets of a typical ZK-rollup are guarded by a L1 `Verifier[.sol]` contract which checks the validity of ZK-proofs.
- Verifier is the ultimate source of truth.
- So, it is vitally important for this contract to be sound, i.e., not accept the proofs of false statements.
- Critically important for decentralization.

PLONK Verifier in LaTeX

$$e([W_{\mathfrak{z}}]_1 + u \cdot [W_{\mathfrak{z}\omega}]_1, [x]_2) \stackrel{?}{=} e(\mathfrak{z} \cdot [W_{\mathfrak{z}}]_1 + u\mathfrak{z}\omega \cdot [W_{\mathfrak{z}\omega}]_1 + [F]_1 - [E]_1, [1]_2)$$

1% of PLONK Verifier in YUL

```
let accumulator
let intermediateValue
//  = alpha * (a(z)^2 - b(z))
accumulator := mulmod(stateOpening0AtZ, stateOpening0AtZ, R_MOD)
accumulator := addmod(accumulator, sub(R_MOD, stateOpening1AtZ), R_MOD)
accumulator := mulmod(accumulator, mload(STATE_ALPHA_SLOT), R_MOD)
// += alpha^2 * (b(z)^2 - c(z))
intermediateValue := mulmod(stateOpening1AtZ, stateOpening1AtZ, R_MOD)
intermediateValue := addmod(intermediateValue, sub(R_MOD, stateOpening2AtZ), R_MOD)
intermediateValue := mulmod(intermediateValue, mload(STATE_POWER_OF_ALPHA_2_SLOT), R_MOD)
accumulator := addmod(accumulator, intermediateValue, R_MOD)
// += alpha^3 * (c(z) * a(z) - d(z))
intermediateValue := mulmod(stateOpening2AtZ, stateOpening0AtZ, R_MOD)
intermediateValue := addmod(intermediateValue, sub(R_MOD, stateOpening3AtZ), R_MOD)
intermediateValue := mulmod(intermediateValue, mload(STATE_POWER_OF_ALPHA_3_SLOT), R_MOD)
accumulator := addmod(accumulator, intermediateValue, R_MOD)

// *= v * [custom_gate_selector]
accumulator := mulmod(accumulator, mload(STATE_V_SLOT), R_MOD)
pointMulAndAddIntoDest(VK_GATE_SELECTORS_1_X_SLOT, accumulator, dest)
```

Security relevant decisions are done by developers

- How to encode data?
- Does encoding matter?
- How to implement group/field operations?
- What to do if prover sends garbage?
- Is it always safe to abort?
- Optimizations of code
- …

Challenge

How do we prove that YUL verifier corresponds to “LaTeX verifier”?

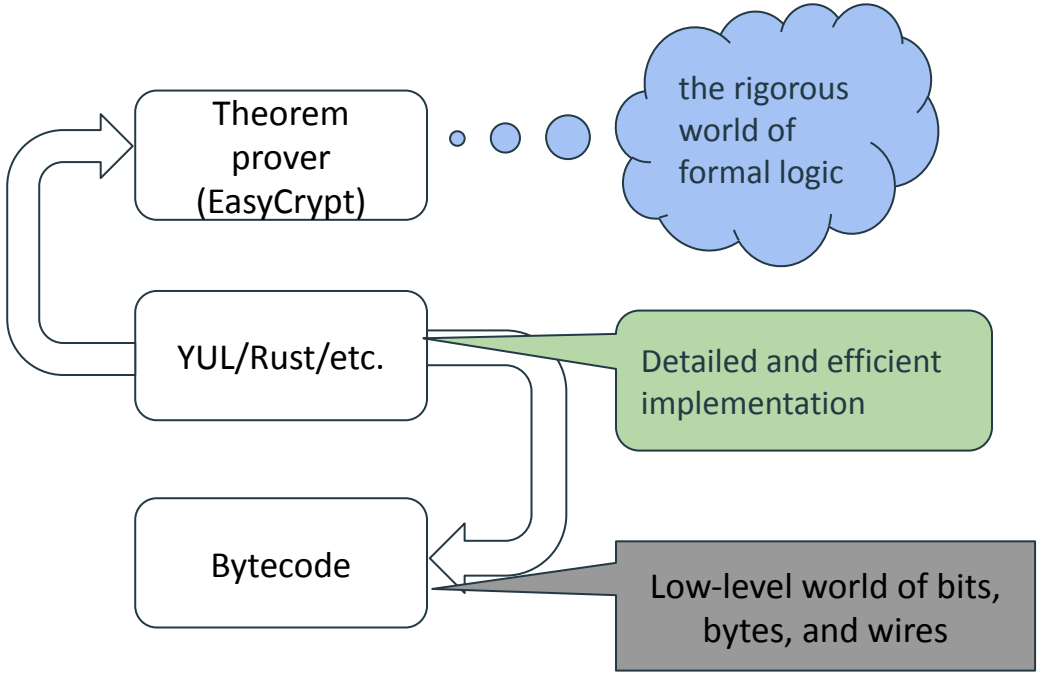
EasyCrypt theorem prover

- EasyCrypt is designed to handle cryptographic proofs (in computational model).
- Incorporates the following specialized program logics:
  - hoare logic;
  - probabilistic hoare logic;
  - probabilistic relational hoare logic;
  - higher-order ambient classical logic.
- Specify syntax and security models for cryptographic constructions.
- Specify cryptographic assumptions.
- Prove correctness (usually easy) and security (always hard).
- EasyCrypt can be connected to other languages and toolchains (e.g., Rust, Jasmin).

Challenge (refined)

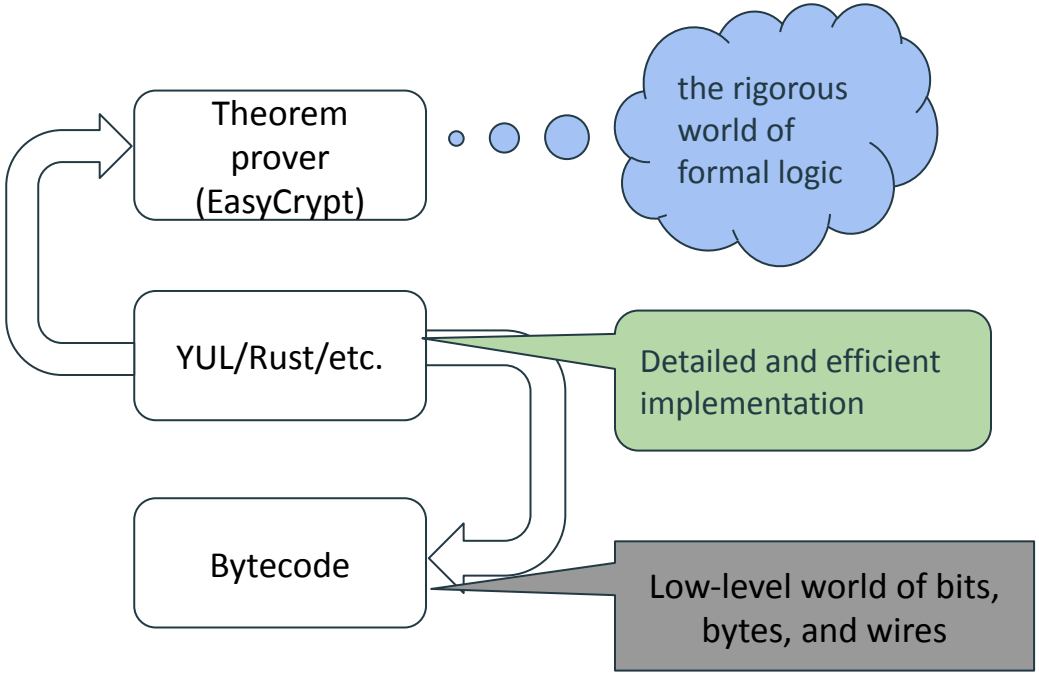
How to prove in EasyCrypt that “LaTeX verifier” and YUL verifier correspond?

Semantics preserving code extraction



Semantics preserving code extraction

1. Develop abstract protocol in the theorem prover.
2. Formally prove that the protocol is correct and secure.
3. Develop efficient implementation in YUL/Rust/etc.
4. Extract implementation to the theorem prover.
5. Carry over properties from abstract protocol to the extracted code.



Formal end-to-end verification

$$e([W_{\mathfrak{z}}]_1 + u \cdot [W_{\mathfrak{z}\omega}]_1, [x]_2) \stackrel{?}{=} e(\mathfrak{z} \cdot [W_{\mathfrak{z}}]_1 + u\mathfrak{z}\omega \cdot [W_{\mathfrak{z}\omega}]_1 + [F]_1 - [E]_1, [1]_2)$$

```
// += alpha^3 * (c(z) * a(z) - d(z))
intermediateValue := mulmod(stateOpening2AtZ, stateOpening0AtZ, R_MOD)
intermediateValue := addmod(intermediateValue, sub(R_MOD, stateOpening3AtZ), R_MOD)
intermediateValue := mulmod(intermediateValue, mload(STATE_POWER_OF_ALPHA_3_SLOT), R_MOD)
accumulator := addmod(accumulator, intermediateValue, R_MOD)
```

Work in progress

- Transpiler from subset of YUL to EasyCrypt.
- Proving that YUL code is a refinement of the high-level specification.
- Note that same technique could be applied to any “side-effect free” YUL code (e.g., precompiled ECC algorithms).

Thank you!