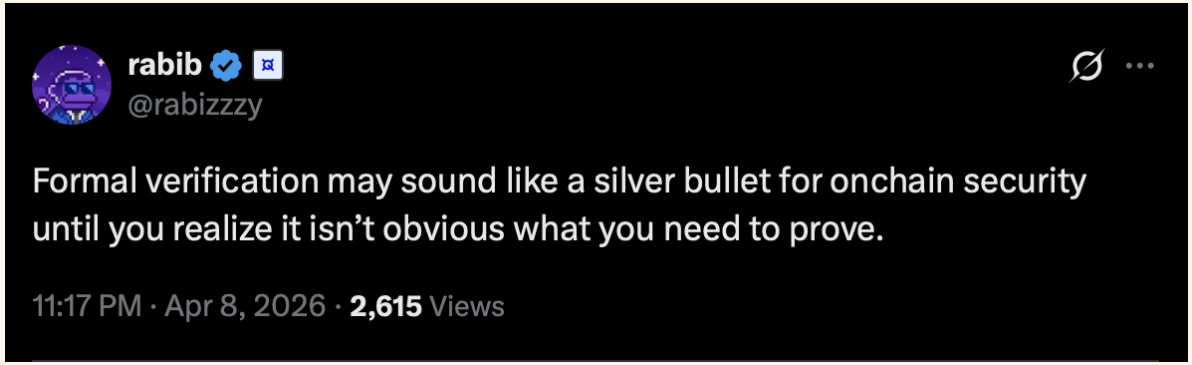
Correct and Computable Specifications in Lean

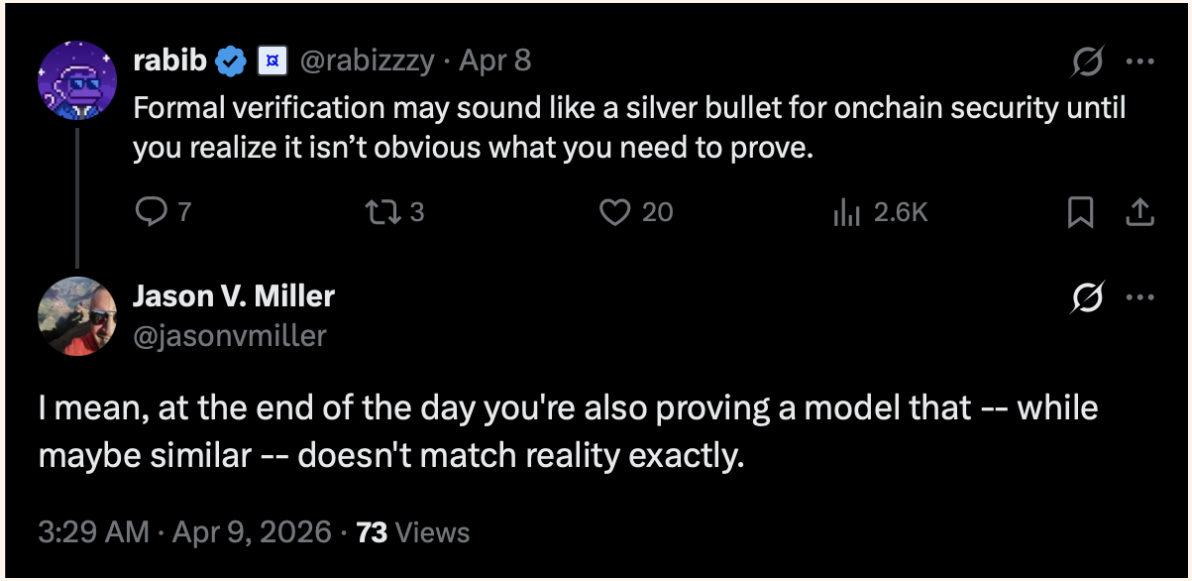
Derek Sorensen
Ethereum Foundation

Software Verification in Lean (SViL26)

Meanwhile on 𝕏 ...



Meanwhile on 𝕏 ...



rabib @rabizzzy · Apr 8

Formal verification may sound like a silver bullet for onchain security until you realize it isn't obvious what you need to prove.

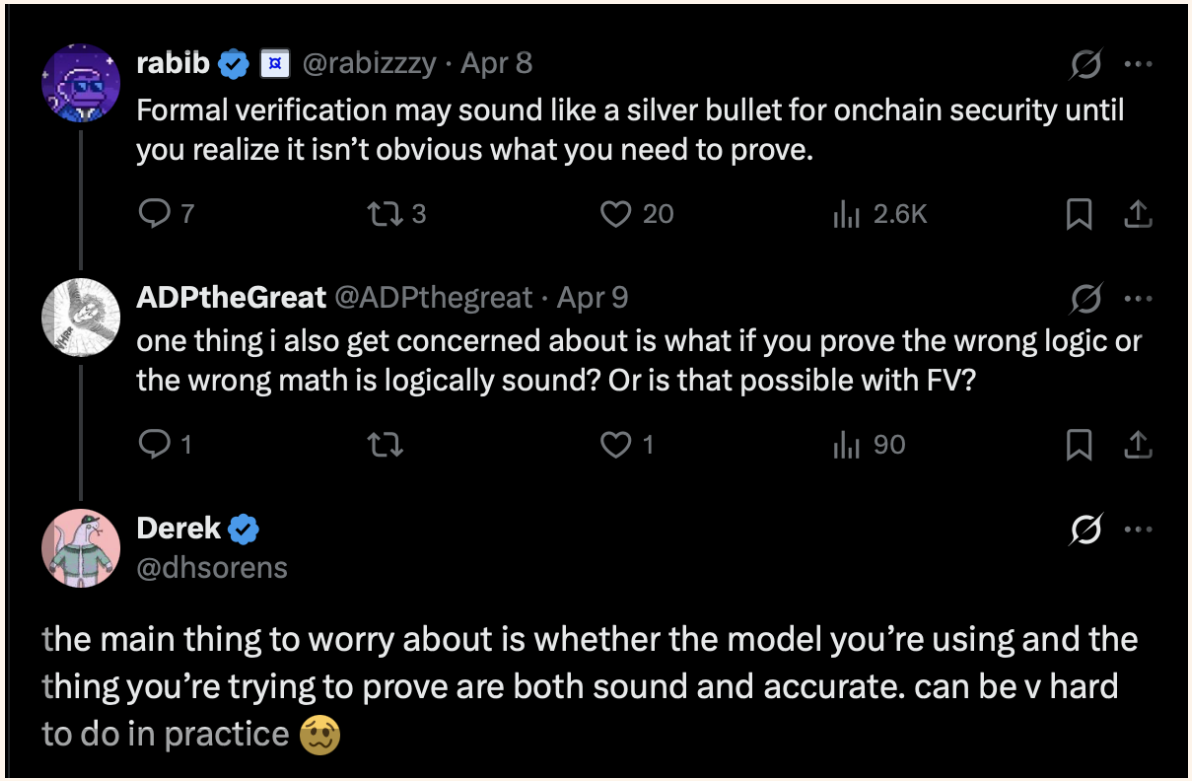
7 3 20 2.6K

Jason V. Miller
@jasonvmiller

I mean, at the end of the day you're also proving a model that -- while maybe similar -- doesn't match reality exactly.

3:29 AM · Apr 9, 2026 · **73** Views

Meanwhile on 𝕏 ...



Objections to Formal Verification

Common objections to formal verification include:

- What about the compiler? Transpiler?
- Too abstract? Bugs in your model?
- How do you know what to prove? Can we trust your spec?
- ...

Objections to Formal Verification

Common objections to formal verification include:

- What about the compiler? Transpiler?
- Too abstract? Bugs in your model?
- How do you know what to prove? Can we trust your spec?
- ...

Asking: What can invalidate the formal proof?

Objections to Formal Verification

Aking: What can invalidate the formal proof?

In more rigorous terms ...

- What is the verification boundary?
- What is the trusted computing base?
- What assumptions are implicit in the model or the specification?
- What is meant by *proof*?

CompCert and its Trusted Code Base

CompCert – A Formally Verified Optimizing Compiler

Xavier Leroy[1] Sandrine Blazy[2] Daniel Kästner[3] Bernhard Schommer[3]
Markus Pister[3] Christian Ferdinand[3]

1: Inria Paris-Rocquencourt, domaine de Voluceau, 78153 Le Chesnay, France
2: University of Rennes 1 - IRISA, campus de Beaulieu, 35041 Rennes, France
3: AbsInt Angewandte Informatik GmbH. Science Park 1, D-66123 Saarbrücken, Germany

Abstract

CompCert is the first commercially available optimizing compiler that is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation. The executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This article gives an overview of the design of CompCert and its proof concept and then focuses on aspects relevant for industrial application. We briefly summarize practical experience and give an overview of recent CompCert development aiming at industrial usage. CompCert's intended use is the compilation of life-critical and mission-critical software meeting high levels of assurance. In this context tool qualification is of paramount importance. We summarize the confidence argument of CompCert and give an overview of relevant qualification strategies.

1 Introduction

Modern compilers are highly complex software systems that try to find a balance between various conflicting goals, like minimal size or minimal execution time of the generated code, maximum compilation speed, maximum however, does typically not include systematic checks for them. When they occur in the field, they can be hard to isolate and to fix.

Whereas in non-critical software functional software bugs tend to have bigger impact than miscompilation errors, the importance of the latter dramatically increases in safety-critical systems. Contemporary safety standards such as DO-178B/C, ISO-26262, or IEC-61508 require to identify potential functional and non-functional hazards and to demonstrate that the software does not violate the relevant safety goals. Many verification activities are performed at the architecture, model, or source code level, but all properties demonstrated there may not be satisfied at the executable code level when miscompilation happens. This is not only true for source code review but also for formal, tool-assisted verification methods such as static analyzers, deductive verifiers, and model checkers. Moreover, properties asserted by the operating system may be violated when its binary code contains wrong-code errors induced when compiling the OS. In consequence, miscompilation is a non-negligible risk that must be addressed by additional, difficult and costly verification activities such as more testing and more code reviews at the generated assembly code level.

The first attempts to formally prove the correctness

CompCert — A Formally Verified Optimizing Compiler

2016

Check for updates

The Trusted Computing Base
of the CompCert Verified Compiler*

David Monniaux and Sylvain Boulmé

Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag
{David.Monniaux,Sylvain.Boulme}@univ-grenoble-alpes.fr

Abstract. CompCert is the first realistic formally verified compiler: it provides a machine-checked mathematical proof that the code it generates matches the source code. Yet, there could be loopholes in this approach. We comprehensively analyze aspects of CompCert where errors could lead to incorrect code being generated. Possible issues range from the modeling of the source and the target languages to some techniques used to call external algorithms from within the compiler.

Keywords: Formally Verified Software · The Coq Proof Assistant

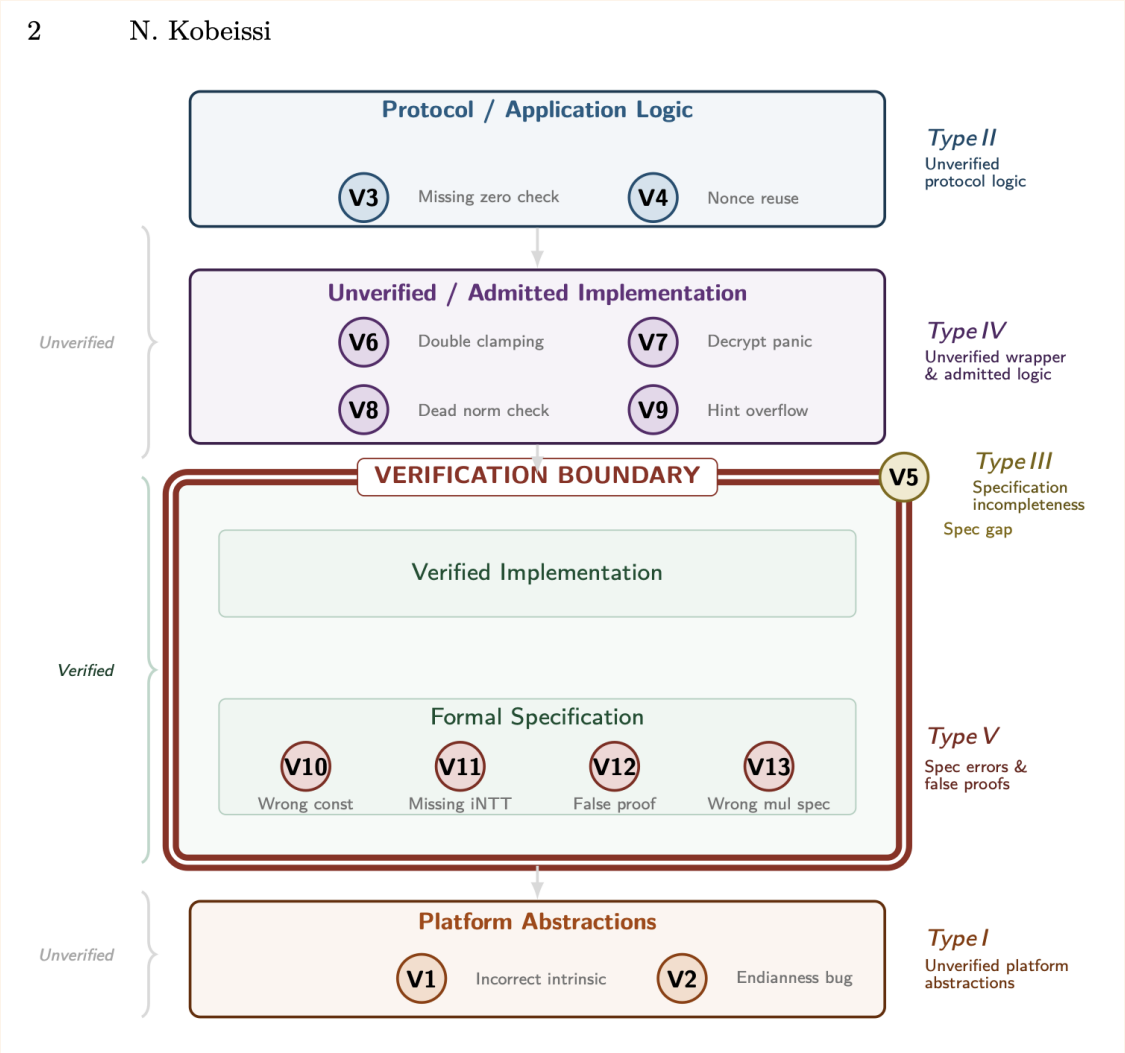
1 Introduction

CompCert [35,34,36] is a formally verified compiler for a large subset of the C99 language (extended with some C11 features): there is a proof, checked by a proof assistant, that if the compiler succeeded in compiling a C program and that program executes with no undefined behavior, then the assembly code produced executes correctly with the same observable behavior. Yet, this impressive claim

The Trusted Computing Base of the CompCert Verified Compiler

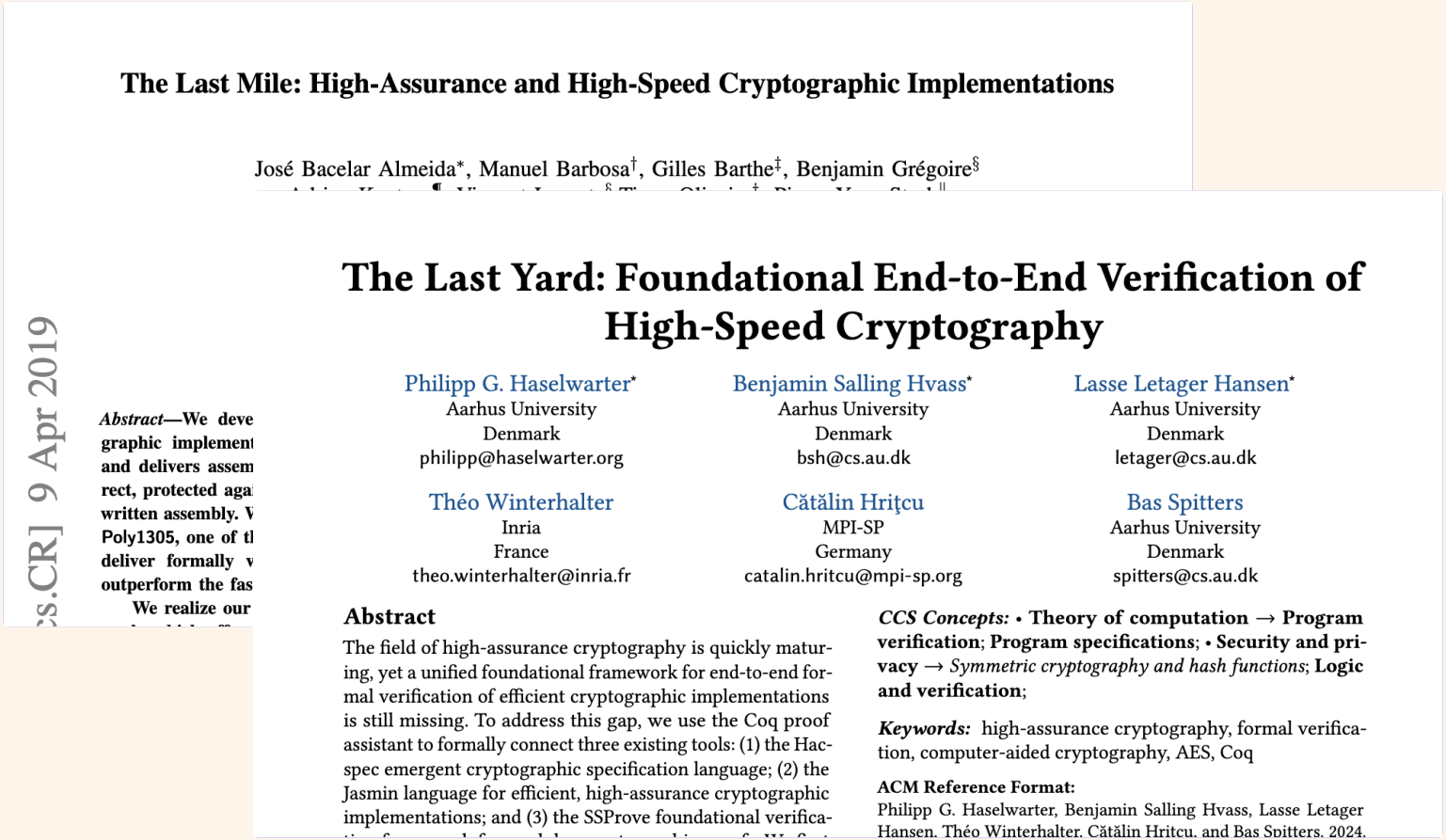
2022

Limitations of Formal Verification



Nadim Kobeissi. *Verification Theatre: False Assurance in Formally Verified Cryptographic Libraries.* eprint.iacr.org/2026/192.pdf

The last ...



The Last Mile: High-Assurance and High-Speed Cryptographic Implementations

José Bacelar Almeida*, Manuel Barbosa†, Gilles Barthe‡, Benjamin Grégoire§

Abstract—We deve
graphic implement
and delivers assem
rect, protected aga
written assembly. V
Poly1305, one of th
deliver formally v
outperform the fas
We realize our

The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography

Philipp G. Haselwarter*
Aarhus University
Denmark
philipp@haselwarter.org

Benjamin Salling Hvass*
Aarhus University
Denmark
bsh@cs.au.dk

Lasse Letager Hansen*
Aarhus University
Denmark
letager@cs.au.dk

Théo Winterhalter
Inria
France
theo.winterhalter@inria.fr

Cătălin Hriţcu
MPI-SP
Germany
catalin.hritcu@mpi-sp.org

Bas Spitters
Aarhus University
Denmark
spitters@cs.au.dk

Abstract

The field of high-assurance cryptography is quickly maturing, yet a unified foundational framework for end-to-end formal verification of efficient cryptographic implementations is still missing. To address this gap, we use the Coq proof assistant to formally connect three existing tools: (1) the Hacspec emergent cryptographic specification language; (2) the Jasmin language for efficient, high-assurance cryptographic implementations; and (3) the SSProve foundational verifica-

CCS Concepts: • **Theory of computation** → **Program verification**; **Program specifications**; • **Security and privacy** → *Symmetric cryptography and hash functions*; **Logic and verification**;

Keywords: high-assurance cryptography, formal verification, computer-aided cryptography, AES, Coq

ACM Reference Format:
Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. 2024.

Correct and Computable Specifications

1. What of these gaps exist at the level of *specification*?
2. How can we bridge those gaps?

(In)Correct Specification—Smart Contracts

A **smart contract** is a program that executes on a blockchain.

Key components:

1. Entry points
2. Storage
3. Native token balance

Key features:

1. Typically manages money
2. Implement a market or game
3. Ideal for FV

(In)Correct Specification—Smart Contracts

1. Entrypoints:
   - `swap`
   - `add_liquidity`
   - `remove_liquidity`
2. Storage:
   - `lp_token_balance`
   - `token_x_balance`
   - `token_y_balance`
3. Native token balance:
   - Possibly nonzero (treasury)

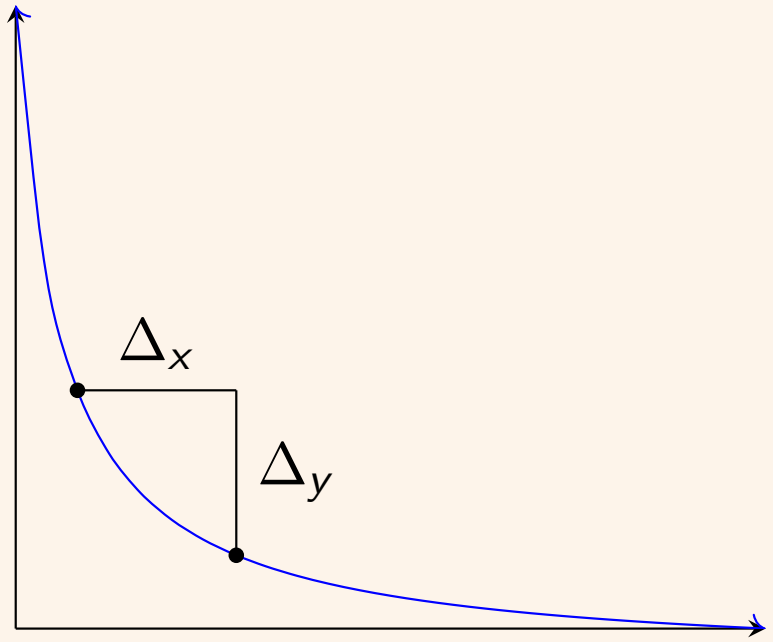


Figure: A trade of Δ_x for Δ_y along the indifference curve $xy = k$.

(In)Correct Specification—Smart Contracts

- **Aave** — pooled lending, variable rates, flash loans
- **Uniswap** — AMM swaps, liquidity positions, fees
- **Maker** — collateralized stablecoin (DAI)
- **Lido** — liquid staking and validator delegation
- **Curve** — stablecoin- and peg-optimized pools
- **dYdX** — perpetuals and margin trading
- **Chainlink** — decentralized price and data feeds
- **Bridges** — cross-chain asset locks and messaging

Not easy to specify formally "all the way up."

(In)Correct Specification—Smart Contracts

Exploits where contracts behaved "as written"—but poorly specified

Game theory / markets

- **Beanstalk** (2022) — flash-loaned governance supermajority; emergency path; ∼**$77M**
- **Spartan / Pancake Bunny** (2021) — spot/oracle prices inside logic manipulated via flash liquidity; ∼**$30M / ∼$45M**
- **Mango Markets** (2022) — large positions and oracle marks within stated rules; ∼**$116M**
- **Compound** (Jul 2024) — token-weighted governance moved treasury; ∼**$25M**
- **Resupply** (Jun 2025) — lending pair exchange rate / empty-vault accounting bypassed solvency intuition; ∼**$9.6M**

Trust model / upgrades

- **Nomad** (2022) — upgrade made null address a trusted root; replicated drains ∼**$190M**
- **Uranium / NowSwap** (2021) — pricing constant `k` inconsistent across upgrade; ∼**$50M / ∼$1M**
- **Prisma** (Mar 2024) — migration + flash-loan callback with weak checks on delegated trove ops; ∼**$12M**

(In)Correct Specification—Smart Contracts

Some solutions:

- High-level, game-theoretic reasoning (in some model)
- Certora: Heuristics at the Solidity level
- ConCert / FinCert
- ... (TBD)

Largely an open problem for formal verification ..

(In)Correct Specification—SNARKS

Succinct Non-interactive ARguments of Knowledge (SNARKS):

- Relatively new in theory and production.
- The specs are continually evolving, in the literature and in practice.
- These are rapidly becoming essential scaling infrastructure for Ethereum.

(In)Correct Specification—SNARKS

ArkLib sits in a stack of formal libraries

Computable polynomials and finite fields.

SNARKs and proof systems.

Oracle computations, underlies core IOR framework of ArkLib.

CompPoly → ArkLib ← VCV-io

(In)Correct Specification—SNARKS

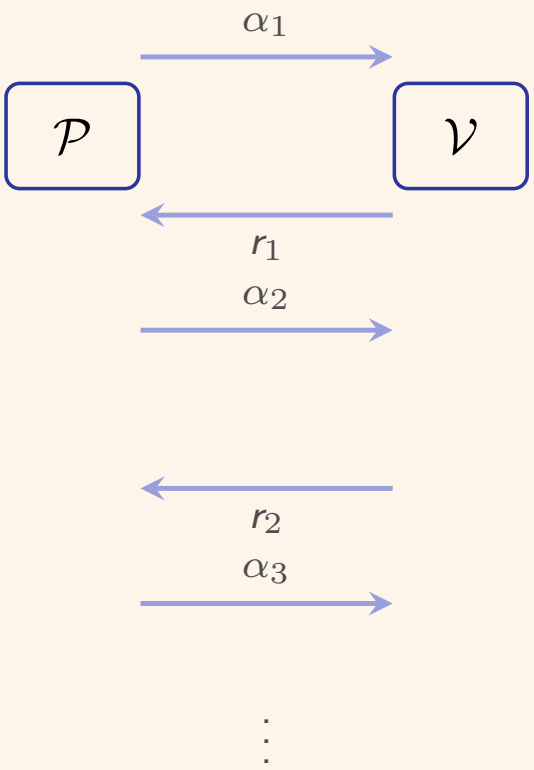
ArkLib: papers, deployments, and lore rarely match verbatim

- **IOPs** — `OracleReduction` is broader than the vector-IOP story; paper is lineage, not a line-by-line spec.
- **FRI** — oracle reduction with explicit round composition and parameters; aims at generalizations, not a frozen "classical FRI" write-up alone.
- **Sum-check** — lives inside the oracle-reduction stack (composed rounds), not as a standalone classical IP object; external formal-verification benchmarks (e.g. BBS24) are comparison context, not API-identical.
- **WHIR** — definitions lifted into reusable abstractions; ePrint vs published variants matter for exact numbering and wording.
- **Polishchuk–Spielman** — historical PS94/Spi95 statements are flawed; ArkLib follows the corrected lemma (BCIKS20 lineage), split across helper modules rather than mirroring the originals verbatim.

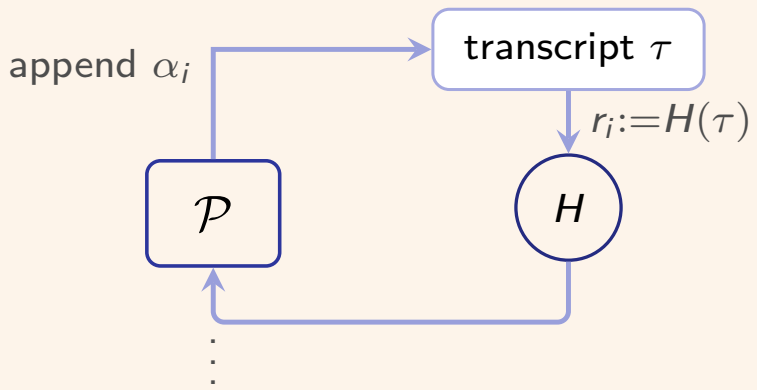
(In)Correct Specification—SNARKS

Fiat–Shamir: specifying the challenge

Interactive



Fiat–Shamir



(In)Correct Specification

Some observations ...

1. **Specs are code** — computability matters
2. **Extensible** — in the theory and syntax
3. **Metaprogramming** — code, specify, verify

Computable Specifications

the Arklib specification is a full-scale codebase

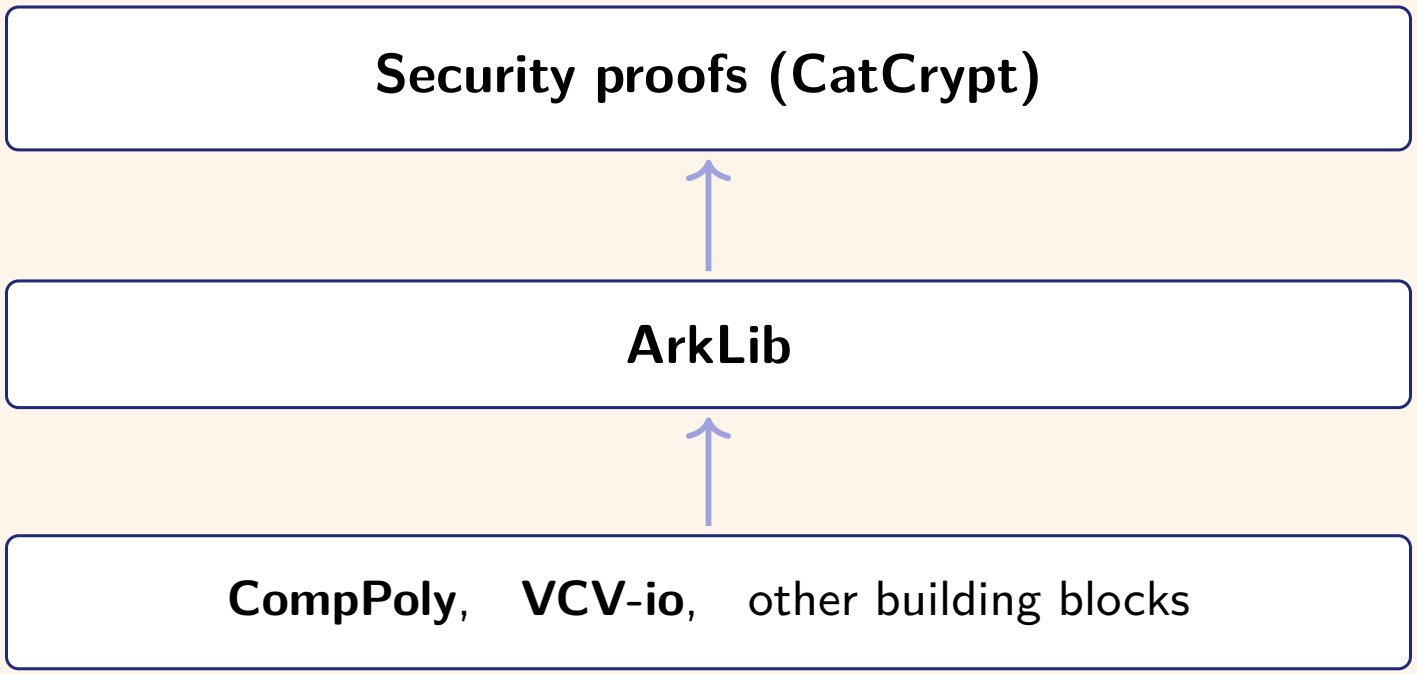
ArkLib

- Sum-check
- Coding theory
- FRI
- Fiat–Shamir

Specifications are ***programs*** + ***proofs***.

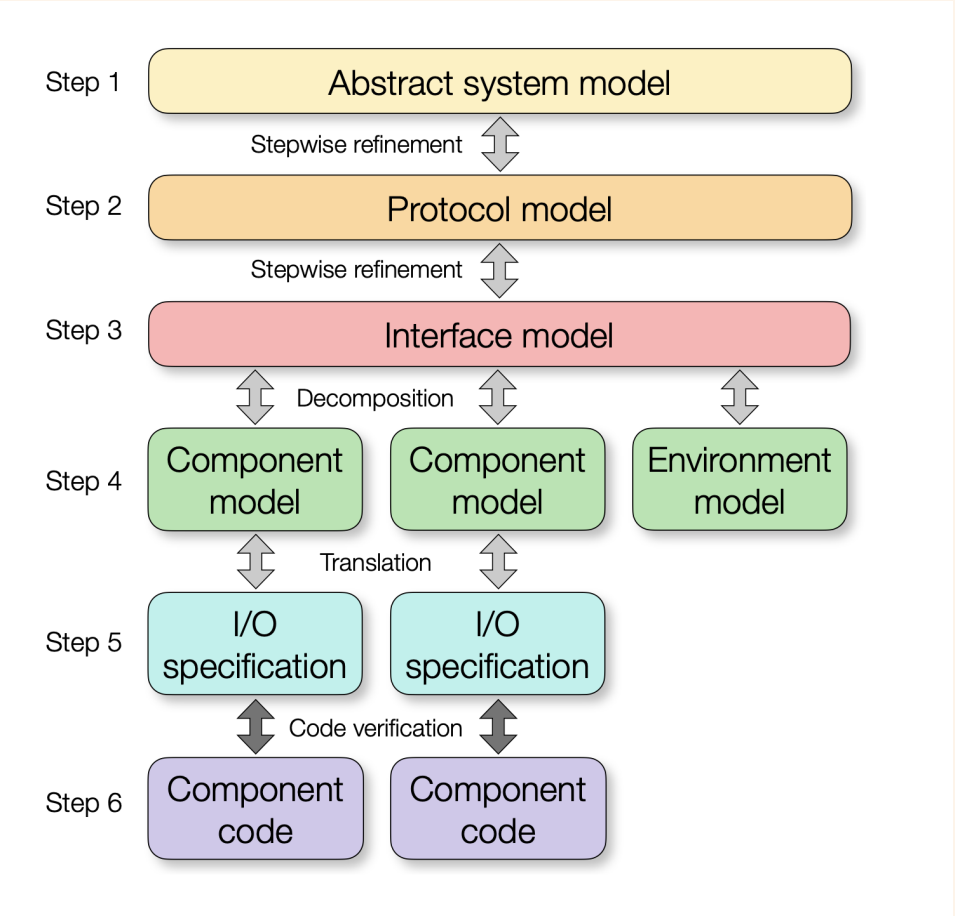
Extensibility in Theory and in Syntax

Arklib, at many levels



Extensibility in Theory and in Syntax

Igloo Systems Specification Framework



Sprenger *et al. Igloo:Soundly LinkingCompositional Refinementand Separation Logic forDistributed System Verification*
`https://dl.acm.org/doi/pdf/10.1145/3428220`

Metaprogramming in Lean

Wouldn't it be nice if we could do it all in one place?

- Lean has a **unique metaprogramming** setup where you can code, specify it, verify it all in one spot. (Runtime in TCB)
- evm-asm, Arklib, Clean, etc (and compiler work)

❤️AI and Formal Specification ❤️

AI is great at ..

- Proving theorems (leaves time for attention to specs)
- Writing formal specifications
- Matching code to spec (and refactoring)

But some caution ...

- Specs are hard to verify as correct, get into the details
- AI can be a great companion, but the same principles apply

Correct and Computable Specifications in Lean

Takeaways

- **Specs are code too.** Specifications are code (and code is specification); computability can help root out bugs.
- **Extensible (theory and syntax).** Lean's extensibility in theory and syntax allows us to build theories, DSLs, and more in the same environment as specifications.
- **Code, specify, and verify in Lean** Lean is extensible and may be the ideal environment from which to code, specify, and verify some real-world software.
- **AI is good, but verify.** AI will still write bad specs, so the same principles apply.

questions / discussion