Maggie Simmons
m8simmon@uwaterloo.ca

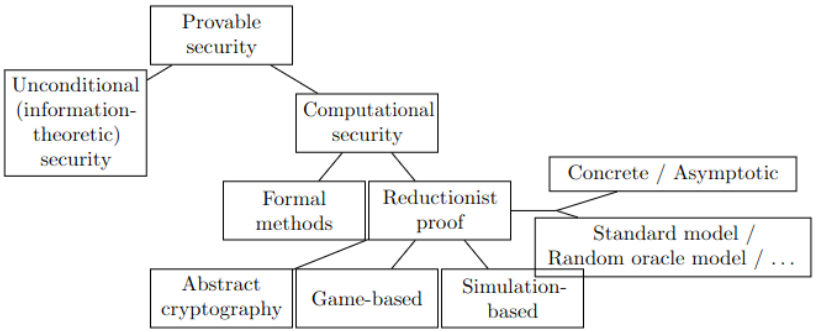
Cheriton School of Computer Science
University of Waterloo

ShannonProver: Towards Automating Formal Cryptographic Proofs

by Ma et al., 2026

Provable Security and Computer-Aided Cryptography

Provable Security aims to show mathematically that a specific property of a scheme cannot be broken by a class of attackers.



Computer-Aided Cryptography

- Cryptographers "generate more proofs than we carefully verify"
- Reductionist proofs and formal methods are starting to merge with computer-verifiable reductionist proofs in EasyCrypt
- Machine-checked proofs offer a way to scale proof verification.
- However, the bottleneck shifts to producing proof scripts

EasyCrypt

What is EasyCrypt?

- Cryptographic proof assistant based on ambient and Hoare logics to prove security properties of cryptographic protocols
- Most cryptographic proofs are game-based, implying that we need the ability to work with pairs of programs.
- EasyCrypt also relies on external SMT solvers for some automation.

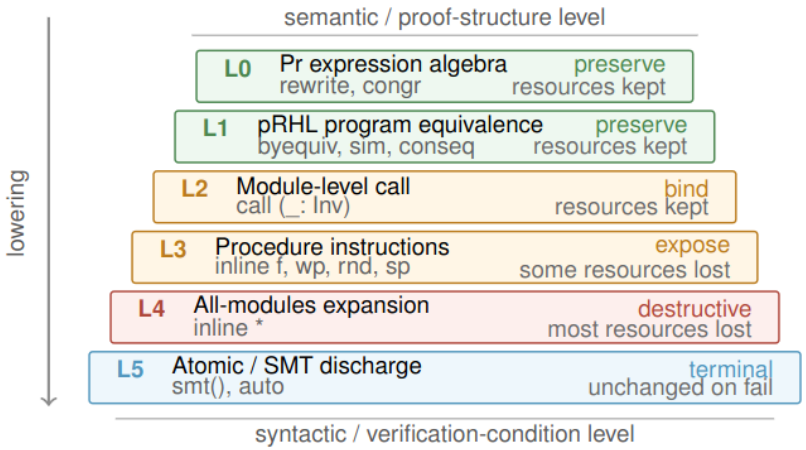
How to Write EasyCrypt Proof Scripts?

- The cryptographer writes games as imperative, code-like programs.
- The proof assistant evaluates the logic line-by-line.
- The user applies logical "tactics" to demonstrate how the internal state of Game A relates to Game B

Can AI agents accelerate writing these proofs?

Observations in Agentic Proving

Writing an EasyCrypt proof script is more than choosing the next tactic that can change the current goal.



Unconscious Lowering

- The agent can mistakenly interpret a local but unhelpful goal change as real proof progress.
- Lowering often necessary, but interface gives no state-aware information to decide when.

Agent Flinch

- Agent can identify a viable high-level proof plan but abandon it when trying to implement plan with a well-typed command.

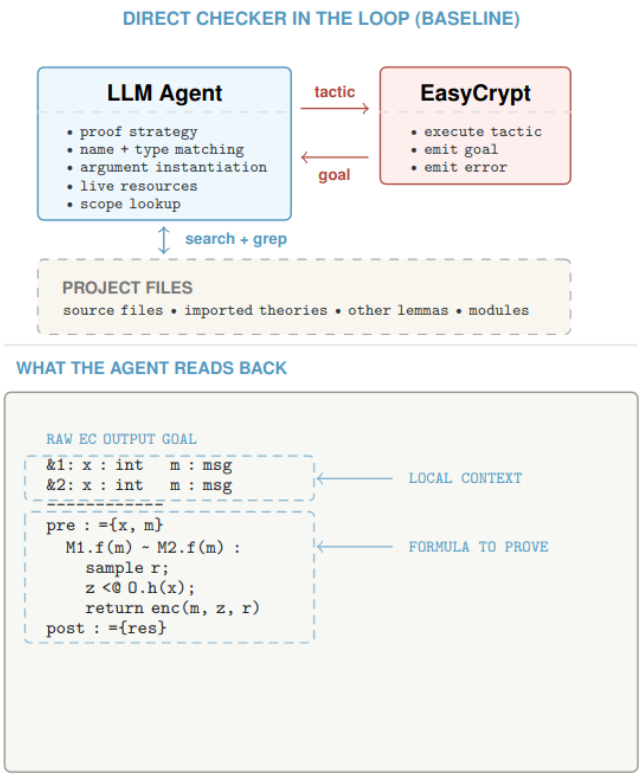
Error Reporting

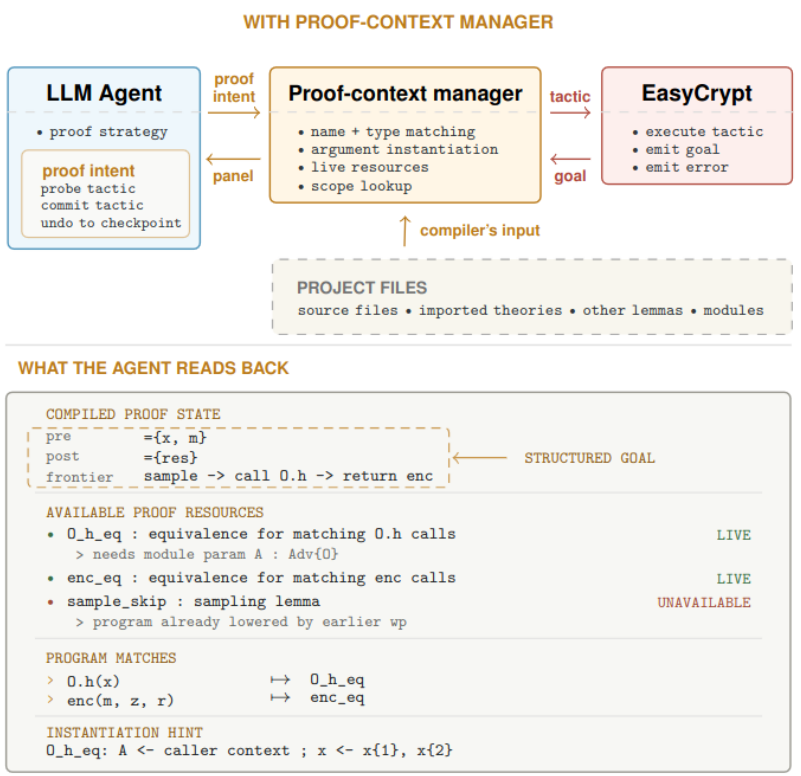
- Broader issue of semantic mislocalization.

ShannonProver Workflow

Failure modes point to an interface problem :

- Flinches: right high-level idea, but need chain of mechanical steps.
- Semantic mislocalization: agent should be given matches derived from the current proof state.
- Unconscious lowering: interface should make resource applicability visible.





Layers 1 and 2

Layer 1: State Projection

- Where am I?
- Front end, turns verifier output, session events, and tactic history into one current snapshot.
- This step only supplies the stable state boundary.

Layer 2: Proof State IR

- What is the current goal?
- Gives snapshot a typed semantic view: the current proof layer, the visible program structure, and the proof resources relevant at that layer.
- Three levels of information: goal, layer-specific, and candidate resources.

Layers 3 and 4

Layer 3: Resource Liveness and Program Frontier

- What remains usable?
- Refines view by tracking which resources are live, blocked, or stale at the current cursor and frontier.
- Output: which resources are live, blocked, stale, what frontier is currently exposed, and which local moves would preserve or destroy useful proof structure

Layer 4: Action Surface

- What can I try safely?
- Turns the resulting state information into targeted inspection and probing actions, previews, missing bindings.
- The eventual layer exposed to the agent - gives agent state-aware entry points into the proof context.
- Entries types: resources, bindings, inspection/probe tools, neutral diagnostics

Proof Trees

Proof construction is rarely a straight line

- There may be several structurally different ways to proceed
- A step that looks reasonable locally may later lead into a dead end.
- Need a way to explore alternatives, recover from wrong turns, and decide what worth pursuing.

Proof Trees

- Node: Proof state
- Edge: Accepted transition between states
- Path: Sequence of nodes from the root to the current node, representing proof attempts
- Tree: Same state can be continued in more than one way.
- Leaf: Either a closed proof (qed.) or a state that is abandoned because it is a dead end or is pruned by the orchestrator.

Backtracking within an Agent

- At each node, the agent reads the compiled surface.
- Agent can also backtrack to the earlier state, revise the proof from that point, and extend a new edge to a new child.
- Local backtracking not sufficient - search needs a second layer of control beyond the individual agent
- "Orchestrator" to ensure backtrack at correct time, explore several feasible options in tandem.

Scheduling across Agents

The orchestrator is not an AI agent - it is a deterministic algorithm:

- **Spawning Children**: orchestrator may spawn additional children if persistent local failure, structural exploration
- **Pruning**: prunes branches and terminates unproductive agents based on a program gap
- **Negative Memory**: For a given node, records which attempted extension led to a dead end and why

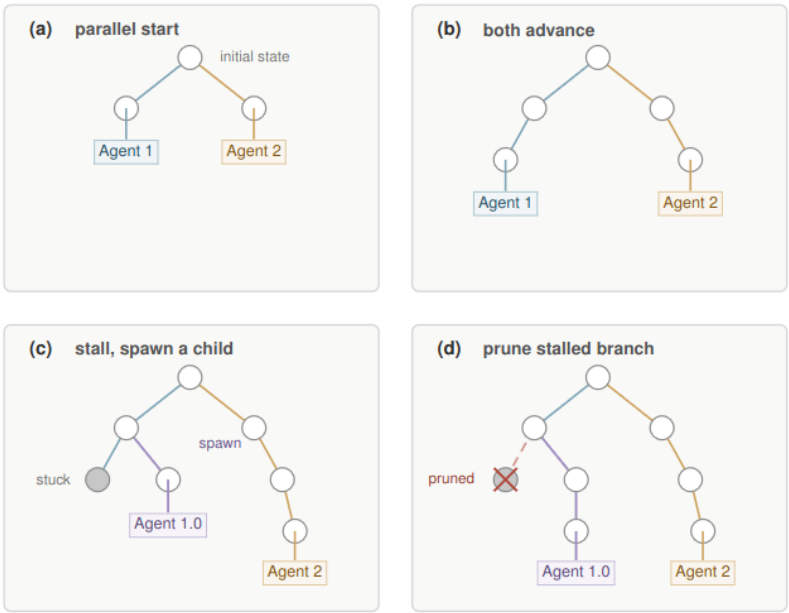


Figure: A proof tree evolving over a search.

Evaluation Metrics

Part One: Controlled Ablations

- Compiler: how much mechanical friction the compiled surface removes from proof search
- Tree policy: how much does multi-agent orchestration accelerate the proof writing relative to a single agent

Part Two: Case Studies

- ChaCha20-Poly1305
- MEE-CBC
- CMAC

Setup

- Two configurations: direct checker-in-the-loop baseline and ShannonProver with Opus 4.8
- Agent is allowed to announce that it gives up at most three times.
- Note: metric measuring whether a lemma is proven or not is too coarse here, new metrics for each ablation

Effects of the Proof-State Compiler

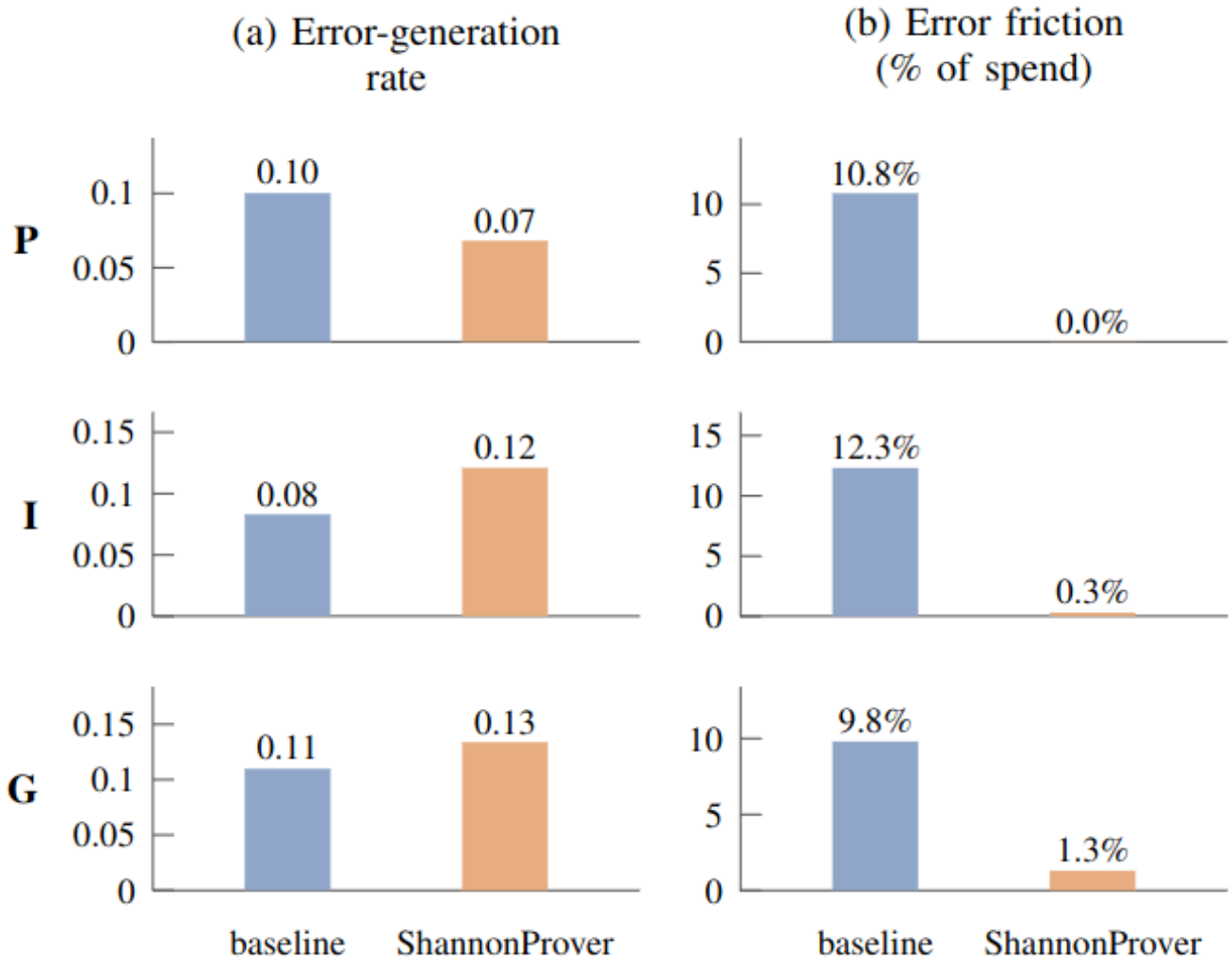


Figure: Across all lemma types, ShannonProver sharply reduces the fraction of spend inside error recovery windows.

Effects of Search Tree

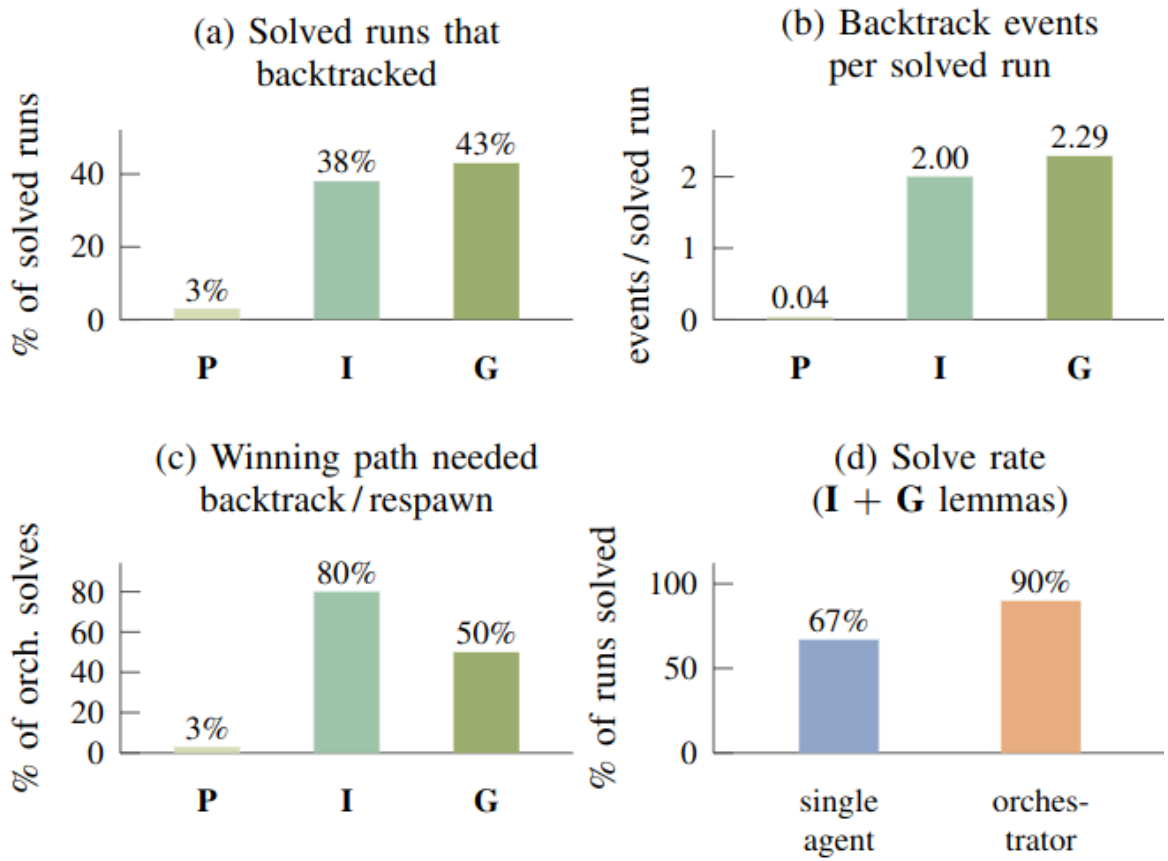


Figure: For the proof types that matter most in cryptographic developments, the successful route is often found through backtracking and diverse path exploration.

Cost of ShannonProver

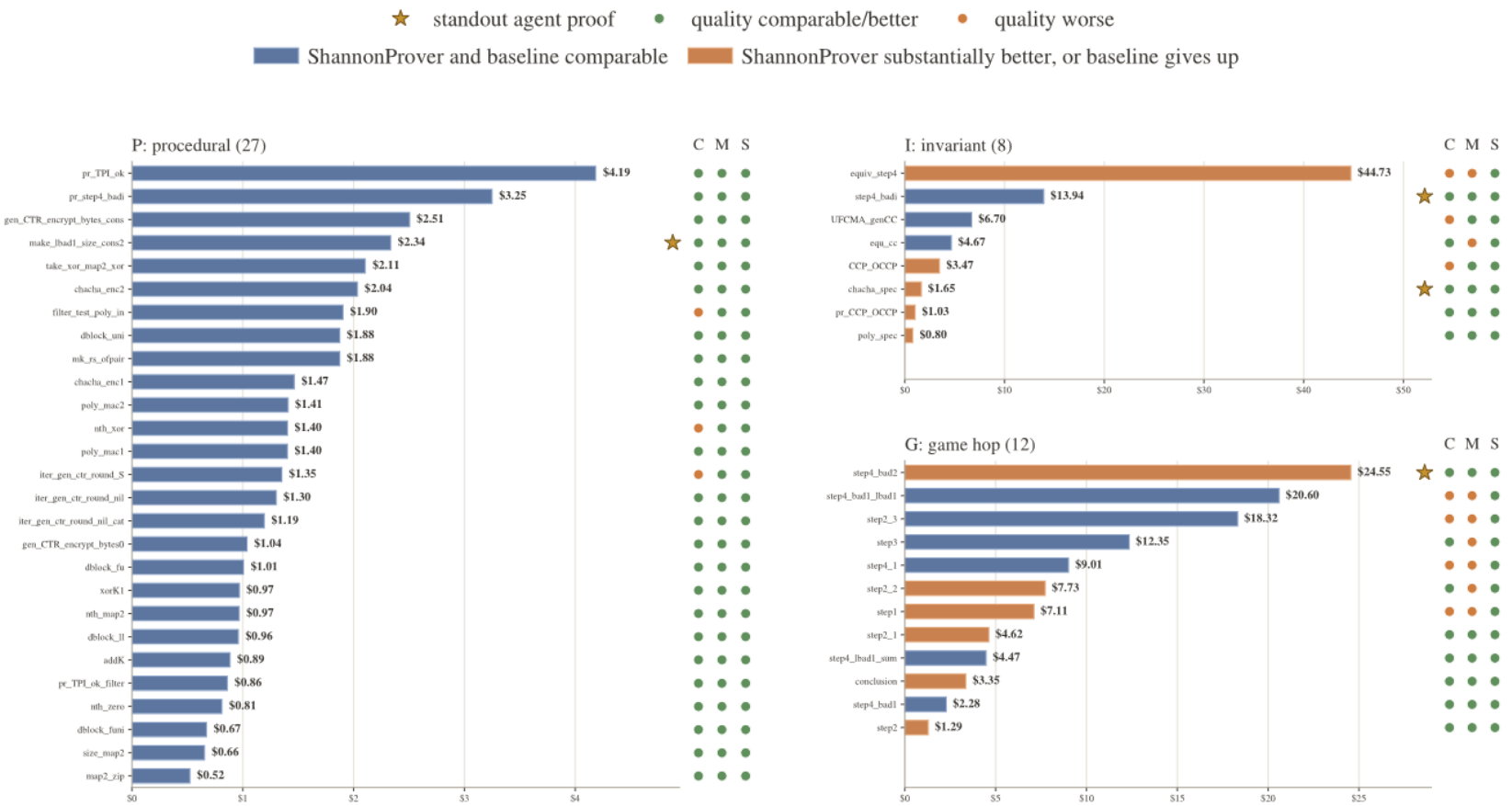
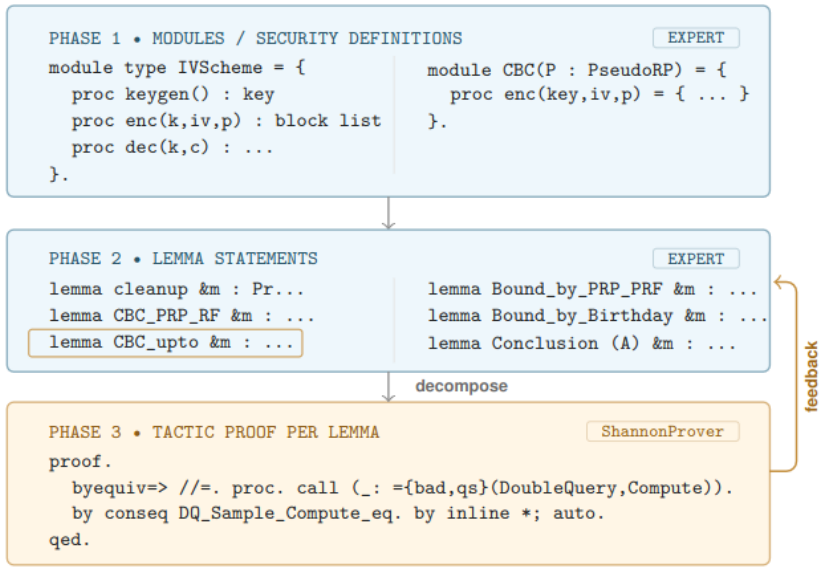


Figure: Per-lemma API cost for ChaCha20-Poly1305 project solved by AI agents, grouped by proof shape (P/I/G). Dots report qualitative proof quality along concision (C), maintainability (M), and semantic proof idiom (S). The star marks lemmas where the agent found a "tesuji" move that makes the proof much shorter than human expert proofs

Conclusion

Takeaways

- Formalizing proofs for large scale cryptographic projects proceeds in three stages, the most time consuming of which can now be done with agents.
- ShannonProver has proven effective for real-world standardized schemes.
- We now have a corpus of 1.6K EasyCrypt lemmas to support systematic evaluation of agents for writing formal proofs.



Discussion Questions

Current AI agents can automate a substantial amount of formal cryptographic proof at reasonable cost when placed behind the right harness... what next?

Discussion Questions

- **Scaling**: Which parts of the proof-state compiler already scale and which need stronger retrieval, caching, or decomposition support?
- **Theorem Decomposition**: How can ShannonProver's system adapt to enable theorem decomposition (as found in other LLM-based theorem proving tools)?
- **Verified Implementations**: How can we extend ShannonProver into the verified implementation setting?
- **Generalization**: Since building the right harness is essentially a compiler problem, can we apply this framing to other formal verification settings like Lean?
- **Cost**: ShannonProver's multi-agent approach is expensive. Are there any other deterministic procedures we can run or state to maintain to keep backtracking and proof search contained?

References I

[1] Yiping Ma et al. *ShannonProver: Towards Automating Formal Cryptographic Proofs*. en. arXiv:2607.02847 [cs.CR]. July 2026. DOI: 10.48550/arXiv.2607.02847. URL: http://arxiv.org/abs/2607.02847 (visited on 07/10/2026).