

Rocq-solid

A story about vericoding

Valentin Bergeron

Thursday, October 1, 2026

Reboot with AI

Follow along



Slides

<https://vbergeron.github.io/rocq-solid/decks/rocq-solid.pdf>



Site

<https://vbergeron.github.io/rocq-solid/>

A story about Jack

A story about Jack

- Full stack developer at a small e-shop startup
- Twelve engineers, one product, a roadmap that doubles every quarter
- Frontend in the morning, database migrations after lunch, on call at night



Claude writes the feature.

Claude writes the tests.

Jack **loves** Claude.

A story about Jack

The past 9 months have been a **blast**!

- Three times as many pull requests merged per week
- Coverage above 90%, and climbing

Jack reads every diff. Well, most of them. *The tests are green anyway.*



A story about Jack

Friday, 10h00

Product asks for **partial refunds**: one button, one click per refund.

Claude does the job from the front to the DB.

The tests are still green.

A story about Jack

Friday, 22h57

A customer notices the refund button **works more than once.**

A story about Jack

Monday, 08h43

€180,000 refunded, on **€40** of orders.

A story about Jack

```
type Order = {  
  id: OrderId;  
  paid: Money;  
  refund?: Refund; // an order is refunded at most once  
};  
  
const refundable = (o: Order) => o.paid - (o.refund?.amount ?? 0);
```

A story about Jack

```
test("a refund never exceeds what was paid", async () => {  
  const refunds = await db.refunds.findMany({ orderId: order.id });  
  for (const refund of refunds) {  
    expect(refund.amount).toBeLessThanOrEqual(order.paid);  
  }  
});
```

A story about Jack

The post-mortem

- Every test was green
- Every pull request was reviewed and approved
- The code did exactly what it said

But nobody wrote down **what must be true**.

It's all about proofs and trust

Yes obviously some guys in the 17th century have thought about it

Rationalists

Descartes, Spinoza, Leibniz

- Knowledge comes from **reason** (innate ideas, deduction)
- Model: **mathematical proof**
- Limit: can pure reason tell us about the world?

Empiricists

Locke, Berkeley, Hume

- Knowledge comes from **experience** (mind as *tabula rasa*)
- Model: **experience and deductions**
- Limit: no amount of observation yields certainty

Collatz: evidence without proof

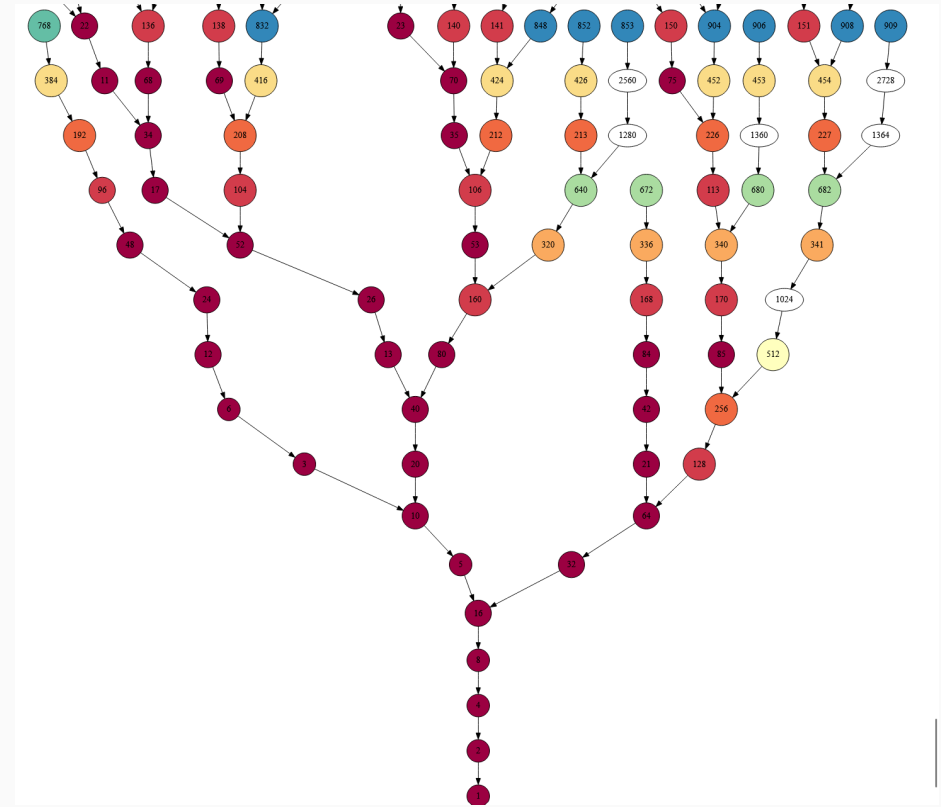
The rule: take any n . If even, $n \rightarrow \frac{n}{2}$; if odd, $n \rightarrow 3n + 1$. Repeat.

Conjecture (Collatz, 1937): you always end up at 1.

$6 \rightarrow 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

- Checked for every n up to $2^{71} \approx 2.36 \times 10^{21}$ (Barina, 2025)
- Still no proof

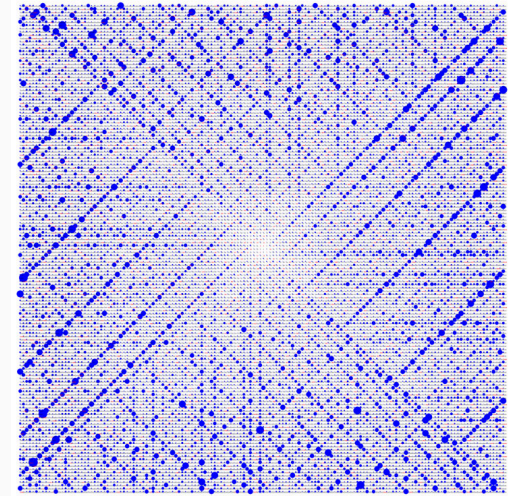
Billions of billions of cases $\neq$ certainty: the next number could be the exception.



Reason alone: truths no experiment can reach

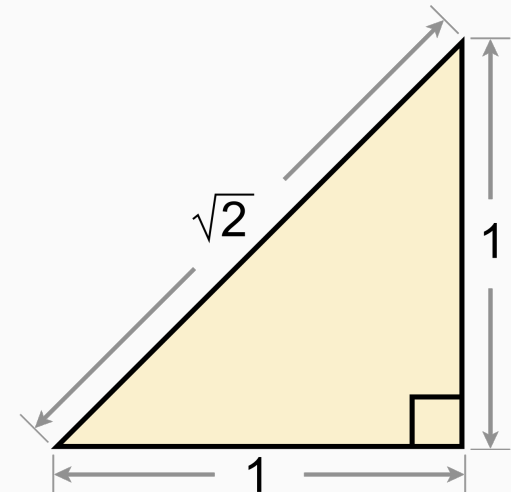
There are infinitely many primes (Euclid, c. 300 BCE)

Suppose the list is finite: $p_1, \dots, p_n$. Then $N = p_1 p_2 \cdots p_n + 1$ leaves remainder 1 when divided by each p_i , so its prime factors are missing from the list. Contradiction.



$\sqrt{2}$ is irrational (Pythagoreans, 5th c. BCE)

Suppose $\sqrt{2} = \frac{p}{q}$ in lowest terms. Then $p^2 = 2q^2$, so p is even: $p = 2k$. Then $q^2 = 2k^2$, so q is even too. Contradiction.



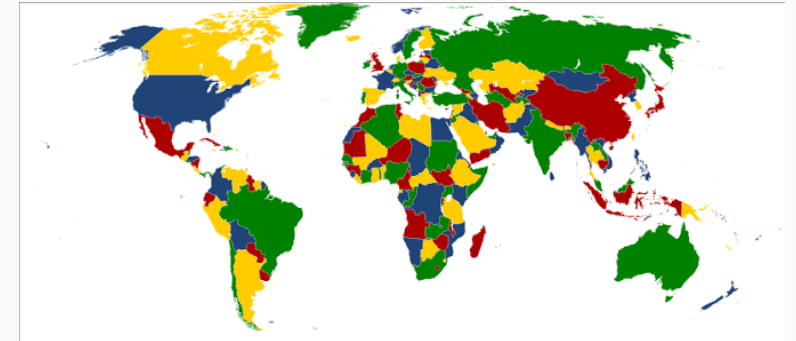
Theorem provers

The Four Colour Theorem: a proof no human can read

Any map can be coloured with 4 colours so that neighbouring regions differ.

- 1879** Kempe publishes a “proof”, accepted for 11 years
- 1890** Heawood finds the flaw: back to square one
- 1976** Appel & Haken: a computer checks 1,900 cases in over 1,000 hours
- 1996** Robertson, Sanders, Seymour & Thomas: a simpler proof, still computer-assisted
- 2005** Gonthier verifies the full proof in the Coq proof assistant

The question: if no human can check it, is it still a proof? Do we *know* the theorem, or do we *trust* the machine?



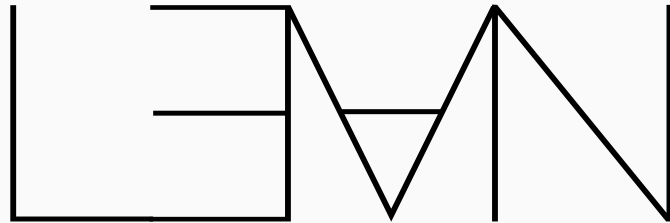
Theorem (Curry 1934–1958, Howard 1969).

Proofs in intuitionistic logic are exactly the programs of the typed λ -calculus: a proposition is provable if and only if the corresponding type has a program.

Checking a proof = checking a program's type.

- 1984** Coquand and Huet start it at Inria
- 1989** First release, named **Coq**
- 2005** **The four colour theorem**, checked
- 2006** CompCert, a C compiler proved correct
- 2013** ACM Software System Award
- 2025** Renamed **Rocq**, after Rocquencourt





Isabelle

An OS kernel, proved

seL4 flew a military helicopter that red teams could not break into

F*

You used it today

HACL*, the proved crypto inside Firefox, Linux and Python

Dafny

At the heart of AWS

The engine that evaluates IAM policies, rewritten and proved

TLA+

Bugs caught before the code

Amazon found subtle design bugs in S3 and DynamoDB

ACL2

The Pentium, never again

AMD proved its floating point division in 1996

Agda

Proofs are programs

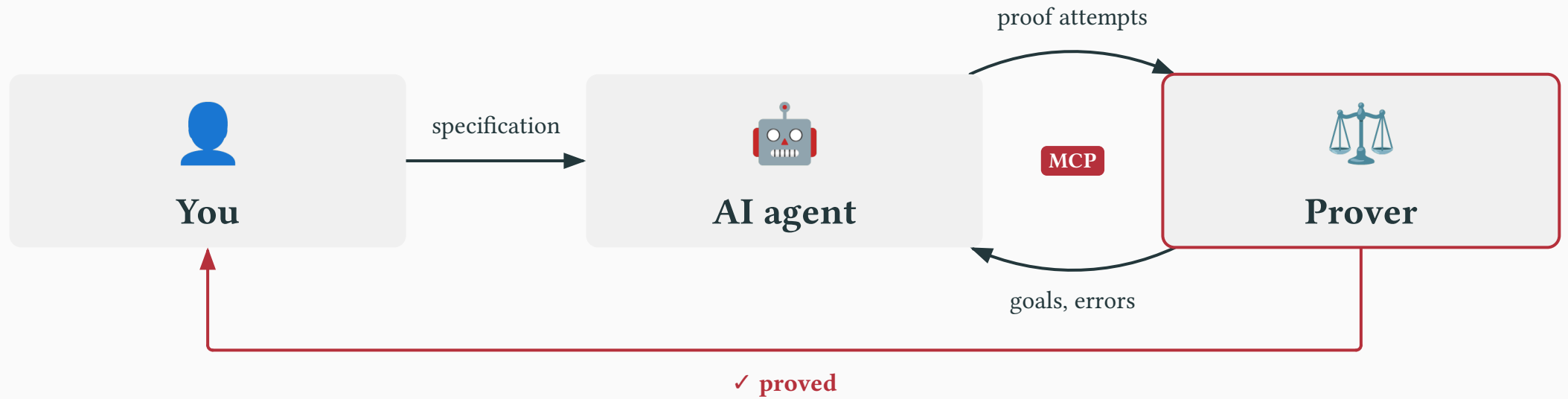
The playground where type theorists try their ideas first

Vericoding

Writing a proof is hard.

Writing **what you want to prove** is much easier.

Who does what



Example: the code

```
Inductive event :=  
| Paid (id amount : nat)  
| Refunded (id amount : nat).  
  
Record order := mk_order {  
  seen : list nat; (* ids already handled *)  
  paid : nat;  
  refunded : nat  
}.
```

```
(* Was this event already handled? *)  
Definition already_seen  
  (o : order) (e : event) : bool := ...  
  
(* Apply one event to an order. *)  
Definition handle  
  (o : order) (e : event) : order := ...  
  
(* The order, rebuilt from its events. *)  
Definition replay  
  (events : list event) : order := ...
```

Example: the specification

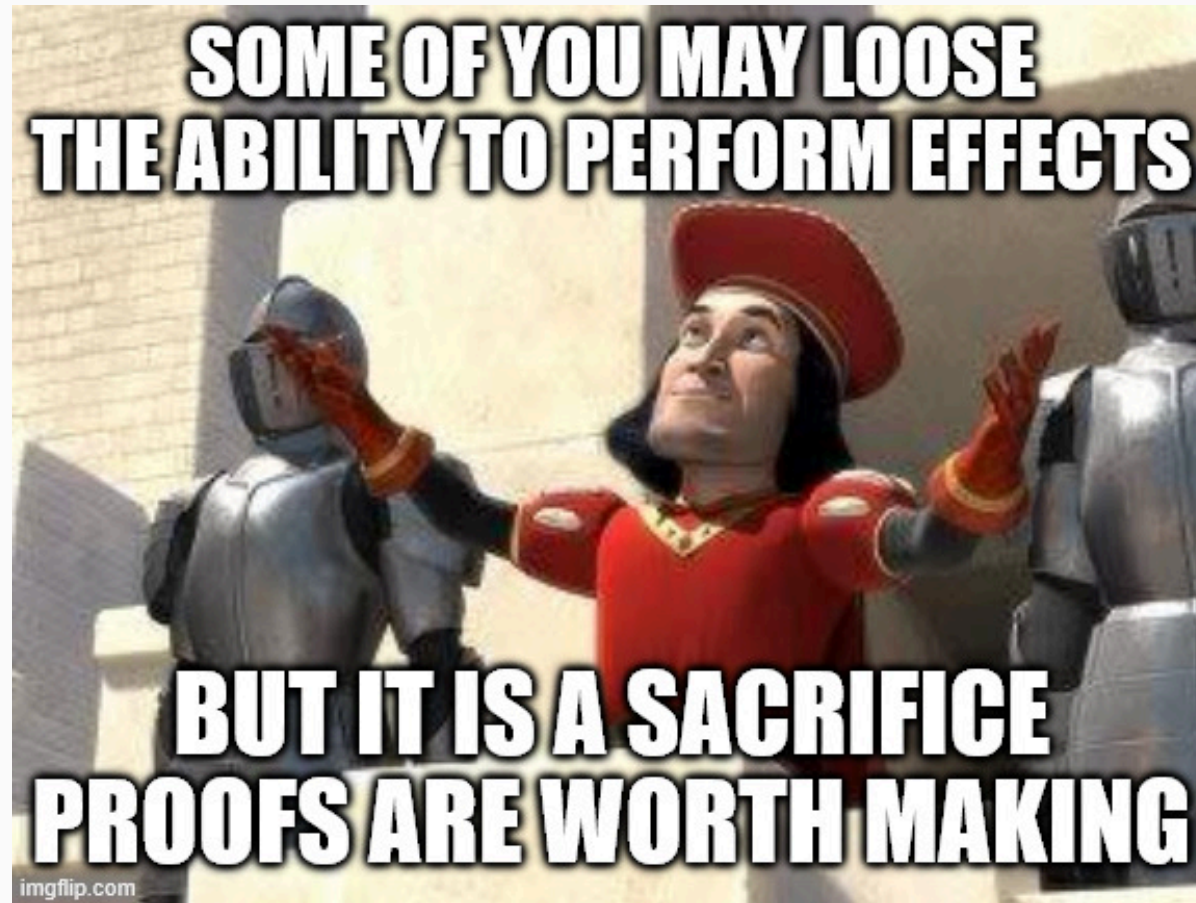
```
Theorem never_refund_more_than_paid : forall (events : list event),  
  refunded (replay events) ≤ paid (replay events).
```

- Written by **you**: an order never refunds more than it was paid
- True for **every** sequence of events: not just the cases you thought of testing
- Two lines to review, readable **without the proof**

Example: the proof

```
Lemma handle_keeps_refunds : forall o e,  
  refunded o ≤ paid o → refunded (handle o e) ≤ paid (handle o e).  
Proof.  
  intros o e H. unfold handle.  
  destruct (already_seen o e); [exact H|].  
  destruct e as [i n | i n]; cbn; [lia|].  
  destruct (refunded o + n ≤? paid o) eqn:E; cbn; [apply Nat.leb_le in E|]; lia.  
Qed.  
  
Theorem never_refund_more_than_paid : forall (events : list event),  
  refunded (replay events) ≤ paid (replay events).  
Proof.  
  intro events. unfold replay.  
  assert (H : refunded empty ≤ paid empty) by (cbn; lia).  
  revert H. generalize empty as o.  
  induction events as [| e es IH]; intros o H; cbn; [exact H|].  
  apply IH, handle_keeps_refunds, H.  
Qed.
```

- Written by **the agent**: first a lemma about **one** event, then induction over **all** of them
- Qed.: Rocq accepted every step. A wrong proof **does not compile**
- You do not have to read it, only the statement



Shipping proved code

Rocq **extracts** the proved function to a real language.

The rest of the app calls it **like any other library**.

Event processing: an order is a list of events

```
Inductive event :=
| Paid (id amount : nat)
| Refunded (id amount : nat).

Record order := mk_order {
  seen : list nat;    (* ids already handled *)
  paid : nat;
  refunded : nat
}.

Definition handle (o : order) (e : event) : order :=
(* The order, rebuilt from its events. *)
Definition replay (events : list event) : order :=
  fold_left handle events empty.
```

- Every change is an **event**: a payment, a refund
- The order is **replayed** from its events, one handle at a time
- Queues deliver **at least once**: the same event can arrive **twice**
- `handle` is a pure function: proved in Rocq, called by the consumer



Event processing: Jack's refunds, proved

```
Theorem never_refund_more_than_paid : forall (events : list event),  
  refunded (replay events) ≤ paid (replay events).
```

Whatever events arrive, in whatever order: **never more refunded than paid**.

```
Theorem delivered_twice_counted_once : forall (o : order) (e : event),  
  handle (handle o e) e = handle o e.
```

An event delivered twice is **counted once**.

Friday, 22h57 **cannot happen**.

Shipping it: extraction to OCaml

```
From Stdlib Require Import Extraction ExtrOcamlBasic ExtrOcamlNatInt.  
From Solid Require Import Orders.  
  
Extraction "orders.ml" handle replay.
```

Rocq: theories/Orders.v

```
Inductive event :=  
| Paid (id amount : nat)  
| Refunded (id amount : nat).  
  
Record order := mk_order {  
  seen : list nat; (* ids already handled *)  
  paid : nat;  
  refunded : nat  
}.
```

OCaml: orders.mli, written by dune build

```
type event =  
| Paid of int * int  
| Refunded of int * int  
  
type order = {  
  seen : int list;  
  paid : int;  
  refunded : int;  
}  
  
val handle : order → event → order  
val replay : event list → order
```

Shipping it: a Kafka consumer

```
(* consumer.ml: the thin shell around the proved core *)
let rec loop state =
  match Kafka.consume ~timeout_ms:1000 topic partition with
  | Kafka.Message (_, _, offset, payload, _) →
      let event = Codec.decode payload in          (* parse *)
      let next = Orders.handle state event in      (* proved *)
      Store.save next;                             (* persist *)
      Kafka.store_offset topic partition offset;   (* acknowledge *)
      loop next
  | Kafka.PartitionEnd _ → loop state
```

- The shell only **parses**, **persists** and **acknowledges**: a few lines, reviewed by hand
- A crash between save and the acknowledgement? Kafka delivers the event **again**:
delivered_twice_counted_once says it is harmless
- The business rule lives in `Orders.handle`, **proved**

Frontend: a reducer is a pure function

```
const [state, dispatch] = useReducer(reducer, initialState);  
// reducer: (state, event) ⇒ state
```

- Same state, same event: **same next state**, and no side effects
- React counts on it: in Strict Mode, it calls your reducer **twice**



Rocqducers: github.com/vbergeron/rocqducers

Remember the refund button?

```
Inductive state := Idle | Loading.
```

```
Inductive event := Click | Success | Failure.
```

```
Definition reducer (s : state) (e : event) : state :=  
  match s, e with  
  | Idle, Click => Loading  
  | Loading, Success => Idle  
  | Loading, Failure => Idle  
  | _, _ => s  
end.
```

Remember the refund button?

```
(* A second click while the request is in flight does nothing. *)  
Theorem click_while_loading_is_ignored :  
  reducer Loading Click = Loading.  
Proof. reflexivity. Qed.
```

A click while the request is in flight **does nothing**.

- **Undo / redo** for any reducer: Undo reverses Do, Redo reverses Undo (Rocqducers' UndoList)
- **Branching history**: no edit is ever lost (UndoTree)
- **Wizards and forms**: Submit is only reachable from a valid state
- **Shopping cart**: the total is the sum of its lines, a promo code applies once
- **Permissions**: a button is shown exactly when the rule allows it
- **Connection and auth**: no request leaves after logout
- **Selection, pagination, drag and drop**: indices stay in bounds, a reorder is a permutation
- **Optimistic updates**: a rollback restores exactly the previous state

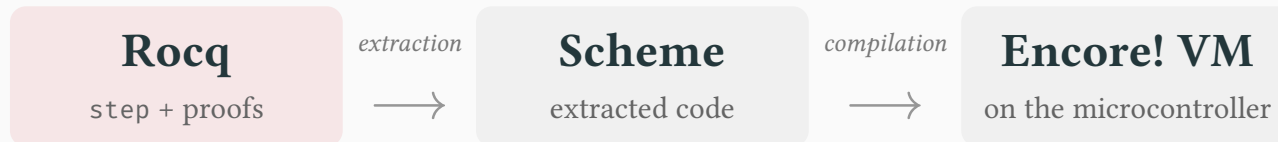
Embedded firmware: the device is a state machine

```
step : state → cmd → state * resp

Record state := mk_state {
  pin : list nat;
  puk : list nat;
  tries : nat;      (* PIN attempts left *)
  puk_tries : nat;  (* PUK attempts left *)
  auth : bool       (* PIN verified *)
}.

Inductive cmd :=
| Verify (guess : list nat)
| Change (new_pin : list nat)
| Unblock (puk_guess new_pin : list nat)
| ...
```

- Your SIM card's PIN: 3 wrong tries and it blocks, the PUK unblocks it, 10 wrong PUKs and the card is dead
- The whole logic is **one pure function**, written and proved in Rocq
- The chip only **reads commands** and **sends answers**
- That same function **runs on the microcontroller**, in the **Encore!** VM: what runs is what was proved



From Rocq to Metal: A Pipeline for Formally Verified Microcontroller Firmware: arXiv:2606.02651 · Encore!: github.com/vbergeron/encore

Embedded firmware: what the card guarantees

```
Theorem no_pin_no_access : forall (s : state) (c : cmd),  
  unlocked (next_state s c) → unlocked s ∨ right_pin s c.
```

The card **only unlocks with the right PIN**.

```
Theorem blocked_forever : forall (s : state) (cmds : list cmd),  
  blocked s → blocked (run s cmds).
```

A card blocked for good **stays blocked**, whatever you send it.

PIN and PUK logic of a SIM card: encore-benchmarks, workload W4

What it changes in your architecture

A proved core, a thin shell: the same shape three times

	The shell reads	The core decides	The shell does
Frontend	clicks	reducer	renders the new state
Service	Kafka messages	handle	saves, acknowledges
Firmware	commands from the phone	step	sends the answer

- The core is **pure**: state and event in, new state out. That is what makes it provable
- The shell does all the I/O, and **no business rule**: small enough to review by hand
- The boundary enforces itself: the core is generated code, with no access to the network, the disk or the clock

What it changes in your codebase

- **Tests move to the shell:** a theorem covers every input of the core, tests cover the I/O around it
- **Review moves to the specification:** read the three lines of the theorem, not the proof. The AI writes the proof, Rocq checks it
- **CI checks the proofs on every build:** a broken proof fails the build like a failing test, and the shipped code is regenerated from the proved one
- **The rules are written down:** “never refund more than was paid” is in the codebase, checked, and cannot rot

Be the one who writes down **what must be true.**

Thank you for your attention

Questions?



Slides

<https://vbergeron.github.io/rocq-solid/decks/rocq-solid.pdf>



Site

<https://vbergeron.github.io/rocq-solid/>