RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Inria

From RFC to Proof

Formally verifying a real-world IoT protocol in Rust

Not tested. Proved.

William Takeshi Pereira · Inria Paris, AIO team

Hello, I’m William

William Takeshi Pereira

Research engineer at Inria Paris, AIO team

What I work on

Embedded Rust and formal verification

Projects

lakers (EDHOC in Rust) and embedded-cal (verified crypto for microcontrollers)

Find me

github.com/williamtakeshi

Part zero: Formal proofs

Let's prove a simple sort function

```
fn sort(v: Vec<u32>) → Vec<u32> {
  /* … */
}

#[test]
fn sorts_example() {
  assert_eq!(sort(vec![3, 1, 2]),
             vec![1, 2, 3]);
}
```

Let's prove a simple sort function

```rust
#[hax_lib::ensures(|r| true)]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

$$\forall i,j.\ i<j<r.len() \Rightarrow r[i]<r[j]$$

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  forall(|i: usize| forall(|j: usize|
    implies(i < j && j < r.len(),
            r[i] < r[j])))))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

$$\forall i,j.\ i<j<r.\mathrm{len}() \Rightarrow r[i]<r[j]$$

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  forall(|i: usize| forall(|j: usize|
    implies(i < j && j < r.len(),
            r[i] < r[j])))))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

$\forall i,j.\ i<j<r.len() \Rightarrow r[i]<r[j]$

$[1,1] \Rightarrow [1,1]$

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  forall(|i: usize| forall(|j: usize|
    implies(i < j && j < r.len(),
            r[i] ⩽ r[j])))))]
fn sort(v: Vec<u32>) → Vec<u32> {
  /* … */
}
```

$$\forall i,j.\ i<j<r.\mathrm{len}() \Rightarrow r[i] \leqslant r[j]$$

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  forall(|i: usize| forall(|j: usize|
    implies(i < j && j < r.len(),
            r[i] ⩽ r[j])))))]
fn sort(v: Vec<u32>) → Vec<u32> {
  /* … */
}
```

$\forall i,j.\ i<j<r.len() \Rightarrow r[i] \leqslant r[j]$

`[3,2,1] ⇒ [3]`

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  Prop::from(r.len() == v.len())
  & sorted(&r))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

$$r.len() = v.len() \land sorted(r)$$

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  Prop::from(r.len() == v.len())
  & sorted(&r))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

`r.len()=v.len() ∧ sorted(r)`

`[3,2,1] ⇒ [3,3,3]`

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  Prop::from(r.len() == v.len())
  & sorted(&r)
  & forall(|i: usize| implies(i < r.len(),
      exists(|j: usize|
        j < v.len() && r[i] == v[j]))))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

```
… ∧ ∀i<r.len() ∃j<v.len()
r[i] = v[j]
```

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  Prop::from(r.len() == v.len())
  & sorted(&r)
  & forall(|i: usize| implies(i < r.len(),
      exists(|j: usize|
        j < v.len() && r[i] == v[j]))))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

… ∧ ∀i<r.len() ∃j<v.len()
r[i] = v[j]

[1,1,2] ⇒ [1,2,2]

Let's prove a simple sort function

```
#[hax_lib::ensures(|r|
  sorted(&r)
  & forall(|x: u32| count(&r, x) == count(&v, x)))]
fn sort(v: Vec<u32>) -> Vec<u32> {
  /* … */
}
```

$$\text{sorted}(r) \land \forall x.\ \text{count}(r, x) = \text{count}(v, x)$$

Works for every possible input! 🥳🎉

Agenda

A real protocol, real Rust, real proofs

1. **From RFC to code:** how protocols reach us
2. **Proving Rust does what it should**
3. **hax and Lean:** proofs for your Rust
4. **Bugs the proofs found,** and AI
5. **Proofs in CI:** zero binary cost
6. **What a proof does not give you**

Part one: From RFC to code

How a protocol is born at the IETF

EDHOC went from Internet-Draft to RFC 9528 in 2024; its PSK extension is still a draft

Internet-Draft

Authors propose a design

Working group

Review, then revision after revision

RFC

A published standard, stable and citable

10 revisions of the PSK draft so far, several after formal analysis

Then the design is checked as a model

Tools such as ProVerif and Tamarin analyze an abstract version of the protocol

What it proves

Secrecy and authentication of the design

What it models

Messages and keys, not bytes or buffers

What it catches

Design flaws, before anyone writes code

Where it stops

It says nothing about any implementation

Usually, this is where the formal work ends.

Then developers implement it, from prose

Specs are written by protocol designers
Prose, CDDL and test vectors, not code

Code is written by developers
Often new to the protocol, sometimes new to the language

Checked with test vectors
They cover the happy path, not malformed input

Nothing links the model to the code
The proved design and the shipped code can drift apart

This talk
Prove that the Rust code does what the spec says

Where we prove things: EDHOC and its PSK extension

IETF LAKE working group: RFC 9528 (2024) and draft-ietf-lake-edhoc-psk (2026)

A key exchange for IoT

TLS-like guarantees in a few dozen bytes per message

PSK extension

Cheaper authentication and session resumption

Attacker-controlled input

Every message is CBOR, parsed by our Rust code

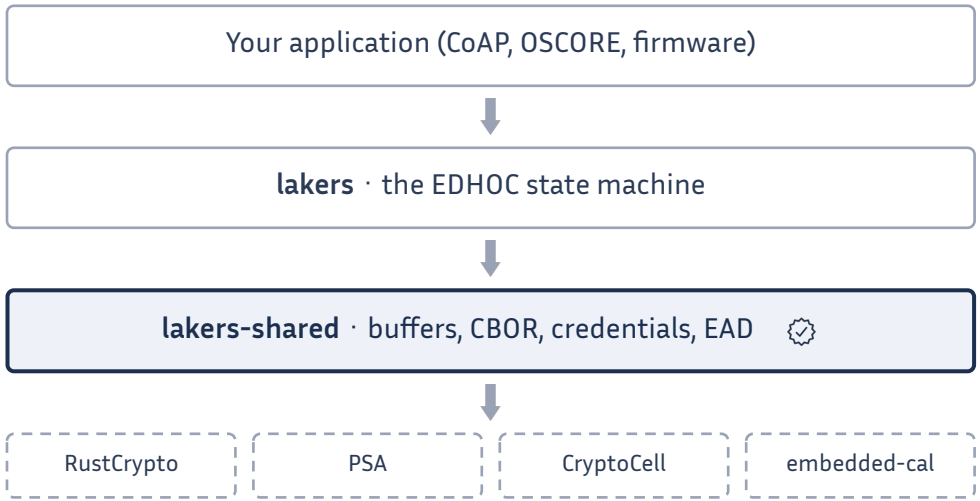
A panic is an outage

No OS, no reset: one bad packet stops the device

> The protocol details don’t matter today. What matters is the code that parses it.

lakers: EDHOC in Rust

no_std · no heap · runs on Cortex-M · github.com/lake-rs/lakers



Bindings

C (RIOT, Zephyr) and Python. In C, a parser bug is memory corruption.

Supports

RFC 9528 methods and the PSK draft

“But it’s Rust. Isn’t it already safe?”

Rust is a great starting point for verification, not the end of it

Rust guarantees	Rust does not guarantee
Memory safety	No panics: unwrap, indexing, overflow in debug
No data races	Correct arithmetic: as casts wrap silently
Types match	The code does what the RFC says

> Rust turns memory corruption into panics. On a device without an OS, a panic is still a remote kill switch.

Part two: Proving Rust does what it should

Testing vs proving

"Program testing can be used to show the presence of bugs, but never to show their absence" — Edsger W. Dijkstra

Tests

Check the inputs you thought of

Fuzzing

Check many inputs you did not think of

Proof

A property holds for **every** input that satisfies the precondition

2^{512} possible 64-byte messages. Tests sample the space; a proof covers all of it.

What can you prove?

Each property is a different tool and a different cost

Panic freedom

No out-of-bounds, no overflow, no unwrap on None, for all inputs

Functional correctness

The output matches a mathematical specification

Secret independence

Control flow and memory access do not depend on secrets

Protocol security

Authentication and secrecy, on a model of the protocol

Today: panic freedom on lakers-shared, plus key postconditions. The cheapest rung, and it found bugs.

Part three: hax and Lean

hax: from Rust to a proof

Verification toolchain



Frontend
Hooks into rustc: sees exactly what the compiler sees

Engine
&mut becomes “take value, return new value”; loops become folds

Obligations are discharged automatically where possible: SMT in F*, tactics such as grind in Lean.

Lean: where our proofs are going

lakers is proved in F today; we are moving the proofs to hax's Lean backend*

Same Rust annotations

requires, ensures and invariants stay in the Rust

Lean backend

hax extracts Rust to Lean 4

Automation

grind closes goals or shows a counterexample

Ecosystem

A large community and libraries such as Mathlib

Annotations live in your Rust code

shared/src/buffer.rs

```rust
#[hax_lib::requires(short.len() ⩽ long.len())]
const fn copy_into_longer(long: &mut [u8], short: &[u8]) {
    let mut cursor = short.len();
    let original_length = long.len();
    while cursor > 0 {
        hax_lib::loop_decreases!(cursor);
        hax_lib::loop_invariant!(cursor ⩽ short.len() && long.len() == original_length);
        cursor = cursor - 1;
        long[cursor] = short[cursor];
    }
}
```

requires · the contract

loop_decreases · termination

loop_invariant · in bounds, length kept

Proving it in Lean: extract, then build

One config file and two commands, like any cargo project

Config

```
# hax.toml
[scenario.lakers]
backend = "lean"
```

Run

```
$ cargo hax extract lakers
$ lake build

Build completed successfully
```

> When a proof fails, grind prints the inputs that break your spec: a counterexample, not a guess.

How panic freedom is proved

In the extracted model, every risky operation carries a precondition

Rust operation	Precondition in the model
a + b	The result does not overflow
s[i]	i < s.len()
x.unwrap()	x is Some
a / b	b ≠ 0

> **If the extracted model checks, none of these can fail.**

Part four: Bugs the proofs found, and AI

Scope: lakers-shared

Where bytes from the wire are parsed, so where an attacker can reach a panic

~3,300

lines of Rust in scope

buffers · CBOR decoder · connection IDs · credentials · EAD parsing · KDF info encoding · errors, consts

Proved

lakers-shared: panic freedom for every function, plus key postconditions

Extracted

lakers (state machine) goes to F* in CI; its full proof is next

Excluded

FFI, Debug impls, test helpers

Bug #1: abs() on i32::MIN

EAD labels: a negative label marks a critical item

```
let is_critical = label < 0;
let label = label.abs();           // overflow when label == i32::MIN
let label = label.unsigned_abs(); // fixed
```

Reachable from the wire

Debug: panic. Release: wraps and stays negative.

One value in 4 billion

A test needs exactly i32::MIN. The prover asks on the first try.

Bug #2: the 0x20 offset in i8()

CBOR negative integers: an opcode offset was reused on the argument byte

```
} else if (0x20..=0x37).contains(&n) {
    Ok(-1 - (n - 0x20) as i8)              // opcode byte: strip 0x20
} else if 0x38 == n {
    Ok(-1 - (self.read()? - 0x20) as i8)   // argument byte: wrong
```

Wrong value

0x38 0x18 decodes to 7 instead of −25.

Any argument below 0x20. Debug: panic.

Release: silent wrap. Hax could not prove b ≥ 32.

Bug #3: s[..len] in ConnId::from_decoder

Connection IDs: the decoder trusted the length the peer sent

```
let len = ConnIdType::classify(decoder.current()?)
    .ok_or(CBORError::DecodingError)?.length();      // up to 24
// fixed: reject IDs the buffer cannot hold
if len > MAX_CONNID_ENCODED_LEN { return Err(CBORError::DecodingError); }
s[..len].copy_from_slice(decoder.read_slice(len)?);   // s holds 8
```

Reachable from the wire

A peer sends a 24-byte C_I in message 1.

No authentication needed

One unauthenticated message stops the device.

AI turns prover errors into plain language

How we used AI on lakers: reading F and Lean errors, not writing the proofs*

F* says

```
* Error 19 at Lakers_shared.fst(…):
  - Subtyping check failed
  - Expected type: n: usize{v n ⩽ length buffer}
  - The SMT solver could not prove the query
```

AI says

```
cursor += consumed may run past the end
of buffer: F* cannot see that
consumed ⩽ input.len().
Fix: add an ensures to parse_single_ead.
```

> The checker stays the judge: AI only helps a person read what failed and why.

Part five: Proofs in CI

Proofs in CI

.github/workflows/build-and-test.yml · every pull request re-extracts and re-proves

```yaml
generate-fstar:
  - uses: hacspec/hax-actions@main
    with: { hax_reference: "<pinned commit>", fstar: v2025.10.06 }
  - run: cargo-hax -C -p lakers-shared \; into -i '-lakers_shared::ffi::**' fstar
  - run: cargo-hax -C -p lakers … \; into fstar
  - run: # fail if any item is "not implemented yet"
  - working-directory: ./shared/proofs/fstar/extraction
    run: make verify
```

Red on regression

A new unchecked index or unwrap fails CI, naming the operation

Pin versions

hax and F* move fast; pin both

Auditable

The F* code ships as a release artifact

Proofs let you change code without fear, by hand or with AI

The guarantees are re-proved on every pull request, whoever wrote the change

```
# a refactor, an optimization, an AI-suggested rewrite: same gate
$ git push

# CI re-extracts the crate and re-proves every function
$ cargo hax into fstar && make verify

  all obligations proved   → the guarantees still hold: merge
  1 obligation failed      → CI names the exact operation
```

Refactor freely

Readability and speed changes keep their guarantees

Review the intent

Reviewers read specs, not indexes

Same gate for AI

AI code passes the same proofs

Zero runtime cost

hax_lib macros only do something under --cfg hax, during extraction

cargo build

```
// #[hax_lib::requires(…)] expands to nothing
pub const fn from_int_raw(raw: u8) -> Self
```

cargo hax into fstar

```
(* becomes an F* precondition *)
val from_int_raw (raw: u8) : Prims.Pure …
```

Same binary	Same size	Same speed

> The proof is a build-time artifact: it runs in CI, not on the device.

Part six: What a proof does not give you

What a proof does *not* give you

The spec can be wrong

If ensures says the wrong thing, the proof proves the wrong thing

There is a trusted base

rustc, hax's translation, F*, Z3, and the hardware

Scope is limited

lakers-shared is proved; FFI and the main crate's full proof are not (yet)

No side-channel guarantee

Panic freedom says nothing about timing or physical leakage

Protocol security is separate

It comes from the analysis of the RFC, not from the implementation proof

Summary

TAKEAWAYS

- ✓ **A proved model is not proved code:** prove the Rust too
- ✓ **hax** proves your existing Rust, no rewrite
- ✓ Start with **panic freedom** on your parsers: the cheapest rung, real bugs
- ✓ **Put it in CI:** zero runtime cost, every change re-proved
- ✓ **AI proposes, the prover decides:** refactor without fear

github.com/lake-rs/lakers · github.com/cryspen/hax · github.com/lake-rs/embedded-cal

NEXT

→ Full proof of the lakers state machine

→ Port the lakers proofs from F* to Lean

→ Functional specs from RFC 9528 and the PSK draft

Thank you - Questions?

Not tested. Proved.

github.com/lake-rs/lakers · github.com/cryspen/hax · github.com/lake-rs/embedded-cal