RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Inria

CE · Labs
CRYPTOGRAPHY ENGINEERING

embedded-cal

A Formally Verified Cryptographic Provider for Embedded Platforms

Robin Hundt
William Takeshi Pereira

Who we are

Robin Hundt
Cryptography Engineer
CE Labs (spun out of Cryspen)

implements formally verified cryptography in Rust. Previously he implemented secure multi-party computation protocols in Rust at the ENCRYPTO group (TU Darmstadt) and the SINE Foundation.

github.com/robinhundt

William Takeshi Pereira
Research Engineer
Inria Paris - AIO team

Rust enthusiast working on embedded Rust and formal verification. Currently focused on embedded-cal, a formally verified cryptographic abstraction layer for microcontrollers.

github.com/williamtakeshi

Motivation - Can you spot the error?

```rust
pub struct KeyPolicy {
    pub exportable: bool,
    pub session_only: bool,
}

pub struct Template {
    pub token: bool,
    pub sensitive: bool,
    pub extractable: bool,
}
```

```rust
/// A key derived under `parent` must be no more permissive
/// than `parent`.
pub fn derived_template(parent: KeyPolicy, requested: KeyPolicy)
    -> Template {
    Template {
        token: !(parent.session_only || requested.session_only),
        sensitive: !parent.exportable || !requested.exportable,
        extractable: parent.exportable || requested.exportable,
    }
}
```

Part one: Abstraction and portability

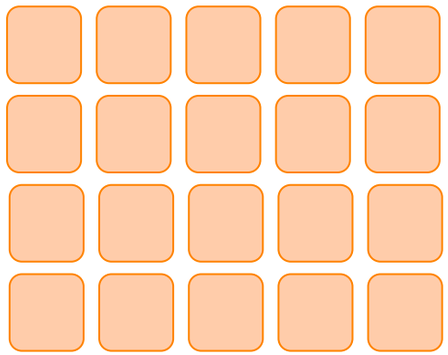
Crypto on microcontrollers is re-implemented, per chipset

A mistake here doesn't crash, it produces a wrong result that looks correct.
This is exactly the class of bug that testing is worst at catching, and formal methods are good at catching.
Building secure applications shouldn't require hardware expertise

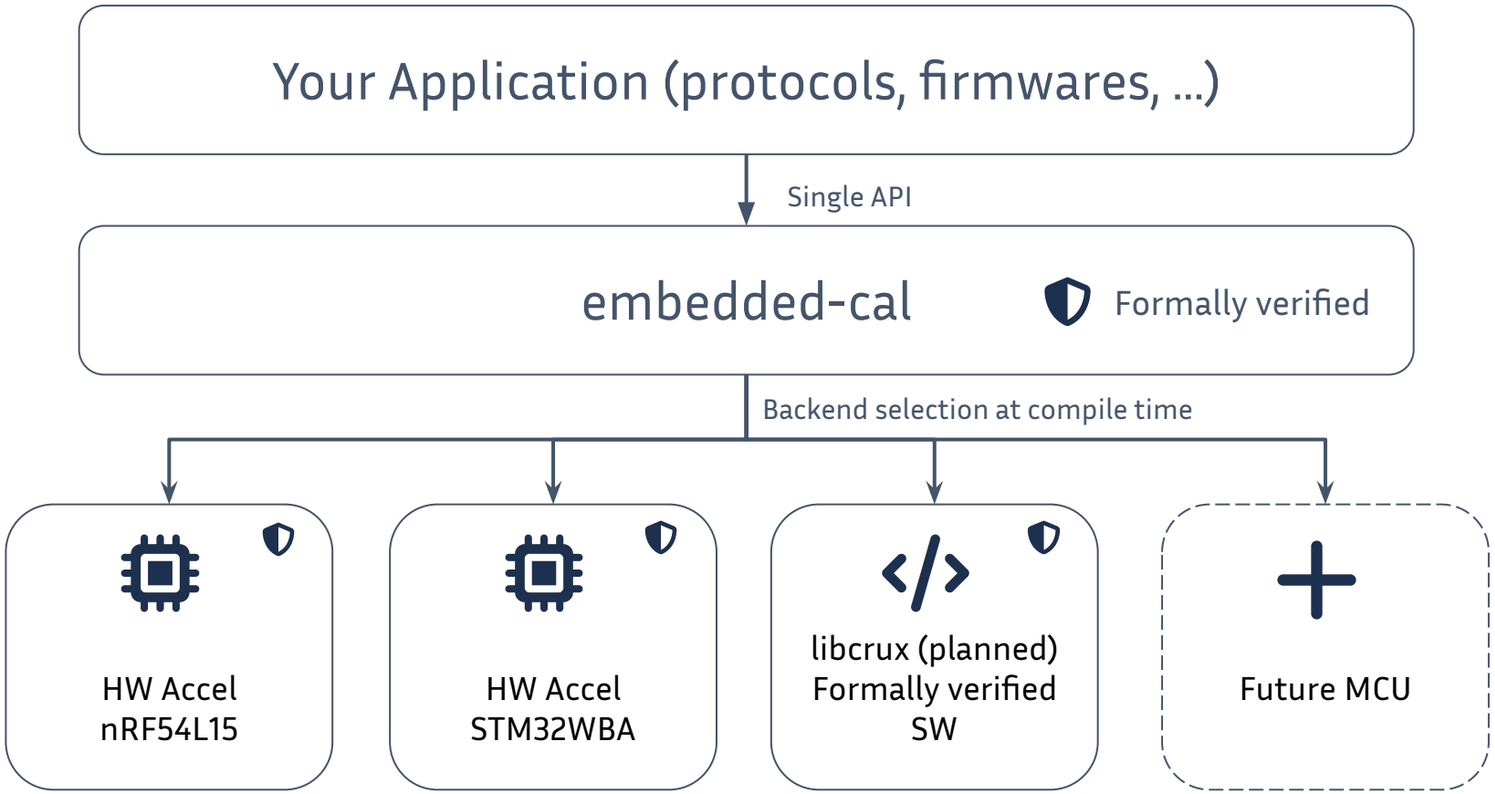
What if the abstraction that talks to the hardware could be proven not to have this class of bug?

Platform-agnostic cryptographic abstraction layer

Before



N × M hand-written integrations



With new embedded-cal updates, every application gains new hardware accelerated capabilities

embedded-cal — Overview

A lightweight cryptographic abstraction layer for constrained embedded systems

Unified Rust API

Trait-based driver layering; verified SW provides top-level fallback

Lightweight

Const evaluation eliminates unused code paths at build time

Hardware-Aware

Native acceleration on nRF54L15 (CRACEN) and STM32WBA55CG; possible to extend to more families

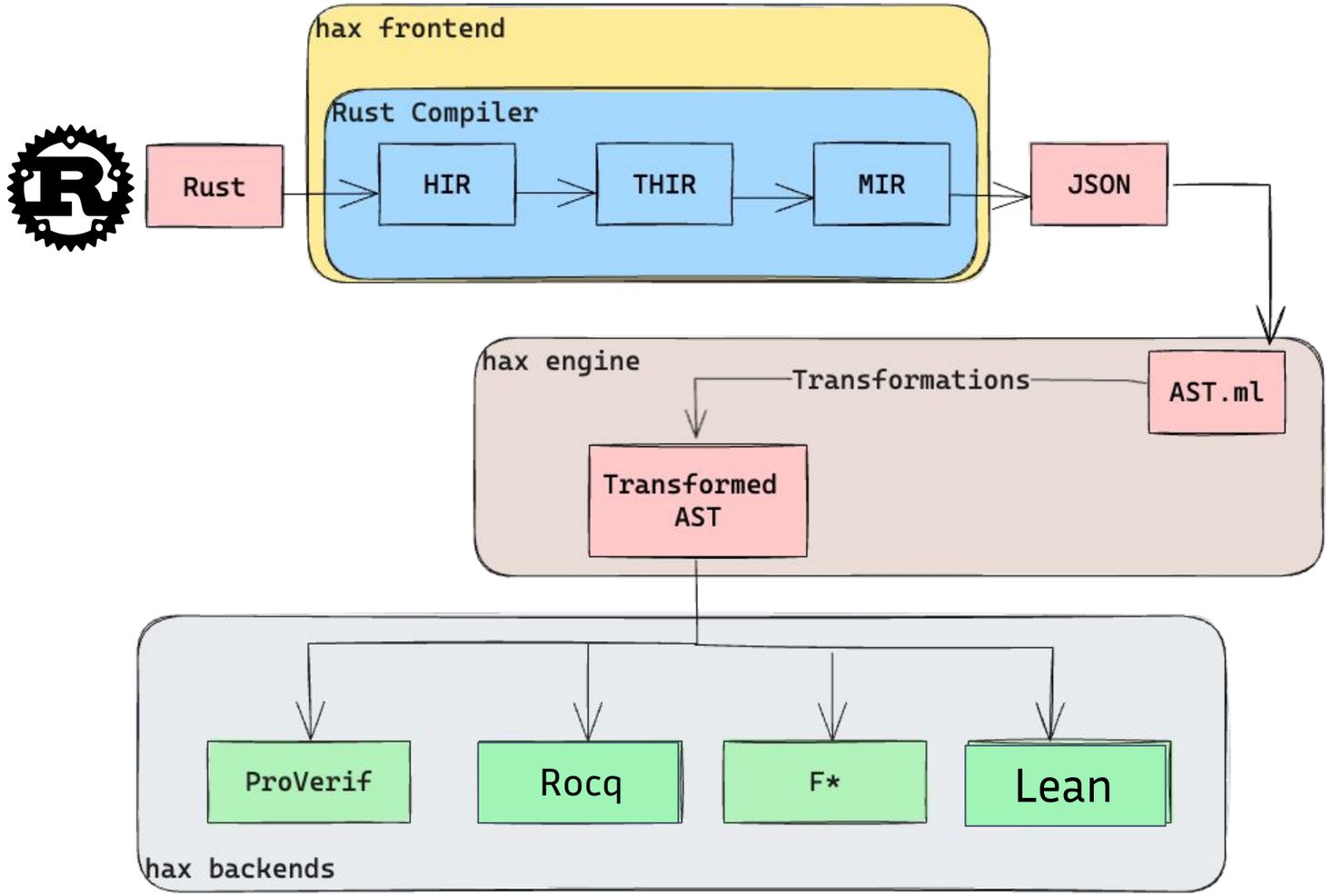
Formal Verification

Correctness, runtime safety & secret independence via the Hax framework, the future SW fallback (libcrux) is already formally verified

Open-source: github.com/lake-rs/embedded-cal

Part two: Formal verification

Hax - A Verification Toolchain for Rust



Can you spot the error?

```rust
pub struct KeyPolicy {
    pub exportable: bool,
    pub session_only: bool,
}

pub struct Template {
    pub token: bool,
    pub sensitive: bool,
    pub extractable: bool,
}
```

```rust
/// A key derived under `parent` must be no more permissive
/// than `parent`.
pub fn derived_template(parent: KeyPolicy, requested: KeyPolicy)
    -> Template {
    Template {
        token: !(parent.session_only || requested.session_only),
        sensitive: !parent.exportable || !requested.exportable,
        extractable: parent.exportable || requested.exportable,
    }
}
```

Can you spot the error?

```rust
pub struct KeyPolicy {
    pub exportable: bool,
    pub session_only: bool,
}

pub struct Template {
    pub token: bool,
    pub sensitive: bool,
    pub extractable: bool,
}
```

```rust
/// A key derived under `parent` must be no more permissive
/// than `parent`.
#[hax_lib::ensures(|t| hax_lib::implies(!parent.exportable, !t.extractable)
                    & hax_lib::implies(parent.session_only, !t.token)
                    & (t.sensitive == !t.extractable))]
pub fn derived_template(parent: KeyPolicy, requested: KeyPolicy)
    -> Template {
    Template {
        token: !(parent.session_only || requested.session_only),
        sensitive: !parent.exportable || !requested.exportable,
        extractable: parent.exportable || requested.exportable,
    }
}
```

Can Hax and Lean spot the error?

```
cargo hax extract showcase

// in hax.toml

[scenario.showcase]
backend = "lean
```

+ A small proof of the derived_template function using automated tactics

```
cd proofs/showcase/lean && lake exe cache get && lake build

error: Showcase/Verification/ProofObligations.lean:42:40: `grind` failed

  [eqc] Equivalence classes
    [eqc] {false, parent.exportable, parent.session_only,}
    [eqc] {true, requested.exportable}
```

Can you spot the error? - Fixed

```rust
/// A key derived under `parent` must be no more permissive
/// than `parent`.
#[hax_lib::ensures(|t| hax_lib::implies(!parent.exportable, !t.extractable)
                    & hax_lib::implies(parent.session_only, !t.token)
                    & (t.sensitive == !t.extractable))]
pub fn derived_template(parent: KeyPolicy, requested: KeyPolicy)
    -> Template {
    Template {
        token: !(parent.session_only || requested.session_only),
        sensitive: !parent.exportable || !requested.exportable,
        extractable: parent.exportable && requested.exportable,
    }
}
```

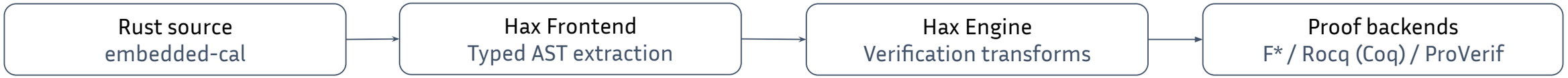
```
i [1738/1740] Built Showcase.Verification.ProofObligations

Build completed successfully (1740 jobs).
```

Verification Toolchain

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

Verification Toolchain



Both execution paths are verified on CI/CD, zero runtime cost

Hardware Path

Rust-level interface code verified for panic freedom and correctness. Hardware interface verification remains an open challenge

Software Path (libcrux)

Fully verified: correctness, safety and secret independence end-to-end

Classes of bug we can rule out entirely

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

Panic-freedom

No panics at runtime, no out-of-bounds access, no integer overflow

Functional Correctness

Input/output behavior matches the mathematical specification of each algorithm

Bug class 1: panic-freedom

- On embedded, there's often no OS to catch a panic
- A panic, exception, or abort can mean total denial-of-service on a device meant to run forever, unattended
- hax proves the glue code cannot panic — for every possible input, not just the ones we tested

Panic-freedom

No panics at runtime, no out-of-bounds access, no integer overflow

Bug class 2: functional correctness of the glue

- Beyond “doesn't crash” and “memory safe” — does the glue code actually do what it claims?
- E.g., correctly marshals data in and out of the hardware crypto engine, in the format and order it expects
- **We can prove functional correctness of this glue code itself**

Functional Correctness

Input/output behavior matches the mathematical specification of each algorithm

What this does NOT give you

Silicon is trusted
Verification assumes the hardware behaves as documented. A lying or buggy chip is out of scope.

No side-channel or timing guarantees
These proofs are about functional behavior, not physical leakage.

Invariants are only as good as specified
If the safety invariant itself is wrong or incomplete, the proof won't catch that.

libcrux & libcrux-iot: what's already verified

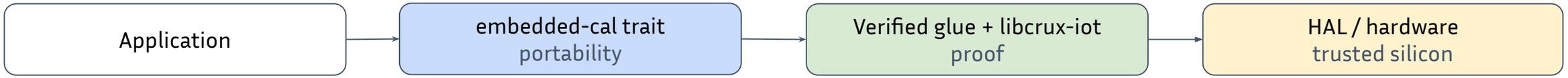
libcrux
A formally verified cryptographic library, built with hax, the crypto core.

libcrux-iot
The subset adapted for constrained embedded platforms — what embedded-cal draws on.

Partially proven: panic freedom and functional correctness of the primitive implementations themselves.

So the crypto core is verified. What about everything wrapped around it?

The full picture, end to end



- Portability: one interface, many boards (traits)
- Proof: glue + crypto core verified with hax / libcrux-iot
- Assumption: silicon behaves as documented

Summary

CONTRIBUTIONS

✓ embedded-cal unifies hardware-accelerated and formally verified software cryptography for the Rust embedded ecosystem

✓ Verified for functional correctness, runtime safety, and secret independence via the Hax framework

✓ Benchmarks on nRF54L15 and STM32WBA55CG, showing the speedup in hardware accelerated cases

FUTURE WORK

→ Replace RustCrypto fallback with libcrux (formally verified)

→ Extend hardware support to additional MCU families

→ Integrate into lakers for real-world validation

github.com/lake-rs/embedded-cal

Thank you - Questions?

github.com/lake-rs/embedded-cal

Can Hax and Lean spot the error?

+ A small proof of the derived_template function using automated tactics

```
@[spec]
theorem derived_template.spec.proof (parent : KeyPolicy)
    (requested : KeyPolicy) : derived_template.spec parent requested := by
  unfold derived_template.spec derived_template.post derived_template
  hax_mvcgen [hax_lib.prop.implies,
    hax_lib.prop.Prop.Insts.CoreOpsBitBitAndTProp.bitand,
    hax_lib.prop.Prop.Insts.CoreConvertFromBool,
    core.convert.Into.Blanket.into,
    core.convert.From.Blanket.from] <;> grind
```

Motivation

Building secure applications shouldn't require hardware expertise

1. **Use of C wrappers...** but they may introduce memory safety and verification gaps, leading to duplicated, inconsistent, and potentially insecure implementations across projects

2. **Use of software-only solutions...** but they are often slower, consume more power, and rarely provide formal guarantees against timing or data-dependent side channels

3. **Use of platform-specific code...** but it ties the user application to one MCU family and forces you to maintain unverified crypto integration code

Question: How do we design a unified, hardware-aware, formally verified crypto layer for constrained IoT?

Formal Verification with Hax

Functional Correctness

Input/output behavior matches the mathematical specification of each algorithm

Runtime Safety

No panics at runtime, no out-of-bounds access, no integer overflow

Secret Independence

Control flow, memory accesses, and timing do not depend on secret data (constant-time)

Verification Toolchain

Rust source
embedded-cal
→
Hax Frontend
Typed AST extraction
→
Hax Engine
Verification transforms
→
Proof backends
F* / Rocq (Coq) / ProVerif

Both execution paths are verified on CI/CD. Zero runtime cost

Hardware Path

Rust-level interface code verified for secret independence. Hardware interface verification remains an open challenge

Software Path (libcrux)

Fully verified: correctness, safety and secret independence end-to-end

Evaluation

Execution time (ms) — Hardware accelerated (HW) vs. RustCrypto software (SW)

Operation	STM HW (ms)	STM SW (ms)	nRF HW (ms)	nRF SW (ms)	*HW speedup*
AES-ECB-128	**0.02**	0.31	**0.02**	0.18	*9–15×*
SHA2-256	**0.02**	0.24	**0.02**	0.15	*8–12×*
ECDSA verify	**181**	1077	**1.82**	599	*6–329×*
ECDSA sign	**172**	1168	**2.31**	653	*7–283×*
ECC mult	**193**	540	**1.68**	300	*3–179×*

- ECC operations up to 329× faster with hardware acceleration
- nRF CRACEN: modern low-power design, 4–12 mA envelope
- STM32: constant current — energy saved via shorter execution time
- HW-backed operations need only thin peripheral drivers → smaller flash

Target Use Cases

Ariel OS

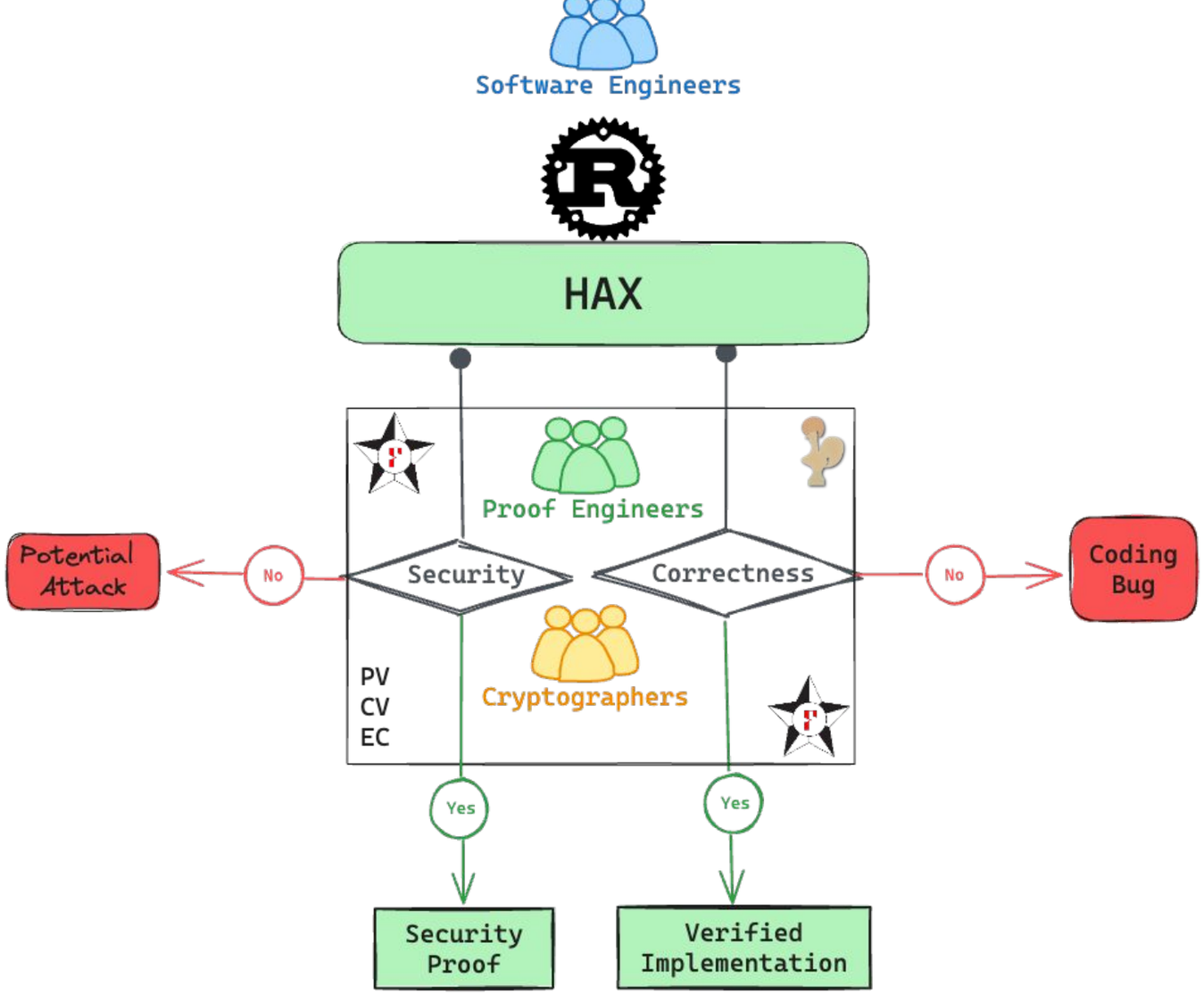
An emerging research OS for secure networked embedded systems. embedded-cal provides its formally verified, hardware-aware crypto module.

lakers / EDHOC (IETF RFC 9528)

Rust implementation of EDHOC. Current crypto provider relies on unverified code with no constant-time guarantees. embedded-cal is a direct drop-in replacement.

libOSCORE

Replaces the library's bespoke crypto layer with a unified, hardware-accelerated, verified backend.


Software Engineers
HAX
Proof Engineers
Potential Attack
No
Security
Correctness
No
Coding Bug
PV
CV
EC
Cryptographers
Yes
Yes
Security Proof
Verified Implementation

hax: Correctness Process